КАУФМАН В.Ш.

ЯЗЫКИ ПРОГРАММИРОВАНИЯ
концепции и принципы

(на правах рукописи)

ноябрь 1986

Московский ордена Ленина, ордена Октябрьской революции и ордена Трудового Красного Знамени государственный университет им. М.В.Ломоносова
Факультет вычислительной математики и кибернетики

МЕТОДИЧЕСКИЙ МАТЕРИАЛ

"ЯЗЫКИ ПРОГРАММИРОВАНИЯ"
Часть 1. ОСНОВНЫЕ АБСТРАКЦИИ.

1. КОНЦЕПТУАЛЬНАЯ СХЕМА ЯЗЫКА ПРОГРАММИРОВАНИЯ.

1.1. Что такое язык программирования

Естественно начать с характеристики изучаемого предмета. Но коротко охарактеризовать, что именно будем изучать, с какой целью и как, не просто (скоро станет понятно, почему). Конечно, нас будут интересовать "языки программирования" (ЯП). На сколь точно эти слова определяют сферу наших интересов? Одни скажут, что язык машин Тьюринга или алгоритмов Маркова - это ЯП, другие не согласятся с этим категорически. Одни признают язык управления заданиями в ОС ЕС языком программирования, другие приведут доводы против.

Такая ситуация на первый взгляд неприятна - собираемся изучать неизвестно что. Сделаем вывод, что нужно определить объем понятия "язык программирования" (его экстенсионал, т.е. множество обьектов, охватываемых этим понятием, множество его частных случаев).

Чтобы создать себе первую точку опоры, пойдем по простейшему пути - явно перечислим те конкретные языки, которые нас заведомо интересуют (их мы уверенно считаем "языками программирования"). Это Фортран, Паскаль, Бейсик, Лисп, Апл, Форт, Рефал, Ада. Однако вряд ли стало намного легче. Хочется иметь возможность на основе определения предсказывать новые частные случаи, в определении не перечисленные. Такое определение должно опираться на существенные свойства выбираемых для изучения языков - оно должно быть интенсиональным. Дадим одно из возможных интенсиональных определений ЯП.

Язык программирования - это инструмент для планирования поведения исполнителя.

Однако остаются основания для неудовольствия. Во-первых, известные нам языки программирования (Фортран, Алгол, Бейсик) служат не только для планирования поведения (машин), но и для обмена программами между людьми. Такая важнейшая функция существенно влияет на их устройство и принципы создания (хотя она все-же вторична - можно показать, что люди должны понимать и читать программы, даже не имея никаких намерений ими обмениваться; просто иначе достаточно крупной программы не создать). Эту функцию языка никак нельзя игнорировать при изучении ЯП.

Во-вторых, в нашем определении каждое слово нуждается в уточнении. Являются ли "инструментами для планирования поведения исполнителя" должностная инструкция, письменный стол, переговорное устройство, правила уличного движения, русский язык?

Но кое-чего мы добились - можем привести примеры ЯП, с которыми все согласны, и указать объекты, заведомо не являющиеся ЯП в соответствии с нашим определением (также рассчитывая на общее согласие) - левая тумба письменного стола, стойка питания БЭСМ-6, рубанок, автомобиль.

1.2. Метауровень

Взглянем на наши действия с позиции стороннего наблюдателя, отвлечемся (абстрагируемся) от своей роли соответственно автора и читателей на только что законченном начальном отрезке работы. Поднимемся, как говорят, на метауровень, с тем чтобы обозревать начальный отрезок (исходный уровень) в целом. Чем мы занимались?

Во-первых, приступили к изучению ЯП. Во-вторых, попытались добиться взаимопонимания (согласованного понимания) в вопросе о том, что такое ЯП. В-третьих, начали применять для достижения взаимопонимания метод последовательных уточнений.

Чего мы добилмсь и что осталось неясным? Стало яснее, что будем изучать. Почувствовали, что добиться взаимопонимания (даже по поводу привычных понятий) очень непросто. Осталось неясным , в частности, с какой позиции и с какой целью мы намерены изучать ЯП.

Постараемся в первом приближении устранить возможные неясности. Однако заниматься последовательными уточнениями многих важных понятий мы будем на протяжении всей нашей работы - ведь она не формальная, а содержательная, нас интересуют реально существующие, возникающие на наших глазах и развивающиеся объекты - живые языки программирования. Поэтому-то и невозможно дать исчерпывающего описания ЯП как понятия (это понятие живет вместе с нами).

Начнем с более внимательного рассмотрения преград, обычно возникающих на пути к взаимопониманию.

1.3. Модель передачи сообщения

Добиться взаимопонимания бывает очень сложно. Чтобы выделить возникающие здесь проблемы, рассмотрим следующую модель передачи сообщения.

Модель передачи сообщения

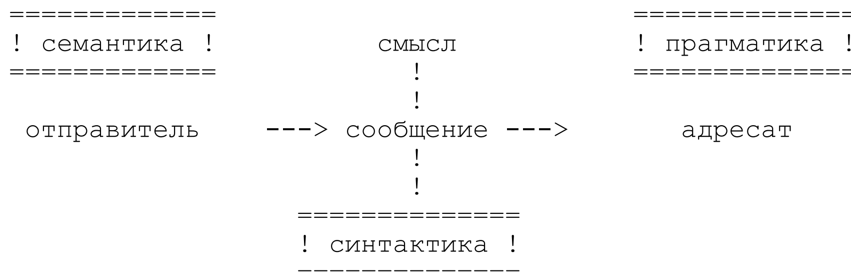


Рис. 1.1

В этой модели выделены понятия "отправитель" (автор, генератор сообщения), "адресат" (получатель, читатель, слушатель сообщения), собственно "сообщение" (текст, последовательность звуков), "смысл" сообщения (нечто обозначаемое сообщением в соответствии с правилами, известными и отправителю, и адресату).

В прямоугольниках обозначены названия наук, занимающихся соответственно правилами построения допустимых сообщений (синтактика), правилами сопоставления таким сообщениям смысла (семантика) и правилами, регулирующими использование сообщений (прагматика).

1.4. Классификация недоразумений

С помощью этой модели займемся классификацией недоразумений, возникающих при попытке установить взаимопонимание.

Автор (отправитель сообщения) может подразумевать одну структуру сообщения, а адресат (получатель) - другую, как в классическом королевском указе: "Казнить нельзя помиловать!". Это так называемое синтаксическое недоразумение. Автор может употребить слово с неточным смыслом, подразумевая один его оттенок, а адресат выберет другой.

Рассмотрим, например, фрагмент рецепта приготовления шоколадной помадки: "изредка помешивая, варить на слабом огне до тех пор, пока капля не станет превращаться в холодной воде в мягкий шарик". Не потому ли кулинарное искусство и является искусством, а не точной наукой, что разные повара, зная один и тот же рецепт, по-разному понимают слова "изредка", "медленно", "холодной", "мягкий", а также с разной частотой станут пробовать "превратить каплю в мягкий шарик". Естественно, и результаты у них будут разные. Это семантическое недоразумение.

Наконец, автору трудно иногда представить себе, какую интерпретацию может придать его сообщению адресат, если у них сильно различаются представление о мире или решаемые задачи.

Например, сообщение лектора о предстоящем коллоквиуме может быть воспринято студентами как призыв не посещать малоинформативные лекции, чтобы иметь время для работы с книгами. Это уже прагматическое недоразумение.

Нетрудно привести и другие примеры синтаксических, семантических и прагматических недоразумений при попытке достичь взаимопонимания.

1.5. При чем здесь взаимопонимание

Почему же в самом начале речь пошла о взаимопонимании и о стоящих на пути к нему преградах? В основном, по двум причинам.

Во-первых, ЯП - это инструмент для достижения взаимопонимания (безопаснее "взаимопонимания") с компьютерами и между людьми по поводу управления компьютерами. Поэтому в принципах построения, структуре, понятиях и конструктах ЯП находит свое отражение и сущность общей проблемы взаимопонимания, и взгляды их творцов на эту проблему, и конкретные методы ее решения.

Во-вторых, способ, которым люди преодолевают преграды на пути к взаимопониманию, содержит некоторые существенные элементы, остающиеся важными и при общении с компьютерами (в частности, при создании и использовании ЯП).

1.6. Как достигают взаимопонимания

Особенно бояться синтаксических недоразумений не стоит. Они касаются отдельных неудачных фраз и легко устраняются немедленным вопросом (устным или письменным). В ЯП это тоже не проблема - таких недоразумений там просто не бывает. Дело в том, что создатели ЯП руководствуются принципом однозначности - язык программирования должен быть синтаксически однозначным (т.е. всякий правильный текст на ЯП должен иметь единственную допустимую структуру).

Замечание 1. Итак, сформулирован один из принципов построения ЯП, отличающих их, скажем, от языков естественных. Такого рода общие принципы и концепции нас и дальше будут интересовать в первую очередь.
Конец замечания.

Семантические недоразумения опаснее. Если, скажем, слово "язык" будет ассоциироваться с кулинарным субпродуктом, ставшим весьма редким гостем прилавка, то недоразумение может не ограничиться пределами одной фразы. Большой язык, свежий язык, красный язык, зеленый и голубой язык - все это может касаться и говяжьего языка и ЯП (в конкурсе языковых проектов, ставшем одним из этапов создания языка Ада, языки-конкуренты получили условные "цветные" наименования; победил "зеленый" язык).

Метод борьбы с семантическими недоразумениями при человеческом общении известен - нужно выделять важные понятия, давать им четкие определения, приводить характерные примеры. Это со стороны говорящего. Слушатели должны, в свою очередь, стараться уловить оставшиеся существенные неясности, приводить контрпримеры (объектов, соответствующих определениям, но повидимому не имевшихся в виду говорящим, и объектов, не соответствующих определениям, но, скорее всего, имевшихся в виду). Этот же метод точных определений широко используется в ЯП (вспомните определения процедур и функций в Алголе и Фортране, а также типов в Паскале), а примеры и контрпримеры применяются, как известно, при отладке программ.

1.7. Отступление об абстракции-конкретизации. Понятие модели

Добиваясь взаимопонимания, мы активно пользуемся аппаратом абстракции-конкретизации (обобщения-специализации).

Создавая понятие, отвлекаемся (абстрагируемся) от несущественных свойств тех конкретных объектов, на основе знания которых понятие создается, и фиксируем в понятии лишь свойства существенные, важные с точки зрения задачи, решаемой с применением этого понятия. Так, в понятии "часы" мы обычно фиксируем лишь свойство быть "устройством, показывающим время" и абстрагируемся от формы, структуры, цвета, материала, изготовителя и других атрибутов конкретных часов.

Приводя пример, мы конкретизируем (абстрактное) понятие, "снабжая" его второстепенными с точки зрения его сущности, но важными в конкретной ситуации деталями. Так, конкретное выполнение процедуры происходит при конкретных значениях ее параметров; конкретный пример ЯП - скажем, Фортран - имеет конкретный синтаксис и конкретную семантику.

Мои часы большие, круглые, позолоченные, с тремя стрелками, марки "Восток", на 17 камнях. Все это немного говорит об их качестве в роли "устройства, показывающего время", но конкретное устройство всегда обладает подобными "лишними" с точки зрения его роли свойствами. Их существование лишь подчеркивает тот факт, что (абстрактное) понятие никогда не исчерпывает конкретного объекта - оно всегда отражает лишь некоторую точку зрения на этот объект, служит компонентой его модели, оказавшейся удобной для решения определенной задачи. В другой ситуации, при решении другой задачи, этот же конкретный объект может играть другую роль. Тогда и точка зрения на него может быть другой, и может потребоваться совсем другая модель того же самого объекта.

На "устройство, показывающее время", в известных условиях можно предпочесть смотреть как на "украшение" и с этой точки зрения (в этой его роли) важнее станут форма, цвет, размер, фирма, чем даже способность правильно показывать время. На процедуру можно смотреть как на "объект, расходующий машинные ресурсы". При такой ее роли совершенно неважно, каков смысл выполняемых в ней действий. На ЯП иногда

приходится смотреть как на объект стандартизации, и тогда важно не столько то, каковы особенности его семантики и синтаксиса, сколько то, найдется ли достаточно много заинтересованных в его стандартизации людей и организаций.

Непроизвольная, а иногда и намеренная, но не подчеркнутая явно смена точки зрения, переход по существу к другой модели объекта мешает взаимопониманию, служит источником прагматических недоразумений. Вы говорите, что часы "плохие", потому что некрасивые, а я говорю "хорошие", так как отлично работают.

Способность без затруднений переходить от одной модели к другой, четко фиксировать и легко изменять уровень рассмотрения, а также угол зрения, отмечается обычно как важнейшее профессиональное качество программиста.

1.8. Синтактика, семантика, прагматика

Устранять прагматические недоразумения бывает особенно сложно, когда они связаны не только с различием точек зрения, но и целевых установок. Если правильно разложить фразу на составляющие может помочь согласование с контекстом, а правильно понять смысл слова или фразы может помочь знание их назначения (роли), то восстановить эту роль, догадаться о ней, если об этом не сказано явно, очень тяжело. Слишком велика неопределенность, свобода выбора.

Представим себе положение человека, которому излагается последовательность определений и не говорится, зачем они вводятся, для решения каких задач предназначены. Это хорошо знакомая всем ситуация - есть такой стиль изложения математических результатов. Слушатель (читатель) при этом лишен всякой опоры для контроля, кроме поиска чисто логических противоречий. Пока он не понял, зачем все это нужно, он готов пропустить любую содержательную ошибку. А попробуйте понять смысл программы, если неизвестно, для чего она написана!

Вывод очевиден - для достижения взаимопонимания необходимо, чтобы отправитель и адресат, во-первых, пользовались одинаковыми правилами разложения сообщения на составляющие (изучением таких правил занимается синтактика); во-вторых, согласованными правилами сопоставления сообщению смысла (такими правилами занимается семантика); и, в-третьих, имели согласованные целевые установки (это предмет прагматики).

Ролью всех перечисленных аспектов для создания и использования языков программирования мы еще займемся, а сейчас уместно поговорить об основной цели изложения (для согласования наших целевых установок).

1.9. Основная цель изложения

Мы намерены изложить основные принципы оценки, создания и использования современных ЯП. Это очень нужная, плодотворная и увлекательная, но далеко еще не устоявшаяся, быстро развивающаяся область. Поэтому нет возможности опираться на освященный традицией опыт предшественников, а также стабильные программы и учебники (как это бывает, скажем, при изучении математического анализа или дифференциальных уравнений). Приходится рисковать и экспериментировать.

Итак, о нашей основной цели. Она состоит в том, чтобы постараться правильно ориентировать читателя в области ЯП, помочь ему осознать навыки и опыт, приобретенные при самостоятельной работе с конкретными ЯП.

Но не слишком ли опасна идея "правильно ориентировать? Ведь если, скажем, представления о профессиональных запросах читателя или о тенденциях развития ЯП окажутся ошибочными, то скорее всего "правильная" ориентация на самом деле окажется дезориентацией. Не лучше ли ограничиться изложением бесспорных положений из области ЯП - уж они-то понадобятся наверняка!?

К сожалению или к счастью, альтернативы у нас по сути нет. Абсолютно бесспорные положения касаются, как правило, лишь конкретных ЯП. Например, "Один из операторов в языке Алгол 60 - оператор присваивания. Устроен он так-то. Служит для того-то". В хорошо известном учебнике программирования это положение обобщено. Сказано так:"Фундаментальным действием в любом алгоритмическом языке является присваивание, которое изменяет значение некоторой переменной". И это уже неверно! Сейчас много внимания уделяется так называемому функциональному программированию, аппликативным ЯП, где присваивание не только не "фундаментальное" действие, но его вообще нет!

Значит, в области ЯП нет достаточно общих бесспорных положений? В некотором смысле есть. Чаще не столь бесспорных, сколь заслуживающих изучения. Правда, их общность - несколько другого характера. Примером может служить упоминавшийся принцип однозначности. Да и приведенная фраза из учебника - вполне бесспорное

положение, если считать, что она характеризует определенный класс ЯП, в который не попадает, скажем, язык Лисп - один из самых "заслуженных", распространенных и в то же время перспективных. Итак, даже если ограничиться лишь относительно бесспорными положениями, их все равно нужно отбирать с определенных позиций, с определенной целью. Естественная цель - стремиться принести читателю максимальную пользу. Опять мы приходим к "угадыванию" будущих потребностей.

1.10. Зачем могут понадобиться знания о ЯП

Во-первых, каждая программа должна общаться (обмениваться информацией) с внешним миром. Соглашения, определяющие способ общения - это язык, так что понимание принципов построения языков - необходимый компонент грамотного программирования. Исключительно важный компонент, как мы еще не раз увидим, потому что непосредственно связан с внешним эффектом программы, со способом ее использования. При разработке внешнего сопряжения своей программы программист обязан проявить истинный профессионализм, представляя пользователю максимум услуг при минимуме затрат. Особенно это важно при создании пакетов прикладных программ, инструментальных систем, вообще любых программных изделий, предназначенных для эксплуатации без участия автора.

Во-вторых, каждый язык - это своя философия, свой взгляд на деятельность программиста, отражение определенной технологии программирования. Даже представлений об Алголе, Фортране и Бейсике достаточно, чтобы почувствовать, что имеется в виду.

Скажем, творцы Алгола (выдающиеся представители международного сообщества ученых в области информатики под руководством Петера Наура) с естественным для них академизмом придавали относительно много значения строгости определения и изяществу языковых конструктов. Считалось, что самое важное в работе программиста - сформулировать алгоритм (и, возможно, опубликовать его). Переписать программу в расчете на конкретные устройства ввода-вывода считалось не заслуживающей особого внимания технической деятельностью. Не привлек должного внимания авторов языка и такой "технический" аспект программистской деятельности, как компоновка программ из модулей.

Творцы Фортрана (сотрудники фирмы ИБМ во главе с Джоном Бэкусом) в значительной степени пренебрегли строгостью и изяществом и со свойственным им в ту пору (1954-57гг.) прагматизмом уже в первых версиях языка уделили особое внимание вводу-выводу и модульности. Но ни Фортран, ни Алгол не рассчитаны на работу в диалоговом режиме. В отличии, как вам известно, от Бейсика (созданного в Дартмундском колледже первоначально для обучения студентов).

Таким образом, изучение ЯП дает знание и понимание разнообразных подходов к программированию. Это полезно при любой программистской деятельности.

В-третьих, понимание общих принципов и концепций, определяющих строение и применение ЯП, позволяет легче и глубже освоить конкретный язык - основной профессиональный инструмент программиста.

В-четвертых, и это хотелось бы подчеркнуть особо, понятия и тенденции в области ЯП с некоторым запаздыванием (в целом полезным), довольно точно отражают понятия и тенденции собственно программирования как науки (и области человеческой деятельности). В этом смысле мы повторяем и закрепляем основные принципы и понятия программирования, но с несколько иной точки зрения.

Все, о чем было сказано до сих пор, касалось интересов потенциального пользователя ЯП. Но читатель может оказаться и руководителем коллектива, которому требуется оценивать и выбирать язык для выполнения конкретного проекта (учитывать, скажем, затраты на освоение этого языка или на обмен написанными на нем программными изделиями).

Если же он станет творцом языка, создателем транслятора или руководства для пользователей, то ему понадобятся столь разнообразные знания о ЯП, что их придется извлекать целеустремленным изучением специальной литературы. Можно надеяться дать лишь первоначальный импульс в нужном направлении.

Конечно, предсказать, для чего именно понадобятся приобретенные знания - сложно. Могут напрямую и вовсе не понадобиться. Но наверняка пригодится приобретенная обсуждениями, размышлениями и упражнениями культура работы со сложными объектами при решении сложных задач. В нашем случае - это такие задачи, как ОЦЕНКА, ИСПОЛЬЗОВАНИЕ, РАЗРАБОТКА И РЕАЛИЗАЦИЯ языков программирования.

Было бы неправильно ставить нашей целью научить свободному владению конкретными языками, пусть даже такими привлекательными или перспективными, как

Бейсик, Паскаль или Ада. Для этого служат специальные учебники, упражнения и, главное, практика.

Наша задача - познакомить с важнейшими понятиями и концепциями, помогающими оценивать, использовать, реализовывать и разрабатывать ЯП, дать представление о направлениях и проблемах их развития. Поэтому займемся, в основном, изучением моделей языков программирования и основных принципов их оценки, использования, реализации и разработки. Наиболее важные по тем или иным причинам языки или их конструкты иногда будут рассматриваться довольно подробно, но прежде всего лишь как примеры, иллюстрирующие более общие положения.

Например, будет важно понимать, что с каждым ЯП связан эталонный (абстрактный) исполнитель, в котором в свою очередь определены данные, операции, связывание, именование, аппарат прогнозирования и контроля, возможно исключений, синхронизации И защиты. Важно понимать перечисленные термины, понимать назначение соотвествующих языковых конструктов и уметь ими пользоваться при решении практических задач. Но не очень важно помнить наизусть все связанные с ними тонкости в конкретных языках. Последнее может оказаться важным лишь тогда, когда тонкости иллюстрируют ключевые концепции рассматриваемого ЯП. Например, жесткие правила выбора обозначений в Бейсике непосредственно связаны с его ориентацией на относительно небольшие программы и простоту реализации.

1.11. Пять основных позиций при рассмотрении ЯП

Итак, будем считать, что целевые установки согласованы в достаточной степени, чтобы сделать следующий шаг - приступить к систематическому изучению нашего предмета.

И сразу вопрос - с чего начать? Легко сказать "систематическому". Но ведь системы бывают разные. Часто начинают "снизу" - с основных конструктов, встречающихся почти во всех существующих ЯП. Тогда мы сразу погружаемся в мир переменных, констант, параметров, процедур, циклов и т.п. Такой путь привлекателен хотя бы тем, что им сравнительно легко пойти. Но на этом пути за деревьями обычно не видно леса, не удается увидеть язык "в целом", построить его адекватную модель.

Поэтому выберем другой путь. Постараемся взглянуть на объект нашего изучения - ЯП - с общих позиций. Нас будут особенно интересовать технологическая, семиотическая и математическая позиции.

Первая названа "технологической" потому, что отражает взгляд человека, желающего или вынужденного пользоваться ЯП как технологическим инструментом на каком-либо из этапов создания и использования программных изделий (другими словами, в течении их жизненного цикла). С таким человеком естественно объясняться в технологических терминах.

Вторая позиция названа "семиотической" потому, что ее можно представить себе как позицию человека, знакомого с некоторыми знаковыми системами (русским языком, дорожными знаками, позиционными системами счисления) и желающего узнать хотя бы в первом приближении, чем выделяются такие знаковые системы, как языки программирования. Следует объяснить ему это в семиотических терминах (семиотика - наука о знаковых системах).

Третья позиция отражает, естественно, взгляд математика, понимающего, что такое математическая модель изучаемого обьекта, и желающего познакомиться с математическими моделями ЯП. С ним желательно объясняться в математических терминах.

Уделим внимание и другим позициям - реализаторской и авторской. Они отражают взгляд на язык соответственно его реализаторов и авторов.

Авторы создают ЯП, делают его известным программистской общественности, исправляют и модифицируют его с учетом поступающих предложений и критических замечаний. Реализаторы обеспечивают возможность пользоваться ЯП как средством практического программирования. Другими словами, они не только создают трансляторы, но и пишут методические руководства, обучающие и контролирующие программы, испытывают трансляторы и т.п.

Сказанное иллюстрирует следующий пятиугольник.

Основные позиции при изучении ЯП

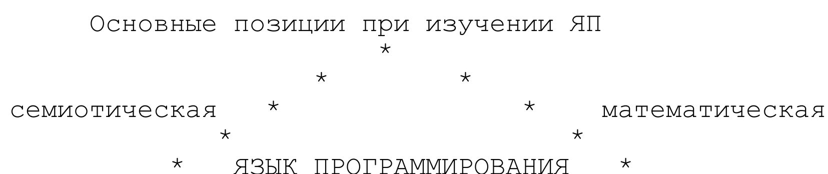


Рис. 1.2

Уместно подчеркнуть, что с разных позиций мы будем рассматривать один и тот же объект. Начнем с технологической позиции. Установим связь ЯП с производством программных услуг.

1.12. Что такое производство программных услуг

Напомним исходные понятия, известные из общего курса программирования ЭВМ (компьютеров). Если пойти по пути уточнения понятия "ЭВМ", то, в зависимости от обстоятельств, можно потратить на это и всю жизнь. К счастью, нам нужно уточнить это понятие лишь в той мере, в которой это необходимо для нашей цели. Для нас важно, что ЭВМ обладает двумя фундаментальными способностями - хранить данные и выполнять планы.

Начнем со второй способности. Назовем исполнителем всякое устройство, способное выполнять план. Так что и компьютер, и робот, и рабочий, и солдат, и сеть ЭВМ, и коллектив академического института может играть роль исполнителя.

Всего одна фундаментальная способность - выполнять план - дает возможность исполнителю предоставить пользователю содержательно разнообразные услуги. Определяя конкретные планы, можно настраивать исполнителя на предоставление конкретных услуг. Важно лишь, чтобы в плане фигурировали задания, посильные для выбранного исполнителя. Посильные - это значит такие, для выполнения которых у исполнителя имеются соответствующие ресурсы.

Для ЭВМ как исполнителей характерны два вида ресурсов - память и процессор. Память реализует первую из двух названных фундаментальных способностей - служит для хранения данных. Это пассивный ресурс. Процессор реализует вторую из названных способностей - служит для выполнения действий, предусмотренных в планах. Это активный ресурс. Процессор характеризуется определенным набором допустимых действий (операций, команд). Действия из этого набора считаются элементарными (в плане не нужно заботиться о способе выполнения таких действий).

Две названные способности связаны - выполнение достаточно сложного плана требует его хранения в доступной исполнителю памяти. В свою очередь, реализация хранения требует способности выполнять план (данные нужно размещать, перемещать, делать доступными).

План для такого исполнителя, как ЭВМ, должен в итоге сводиться к указанию конечной последовательности элементарных действий. Такой план называют программой.

[Людей как исполнителей характеризует прежде всего наличие у них модели реального мира, в достаточной степени согласованной с моделью мира у создателя плана. Поэтому в плане для людей можно указывать цели, а не элементарные действия.]

Ресурс, существенный почти для всех реальных исполнителей - это время. Важное свойство ЭВМ как исполнителей - способность выполнять элементарные действия исключительно быстро (порядка микросекунды на действие). Не менее важное свойство ЭВМ - способность хранить огромные объемы данных (в оперативной памяти - мегабайты; на внешней - практически неограниченно).

Способность выполнять весьма длинные последовательности элементарных действий над данными любого нужного объема за практически вполне приемлемое время делает принципиально возможным применение исполнителей класса ЭВМ для предоставления пользователям весьма разнообразных по содержанию услуг.

[ЭВМ способна развлекать пользователя, играя с ним в интеллектуальные и (или) азартные игры, давать справки, помогать в составлении планов, управлять самолетами и танками, поддерживать светскую беседу.]

Чтобы настроить ЭВМ на конкретный вид услуг, нужно снабдить ее соответствующими этому виду услуг знаниями в форме, воспринимаемой ЭВМ. Принципиально важный для нас факт, ставший очевидным лишь относительно недавно (по мере расширения и развития сферы услуг, представляемых пользователям с помощью ЭВМ), состоит в том, что самое сложное (и дорогое) в этом деле - не сами ЭВМ, а именно представление знаний для них. В наше время аппаратура в компьютере по своей относительной стоимости часто сравнима с упаковкой, в которую "заворачивают" знания.

Знания, представленные в ЭВМ, традиционно называют программами (таким образом, под программой в широком смысле слова понимают не только планы). Соотношение стоимости аппаратуры и программ (а также иные соображения) во многих случаях дает основание абстрагироваться от используемой аппаратуры и говорить об услугах, представляемых непосредственно программами - о производстве программных услуг.

[Из общего бюджета пользователей в США (в 1980 г. - 80 млрд. долларов) на программирование израсходовано 60 млрд., причем на закупку программ - 5 млрд., на разработку собственных программ 15 млрд., на сопровождение программ - 40 млрд. Задел прикладного программного обеспечения в США (стоимость формализованного в программах знания) до 1975 года - 50 млрд. долларов.]

Подчеркнем, что пока далеко до полной независимости программ от используемой аппаратуры. Проблема переноса программ на исполнители других типов - одна из острейших проблем современного программирования. И тем не менее затраты на создание достаточно сложной программы определяются в основном сущностью представляемой услуги (решаемой задачи), а не используемой аппаратуры. В этом смысле и говорят об абстракции от аппаратуры.

[Между прочим, правомерность такой абстракции в современных условиях подчеркивает определенную искусственность проблемы переноса программ. Если пока в значительной степени безразлично, на чем программировать, то разнообразие и несовместимость исполнителей вызваны не объективными, а социальными причинами.]

Знания, представляемые в компьютерах, можно разделить на пассивные и активные. Первые отражают факты, связи и соотношения, касающиеся определенного вида услуг. Вторые - это рецепты, планы действий, процедуры, непосредственно управляющие исполнителем.

Представление пассивного знания ориентировано в первую очередь на такой ресурс компьютера, как память, а представление активного - на процессор. Исторически самым первыми сферами применения ЭВМ оказались такие, где главенствовало активное знание - эксплуатировалась не столько способность ЭВМ много знать, сколько быстро считать.

Кстати, и память на первых ЭВМ была очень мала по сравнению со скоростью работы процессора. Всю свою память они могли просмотреть (или переписать) за десятую долю секунды. Недалеко ушли в этом отношении и современные компьютеры (если говорить об оперативной памяти). Соотношение между объемом оперативной памяти и скоростью процессоров активно влияет на мышление программистов, инженеров, пользователей, а также всех связанных с ЭВМ людей. Изменение этого соотношения (а его разумно ожидать) способно произвести революцию в практическом программировании (см. далее о модифицированной модели Маркова и функциональном стиле программирования (модель Бэкуса); есть и другие перспективные модели, например, реляционная). Пока же программисты вовсю экономят память, помещая новые значения на место старых и затрудняя написание, понимание и проверку программ, а преподаватели учат искусству "эффективного" программирования. Тенденция развития состоит в том, что роли активного и пассивного знания в производстве программных услуг становятся более симметричными.

1.13. Производство программных услуг - основная цель программирования

Измерять эффективность того или иного производства разумно лишь по отношению к его цели (конечному результату). Поэтому важно понимать, что конечная цель программирования - не создание программ самих по себе, а предоставление программных услуг. Другими словами, программирование в конечном итоге нацелено на обслуживание пользователей. А настоящее обслуживание должно руководствоваться известным принципом: "клиент всегда прав". В применении к программированию этот принцип означает, что программы должны быть надежными и робастными.

Первое означает, что в программе должно быть мало ошибок, второе - что она должна сохранять работоспособность в неблагоприятных условиях эксплуатации.

Что значит "мало ошибок", зависит от назначения программы (ясно, что программа обучения русскому языку и программа автопилота могут иметь различную надежность). Под "неблагоприятными условиями" понимается ограниченность выделяемых программе ресурсов (память, каналы ввода-вывода, число процессоров), перегрузка (много пользователей, большие объемы данных), ошибки пользователей, сбои и отказы аппаратуры, попытки намеренно вывести программу из строя и т.п.

Сказанное относится к работе программы. Однако важно понимать, что программная услуга - это не только услуга, оказываемая пользователю непосредственно при работе исполнителя под управлением программы, но и проверка,

оценка, продажа, подбор программ, их перенос на другой исполнитель, сопровождение и аттестация программ и т.п.

Когда производство программных услуг достигает некоторой степени зрелости, из кустарного производства вырастает индустрия программных услуг и обслуживающая ее потребности теория программирования. Как индустрия, так и кустарное производство пользуются при этом той или иной технологией - технологией производства программных услуг - технологией программирования.

Технология - это систематическое применение знаний к практическим задачам производства. (О связи науки, искусства, теории и технологии в программировании см. Тьюринговскую лекцию Дональда Кнута в Communication of the ACM, 1974,v,12,667-673).

1.14. Сложность как основная проблема программирования.

Итак, основная цель программирования - производство программных услуг. Известно, что этот род человеческой деятельности в развитых странах уверенно выходит на первое место среди всех других производств (скажем, в США соотвествующая отрасль хозяйства уже опередила недавних лидеров - нефтяную и автомобильную отрасли). Вместе с тем известно, что создание программ и другие связанные с ними услуги остаются слишком дорогим и относительно длительным делом, в котором трудно гарантировать высококачественный конечный результат. В чем же основная причина такого положения? Связана ли она с самой природой программирования или носит субъективный характер? В настоящее время краткий ответ можно сформулировать так; "сложность - основная проблема программирования; связана с самой его природой; можно надеяться на ее понижение для освоенных классов задач".

1.15. Источники сложности.

Попытаемся подробнее разобраться с тем, почему же сложность объектов, с которыми приходится иметь дело - отличительная черта программирования компьютеров. Найдя источники сложности, можно с большей надеждой на успех искать пути ее преодоления.

Когда исполнители работают медленно, а действия, которые считаются элементарными, специфичны для вида предоставляемых услуг, то планирование деятельности исполнителя и критерии качества такого планирования существенно зависят и от услуги, и от конкретного набора элементарных действий.

Вряд ли стоит поручать, скажем, заводскому технологу, специалисту по обработке металлов резанием, планирование индивидуального пошива верхней одежды. Для этого есть закройщики, которые , в свою очередь, вряд ли смогут применить свое искусство при создании заводских технологических карт. Другими словами, набор "элементарных" действий двух рассмотренных категорий исполнителей считать эквивалентными неестественно.

Компьютеры работают быстро и наборы их команд в известном смысле эквивалентны. Причем уже в течении многих лет сохраняется тенденция к увеличению скорости и увеличению объема памяти при фиксированной цене (примерно на порядок за десятилетие). В таких условиях возникает принципиальная возможность настроить исполнитель на предоставление услуг из очень широкого класса (во всяком случае, границы этого класса неизвестны). Для этого достаточно снабдить исполнитель подходящим планом (написать программу).

Эта принципиальная возможность соблазняет настраивать компьютеры на виды услуг, очень далеких от элементарных возможностей исполнителя. Более того - таковы почти все практически значимые услуги. Поэтому в общем случае план должен предусматривать огромное количество элементарных действий над огромным количеством элементарных объектов. Но этого мало. Самое главное - огромное количество связей между этими объектами. Поэтому и сами программы становятся огромными (уже имеются программы из миллионов команд; например, для управления военными и космическими объектами).

Между тем способности человека работать с большим числом связанных объектов, как хорошо известно, весьма ограничены. В качестве ориентира при оценке этих способностей указывают обычно на так называемое "число Ингве", равное семи (плюс-минус 2). Другими словами, человек обычно не в состоянии уверенно работать с объектом, в котором более семи компонент с произвольными допустимыми взаимными связями. До тех пор, пока программирование остается в основном человеческой деятельностью, с указанным ограничением необходимо считаться.

Таким образом, предоставив универсальность, скорость и потенциально неограниченную память, создатели компьютеров, с одной стороны, соблазнили человечество неслыханными возможностями, а с другой, поставили лицом к лицу с проблемами невиданной потенциальной сложности (при попытке осуществить эти гипотетические возможности).

В этой связи упомянем принцип, формулируемый на программистском жаргоне как "труд на юзера спихнуть" (юзер - пользователь). Изобретатели ЭВМ, предоставив в распоряжение программистов исключительное по своим возможностям абстрактное устройство, "спихнули" на них труд по настройке этого абстрактного устройства на предоставление конкретных услуг. Но такая конкретизация оказалась далеко не тривиальной. Программисты, в свою очередь, создавая "универсальные" программы, "спихивают" труд по их применению в конкретных условиях на потенциальных пользователей этих программ.

Итак, первый источник сложности в программировании - разрыв между уровнем и характером элементарных объектов и операций, с одной стороны, и потенциально возможных услуг - с другой. Иными словами, это проблема согласования масштаба - ювелирными инструментами предлагается сооружать города.

Именно этот источник имелся в виду, когда шла речь об объективном характере присущей программированию сложности. Занимаясь определенным классом услуг (задач) можно стремиться выделить характерный именно для этого класса набор элементарных объектов и операций, построить соответствующий исполнитель (аппаратным или программным способом) и программировать на таком более подходящем исполнителе. Фактически это означает создать адекватный выбранному классу услуг ЯП. На практике это самый распространенный способ борьбы со сложностью и одновременно - основная причина роста так называемых проблемно-ориентированных ЯП (ПОЯ).

Имеется еще один принципиально важный источник сложности, затрудняющий "взаимопонимание" с компьютерами. Речь идет об отсутствии в современных компьютерах модели реального мира, согласованной с представлениями о мире у программистов и пользователей. Поэтому в общем случае компьютер не в состоянии контролировать указания программиста или действия пользователя с прагматической точки зрения (контролировать соответствие между действиями и теми целями, ради которых эти действия совершаются - цели компьютеру неизвестны).

Из-за этого самая "мелкая" с точки зрения создателя программы описка может привести к совершенно непредсказуемым последствиям (широко известен случай, когда из-за одной запятой в программе на Фортране взорвалась космическая ракета, направлявшаяся на Венеру; пропали усилия, стоившие миллиарды долларов).

Итак, в качестве второго источника сложности в современном программировании следует назвать абсолютную точность планирования. Лишенный знания целей исполнитель не может не только скорректировать неточно указанные в программе действия, но не может и проинформировать об отклонениях от направления на цель. Управление исполнителем посредством действий, а не целей требует учета мельчайших нюансов всех обстоятельств, в которых может оказаться исполнитель в процессе предоставления нужной услуги. Это резко увеличивает число объектов, с которыми приходится иметь дело при создании программ. Отсюда повышение сложности программирования, увеличение размера программ, понижение их надежности и робастности.

Второй источник сложности, в отличие от первого, есть надежда победить (или хотя бы существенно ослабить) за счет развития методов представления в компьютерах знаний о реальном мире и эффективном учете этих знаний при создании и исполнении программ.

1.16. Два основных средства борьбы со сложностью

Рассмотренные источники сложности оказывают определяющее влияние на теорию и практику в области ЯП. Важнейшим средством борьбы с первым из них служит аппарат абстракции-конкретизации. Он обеспечивает базу для проблемной ориентации языковых выразительных средств.

Например, в Фортране характерным средством абстракции служит подпрограмма, а соответствующим средством конкретизации - обращение к ней с фактическими параметрами. Важнейшим средством борьбы со вторым источником сложности служит аппарат прогнозирования-контроля. Он обеспечивает базу для повышения надежности и робастности программ.

Например, в Фортране характерным средством прогнозирования служит объявление типа, соответствующий контроль предусмотрен семантикой языка, но средств управления таким контролем в языке нет.

Упражнение. Приведите известные вам примеры средств абстракции-конкретизации и прогнозирования-контроля. Постарайтесь подобрать симметричные, взаимнодополнительные средства. Убедитесь, что в известных вам ЯП эта дополнительность обеспечена не всегда.

Теперь мы в состоянии сформулировать следующий основной критерий качества ЯП (как инструмента для планирования поведения исполнителя): язык тем лучше, чем более он способствует СНИЖЕНИЮ СЛОЖНОСТИ производства программных услуг.

Удовлетворимся временно этим результатом разработки технологической позиции и уделим теперь немного внимания семиотической позиции.

1.17. Язык программирования как знаковая система

Продолжим уточнение понятия "язык программирования". Наше новое интенсиональное определение таково:

Язык программирования - это знаковая система для планирования поведения компьютеров.

Итак, не любой "инструмент", а "знаковая система" и не для планирования произвольных "исполнителей", а только из класса ЭВМ (или "компьютеров"). К ограничению класса исполнителей в этом определении мы подготовились заранее, а вот о знаковых системах еще подробно не говорили.

Знаковая система - это совокупность соглашений (явных или неявных), определяющих класс знаковых ситуаций.

Понятие знаковой ситуации в семиотике относят к первичным понятиям, представление о которых создают с помощью примеров, а не явных определений. Необходимые компоненты знаковой ситуации - знак и денотат. Говорят, что знак обозначает денотат (знак называют также обозначением или именем, а денотат - обозначаемым или значением). Так, в модели передачи сообщения само сообщение служит знаком, его смысл - денотатом.

Вот еще знаковые ситуации (первым укажем знак, вторым - денотат): буква и соответствующий звук, дорожный знак ("кирпич") и соответствующее ограничение ("въезд запрещен"), слово и соответствующее ему понятие. Каждый без затруднений пополнит этот список.

Когла класс знаковых ситуаций определяется совокупностью соглашений (правил), устанавливающих закономерную связь между структурой знака и его денотатом, говорят, что эти соглашения образуют знаковую систему (или язык). При этом правила, определяющие структуру допустимых знаков, называются синтаксисом языка, а правила, определяющие соответствующие допустимым знакам денотаты, называются семантикой языка. (Науку о синтаксисах языков называют синтактикой, а слово "семантика" используется как для обозначения конкретных правил некоторого языка, так и для обозначения общей науки о таких правилах).

Одним из примеров знаковой системы служит позиционная система счисления (скажем, десятичная). Правила, определяющие перечень допустимых цифр и их допустимое расположение (скажем, справа налево без разделителей) - это синтаксис. Правила вычисления обозначаемого числа - семантика. При этом запись числа в позиционной системе - знак, а само обозначаемое число - денотат. Известные вам ЯП - также знаковые системы.

Упражнение. Приведите пример синтаксического и семантического правила из таких знаковых систем, как Фортран, Бейсик, Ассемблер.

В общем случае в ЯП знаки - это элементы программ (в том числе полные программы), а денотаты - элементы и свойства поведения исполнителя (атрибуты его поведения), в частности, данные, операции, управление, их структура, их связи и атрибуты. Например, знаку, составленному из шести букв "arctan" (элементу программы на Фортране), использованному в этой программе в подходящем контексте, соответствует в качестве денотата такой элемент поведения исполнителя, как операция вычисления арктангенса.

Знаку, составленному из пяти букв "begin" (элементу программы на Алголе) в одном контексте в качестве денотата может соответствовать такой элемент поведения, как вход в блок, а в другом - переменная вещественного типа, в третьем - массив целого типа.

Упражнение. Выпишите подходящие контексты.

Итак, знаковая система - это правила образования знаков (синтаксис) и согласованные с ними правила образования денотатов (семантика). Подчеркнем, что правила использования денотатов для целей, выходящих за рамки семантики (т.е. прагматика) обычно не включаются в знаковую систему. Например, в Фортране нет каких-либо правил, ограничивающих применение соответствующих вычислительных процессов для неблаговидных целей.

Теперь уточненное определение ЯП как знаковой системы для планирования поведения компьютеров должно быть полностью понятным.

1.18. Разновидности программирования

Чтобы создать себе более удобную основу для формирования оценок, принципов и требований, примем соглашения, сужающие область наших рассмотрений.

Во-первых, программировать можно с различной целью. Скажем, для развлечения и обучения (игровое программирование); для отработки идей, приемов, инструментов, методов, критериев, моделей (экспериментальное программирование, его характерное свойство - созданная программа не предназначена для применения без участия автора, т.е. результат такого программирования неотчуждаем).

В дальнейшем будем рассматривать только индустриальное программирование, цель которого - создание программных изделий (программных продуктов) на заказ или на продажу. Характерное свойство - отчуждаемость результата.

Во-вторых, может быть различным характер использования заготовок программ. По этому критерию различают по крайней мере три разновидности программирования:

сборочное - программа составляется из заранее заготовленных модулей (так обычно сейчас работают пакеты прикладных программ);

конкретизирующее - программа получается в результате преобразования универсальных модулей-заготовок (в результате специализации) в расчете на конкретные условия применения; цель специализации - повышение эффективности (снижение ресурсоемкости) универсальной программы;

синтезирующее - роль заготовок относительно невелика.

В дальнейшем нас, как правило, будет интересовать лишь синтезирующее индустриальное программирование.

В-третьих, стадии жизненного цикла программного изделия предъявляют различные, иногда противоречивые, требования к ЯП. Выделим стадии проектирования, эксплуатации и сопровождения. В первую очередь будем интересоваться стадией проектирования изделия, так как на ней в той или иной форме следует учитывать и требования всех остальных стадий жизненного цикла.

Можно надеяться, что вдумчивый читатель сможет применить полученные навыки анализа ЯП и при иных исходных соглашениях.

1.19. Понятие о базовом языке

Выделенные нами два источника сложности в программировании полезно трактовать как два различных аспекта единого источника - рассогласования моделей проблемной области (области услуг, задач, операций, сокращенно - ПО) у пользователей и исполнителей.

При таком взгляде создаваемая программа выступает как средство согласования этих моделей. Чем ближе исходные модели, тем проще программа. При идеальном исходном согласовании программа вырождается в прямое указание на одну из заранее заготовленных услуг (например, "распечатать файл", "взять производную", "выдать железнодорожный билет").

Но мы уже говорили об исключительном разнообразии моделей даже одного-единственного объекта, рассматриваемого с различных точек зрения. Поэтому невозможно построить исполнитель, непосредственно пригодный для выполнения любой услуги. Однако можно ориентировать его на фиксированный класс услуг. Для управления такими специализированными исполнителями строятся проблемно-ориентированные языки программирования (ПОЯ). В качестве хорошо известного примера годится, скажем, язык управления заданиями в операционной системе.

Итак, ПОЯ опирается на определенную модель соответствующей ПО (иногда говорят, что эта модель встроена в такой язык; точнее говоря, ПОЯ - это знаковая система, для которой модель соответствующей ПО служит областью денотатов).

Мы установили, что безнадежно строить язык с моделями, заготовленными "на все случаи жизни". Однако можно попытаться построить язык, на базе которого будет

удобно (относительно несложно, с приемлемыми затратами) строить модели весьма разнообразных ПО. Такой язык называют базовым языком программирования.

Обычная схема применения базового языка в определенной ПО состоит из двух этапов. На первом (инструментальном) создается модель ПО и соотвествующий ПОЯ (их создают с помощью базового языка программисты-конструкторы). На втором (функциональном) этапе программисты-пользователи решают прикладные задачи, пользуясь созданным ПОЯ.

Итак, базовый ЯП - это по существу ПОЯ, предназначенный для построения моделей других ПО и соответствующих ПОЯ. Нас будут интересовать в первую очередь именно базовые языки, в особенности базовые языки индустриального программирования.

1.20. Концептуальная схема рассмотрения ЯП

Завершая подготовку к систематическому изучению ряда моделей ЯП, зафиксируем единую схему их рассмотрения. Эта схема поможет сопоставить и оценить различные ЯП прежде всего с точки зрения их пригодности служить базовым языком индустриального программирования.

Если бы нас интересовала, скажем, оценка языков с точки зрения легкости их усвоения начинающими программистами, мы предложили бы, конечно, другую схему их рассмотрения, начав с выявления основных идей и проблем обучения, наиболее подходящих средств решения этих проблем и т.д., подобно нашему пути к базовому языку.

Тем самым, предлагаемую ниже схему нужно воспринимать и как демонстрацию существенного элемента систематического метода сравнительной оценки языков. (Конечно, наша учебная схема намного проще той, которую следовало бы строить при практическом решении вопроса о пригодности конкретного языка служить базовым языком индустриального программирования.)

Главное назначение базового языка - строить модели ПО с тем, чтобы уменьшить сложность программирования в них. В качестве основных средств понижения сложности мы выделили абстракцию-конкретизацию и прогнозирование-контроль.

Первый будем кратко называть аппаратом развития (так как по существу он служит для построения над исходным языком новой знаковой системы, денотатами в которой выступают введенные абстракции и их конкретизации).

Второй будем называть также аппаратом защиты (так как он используется, в частности, для защиты построенных абстракций от разрушения).

Исключительная технологическая роль названных средств дает основание уделить им особое внимание в предлагаемой ниже единой концептуальной схеме рассмотрения ЯП.

Опишем первую версию единой схемы. При необходимости она будет корректироваться и уточняться.

В каждом ЯП нас будет интересовать пять аспектов: базис, развитие, защита, исполнитель, архитектура. Охарактеризуем каждый из этих аспектов.

Базис ЯП - это, во-первых, так называемая скалярная сигнатура (т.е. элементарные типы данных и элементарные операции) и, во-вторых, структурная сигнатура (т.е. допустимые структуры данных и операций; другими словами, структуры памяти и управляющие структуры).

Об аппарате развития языка (абстракции-конкретизации) уже сказано. Добавим лишь, что будем различать развитие вверх - аппарат определения и использования новых абстракций, и развитие вниз - уточнение и переопределение компонент базиса.

Об аппарате защиты также сказано. Имеется в виду прогнозирование (объявление) свойств поведения объектов (принадлежности к определенному типу, указание области действия, указание ограничений на допустимые значения в определенных контекстах) и контроль за соблюдением ограничений (в частности, управление реакцией на нарушение объявленного поведения).

Характеризуя исполнитель, будем говорить об основных его ресурсах, позволяющих хранить и выполнять программы (не о типах данных и операций, как в базисе, а о конкретных индивидуальных устройствах: разновидностях памяти, процессоров, обмена с внешней средой и т.п.).

Наиболее "капризный" из выделенных аспектов - "архитектура". Здесь критерии весьма расплывчаты. Тем не менее постараемся оценивать общее строение языка с точки зрения таких архитектурных понятий, как концептуальная целостность (возможность предсказать одни решения авторов языка по другим, т.е. увязанность, согласованность решений), модульность, ортогональность (возможность свободно

комбинировать небольшое число относительно независимых фундаментальных понятий) и другим, знакомство с которыми нам еще предстоит.

В заключение подчеркнем, что пока в нашей схеме - только внутренние аспекты ЯП как знаковой системы. Совершенно не затронуты такие важнейшие для выбора и оценки языка аспекты, как распространенность на различных типах компьютеров, наличие высококачественных реализаций, уровень их совместимости и т.п.

Следующий раздел начнем с применения нашей схемы к трем моделям ЯП - модели Неймана, модифицированной модели Маркова и модели Бэкуса.

2. ТРИ МОДЕЛИ ЯЗЫКА

Стремясь дать представление о разнообразии подходов к практическому программированию и одновременно подтвердить дееспособность концептуальной схемы, применим ее к трем конкретным моделям ЯП. Первая модель отражает свойства первых ЭВМ, вторая восходит к нормальным алгоритмам Маркова. Вместе с тем это модели вполне реальных языков практического программирования. Третья модель, с одной стороны, опирается на такую почтенную форму планирования, как алгебраическая формула (выражение), а с другой стороны, ориентирует на такие современные области исследований и разработок, как функциональное программирование и алгебра программ.

2.1. Модель фон-Неймана (модель Н)

Рассмотрим модель, отражающую свойства первых ЭВМ - модель весьма примитивную, но способную послужить для нас своеобразным "началом координат", создать исходную точку отсчета.

2.1.1. Базис.

Два скалярных типа данных: адреса и значения. Конечный набор базисных скалярных операций (система команд): присваивание, условные операции, останов и другие. Единственная структура данных - кортеж ячеек (т.е. пар адрес -> значение) с линейно упорядоченными адресами (память). Никакой явной структуры операций - каждая операция сама определяет своего преемника. Есть выделенная ячейка С (регистр команд), в которой хранится адрес подлежащей выполнению команды.

2.1.2. Развитие

Никаких выделенных явно средств развития - все они скрыты в универсальности набора операций, среди которых ключевую роль играет оператор присваивания ячейке нового значения и зависимость выбора преемника от состояния памяти. Любое развитие возможно только путем явного моделирования новых операций за счет универсальности системы команд. Что такое значение - не уточняем. Достаточно считать, что это целые и строки литер (для выделенных ячеек ввода-вывода).

2.1.3. Защита

Полностью отсутствует.

2.1.4. Исполнитель

Память - базисная структура данных (кортеж ячеек), процессор - устройство, последовательно выполняющее указанные (в С) операции, поведение - последовательность состояний памяти, план (программа) - исходное состояние (или его выделенная часть), результат - заключительное состояние (если оно есть; при этом содержательно результатом обычно служит лишь выделенная часть заключительного состояния).

Указанные "части" для каждой программы свои. Так что в общем случае программа формально не отличается от исходных данных и результатов - одни и те же ячейки исполнитель может интерпретировать либо как содержащие команды, либо как содержащие данные. Все дело в том, в какой роли адреса ячеек используются в исполняемых командах.

2.1.5. Знаки и денотаты в модели Н

Сведения о базисе можно выразить с помощью следующих обозначений.

Пусть А - тип данных "адрес" (т.е. множество адресов в модели Н), V - тип данных "значение" (т.е. множество содержимых ячеек с адресами из А). Тогда конкретное состояние памяти можно представить функцией s типа

S:A->V

т.е. конкретным отображением адресов в значения.

Тип функции "состояние" выражает первый принцип фон-Неймана - принцип произвольного доступа к памяти (в конкретном состоянии s из S равнодоступны все ячейки).

Операции (операторы) в модели фон-Неймана - это объекты типа

St:S->S.

Кроме того, модель фон-Неймана характеризуется функцией декодирования операций (частично-определенной)

d:V->Com,

где Com - команды т.е. операции, встроенные (элементарные) в Н.

В этих обозначениях второй принцип фон-Неймана - принцип хранимой программы отражается формулой

(A c из Com)(E v из V):d(v) = c ,

где A обозначает "для всех", а E - "существует". Т.е. всякую команду можно записать в память (найдется способ ее закодировать).

[Фактически здесь использованы элементы некоторого языка для описания семантики ЯП - семантического метаязыка. Язык для описания синтаксиса ЯП знаком из курса программирования. Таким синтаксическим метаязыком служит, например, БНФ (форма Бэкуса-Наура).]

2.1.6. Основное семантическое соотношение в модели Н.
(денотационная семантика)

Каков же денотат программы s в модели Н? Другими словами, какова та функция, которую реализует (обозначает) программа s?

Рассмотрим функцию r типа St

r:S->S

которая обозначает результат выполнения программы s, т.е. r(s) - это состояние s1, в котором выполняется операция остановки ("stop"). Оно не всегда достигается, т.е. функция r - частично-определенная - ведь, во-первых, не всякое состояние может служить программой и, во-вторых, не всякая программа завершает работу.

Другими словами, если C - регистр команды, то

d(s1(s1(C))) = stop.

(такому семантическому соотношению удовлетворяет заключительное состояние s1).

Обозначим через k=d*s*s композицию функций d,s,s. Тогда основное семантическое соотношение, определяющее денотат r(s) программы s в модели Н, записывается так:

r(s)=если k(C)=stop, то s, иначе r(k(C)(s)).

Другими словами, нужно выполнить над состоянием s операцию, получающуюся декодированием содержимого ячейки с адресом, взятым из C, и вычислить функцию r от полученного нового состояния, пока не окажется, что нужно выполнить операцию stop.

Что можно извлечь из формулы для r?

Во-первых, то, что один шаг выполнения программы требует, в общем случае, трех обращений к памяти (в нашей модели регистр команд - в основной памяти), ведь переход к новому состоянию описывается как d(s(s(C)))(s).

Во-вторых, становится очевиднее, что средства развития в модели Н не выделены - денотат программы разлагается лишь на очень мелкие части - денотаты отдельных команд (выраженные, кстати, функцией k). Отсутствуют средства для явного обозначения композиций функции k, т.е. для явного укрупнения денотатов.

Пусть P = {p} - множество программ, R = {r} - множество функций типа S->S. Функциональной или "денотационной" семантикой программ называют функцию типа P->R, отображающую программу (т.е. исходное состояние p) в соответствующую ей функцию (оператор) r, удовлетворящую основному семантическому соотношению.

Название "денотационная" возникло исторически. Всякая семантика денотационная в том смысле, что сопоставляет знаку (программе) некоторый ее денотат (смысл).

Обратите внимание, сколь концептуально сложной оказалась знаковая система Н. Во всяком случае, нам потребовались так называемые функции высших порядков (т.е. функции, среди аргументов и (или) результатов которых встречаются снова функции). Действительно, посмотрите на перечень примененных функций:

s: A --> V
St: S --> S
d: V --> Com
r: S --> S
sem: P --> R

очевидно, что операция - функция высшего порядка, отображающая состояния (функции из адресов в значения); декодирующая функция - также высшего порядка; семантическая функция - также высшего порядка по отношению к функциям p и r (первая из них отображает адреса в значения, вторая - исходное состояние в заключительное).

2.1.7. Архитектура

Архитектура модели Н сильно зависит от конкретного набора команд. Может быть весьма изящной, как, например, архитектура команд машин серии ЕС или Сетуни.

Примеры программ в модели Н в достаточном числе содержатся в любом учебнике по программированию.

2.2. Модифицированная модель Маркова (модель М)

Модель Н возникла как обобщение такого поведения, когда после предыдущего действия ясно, какое должно быть следующим (команда сама устанавливает следующую). Такое поведение типично для рутинных вычислений, на автоматизацию которых ориентировались первые компьютеры (они были предназначены, как известно, для расчетов, связанных с созданием атомной бомбы).

Расчеты такого рода характеризуются данными относительно простой структуры - программы имеют дело с числами. Вся сложность поведения исполнителя определяется сложностью плана (т.е. числом и связями указанных в нем действий). Управление последовательностью действий зависит от сравнения простых данных. Еще Джон фон-Нейман хорошо понимал, что для других классов применений могут потребоваться компьютеры, характеризующиеся другим типом поведения.

2.2.1. Перевод в польскую инверсную запись (ПОЛИЗ)

Рассмотрим, например, задачу перевода арифметической формулы в постфиксную форму. Другими словами, исходными данными для нашей программы должны быть обычные арифметические формулы, а в результате нужно получить их запись в ПОЛИЗе. Например,

(a+b)*(c+d) --> ab + cd + *.

Мы уже говорили о том, что данные ко всякой программе записываются на некотором языке (являются знаками в некоторой знаковой системе). Чтобы их обработать, нужно воспользоваться правилами построения знаков в этой системе (синтаксисом языка) для распознавания структуры знака, затем воспользоваться семантикой знаковой системы, чтобы связать со структурой знака его смысл (денотат) и обработать данное в соответствии с его смыслом.

Пусть синтаксис языка формул, которые мы хотим обрабатывать, задают следующие правила БНФ:

<формула>::=<сумма>I<произведение>I<первичная>.
<сумма>::=<сумма>+<произведение>I<первичная>.
<произведение>::=<произведение>*<первичная>I<первичная>.
<первичная>::=<число>I<переменная>I(<формула>).

Числа и переменные точно определять не будем, оставляя представление о них на интуитивном уровне (23 и 305 - числа, x, y, a, b, АЛЬФА - переменные).

Тогда 23 - формула (первичная, произведение, сумма), a+b*23 - также формула (сумма), (a+b)*23 - также формула (произведение); (a+*b) - не формула.

Семантика формул - общепринятая. Смыслом (денотатом) формулы будем считать число, получающееся из чисел, входящих в формулу, применением указанных операций в общепринятом порядке. Задача состоит в том, чтобы получить перевод в ПОЛИЗ, сохраняющий денотат (т.е. в данном случае - над теми же числами нужно выполнить те же операции и в том же порядке).

Другими словами, было бы идеально, если бы вся программа записывалась фразой примерно такого вида:

перевод(<формула1><операция><формула2>) =
перевод(<формула1>)

<формула2>)
<операция>

Можно заметить, что перевод текста с языка формул четко распадается на действия двух сортов - на распознавание компонент структуры исходной формулы и на компоновку ее образа в ПОЛИЗЕ из результатов перевода выделенных компонент. Когда действия этих сортов переплетены некоторым нерегулярным способом, то планировать, понимать, выполнять и проверять сложно. Чтобы уменьшить сложность, полезно выделить две ключевых абстракции (два понятия): анализ исходной структуры и синтез результирующей структуры, и предложить знаковую систему для их взаимосвязанной конкретизации в рамках единой программы.

Тут уместно вспомнить язык нормальных алгоритмов Маркова (для единообразия назовем этот язык моделью Маркова).

Охарактеризуем эту модель с точки зрения нашей концептуальной схемы.

Базис: единственный скалярный тип данных - литера; единственная базисная операция - поиск-подстановка; единственная структура данных - строка (текст); единственная структура операций - цикл по подстановкам.

Развитие: Явных средств нет. Только моделированием.

Дальнейший анализ модели можно предложить в качестве упражнения.

В модели Маркова анализ структуры встроен в исполнитель и управляется левой частью подстановки. Синтез структуры отделен от анализа - он управляется правой частью подстановки. Исполнитель распознает тривиальную структуру (слово), указанную слева, и заменяет ее столь же тривиальной структурой (словом), указанной справа.

С точки зрения нашей задачи эта модель недостаточно развита. Дело в том, что вид распознаваемых структур слишком тривиален. Хотелсь бы приблизить средства описания вида структур, скажем, к БНФ. Шаги в нужном направлении сделаны в языке, созданном в ИПМ АН СССР в 1966-68 г.г. и получившем название "рефал" (рекурсивных функций алгоритмический язык). В его основу положены следующие три модификации модели Маркова.

2.2.2. Три основные модификации модели Маркова (введение в рефал)

Мы изложим модель М языка программирования, особенно интересного с точки зрения нашей концептуальной схемы потому, что он был задуман и реально используется как средство для эффективного определения других языков (другими словами, в него заложены достаточно мощные средства развития).

Первая модификация состоит в том, что в качестве (по-прежнему единственной) базисной структуры данных вместо произвольной строки (слова) используется так называемое "выражение" - строка, сбалансированная по скобкам.

Вторая модификация касается подстановки. Ее левая часть должна быть так называемым функциональным термом с возможными переменными. Правая часть должна быть выражением, в котором можно использовать переменные из левой части подстановки (и только их).

Третья модификация касается поиска применимой подстановки. В отличие от модели Маркова, где заранее не фиксируется заменяемая часть обрабатываемого слова, в рефале заменяемая часть обрабатываемого выражения фиксируется перед поиском применимой подстановки - это всегда так называемый ведущий функциональный терм. Применимой считается подстановка с минимальным номером, левая часть которой согласуется с ведущим термом. Другими словами, применима такая подстановка, в левой части которой указан общий вид структуры (образец), частным случаем которой оказался ведущий терм.

Займемся теперь каждой из модификаций подробнее. Нам нужно уточнить смысл слов "выражение", "ведущий функциональный терм", "переменная" и "согласуется".

2.2.2.1. Строение выражений; поле зрения

Выделено три типа скобок - так называемые символьные (открывающая ` и закрывающая ' кавычки), структурные (обычные круглые скобки) и функциональные (мы будем использовать фигурные скобки "{" и "}").

Выражением называется всякая последовательность литер, сбалансированная по всем трем типам скобок; термом - выражение в скобках либо совсем без скобок; символом - отдельная литера либо последовательность литер в символьных скобках.

Например,

(a+b) - выражение - структурный терм;
{a+b (c `АЛЬФА')} - выражение - функциональный терм;

`АЛЬФА' - символ, терм, выражение;
}ab{ - не выражение.

По существу, выражение - это линейное представление дерева - структура этого вида часто используются в программировании именно потому, что наглядно воплощает идею иерархии, частичного порядка, пошаговой (последовательной) декомпозиции.

Дерево - это ориентированный граф (орграф) без циклов, в котором выделена вершина, называемая корнем дерева, и в каждую вершину, кроме корня, входит ровно одна дуга, причем из корня доступны все вершины. В дереве легко вводятся уровни иерархии (по длине пути из корня).

Так, выражение {a+b(c `АЛЬФА')} может быть представлено деревом вида

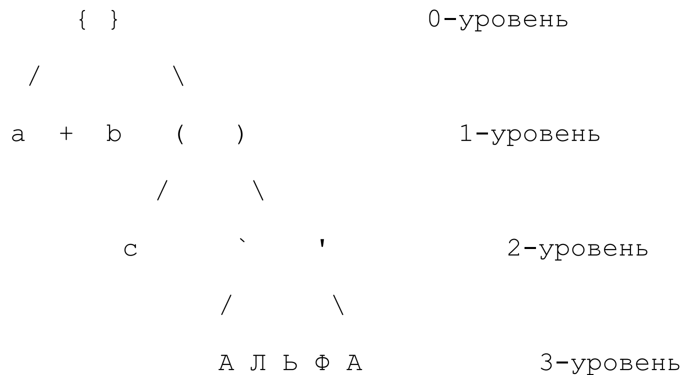


Рис. 2.1

Ведущим (функциональным) термом называется самый левый функциональный терм, не содержащий других функциональных термов. В примерах ниже ведущие термы выделены.

(a+b{c+d}); { АЛЬФА (a*b)}{cd}x10
(100 DO 3 {I={1}(,3)}).

Таким образом, мы полностью описали допустимую структуру поля зрения рефал-исполнителя (рефал-машины). В этом поле помещается обрабатываемый объект, который может быть только выражением. В качестве очередной заменяемой части всегда выбирается ведущий терм. Если такового нет, то считается, что делать исполнителю нечего, и он останавливается.

Выражение, оставшееся в поле зрения, считается результатом выполнения программы, находящейся в поле определений исполнителя.

[В авторской терминологии это поле называется "поле памяти". Так говорить нам неудобно. В модели Н и программа, и данные находились в памяти. Естественно считать, что поле зрения рефал-машины - также часть памяти. Термин "поле определений" лучше отражает суть дела.]

2.2.2.2. Поле определений; рефал-предложения

[Мы изучаем модели ЯП. Поэтому будем позволять себе "вариации на тему" рассматриваемого языка-прототипа, когда такие вариации упрощают рассмотрение. Следовательно, сведения о конкретном языке не следует воспринимать как руководство по программированию на нем.

Например, говоря о рефал-предложениях, мы не будем строго следовать их авторской трактовке.]

В нормальном алгоритме Маркова средства описания правил анализа и синтеза бедны - можно лишь явно выписывать заменяемое и заменяющее подслова (левую и правую части марковской формулы соответственно).

Основная идея обобщения марковской формулы состоит в том, чтобы за счет введения локальных переменных наглядно изображать одной (обобщенной) формулой сразу целый класс подстановок (применимых к функциональным термам определенной структуры).

Ключевыми понятиями при этом служат интерпретация переменных и согласование (терма с обобщенной подстановкой при определенной интерпретации ее переменных).

Интерпретация переменных - это функция типа I:N->V, где N - множество обозначений переменных, V - множество их допустимых значений.

[Интерпретация напоминает состояние в модели Н. Только вместо адресов - обозначения переменных. Это по сути одно и то же. Но называем мы их по-разному,

так как они играют разные роли. Состояние в модели Н - глобальный объект, сохраняющийся между последовательными операциями, а интерпретация в модели М - локальный объект, действующий внутри операции подстановки.]

При конкретной интерпретации переменных обобщенная подстановка (в рефале ее называют предложением или рефал-предложением) изображает конкретную марковскую формулу подстановки.

Например, предложение

{10 e 00 s 1} -> s 101 e

где e и s - (локальные) переменные, при интерпретации

i1={e->00, s->11}

(здесь фигурные скобки - обозначение множества пар, составляющих интерпретацию) изображает марковскую формулу

{100000111}-->1110100 ,

а при интерпретации

i2={e->ABC,s->D}

- изображает марковскую формулу

{10ABC00D1}-->D101ABC.

Соответственно левая часть предложения изображает левую часть марковской формулы, а правая часть предложения - правую часть формулы.

Согласование - это тройка (t,i,s), где t - ведущий терм, s - предложение и i - интерпретация, при которой левая часть s изображает t.

Итак, за счет различных интерпретаций переменных одна обобщенная марковская подстановка (предложение) способна изображать целый класс марковских подстановок (что и требовалось).

Однако этот класс не должен быть слишком широким - ведь каждое предложение должно быть приспособлено для (наглядного) изображения вполне определенного содержательного преобразования поля зрения. Поэтому следует принять меры к тому, чтобы ,во-первых, изображаемые подстановки не нарушали структуру поля зрения, и ,во-вторых, чтобы можно было управлять допустимыми значениями переменных (другими словами, управлять их типом).

Наконец, в-третьих, необходимо установить такие правила согласования предложения с ведущим термом, чтобы анализ и синтез были однозначными. Другими словами, правила согласования должны обеспечивать единственность подразумеваемой программистом согласующей интерпретации (при фиксированном поле зрения).

Первое и второе достигаеттся за счет ограничений на класс допустимых интерпретаций, третье - за счет ограничений на класс допустимых согласований.

Именно, допустимые интерпретации должны удовлетворять двум условиям.

2.2.2.2.1. Значения переменных, а также обе части изображаемой подстановки должны быть выражениями.

Так что М-преобразования не выводят за класс выражений.

2.2.2.2.2. Значение переменной должно соответствовать так называемому спецификатору, который указывается непосредственно после обозначения переменной и отделяеется двоеточием ":".

[Понятие спецификатора связано с еще одним (в некотором смысле ортогональным) направлением обобщения марковской формулы подстановки. Это направление мы оставим открытым и будем использовать пока только очень простые спецификаторы. Именно, в качестве спецификатора можно написать "символ" или "терм" (это значит, что значениями переменной могут быть только символы (только термы)) или в круглых скобках можно явно перечислить допустимые значения переменной.

Например, s:символ - переменная, значениями которой могут быть только символы, t:терм - только термы, s:(+I-) - значениями s могут быть только литеры "+" или "-".]

Ограничения на согласования состоят в том, что допустимыми считаются только так называемые ориентированные согласования. Они бывают левыми или правыми.

2.2.2.2.3. Определение ориентированного согласования

Определим левое (левоориентированное) согласование. Правое определяется по симметричным правилам.

Будем называть переменную y1 в функциональном терме левой для переменной y2, если самое левое вхождение y1 расположено левее самого левого вхождения переменной y2.

Будем говорить, что согласование (t,i',s) короче согласования (t,i,s), если в t найдется переменная y1, для которой i'(y1) короче i(y1), причем для любой переменной z, левой для y1 в терме t, i'(z) совпадает с i(z).

Согласование (t,i,s) называется левым, если оно самое короткое из возможных согласований t и s.

Таким образом, основная идея левого согласования - левые переменные при поиске согласующей интерпретации удлиняются в последнюю очередь.

Конец определения.

По умолчанию предполагается, что допустимы только левые согласования. Допустимость только правых согласований указывается буквой R после закрывающей функциональной скобки в левой части предложения.

Например, предложение

{e1+e2}-->{e1}{e2}+

согласуется с термом {a+b+c+d} за счет интерпретации

{e1-->a, e2-->b+c+d}

и изображает формулу подстановки

{a+b+c+d} --> {a}{b+c+d}+ ,

а предложение

{e1+e2}R --> {e1}{e2}+

согласуется с тем же термом за счет интерпретации

{e1-->a+b+c, e2-->d}

и изображает формулу подстановки

{a+b+c+d} -> {a+b+c}{d}+.

В рефале принимаются меры к тому, чтобы всегда можно было отличить переменные от постоянных частей предложения. Если есть опасность спутать переменную и постоянную, то постоянную будем выделять.

Подводя итог, можно сказать, что идея подстановки работает в рефале три раза.

Во-первых, интерпретация i определяет подстановку значений переменных вместо их обозначений.

Во-вторых, тем самым она определяет соответствие обобщенной и конкретной марковских подстановок (т.е. "подстановку" конкретной подстановки вместо обобщенной).

Наконец, в-третьих, правая часть этой конкретной подстановки заменяет ведущий терм.

При этом подбор согласующей интерпретации есть, по существу, анализ ведущего терма, а порождение конкретной правой части подстановки при найденной интерпретации - синтез заменяющего выражения (в правой части всегда должно быть правильное выражение - это еще одно требование рефала). В этом смысле левая часть предложения служит образцом структуры ведущего терма (терм и предложение согласуются, если структура терма соответствует образцу), а правая - образцом для синтезируемого заменяющего выражения.

Упражнение. Покажите, что если ведущий терм согласуется с некоторым предложением, то соответствующее согласование единственно.

Подсказка. Оно либо левое, либо правое.

2.2.3. Исполнитель (рефал-машина)

Теперь легко объяснить, как действует исполнитель, имея в поле зрения обрабатываемое выражение, а в поле определений - программу (т.е. кортеж предложений). Он выполняет следующий цикл.

Во-первых, выделяет ведущий терм. Если такового нет, останавливается. Выражение в поле зрения считается результатом.

Во-вторых, ищет первое по порядку предложение, которое согласуется с ведущим термом. Соответствующее согласование всегда единственно. Значит, единственна и изображаемая при соответствующей интерпретации переменных марковская подстановка. Она и применяется к ведущему терму. И цикл начинается сначала с обновленным полем зрения.

Если нет согласующихся с ведущим термом предложений, то исполнитель останавливается с диагностикой "согласование невозможно".

2.2.4. Программирование "в стиле рефала"

Задачу перевода в ПОЛИЗ (с учетом старшинства операций) решает следующая программа.

```
{e1+e2}R -> {e1}{e2}+
{e1*e2}R -> {e1}{e2}*
{(e)} -> {e}
{e} -> e
```

Упражнение 1. Доказать, что это правильная программа.

[Обратите внимание: действиями исполнителя полностью управляет структура обрабатываемых данных.] .

Упражнение 2. Можно ли эту программу написать короче? Например, так:

```
{e1 s:(+I*) e2}R -> {e1}{e2}S
{(e)} -> {e}
{e} -> e
```

Упражнение 3. Можно ли здесь отказаться от правого согласования?

Задача. Напишите на рефале программу аналитического дифференцирования многочленов по переменной "x".

2.2.5. Основное семантическое соотношение в модели М

Рассмотрим функцию sem, реализуемую рефал-программой p. Ее тип, очевидно

sem:P x E -> E

где P - программы, E - выражения.

[Уже тип функции sem указывает на принципиалъное отличие от модели Н - программа не меняется. В модели Н программа - часть (изменяемого) состояния.]

Пусть ft - функция, выделяющая в выражении ведущий функциональный терм, l и r - функции, выделяющие соответственно левую и правую части выражения, оставшиеся после удаления ведущего терма. Конкатекацию (соединение) строк литер будем обозначать точкой ".". Удобно считать, что если ведущего терма в выражении e нет, то ft = <>, r(e) = e, где <> обозначает пустое слово. Все эти три функции типа E -> W, где W - тип "слов" (произвольных последовательностей литер), так как результаты могут и не быть выражениями.

Пусть, далее step - функция типа

P x T' -> E ,

где T' = T U {<>}. Эта функция реализуется одним шагом работы рефал-машины - функция step отображает пару

(программа, ведущий терм или пусто)

в выражение, получающееся из этого терма применением соответствующей марковской подстановки. Функция step, естественно, частичная - она не определена, если согласование с p невозможно; step(p,<>) = <> по определению.

Учтем, что p не меняется и вся зависимость sem от p скрыта в функции step. Поэтому позволим себе для краткости явно не указывать p среди аргументов функций sem и step. Тогда можно выписать следующее соотношение для sem:

sem(e) = sem(l(e).step(ft(e)).r(e))

Если обозначить l(e), r(e) и ft(e) соответственно через l, r и f, то получим более выразительное соотношение:

(a) sem(l.ft.r) = sem(l.step(ft).r)

Покажем, что на самом деле справедливо следующее основное соотношение

(b) sem(l.ft.r) = sem(l.sem(ft).r)

Действительно, если step(ft) не содержит функциональных термов, то

sem(ft) = step(ft)

и (b) следует из (a).

Если же step (ft) содержит функциональные термы, то так как l таких термов не содержит, все функциональные термы из step(ft) будут заменены раньше, чем изменится l или r. Но последовательные замены термов в step(ft) - это и есть вычисление sem(ft).

Если такое вычисление завершается и между l и r не остается функциональных термов, то вычисление sem от исходного выражения будет нормально продолжено с выражения l.sem(ft).r.

Если же sem(ft) вычислить не удается из-за отсутствия согласования, то на этом же месте окажется невозможным согласование и для исходного выражения. Тем самым равенство доказано.

В соотношении (b) зафиксированы следующие свойства семантики рефала.

Во-первых, результат применения программы к ведущему терму не зависит от его контекста, а значит, и от истории применения программы к исходному выражению.

Во-вторых, "область изменения" в выражении e до полного вычисления его ведущего терма ограничена этим термом.

В-третьих, если l и r не содержат функциональных скобок, они никогда не могут быть изменени.

Аналогичными рассуждениями можно обобщить соотношение (a). Обозначим через ft1,...,ftn последовательные терминальные функциональные термы в e (т.е. не содержащие других функциональных термов), а через r0,...,rn - слова , не содержащие функциональных термов и такие, что

e = r0.ft1.,...,.ftn.rn

Тогда справедливо следующее соотношение:

(c) sem(r0.ft1.,...,.ftn) = sem(r0.sem(ft1).,...,.sem(ftn).rn)

Упражнение. Докажите справедливость этого соотношения.

Не забудьте, что участок r0.sem(ft1).,...,.sem(ftn).rn может содержать функциональные термы.

Отметим также очевидное соотношение

sem(sem(e)) = sem(e).

Таким образом, обработка в модели M обладает четкой иерархической структурой. Другими словами, выполнение программы p над выражением e можно представлять себе как "вычисление" этого выражения, начиная с любого из "терминальных функциональных поддеревьев" соответствующего дерева.

2.2.6. Пример вычисления в модели M

Сопоставим вычисление по школьным правилам выражения (10+2)*(3+5) с обработкой в модели M выражения {10+2} {3+5}* по программе перевода в ПОЛИЗ. Изобразим последовательно получаемые деревья, соответствующие обрабатываемым выражениям (слева - для школьной арифметики, справа - для рефала).

Шаг 1 (исходные деревья).

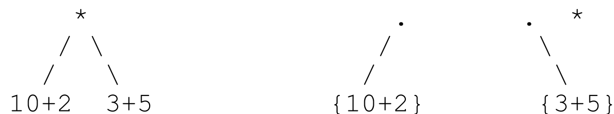


Рис. 2.2

Деревья явно похожи (вершины изображают операции, дуги - отсылки к тем операндам, которые еще следует вычислить).

Шаг 2 (применение одной из операций, для которых готовы операнды).

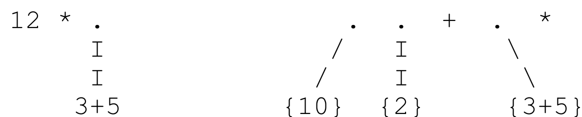


Рис. 2.3

Видно, что дерево справа "отстает" от дерева слева. Сказывается различие результатов функций step и sem. Последим за правым деревом до завершения вычисления функции sem({10+2}).

Шаг 2.1.

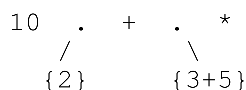


Рис.2.4

Шаг 2.2.

```
10  2  +  .  *
           \
          {3+5}

       Рис. 2.5
```

Вот теперь деревья снова похожи!

Слово "вычисление" означает здесь процесс, вполне аналогичный вычислению значения обычного школьного алгебраического выражения после подстановки вместо переменных их значений. Однако, аналогия касается не типа допустимых значений (в школьной алгебре - числа, а здесь - сбалансированные по скобкам тексты), а способа планирования обработки (способа программирования).

И в школьной алгебре, и в рефале план обработки в определенном смысле содержится в обрабатываемом (вычисляемом) выражении. Роль исполнителя состоит в том, чтобы выполнять указанные операции над допустимыми операндами, подставляя результат операций в обрабатываемое выражение на место вычисленного терма.

[Существенные отличия состоят в том, что, во-первых, школьные операции считаются заранее известными, предопределенными, а смысл единственной рефал-операции step задается полем определений; во-вторых, результат школьных операций - всегда "окончательный" (новых операций в нем не содержится - это число), а результат операции step - в общем случае "промежуточный" - им может оказаться выражение с новыми функциональными термами. Заметим, что второе отличие исчезает, если от функции step перейти к функции sem - ее результат всегда "окончательный", ведь (sem(sem(e)) = sem(e)).].

Шаг 3.

```
12*8              10  2  +  .  .  +  *
                            /    \
                          {3}   {5}
          Рис. 2.6
```

Шаг 3.1.

```
                      10  2  +  3  .  +  *
                                    \
                                    {5}
               Рис. 2.7
```

Шаг 3.2.

```
                      10  2  +  3  5  +  *
               Рис. 2.8
```

Шаг 4.

```
    96                10  2  +  3  5  +  *
     (нет функциональных термов)

               Рис. 2.9
```

Итак, мы убедились, что вычисления в рефале очень похожи на вычисления обычных арифметических формул.

2.2.7. Несколько замечаний

2.2.7.1. Вычисления по формулам очень поучительны для программистов. Из этого древнейшего способа планирования вычислений можно извлечь много полезных идей.

Во-первых, это четкая структура вычислений - она, как мы видели, древовидная.

Во-вторых, операнды рядом с операциями (их не нужно доставать из общей памяти).

В-третьих, результат не зависит от допустимого изменения порядка действий - (с сохранением иерархии в соответствии с деревом выражения). Отсюда - путь к параллельному вычислению, если позволяют вычислительные ресурсы (когда есть несколько процессоров).

В-четвертых, принцип синхронизации таких вычислений прост - всякая операция должна ждать завершения вычислений своих операндов, и только этого события она должна ждать (ничто другое на ее выполнение не влияет). На этом принципе основаны так называемые конвейерные вычисления и вычисления "управляемые потоком данных" (data flow).

В-пятых, результаты операций никуда не нужно посылать - они нужны там, где получены.

Наконец, отметим еще одну идею, в последние годы привлекающую внимание исследователей, стремящихся сделать программирование надежным, доказательным, систематическим. Речь идет о том, что над школьными формулами можно выполнять систематические преобразования (упрощать, приводить подобные члены, явно выражать неизвестные в соотношениях и т.п.). Есть надежда определить практичную алгебру преобразований и над хорошо организованными программами. Это позволит систематически выводить программы, проверять их свойства, оптимизировать и т.п.

2.2.7.2. Обратите внимание: значение функции sem не зависит от порядка вычисления терминальных функциональных термов. А в нашем исходном определении модели М требовалось, чтобы всегда выбирался самый левый из всех таких термов. При отсутствии взаимного влияния непересекающихся термов такое требование несущественно. [В реальном рефале указанное влияние возможно.]

2.2.8. Аппликативное программирование

Модель М относится к широкому классу так называемых аппликативных моделей вычислений. Это название (от слова apply - применять) связано с тем, что в некотором смысле единственной операцией в таких моделях оказывается операция применения функции к ее аргументу, причем единственной формой влияния одного применения на другое служит связь по результатам (суперпозиция функций). В частности, функции не имеют побочного эффекта.

[Напомним, что побочным эффектом функции называется ее влияние на глобальные объекты, не являющиеся аргументами; в модели М переменные локальны в предложениях, а отсутствие побочного эффекта на поле зрения мы уже обсуждали.]

Аппликативные модели привлекательны тем, что сохраняют многие полезные свойства вычислений по формулам. Самое важное из них - простая и ясная структура программы, четко отражающая требования к порядку вычислений и связям компонент.

Вместе с тем по своей алгоритмической мощности аппликативные модели не уступают другим моделям вычислений.

Задача. Доказать, что модель М алгоритмически полна, т.е. для всякого нормального алгоритма А найдется эквивалентная ему рефал-программа (допускается заменять алфавит, в котором работает А).

Однако, пока наша модель М бедна в том отношении, что ко всем термам применяется одна и та же функции step. Это плохо и потому, что программу трудно понимать (особенно, если она длинная) и потому, что она будет медленно работать, если каждый раз просматривать все предложения поля определений.

К счастью, модель М легко приспособить к более гибкому стилю аппликативного программирования.

2.2.9. Структуризация поля определений; рефал-функции

Допустим, что имеется неограниченный набор различных пар функциональных скобок (как это можно обеспечить?). Будем группировать предложения, записывая подряд друг за другом такие предложения, левая часть которых заключена в одинаковые функциональные скобки.

Тогда ведущий терм будет однозначно указывать на соответствующую группу предложений (в ней и только в ней достаточно искать согласование).

В этом случае функция step распадается на отдельные функции, а программа - на определения этих функций (за что соответствующее поле, где помещается рефал-программа, мы и назвали полем определений).

Достаточно различать только левые функциональные скобки (почему?).

Будем считать левой функциональной скобкой название (идентификатор) функции вместе с непосредственно следующей за ним открывающей фигурной скобкой.

Например, программу перевода в ПОЛИЗ запишем так:

```
перевод {e1+e2}R -> перевод {e1}  перевод {e2} +
     перевод {e1*e2}R -> перевод {e1}  перевод {e2} *
     перевод {(e)} -> перевод {e}
     перевод {e} -> e.
```

Такую совокупность подстановок естественно считать определением рефал-функции "перевод". Его удобно использовать в большой программе среди других подобных определений.

Поле зрения с исходными данными для перевода может выглядеть при этом так:

```
перевод {(a+b) * (c+d)}
```

Как видим, и запись самой программы в модели М, и обращение к ней весьма напоминает то, что мы выбрали в качестве идеала в самом начале разговора об анализе и синтезе. [Недостаточна, правда, выразительная сила применяемых в нашей модели образцов. Поэтому приходится писать подробнее, чем в БНФ].

До сих пор мы смотрели на поле определений как на определение одной функции. Это была либо функция step, если результат считался полученным после одного применения подстановки, либо (в общем случае рекурсивная) функция sem, если результатом признавалось только выражение без функциональных термов.

Когда поле определений разбито на группы подстановок с одинаковыми левыми функциональными скобками, каждую такую группу естественно считать определением отдельной функции. С точки зрения одного шага рефал-машины - это функция, представляющая собой сужение функции step на ведущие термы с конкретной функциональной скобкой. С технологической точки зрения (с точки зрения программиста) - это (рекурсивная) рефал-функция, представляющая собой сужение функции sem на те же термы.

Замечание. Применение рефал-функций предполагает уже некоторый элемент прогнозирования со стороны программиста и контроля со стороны рефал-машины, отсутствовавший в исходной модели.

Именно, употребляя конкретную функциональную скобку в правой части предложения, программист прогнозирует, что при определенном поведении исполнителя (если будет выбрано именно это предложение) потребуется определение соответствующей функции.

Рефал-машина, со своей стороны, получает возможность просмотреть поле определений и проверить, что в нем присутствуют определения всех использованных рефал-функций. Другими словами, становится возможным статический контроль программ (т.е. контроль программ до их выполнения, без учета исходных данных).

Конец замечания.

Итак, мы можем определять в программе столько (рекурсивных) функций, сколько нужно.

Вот, например, как выглядит программа аналитического дифференцирования, в которой используется частная производная по x и частная производная по y.

```
Dx{e1+e2}R -> Dx{e1} + Dx{e2}
Dx{e1*e2}R -> e1*(Dx{e2}) + e2*{Dx{e1})
Dx{(e)} -> Dx{e}
Dx{`x'} -> 1
Dx{s: символ} -> 0
Dy{e1+e2}R -> Dy{e1} + Dy{e2}
. . . . . .
. . . . . .
Dy{`y'} -> 1
Dy{s: символ} -> 0
```

Задача. Можно ли объединить эти функции? Как это сделать?

2.2.10. Функциональное программирование

В соответствии с определением А.П.Ершова функциональное программирование - это способ составления программ, в которых единственным действием является вызов (применение) функции, единственным способом расчленения программ на части - введение имени для функции и задание для него выражения, вычисляющего значение этой функции, единственным правилом композиции (структурой операций) служит суперпозиция функций.

Ясно, что модель М с учетом последней "функциональной" модификации позволяет программировать в строго функциональном стиле. Другими словами - это одна из моделей функционального программирования.

[Таким образом, одно из отличий "функционального" программирования от "аппликативного" - возможность явно определять (в общем случае рекурсивные) функции].

Дополнительные примеры программирования в "функциональном стиле" мы приведем чуть позже, а пока завершим введение в рефал кратким обзором "функциональной" модели М с точки зрения нашей концептуальной схемы.

2.2.11. Модель М с точки зрения концептуальной схемы

Базис: скалярные данные - только литеры, скалярные операции - только обобщенная поиск-подстановка. Структурные данные - только выражения (есть подтипы: символ и терм), структурные операции - встроенный цикл, легко приводящий к комбинациям функций.

[Говорят, что функции комбинируются горизонтально, если их результаты являются непосредственными составляющими одного функционального терма.

Говорят, что функции комбинируются вертикально, если одна из них не может быть вычислена до завершения вычисления другой. В такой комбинации первая называется внещней, а вторая - внутренней.

В модели М применяется и горизонтальная, и вертикальная комбинация функций. Горизонтальная комбинация называется также конструкцией, а вертикальная, при которой результат внутренней служит полным аргументом внешней - композицией. Произвольная комбинация - суперпозицией.]

Развитие: вверх - только функции типа E -> E (однако за счет структурированности выражений это весьма мощное средство развития (как будет показано)); вниз - средств нет.

Защита: в базисе средств нет.

2.2.12. Модель М и Лисп

Можно показать, что модель М отражает не только свойства такого реального языка, как рефал, но и свойства еще одного заслуженного языка, языка Лисп, созданного Джоном Маккарти в 1960 году и с тех пор прочно удерживающего позиции

одного из самых распространенных ЯП (особенно в качестве инструментального языка в области искусственного интеллекта). В последние годы интерес к нему усилился еще и как к первому реальному языку функционального программирования.

Единственной базисной структурой данных в Лиспе служит список (так называемое S-выражение). Оно естественно представимо в модели М выражением в круглых скобках. Элементарные селекторы и конструкторы Лиспа (предопределенные функции, позволяющие выбирать из списков компоненты и строить новые списки из заготовок) легко программируются в модели М.

[Приведем упрощенные определения рефал-функций, способных играть роль селекторов и конструкторов. Для краткости всюду ниже будем считать, что с обозначениями рефал-переменных, начинающихся с буквы s и t, связаны соответственно спецификаторы "символ" и "терм" (так и делается в реальном рефале)].

Выбор головы (первого элемента) списка:

```
первый {(t e)} -> t.
```

Выбор хвоста списка:

```
хвост {(t e)} -> (e).
```

Конструирование (создание) списка:

```
создать {e} -> (e).
```

Соединение списков:

```
соединить {(e1)(e2)} -> (e1 e2).
```

Подобным образом программируются и другие функции, аналогичные примитивам Лиспа.

Упражнение. Возьмите руководство по языку Лисп и аккуратно выпишите рефал-определения примитивов (базисных функций) Лиспа. Учтите все их тонкости. Рассмотрите отличия функций первый, хвост и создать от функций car, cdr и cons Лиспа.

Обратите внимание: по существу мы продемонстрировали способность модели М к развитию - довольно легко определить в модели М новый язык, аналогичный Лиспу.

2.3. Критерий концептуальной ясности и функции высших порядков

Предыдущий раздел мы закончили программированием в модели М базисных примитивов языка Лисп - использовали средства развития модели М.

Напомним, что с точки зрения нашей концептуальной схемы способность к развитию - одно из важнейших свойств модели.

Продолжая рассматривать модели ЯП с технологической позиции, продемонстрируем технологическую потребность в функциях высших порядков (т.е. функциях, аргументами и (или) результатами которых служат функции). Затем мы, во-первых, покажем, как их можно ввести в модели М, и, во-вторых, рассмотрим модель Бэкуса (модель Б), в которой функции высших порядков играют ключевую роль (введены в базис).

Напомним, что к модели М мы пришли от идеи разделения анализа и синтеза в обработке данных. И получили мощные средства развития, как только ввели удобную базисную структуру данных (выражение), локализовали область воздействия на эту структуру (ведущий терм) и упростили отбор возможных воздействий (ввели рефал-функции).

Теперь у нас в руках аппарат, который можно развивать в различных направлениях и (или) использовать в различных целях.

[Например, в реальном рефале введены операции, позволяющие изменить поле определений в процессе исполнения программы. Это так называемые операции "закапывания" и "выкапывания" определений по принципу магазина. При таком развитии получается стиль программирования, более близкий к традиционному, с присваиванием глобальным переменным и взаимным влиянием непересекающихся термов. Нас больше интересует развитие в функциональном стиле.]

Воспользуемся аппаратом развития, чтобы показать богатейшие возможности функционального программирования с точки зрения достижения концептуальной ясности программ.

Идеалом будет служить такая программа, в которой в некотором смысле нет ничего лишнего. Другими словами этот критерий концептуальной ясности можно выразить так - структура функции, реализуемой программой, совпадает со структурой программы.

[Однако при этом функция "состоит" из соответствий, а программа - из операций.]

Важнейшая абстракция, способствующая приближению к намеченному идеалу - функция высшего порядка (или, как мы ее назовем, следуя Бэкусу, форма). Ближайшая задача - показать это на достаточно убедительных примерах.

Замечание. Важно понимать, что хотя модель М, конечно, алгоритмически полна, она (как и любая другая модель) не универсальна в том смысле, что в ней не всегда легко вводить любые абстракции. Однако формы в ней вводить довольно легко.

Конец замечания.

2.3.1. Зачем нужны функции высших порядков

Они возникают совершенно естественно. Классический пример - программа интегрирования (вычисления определенного интеграла). Она реализует некоторую форму, аргументом которой служит подынтегральная функция, а результатом - число. Программа аналитического дифференциирования реализует форму, аргументом которой служит некоторая функция (заданная, скажем, многочленом), а результатом - ее производная, т.е. снова функция.

Любая из рассмотренных нами функций, выражающих денотационную семантику модели Н или М, получается, как мы видели, определенной комбинацией исходных функций, соответствующих базисным конструкциям. Если изменить эти исходные функции, не меняя зафиксированной нами формы, представленной их комбинацией, то получим другую семантику модели.

Так, если в модели Н изменить семантику операций - изменится семантика программы. В модели М также можно варьировать, скажем, правила согласования или подстановки без всякого изменения денотационных соотношений - они-то и фиксируют вполне определенную форму, отображающую пару (step,p) в sem.

2.3.2. Замечания о функциях высших порядков

2.3.2.1. Чтобы говорить точно, напомним, что рассматриваем мы функции только типа

E -> E.

В частности, это означает, что все они формально имеют один аргумент. Фактически может быть столько аргументов, сколько нужно - ведь аргументами можно всегда считать последовательные термы выражения. Отдельные аргументы можно всегда заключить в круглые скобки.

Однако чтобы не загромождать примеры, договоримся, что отделение аргументов пробелами эквивалентно заключению в скобки. Другими словами, будем в значительной степени абстрагироваться от "проблемы круглых скобок", концентрируя внимание на принципиальных моментах (хорошо понимая, что в практическом программировании от этой проблемы никуда не деться - в Лиспе, например, она одна из самых неприятных).

2.3.2.2. Как только мы сказали, что имеем дело с функциями

E -> E

сразу возникает вопрос, как же быть с формами. У них-то аргументы - функции, а не выражения. Ответ состоит в том, что и аргументы, и результаты форм всегда будут представлены некоторыми выражениями (например, символами - названиями функций).

2.3.2.3. Примем стиль изложения, при котором смысл вводимых программистских абстракций будем объяснять с помощью определений в модели М. Иногда это может показаться трудным для восприятия. Однако зато мы, во-первых, постоянно упражняемся в программировании в модели М; во-вторых, немедленно демонстрируем конкретизацию вводимой абстракции - а именно ее реализацию в известной модели.

[Такой стиль можно назвать проекционным - вместе с новым понятием излагается его проекция (перевод) на уже известный инструментальный язык. В нашем случае основу этого языка предоставит модель М.]

2.3.2.4. Первая из форм, которую следовало бы рассмотреть - это, конечно, аппликация (которую обозначим двоеточием ":"). Она применяет указанную в ее аргументе функцию (возможно, форму) к остальным компонентам аргумента. Можно было бы определить аппликацию в общем виде, однако нам удобнее считать, что определение рефал-функции ":" формируется постепенно. А именно, группа предложений со специальной функциональной скобкой вида ":{" пополняется новыми предложениями по мере введения новых форм.

Таким способом (за счет возможностей рефальских образцов) можно определять новые формы (и обычные функции), не требуя, чтобы обращение к ним было обязательно

префиксным (т.е. чтобы название функции предшествовало аргументам). Префиксный способ требует слишком много скобок, поэтому его желательно избегать, когда функция (форма) обладает, скажем, свойством ассоциативности.

Упражнение. Покажите, как можно вводить инфиксные функции.

Подсказка. Вспомните о переводе в ПОЛИЗ.

Пока будем считать, что в группе аппликации лишь два предложения

```
            :{(f) e} -> :{f e}
   (апл)
            :{s_f  e} -> s_f{ e }
```

где f - переменная, обозначающая вызов некоторой формы, а s_f - переменная, обозначающая название применяемой рефал-функции.

Первое предложение снимает скобки, ограничивающие вызов формы (они могли остаться после вычисления значения ее результата, если он был задан инфиксным выражением), а второе выписывает функциональный терм, который служит вызовом применяемой функции.

Подразумевается, что определения применяемых функций в рефал-программе имеются. Предложения (апл) будут оставаться последними в группе аппликации. Новые будем добавлять в ее начало (чтобы сначала действовали формы, а лишь затем их результаты - обычные функции).

2.3.3. Примеры структурирующих форм

Намеченный идеал концептуальной ясности наводит на мысль, что наиболее важными могут оказаться формы, помогающие рационально структурировать программу - выражать ее смысл (реализуемую функцию) простой и понятной комбинацией других функций. Рассмотрим несколько таких структурирующих форм.

Первая из них - композиция (ее часто обозначают звездочкой "*"). Применить результат композиции двух функций f и g - значит применить функцию f к результату применения g. "Применить" - это значит использовать аппликацию. В модели М определение композиции выглядит так:

```
        :{(f*g)e} -> :{(f) :{(g) e}}.
```

Точнее говоря, чтобы это предложение заработало как определение новой формы (а именно композиции), им следует пополнить группу (апл).

Вторая полезная форма - "общая аппликация" (применение указанной в аргументе функции ко всем непосредственным составляющим обрабатываемого выражения). Обозначим ее через "A" по аналогии с квантором всеобщности. Для ее определения через аппликацию в группу (апл) следует добавить два рефал-предложения

```
:{(Af)t e} -> :{(f)t} :{(Af)e}
:{(Af)  } -> <>    .
```

Итак, указанная выражением f функция применяется к компонентам обрабатываемого выражения. Получается выражение, составленное из результатов всех применений.

Вопрос. Зачем понадобилось второе предложение?

Третья структурирующая форма - конструкция (ее обозначим запятой ","). Применить результат конструкции двух функций f и g к выражению e - значит получить конкатенацию выражений f(e) и g{e}.

Определить конструкцию в модели М можно так.

```
:{(f,g) e} -> :{(f)e}  :{(g)e}   .
```

Еще один пример формы - редукция, которую обозначим через "/". Название, идея и обозначение восходят к Айверсону, автору языка Апл - одного из самых распространенных диалоговых языков. Своей исключительной лаконичностью этот язык в значительной степени обязан функциям высших порядков.

```
:{(/f) t1 t2 e} -> :{(f) t1 :{(/f) t2 e}}.
:{(/f) t} -> t.
```

Идея редукции в том, что бинарная операция f (двухместная функция) последовательно применяется, начиная с конца выражения вида (t1 t2 e) - т.е. выражения, в котором не меньше двух составляющих. Название этой формы подчеркивает, что обрабатываемое выражение сворачивается к одному терму (редуцируется) за счет последовательного "сьедания" пар компонент выражения, начиная с его конца.

Например, с помощью редукции можно определить функцию "сумма".

```
 сумма{e} -> :{(/+) e}     .
```

Тогда если считать, что бинарная операция "+" предопределена и ее можно использовать префиксным способом, получим

сумма{10 20 30} = :{(/+) 10 20 30} =
= :{+10 :{(/+) 20 30}} =
= :{+10 :{+20 :{(/+) 30}}} =
= :{+10 :{+20 30}} = :{+10 50} = 60 .
Обратите внимание, сколь прост и привычен вид программы-формулы
сумма{10 20 30} = 60.

Итак, мы определили конструкцию, общую аппликацию, композицию, редукцию, В том же стиле с помощью аппликации можно определить и другие полезные формы.

Если программировать с использованием таких форм (и некоторых других), то по существу мы будем работать в модели БЭКУСА (модели Б). И снова развитие рефала новыми функциями дает новый язык - язык Бэкуса.

Отличительная черта модели Бэкуса - "фундаментализация" идеи функциональных форм. В частности, четыре названные выше формы считаются примитивными (предопределенными, т.е. определенными средствами, выходящими за рамки модели). Аналогичная идея - одна из основных в языке Апл Айверсона. Однако Айверсону, в отличие от Бэкуса, не удалось ее фундаментализировать (выделить как важнейшую, как основу целого направления в программировании).

2.3.4. Еще несколько функций над выражениями

Определим в модели М еще несколько функций, полезных для работы с выражениями.

реверс{t e} -> реверс{e} t .
реверс{ } -> <>.

Эта функция преобразует выражение вида t1 ... tn в выражение вида tn ... t1, где ti - термы.

Следующая функция - транспонирование (для краткости будем обозначать ее "транс"). По сути дела это обычное транспонирование матриц. Ее действие представим таблицей примеров.

e	ТРАНС (e)
(a b c) (k l m)	(a k) (b l) (c m)
(a b) (c k) (l m)	(a c l) (b k m)
(a b c) (k l m) (o p r)	(a k o) (b l p) (c m r)

Рис. 2.10

Здесь строки матрицы представлены последовательными термами рефал-выражения (это обычный способ представления структур в рефале).

Определим теперь фуекцию "транс" точно.

транс{e} -> первые{e} транс{хвосты{e}} .

транс{ } -> <>, где "первые" - функция, выделяющая список первых элементов последовательных подвыражений

e	ПЕРВЫЕ (e)
(a b c) (k l m)	(a k)
(a b) (c k) (l m)	(a c l)
(a b c) (k l m) (o p r)	(a k o)

Рис. 2.11

а "хвосты" - функция, выдающая список хвостов от последовательных подвыражений

e	ХВОСТЫ
(a b c) (k l m)	(b c) (l m)

```
I (a b) (c k) (l m)    I   (b)   (k)   (m)  I
I ((a b) c) ((r l) m)  I   (c)   (m)        I
---------------------------------------------
```

Рис. 2.12

Определим эти функции точно.

```
первые{(t1 e1) e2} -> (t1 первые{e2} )   .
первые{  } -> <>.
хвосты{(t1 t2 e1) e2} -> (t2 e1) дл-хвосты{e2}  .
хвосты{(t1) e} -> кор-хвосты{e}  .
дл-хвосты{(t1 t2 e1) e2} -> (t2 e1) дл-хвосты{e2}  .
дл-хвосты{  } -> <>.
кор-хвосты{(t) e} -> кор-хвосты{e}  .
кор-хвосты{  } -> <>.
```

Вопрос. Для чего понадобилось вводить функции дл-хвосты и кор-хвосты?
Подсказка. Мы хотим транспонировать только матрицы.

2.3.5. Пример программы в стиле Бэкуса

Теперь можно написать программу, выражающую в некотором смысле "идеал" программирования в стиле Бэкуса. Точнее, мы напишем программу-формулу, вычисляющую скалярное произведение двух векторов. Будем действовать методом пошаговой детализации.

Допустим, что предопределены функции "сложить" (+) и "умножить" (x). Представим подлежащие перемножению векторы выражением вида (e1)(e2), где e1 - первый вектор, e2 - второй. Исходная пара векторов представляет собой матрицу с двумя строками e1 и e2.

Вспомним определение скалярного произведения - это

```
          Сумма всех произведений
(d)       попарно соответствующих компонент
          подлежащих перемножению векторов.
```

Прочитаем это определение "с конца". Нужно, во-первых, получить попарно компоненты векторов e1 и e2, во-вторых, получить все произведения этих пар, в-третьих, получить сумму (сложить) все эти произведения.

Итак, план (программа) наших действий состоит из трех последовательных шагов, причем результат предыдущего шага непосредственно используется последующим шагом.

Следовательно, наша программа представляет собой композицию функций

    f3 * f2 * f1

Какие же это функции?

Функция f1 определяет то, что нужно сделать "во-первых". Если даны два вектора, скажем,

    (b1)   (10 20 30) (3 2 1)

то нужно получить их компоненты попарно.

    (b2)   (10 3) (20 2) (30 1)

С этим мы уже встречались, так работает функция "транс". Значит, естественно положить f1 = транс.

Функция f2 определяет то, что нужно сделать "во-вторых". Нужно получить все произведения пар. В нашем примере - это выражение

    (b3)   30 40 30

Такое выражение получится, если функцию "умножить" применить к каждому подвыражению выражения (b2). С подобным мы тоже встречались - так работает общая аппликация "A" с аргументом "умножить" (x).

Значит, естественно положить f2 = (Ax).

Наконец, f3 определяет, что нужно сделать "в-третьих". Нужно получить общую сумму всех компонент (b3), т.е.

    (b4)   100

Такое выражение получится, если к (b3) применить форму "редукция", с аргументом "сложить" (+). Значит, естественно положить

    f3 = (/ +).

Итак, можно выписать нашу программу-формулу полностью:

    (/+) * (Ax) * транс .

Эта формула описывает именно ту функцию, которая решает нашу задачу, т.е. вычисляет скалярное произведение. Использовать ее можно, как и раньше, двумя способами - либо непосредственно применять к обрабатываемому выражению:

```
:{((/+)*(Ax)*транс) (10 20 30) (3 2 1)} = 100  ,
```

либо ввести для нее название, скажем, IP

```
IP{e} -> :{((/+)*(Ax)*транс) e}  .
```

и использовать как обычную рефал-функцию:

```
IP{(10 20 30) (3 2 1)} = 100  .
```

Как видим, наша программа полностью соответствует определению скалярного произведения - все слова в этом определении использованы и ничего лишнего не понадобилось вводить (мы записали программу, не использовав ни одного лишнего понятия).

Намеченный идеал концептуальной ясности для данной программы достигнут. Для других программ вопрос открыт, но направление должно чувствоваться. С другой стороны, мы показали, как средства развития в модели М, позволяя вводить адекватные понятия (абстракции), помогают бороться со сложностью создания программ.

Задача. Можно ли аналогичные средства ввести в Алголе-60, Фортране, Бейсике? Дайте обоснованный ответ.

2.3.6. Сравнение с программой на Алголе 60

Рассмотрим фрагмент программы на Алголе 60:

```
        c := 0;
(ap)    for  i:=1  step 1  until  n  do
        c := c + a[i] x b[i];
```

Такой фрагмент вычисляет скалярное произведение двух векторов a и b.

Попытаемся сопоставить его с определением скалярного произведения (d).

Во-первых, сразу видно, что естественная композиция функций в программе (ap) не отражена. Пришлось заменить ее последовательными действиями с компонентами векторов.

Во-вторых, пришлось ввести пять названий c,i,n,a,b, никак не фигурирующих в исходной постановке задачи. Причем, если по отношению к a,b и c еще можно сказать, что это обозначения исходных данных и результата, то что такое i и зачем понадобилось n?

Ответ таков, что на Алголе со структурами-массивами по-другому работать нельзя. Мы работали с выражением в модели М как с целостным объектом, а в Алголе 60 над массивами возможны лишь "мелкие" поэлементные операции (для этого понадобилась переменная i). К тому же нельзя узнать размер массива, необходимо явно указывать этот размер (n).

В-третьих, мы уже говорили о возможности распараллелить работу по функциональной программе-формуле. А как это сделать в программе (ap)? Опять сравнение не в пользу Алгола.

Задача. Найдите аргументы в пользу Алгола.

Замечание. Программа скалярного произведения в модели Б - это формула, операциями в которой служат формы, а операндами - основные скалярные функции (+, x) и некоторые другие (транс). В этой связи интересно напомнить, что Джон Бэкус - "отец" Фортрана, который тоже начинался как Formula Translation (и "испортился" под натиском "эффективности"). Так что Джон Бэкус пронес идею "формульного" программирования через многие годы, от своего первого знаменитого Фортрана до теперь уже также знаменитого "функционального стиля". Излагая модель Б, мы пользуемся лекцией, прочитанной Джоном Бэкусом по случаю вручения ему премии Тьюринга за выдающийся вклад в информатику [3].

Мы показали, как функции высших порядков помогают писать концептуально ясные программы. В дальнейшем нам предстоит заняться моделью Б подробнее. Основная цель - познакомить с алгеброй программ, разработанной в этой модели, и с применением алгебры для доказательного программирования.

2.4. Модель Бэкуса

Мы показали, как функции высших порядков помогают писать концептуально ясные программы. Теперь займемся моделью Б подробнее. Основная цель - познакомить с разработанной в этой модели алгеброй программ и с ее применением для

доказательства эквивалентности программ. Чтобы законы в этой алгебре были относительно простыми, нам понадобится, во-первых, ограничить класс обрабатываемых объектов - считать объектами не произвольные выражения, а только М-термы (т.е. термы в смысле модели М); во-вторых, так подправить определения форм, чтобы их применение всегда давало объекты. Наконец, придется ввести функции, позволяющие раскрывать и создавать термы.

Для выразительности и краткости при наших определениях будем пользоваться общематематической символикой. Однако все нужные объекты, функции и формы можно без принципиальных трудностей ввести и средствами модели М.

2.4.1. Модель Бэкуса с точки зрения концептуальной схемы

Имея опыт работы со структуризованными объектами (выражениями), формами и рекурсивными определениями (который мы приобрели, работая в модели М) можно с самого начала рассматривать модель Бэкуса (модель Б) по нашей концептуальной схеме. Чтобы не загромождать изложение, не будем постоянно подчеркивать различие между знаками в языке Б (модели Б) и их денотатами, надеясь, что из контекста всегда будет ясно, о чем идет речь. Например, будем называть формой как функцию высшего порядка (денотат), так и представляющее ее выражение (знак). Соответственно примитивной функцией будем называть как ее идентификатор (знак), так и обозначаемое этим идентификатором отображение из объектов в объекты (денотат).

Базис. В модели два скалярных типа - атомы и примитивные функции. Первые служат для конструирования объектов, вторые - для конструирования функций. Объекты и формы - это два структурных типа. Имеется единственная операция - аппликация.

Развитие. Единственным средством развития служит возможность пополнять набор D определений функций. Делается это с помощью фиксированного набора форм и примитивных функций. Определения могут быть рекурсивными.

2.4.2. Объекты

Объект - это либо атом, либо кортеж (последовательность) вида

< X1, ... , Xn >

где Xi - либо объект, либо специальный знак <?> - "неопределено".

Таким образом, выбор фмксированного множества А атомов полностью определяет множество всех объектов О.

Будем считать, что в А входят (т.е. служат атомами) идентификаторы, числа и некоторые специальные знаки (T,F и т.п.). Выделен специальный атом <> - это единственный объект, который считается одновременно и атомом, и (пустым) кортежем.

Замечание. Аналогично спискам Лиспа нетрудно предствить Б-объекты М-выражениями, введя подходящие обозначения для специальных объектов и заключая последовательности объектов в круглые скобки. Это же относится и к последующему неформальному изложению модели Б (хороший источник полезных упражнений по представлению Б-понятий М-понятиями).

Конец замечания.

Все объекты, содержащие <?> в качестве элемента, считаются по определению равными <?> (т.е. знаки различны, а денотаты - равны). Будем считать, что все такие объекты до применения к ним каких бы то ни было операций заменяются "каноническим" представлением "<?>".

Примеры объектов: <?>, 15, AB3, <AB,1,2,3>, <a,<<B>,C>,D>.

2.4.3. Аппликация

Смысл этой операции известен. Обозначать ее будем по-прежнему через ":", однако использовать - не как префиксную, а как инфиксную операцию. Так что если f - функция и X - объект, то

f:X

обозначает результат применения функции f к объекту X. Например

+:<1,2>=3, 1:<A,B,C>=A, 2:<A,B,C>=B, t1:<A,B,C>=<B,C>

где слева от знака аппликации ":" выписаны знаки функций сложения, извлечения первого элемента, извлечения второго элемента и хвоста кортежа соответственно.

2.4.4. Функции

Все Б-функции отображают объекты в объекты (т.е. имеют тип O => O) и сохраняют неопределенность (т.е. f : <?> = <?> для всех f).

Каждый знак Б-функции - это либо знак примитивной функции, либо знак формы, либо знак функции, определенной в D. Другими словами, в модели Б различаются, с одной стороны, предопределенные функции и формы, и ,с другой стороны, функции, определяемые программистом с помощью пополнения D.

Равенство f : X = <?> возможно всего в двух случаях, которые полезно различать. Во-первых, выполнение операции ":" может завершаться и давать в результате <?>, и, во-вторых, оно может оказаться бесконечным - тогда это равенство считается справедливым по определению операции ":". Другими словами, виды ненормального выполнения аппликации в модели Б не различаются.

2.4.5. Условные выражения Маккарти

При объяснении смысла вводимых ниже примитивных функций и функциональных форм будем пользоваться модификацией так называемых условных выражений Маккарти. Условное выражение Маккарти - это запись вида

(P1 --> E1, ... ,Pn --> En, T --> E),

где через P с возможными индексами обозначены условия (предикаты, логические функции), а через E с возможными индексами - выражения, вычисляющие произвольные объекты. При конкретной интерпретации входящих в эту запись переменных ее значение вычисляется по следующему правилу. Последовательно вычисляются условия, начиная с первого, пока не найдется истинное (такое всегда найдется. Почему?). Затем вычисляется соответствующее выражение, значение которого и становится значением всего условного выражения Маккарти.

Аналоги такой конструкции широко используются в ЯП. В сущности, возможность согласования с левой частью конкретного рефал-предложения можно рассматривать как аналог условия Pi, а правую часть рефал-предложения - как аналог выражения Ei.

Упражнение. Укажите различия между Pi и Ei и названными их рефал-аналогами.

Вслед за Джоном Бэкусом будем записывать условные выражения Маккарти в виде

P1 --> E1; ... ; Pn --> En; E

т.е. опуская внешние скобки и последнее тождественно истинное условие, а также используя точку с запятой в качестве разделителя (запятая занята под знак формы "конструкция").

2.4.6. Примеры примитивных функций

Многие из приведенных ниже функций вам уже знакомы. Мы опишем их заново, во-первых, чтобы можно было освоиться с введенными обозначениями и, во-вторых, некоторые из них играют важную роль в алгебре программ (АП), к описанию и применению которой мы стремимся. Определения некоторых из ранее рассмотренных функций уточнены здесь с тем, чтобы упростить АП (соответствующая модификация М-определений может служить полезным упражнением). Отличия касаются, как правило, тонкостей (действий с пустыми и неопределенными объектами, а также учета внешних скобок в представлении объектов), однако эти тонкости существенны с точки зрения АП.

2.4.6.1. Селекторы (селекторные функции). Будем использовать целые числа для обозначения функций, выбирающих из кортежей-объектов элементы с соответствующим номером.

1 : X :: X = <X1, ... , Xn> --> X1 ; <?>.

т.е. функция определяется через аппликацию (этот прием мы уже применяли в модели М). Знак "::" используется, как и раньше, в смысле "есть по определению", чтобы отличать определение от обычного равенства. Приведенная запись означает, что применение функции 1 к объекту X дает результат X1 (первый элемент кортежа), если X - кортеж, иначе - неопределено (т.е. аппликация завершает вычисление с результатом <?>).

Вообще, для положительного целого s

s : X :: X = <X1, ... ,Xn> & n > = s --> Xs; <?>.

Здесь в условном выражении Маккарти употреблено более сложное условие вида

X = <X1, ... ,Xn> & n > = s.

Замечание. Такого рода определения (и это, и предыдущее) следует рассматривать лишь как достаточно понятное сокращение точного определения, которое может быть дано, скажем, в модели М. Модель М неплохо подходит для таких определений, потому что в ее базисе имеются мощные средства анализа и синтеза. Ведь нужно сказать следующее: если объект X имеет вид

<X1, ... , Xn>

и длина кортежа больше s, то заменить X на s-ю компоненту кортежа. Если же X не имеет такого вида ("согласование невозможно" с соответсвующим рефал-предложением), то выдать <?>.

Например:

5{ (t1 t2 t3 t4 t5 e)} --> t5 .

5{ e } --> <?> .

Конец замечания.

2.4.6.2. Хвост

t1 : X :: X = <X1> --> <>;

X = <X1, ... ,Xn> & n > = 2 --> <X2, ... ,Xn>; <?>.

Отличие от М-функции "хвост" в том, что результат всегда в скобках, если определен и не пуст.

2.4.6.3. Тождественная функция

id : X :: X

2.4.6.4. Логические функции - атом, равенство, пустой

атом : X :: (X - это атом) --> T; X /= <?> --> F; <?>.

Таким образом, атом:<?> = <?> (сохраняет неопределенность).

eq : X :: X = <Y,Z> & Y = Z --> T;
X = <Y,Z> & Y /= Z --> F; <?>.

Таким образом, скажем, eq:<2,2,3> = <?>, так как аргумент не пара, а тройка.

null : X :: X = <> --> T; X /= <?> --> F; <?>.

2.4.6.5. Сложение, вычитание, умножение, деление

\+   : X :: X = <Y,Z> & числа (Y,Z) --> Y+Z; <?>.
\-   : X :: аналогично.
mult : X :: аналогично. [Звездочка занята]
div : X :: аналогично. [Косая черта занята]

2.4.6.6. Расписать левым, расписать правым

distl : X :: X = <Y,<>> --> <>;
X = <Y,<Z1, ... , Zn>> --> <<Y,Z1>, ... , <Y,Zn>>; <?>.

distr : X :: X = <<>,Y> --> <>;
X = <<Z1, ... , Zn>,Y> --> <<Z1,Y>, ... , <Zn,Y>>; <?>.Таким образом, функция distl "расписывает" левым элементом аргумента все элементы кортежа, который служит правым элементом аргумента функции, а функция distr, наоборот, "расписывает" правым элементом аргумента. Например

distl:<<A,B>,<C,D>> = <<<A,B>,C>,<<A,B>,D>>.
distr:<<A,B>,<C,D>> = <<A,<C,D>>,<B,<C,D>>>.

2.4.6.7. Транспонирование

trans : X ::

```
 X = <<X11, ... ,X1m>,           <<X11, ... ,Xn1>,
      <X21, ... ,X2m>,   -->      <X12, ... ,Xn2>,
            ...                        ...
      <Xn1, ... ,Xnm>>            <X1m, ... , Xnm>> ; <?>.
```

Здесь для наглядности строки матрицы выписаны друг под другом. С подобной функцией мы работали в модели М. К не-матрицам она не применима (т.е. результат - <?>).

2.4.6.8. Присоединить

```
appendl : X :: X = <Y,<>> --> <Y>;
 X = <Y,<Z1, ... ,Zn>> --> <Y,Z1, ... ,Zn>; <?>.
```

Таким образом, эта функция присоединяет левый элемент аргумента в качестве самого левого элемента правой компоненты аргумента. Поэтому ее естественно назвать "присоединить левый". Например

```
appendl:<<A,B>,<C,D>> = <<A,B>,C,D>.
```

```
appendr : X :: X = <<>,Y> --> <Y>;
 X = <<Y1, ... ,Yn,Z> --> <Y1, ... ,Yn,Z>; <?>.
```

Такую функцию естественно назвать "присоединить правый". Например

```
appendr:<<A,B>,<C,D>> = <A,B,<C,D>>.
```

2.4.7. Примеры форм

Почти все определенные ниже формы уже известны по работе в модели М.

2.4.7.1. Композиция (обозначение - *).

```
(f * g) : X :: f : (g :X).
```

2.4.7.2. Конструкция (обозначение - ,).

```
(f1, ... ,fn) : X :: <f1:X, ... , fn:X>.
```

Замечание.. В отличие от М-конструкции, здесь результат - целостный объект в скобках, причем все n результатов применения каждой функции - также целостные объекты (т.е. каждая компонента общего результата явно отделена от остальных). В М-конструкции отделить результаты функций-компонент в общем случае невозможно, так как это могут быть не термы, а выражения, не разделенные между собой явно. По указанной причине для М-конструкции не выполняются некоторые важные алгебраические законы, верные в случае Б-конструкции. Так что указанные отличия М- и Б-конструкций существенны. Конец замечания.

Упражнение. Определите М-конструкцию со свойствами, аналогичными Б-конструкции.

Подсказка. Нужно использовать не конкатенацию выражений, а функцию, аналогичную созданию списка из атомов и (или) списков.

Вопрос. Какие важные алгебраические законы не выполняются для М-конструкции?

Подсказка. Что будет, если некоторый объект присоединить левым, а затем выбрать первый (т.е. левый) элемент? Зависит ли результат от структуры этого объекта в случае Б-конструкции? А в случае М-конструкции?

2.4.7.3. Условие

```
(p --> f;g):X :: (p:X) = T --> f:X; (p:X) = F --> g:X; <?>.
```

В отличие от условного выражения Маккарти, в форме "условие" все аргументы - функции, а не объекты. Вслед за Бэкусом вместо

```
(p1 --> f1; (p2 --> f2;g))
```

будем писать без скобок: p1 --> f1; p2 --> f2 ; g.

2.4.7.4. Генератор постоянных

```
const(X) : Y :: Y = <?> --> <?>; X.
```

Таким образом, аргумент этой формы - некоторый объект, а результат - функция, значение которой на всех определенных объектах совпадает с объектом, заданным в качестве аргумента формы. Другими словами - результатом формы служит функция-константа с заданным значением. Например, для любого Y /= <?>

```
const (<A,B>):Y = <A,B>.
```

2.4.7.5. Редукция (обозначение - /).

```
/f : X :: X = <X1> --> X1;
     X = <X1, ... ,Xn> & n >= 2 --> f:<X1,/f:<X2, ... ,Xn>>;
     <?>.
```

2.4.7.6. Общая аппликация (обозначение - A).

```
Af : X :: X = <> --> <>; X = <X1, ... ,Xn> -->
                               <f:X1, ... , f:Xn>; <?>.
```

Отличие от М-общей аппликации аналогичны отличиям М- и Б-конструкций.

Упражнение. Написать определение М-общей-аппликации, соответствующее приведенному Б-определению.

2.4.7.7. Итерация

```
(while p f) : X :: (p:X) = T --> (while p f):(f:X);
                   (p:X) = F --> X ; <?>.
```

Таким образом, смысл этой формы в том, что функция f в общем случае многократно применяется к объекту X до тех пор, пока не станет ложным результат применения к X логической функции (условия) p.

2.4.7.8. Специализатор (обозначение - s).

(s f X) : Y :: f:<X,Y>.

У этой формы два аргумента - бинарная операция f (функция от двухэлементных объектов) и объект X. Результат формы - унарная операция, получающаяся из f при ее специализации (иногда говорят "конкретизации") за счет отождествления первого операнда с объектом X. Например

(s + 1) : Y = +:<1,Y>   т.е. 1 + Y в обычных инфиксных обозначениях.

2.4.8. Определения

Б-определение (новой функции) - это выражение вида

DEF I :: r .

где в качестве I указано неиспользованное ранее название функции (функциональный символ), а в качестве r - функциональная форма (которая может зависеть от I - допустимы рекурсивные определения). Например

DEF last :: null * tl --> 1; last * tl.

где справа от знака "::" - форма "условие", в которую входят в качестве p композиция null * tl, в качестве f - селекторная функция "1", в качестве g - композиция last * tl.

Так что last:<A,B> = B. [Убедитесь, что это действительно верно!].

Нельзя выбирать названия новых функций так, чтобы они совпадали с уже введенными или предопределенными. Формально использование в D функционального символа всюду вне левой части определения (т.е. в качестве I) означает, что вместо него нужно подставить соответствующее r и попытаться вычислить полученную аппликацию. Если в процессе вычисления снова встретится функциональный символ, определенный в D, то снова заменить его соответствующей правой частью определения и т.д.

Ясно, что существует потенциальная возможность зациклиться. Как уже было сказано, это один из способов получить <?> в качестве результата. Конечно, в разумных рекурсивных определениях прямо или косвенно применяются условные выражения Маккарти, которые позволяют вычислить аппликацию за конечное число шагов (за счет чего?).

Б-определение, в сущности, еще одна форма, ставящая в соответствие функциям-аргументам определяемую функцию. Тем самым мы закончили описывать базис модели Б, а также основное (и единственное) средство развития - форму DEF.

2.4.9. Пример: программа вычисления факториала

Продемонстрируем сказанное на примере рекурсивной программы, по-прежнему стремясь к идеалу концептуальной ясности.

Рассмотрим задачу вычисления факториала. Конечно, наша цель - не просто запрограммировать факториал, а показать, как это делается с помощью форм. Начнем, как и раньше, с математического определения нужной функции "факториал" (обозначение - !) и снова применим пошаговую детализацию.

Обычное математическое определение:

!n равен 1, если n = 0; иначе равен n, умноженному на !(n-1).

Как такое определение переписать в стиле Бэкуса?

На первом шаге нужно выявить функции, на которые непосредственно разлагается исходная функция "факториал". Ее исходное определение не устраивает нас потому, что оно не разлагает факториал на компоненты-функции, а описывает его результат через результаты применения некоторых функций.

С учетом сказаного вернемся к исходному определению факториала. Что известно про функцию "!"? То, что она разлагается на различные составные части в

зависимости от некоторого свойства аргумента (в зависимости от его равенства нулю).

Следовательно, можно представить факториал условной формой

DEF ! :: (p --> f;g) .

где p,f и g - пока не определенные функции.

Вот и выполнен первый шаг детализации. Теперь займемся введенными функциями.

Что известно про p? Это функция, проверяющая равенство нулю исходного аргумента. Итак, p можно представить в виде композиции функции eq и некоторой пока еще не определенной функции, которая готовит для eq аргументы (точнее, формально один аргумент - кортеж из двух содержательных аргументов). Получаем

DEF p :: eq * f1 .

Что делает f1? Она ведь должна приготовить аргументы для проверки исходного аргумента на равенство другому объекту, а именно нулю. Другими словами, ее результатом должна быть пара, составленная из исходного аргумента факториала и нуля. Пару естественно получить формой "конструкция". Так что

DEF f1 :: f2,f3 .

Что делает f2? Поставляет первый аргумент пары. Но ведь это исходный аргумент без изменений! Следовательно,

DEF f2 :: id .

А что делает f3? Поставляет второй аргумент пары. Но ведь это нуль! Отлично, значит f3 - постоянная, которую естественно определить через форму const

DEF f3 :: const(0) .

Итак, функция p определена полностью. Продолжим детализацию для f и g. Что делает f? Всегда дает в результате единицу. Значит, это постоянная, которую также легко выразить через const.

DEF f :: const(1) .

А что делает g? Вычисляет произведение двух объектов, каждый из которых, как теперь уже нетрудно понять, должен доставляться своей функцией. Значит, g естественно представить композицией функций "mult" (умножить) и некоторой конструкции двух функций.

DEF g :: mult * (g1,g2) .

Очевидно, что g1 совпадает с id (почему?). А g2 представляет собой композицию определяемой функции "!" и функции g3, вычитающей единицу из исходного аргумента. Поэтому

DEF g2 :: ! * g3 .

где g3, в свою очередь предсталяется как композиция вычитания и конструкции, готовящей для этого вычитания аргумент. Позволим себе пропустить шаг подробной детализации и сразу написать

DEF g3 :: - * (id , const(1)) .

Пошаговая детализация полностью завершена. Программа в модели Б, вычисляющая факториал, написана. Метод пошаговой детализации продемонстрирован. Но, конечно, программа не обязана быть такой длинной и не все шаги обязательны. Полученные на промежуточных шагах определения можно убрать, подставив соответствующие формулы вместо обозначений функций. В нашем случае получим следующее функциональное соотношение, определяющее факториал

DEF ! :: eq0 --> const(1); mult * (id , ! * sub1)

где через eq0 переобозначена для наглядности функция p, а через sub1 - функция g3.

[Кстати, eq0 = (s eq 0). Верно ли, что sub1 = (s - 1)?].

Замечательное свойство пошаговой детализации (функциональной декомпозиции) в модели Б состоит в том, что нам ни разу не понадобилось исправлять определения ранее введенных функций. Другими словами, функциональная декомпозиция не зависит от контекста (она, как говорят, контекстно-свободна). Это важнейшее преимущество функционального программирования с точки зрения борьбы со сложностью программ.

Вопрос. За счет чего оно достигнуто?

Упражнение.. Запрограммируйте в стиле Бэкуса другое определение факториала:

Для натурального n

!n равен произведению всех различных натуральных чисел, меньших или равных n.

2.4.10. Программа перемножения матриц

Напишем в стиле Бэкуса программу, перемножающую две прямоугольные матрицы (согласованных размеров). Снова применим метод пошаговой детализации. Начнем, как обычно, с постановки задачи, т.е. с определения функции MM (matrix multiply), которую предстоит запрограммировать.

```
             Результат умножения двух матриц B1(m,n) и B2(n,k)
       (def)     - это такая матрица C(m,k), каждый элемент c(i,j)
                 которой - скалярное произведение i-ой строки
                 матрицы B1 на j-ый столбец матрицы B2.
```

До сих пор в наших примерах мы имели дело с программами, работающими с относительно просто устроенными данными. Поэтому можно было не заниматься проблемой представления данных специально. Между тем в программистском фольклоре бытует крылатая фраза, довольно точно отражающая трудоемкость и значимость отдельных аспектов программирования: "Дайте мне структуру данных, а уж программу я и сам напишу!". Мы еще не раз поговорим о данных, а пока займемся представлением исходных данных в нашей задаче, т.е. представим матрицы в виде Б-объектов (других типов данных в модели Б просто нет).

При описании представления нам потребуется говорить о длине объектов-кортежей. Поэтому введем еще одну примитивную функцию

leng : X :: X = <> --> 0; X = <X1, ... ,Xn> --> n; <?>.

Другими словами, функция leng вычисляет число непосредственных компонент объекта X (его длину как кортежа). Например

leng:<A,<B,C>,D> = 3, leng:(2:<A,<B,C>,D>) = 2 .

Представим аргумент X функции MM парой (объектом длины 2), компоненты которой представляют исходные матрицы. Так что

1:X = B1 , 2:X = B2

(здесь B1 и B2 используются как обозначения матриц, а не как Б-атомы!). При этом матрицы, в свою очередь, будут представлены кортежами строк. Например, если

```
 B1 = 3 5 7             ,            B2 = 9 5
      2 4 6                               6 3
                                          1 2
```

то X = < <<3,5,7>,<2,4,6>> , <<9,5>,<6,3>,<1,2>> >.

Так что

leng:(1:X) = 2 и это число строк в матрице B1,
leng:(1:(1:X)) = 3 и это число столбцов в матрице B1,
leng:(2:X) = 3 и это число строк в матрице B2,
leng:(1:(2:X)) = 2 и это число столбцов в матрице B2.

Итак, мы научились предсталять матрицы "по строкам" и говорить об их размерах в терминах "объектного представления". Осталось отметить, что если объект Y представляет матрицу, то ее элемент, стоящий на пересечении i-ой строки и j-го столбца, можно получить с помощью функции

j * i .

При этом элементы матрицы B1 можно получить из X функцией

j * i * 1 ,

а элементы матрицы B2 - с помощью функции

j * i * 2 .

Теперь мы готовы приступить к первому шагу функциональной декомпозиции. Как и раньше, начнем с "конца" определения (def). Там сказано, что "каждый элемент" результата - это скалярное произведение некоторых объектов. Следовательно, MM можно определить как композицию двух функций, из которых внутренняя готовит сомножители для скалярных произведений, а внешняя выполняет умножение для всех заготовленных аргументов.

DEF MM :: f2 * f1 .

Можно ли сразу же продолжить функциональную декомпозицию? На первый взгляд можно, тем более что совсем недавно шла речь о ее независимости от контекста. Но как же детализировать, скажем, f2, не зная, какова структура аргумента у этой функции? Все дело в том, что мы по существу еще не завершили предыдущего шага декомпозиции - мало обозначить компоненты функциональной формы, нужно еще описать их внешний эффект (или, как говорят, спроектировать их внешнюю спецификацию). Другими словами, нужно решить, каково у каждой из этих функций множество исходных

данных, каково множество допустимых результатов и какое именно отображение из первого множества во второе каждая из функций осуществляет.

В нашем случае для ММ все это ясно (и тем самым определены исходные данные для f1 и результаты для f2). Нужно "разделить сферы влияния" f1 и f2, выбрав представление для аргумента функции f2 (и тем самым для результата функции f1).

Вспомнив (def), нетрудно понять, что строение аргумента функции f2 должно отличаться от строения результата лишь тем, что на местах элементов матрицы C должны находиться те самые пары объектов-кортежей, из которых соответствующие c(i,j) будут получены скалярным умножением. Например, в случае наших B1 и B2 аргумент функции f2 должен представлять собой объект, соответствующий матрице размером 2 на 2, из которого функция, скажем, 2 * 1 извлекает пару <<3,5,7>,<5,3,2>>. Назовем поэтому аргумент функции f2 матрицей пар.

Вот теперь можно продолжить декомпозицию. На втором шаге займемся функцией f2. Она должна применять функцию IP (скалярное умножение) КО ВСЕМ элементам матрицы пар. Если бы в нашем распоряжении была форма, которая применяет свой аргумент-функцию ко всем элементам матрицы (т.е. кортежа кортежей), то было бы ясно, как представить f2 через эту форму и IP (кстати, как это сделать?). Но у нас есть лишь общая аппликация A (применяющая заданную ей в качестве аргумента функцию ко всем элементам кортежа). Таким образом, если ввести определение

DEF f2 :: (A f3) ,

то f2 окажется представленной через общую аппликацию с функцией f3, применяемой ко всем "строкам" матрицы пар. Осталось обеспечить, чтобы при каждом своем применении f3 применяла IP ко всем элементам "строки" (т.е. кортежа пар). Ясно, что нужная функция легко выражается через A и IP.

DEF f3 :: (A IP) .

Подставляя вместо f3 ее определение и вслед за Бэкусом убирая вторые скобки, завершим декомпозицию f2 определением

DEF f2 :: (AA IP) .

Вполне можно считать, что через AA обозначена та самая форма, которая применяет свой аргумент ко всем элементам кортежа кортежей. Ее называют двойной общей аппликацией.

На третьем шаге детализации займемся f1. Как мы выяснили, эта функция из пары исходных матриц получает матрицу пар. При этом элемент матрицы пар составлен из i-ой строки первой исходной матрицы и j-го столбца второй матрицы. Другими словами, каждая строка первой матрицы сочетается с каждым столбцом второй. Например, в случае матриц B1 и B2 функция 2 * 1 должна выбрать из матрицы пар объект <<3,5,7>,<2,4,6>>. Но раз каждая сочетается с каждым, естественно возникает идея получить матрицу пар "расписывающими" функциями - distl и distr. Однако чтобы "расписать", нужно иметь, что "расписывать". И если строки первой матрицы представлены объектами-кортежами, то объектов, представляющих столбцы второй матрицы, у нас нет - ведь матрицы представлены "по строкам"! Поэтому следует представить f1 композицией функций, внутренняя из которых располагает вторую матрицу "по столбцам", внешняя - "расписывает" матрицу пар. Итак, третий шаг детализации завершает определение

DEF f1 :: f5 * f4 .

(при этом считаем, что внешняя спецификация новых функций очевидна; кстати, понимаете ли Вы, что у них на входе и что на выходе?).

На четвертом шаге функциональной декомпозиции займемся f4. Содержательно ее назначение - подготовить из исходной новую пару матриц, сохранив ее первую компоненту и переписав "по столбцам" вторую. При уже имеющемся у нас опыте Б-программирования можно сразу выписать определение

DEF f4 :: (1,trans * 2) .

Действительно, такая конструкция составляет новую пару из первой компоненты исходной пары и транспонированной второй.

На пятом шаге рассмотрим f5. Ее назначение - по двум матрицам получить матрицу пар, сочетая каждую строку первой матрицы с каждой строкой второй. При этом i-ая строка матрицы пар (т.е. ее i-ый элемент как кортежа) представляет собой кортеж, полученный сочетанием i-ой строки первой матрицы с каждой строкой второй матрицы в естественном порядке. Значит, если удастся сначала составить кортеж пар, в которых первым элементом будет i-ая строка первой матрицы, а вторым - вся вторая матрица целиком, то затем можно каждую такую пару <строка,матрица> превратить в кортеж пар <строка,строка>.

Здесь пригодятся наши "расписывающие" функции. Действительно, кортеж пар <строка,матрица> получается применением к паре матриц функции distr ("расписать

правым" - ведь правая компонента сочетается со всеми компонентами-строками левой матрицы), а затем из каждой такой пары можно получить кортеж вида <строка,строка> применением общей аппликации с функцией distl (ведь левая компонента ее аргумента сочетается со всеми компонентами правой матрицы). Итак, декомпозицию f5 завершает определение

```
DEF f5 :: (A distl) * distr .
```

Тем самым оказывается полностью завершенным и процесс пошаговой функциональной декомпозиции функции MM. Подставляя вместо обозначений промежуточных функций их определения, получаем

Определение MM

```
DEF MM :: (AA IP) * (A distl) * distr * (1,trans * 2) .
             *******   ----------------   =============
```

Рис. 2.11

Таким образом, MM разлагается на подготовку соответствия строк первой матрицы столбцам второй (выделено двойной чертой), подготовку пар (выделено одинарной чертой) и собственно перемножение (выделено звездочками).

И опять - ничего лишнего, структура программы-формулы получена непосредственно из постановки задачи! Принцип концептуальной ясности снова действует. Конечно, сначала такая программа может показаться и не слишком понятной. Новое, тем более принципиально новое, усваивается не сразу. Однако излагаемый стиль программирования стоит затрачиваемых на его освоение усилий! Обратите внимание - нет имен для промежуточных данных, нет переменных, нет особых управляющих конструктов, нет процедур, нет инициализации (установки начальных значений).

Как видите, многого нет с точки зрения традиционного программирования в стиле фон Неймана. Зато есть концептуальная ясность, есть обобщенность (в качестве исходных данных пригодны любые согласованные по размерам матрицы; кстати, где учтено требование согласованности размеров?), есть возможность распараллеливания, есть, наконец, возможность оптимизации (так как вычисление скалярных произведений независимо, их можно вычислять и последовательно, не требуя большой памяти - эту возможность мы еще продемонстрируем)

Замечание.. Важно понять, что стиль программирования, ориентированнный на концептуальную ясность, предполагает концентрацию внимания исключительно на сути решаемой задачи при возможно более полном игнорировании таких второстепенных на этом этапе вопросов, как ресурсоемкость решения с точки зрения исполнителя. Самое главное - найти правильное и понятное (убедительно правильное, концептуально ясное) решение. Вот когда есть правильное решение, и оно не устраивает по ресурсоемкости, осмысленно тратить время и силы на постановку и решение новой задачи - искать лучшее решение. (Кстати, это тоже пошаговая детализация - детализация поиска оптимального решения. Выделить первый шаг такой детализации (на котором в сущности конструктивно доказывается теорема существования решения) принципиально важно с методологической и технологической точек зрения. Ведь на последующих шагах оптимальное решение ищется в надежных и комфортных условиях - есть куда отступать и есть с чем сравнивать.

Такой подход требует определенной психологической подготовки. Например, когда в процессе пошаговой детализации мы обнаружили, что i-ая строка понадобится для IP много раз, то с точки зрения концептуальной ясности вывод очевиден - размножить каждую строку в нужном количестве экземпляров! При другом стиле программирования такое даже в голову не придет "квалифицированному" программисту - он интуитивно игнорирует его как расточительное по ресурсам. Но именно такое решение ведет не только к ясности, но и к эффективности, когда память недорога, а процессоров сколько угодно. И во всяком случае оно открывает путь к анализу имеющегося правильного решения вместо нерационального расхода человеческих ресурсов на возможно ненужную оптимизацию или попытку воспользоваться неправильной, но показавшейся привлекательной программой.

Конец замечания.

2.5. Алгебра программ в модели Б

Наша ближайшая цель - показать, как простота (скорее регулярность, свобода от контекста) рассмотренных функциональных форм способствует построению системы регулярных преобразований, сохраняющих смысл программы (построению алгебры

программ). В этой алгебре носитель (область значений переменных) - это класс функций в модели Б, а операции - формы в модели Б. Например,

(f * g) * h

- это выражение (формула) в алгебре программ. Результат "вычисления" этого выражения различен в зависимости от значений переменных f,g,h - при конкретной интерпретации переменных это вполне определенная функция. Например, при интерпретации

(f -> id,g -> id,h -> id)

получаем

(f * g) * h = id .

В этой алгебре с программами-выражениями можно обращаться аналогично тому, как в школьной алгебре обращаются с формулами, уравнениями, неравенствами. Применять это умение можно для упрощения программ и доказательства их эквивалентности (а значит и корректности, если правильность эквивалентной программы установлена или считается очевидной).

Основой для преобразований служит серия законов. Некоторые из них мы перечислим. Например,

(f,g) * h = ((f * h),(g * h)) .

К законам этим можно относиться либо как к аксиомам, либо как к теоремам (например, когда рассматривается конкретная реализация форм, скажем, в модели М). Будем относиться к перечисленным ниже законам как к аксиомам. Лишь для одного из них в качестве примера приведем доказательство его справедливости.

2.5.1. Законы алгебры программ

2.5.1.1 (f1, ... ,fn) * g = (f1 * g, ... ,fn * g) .

2.5.1.2 A f * (g1, ... ,gn) = (f * g1, ... ,f * gn) .

Здесь "A" обозначает общую аппликацию.

2.5.1.3 /f * (g1, ... ,gn) = f * (g1,/f * (g2, ... ,gn))
для n >= 2 .

/f * <g> = g .

2.5.1.4 (f1 * 1, ... ,fn * n) * (g1, ... ,gn) =
(f1 * g1, ... ,fn * gn).

2.5.1.5 appendl * (f * g, Af * h) = Af * appendl (g,n) .

2.5.1.6 pair & not * null * 1 ->->
appendl * ((1 * 1,2), distr * (t1 * 1,2)) = distr .

2.5.1.7 A(f * g) = Af * Ag.

Теорема. pair & not * null * 1 ->->
appendl * ((1 * 1,2), distr * (tl * 1,2)) = distr .

Другими словами, равенство, которое написано справа от знака "->->", выполнено для класса объектов, удовлетворяющих условию, выписанному слева от "->->" (т.е. для пар с непустой первой компонентой).

Доказательство. Идея : слева и справа от знака равенства получается тот же результат для любого объекта из выделяемого условием класса.

Случай 1. x - атом или <?> .

distr : (x,y) = <?>. (см. опр distr)
tl * 1 : (x,y) = <?>. (опр tl).

И так как все функции сохраняют <?>, получаем утверждение теоремы.

Случай 2. x = <x1, ... ,xn> (т.е. x - кортеж). Тогда

appendl * ((1 * 1,2),distr * (tl * 1,2)) : <x,y> =
= appendl:<<1:x,y>, distr:<tl:x,y>> =

Если tl:x = <> [<> обозначает "пусто"], то

= appendl:<<x1,y>,<>> = <<x1,y>> = distr:<x,y> .

Если tl:x /= <>, то

= appendl:<<x1,y>,<<x2,y>, ... ,<xn,y>>> =
= distr:<x,y>.

Что и требовалось доказать.

2.5.2. Эквивалентность двух программ перемножения матриц

Наша ближайшая цель - продемонстрировать применение алгебры программ для доказательства эквивалентности двух нетривиальных программ. Одна из них - программа MM, при создании которой мы совершенно игнорировали проблему эффективности (в частности, расхода памяти исполнителя). Вторая - программа MMR, которую мы создадим для решения той же задачи (перемножения матриц), но позаботимся теперь о рациональном использовании памяти исполнителя. Естественно, MMR получится более запутанной. Уверенность в ее правильности (корректности) будет, конечно, более высокой, если удастся доказать ее эквивалентность MM, при создании которой именно правильность (концептуальная ясность) и была основной целью.

Естественно считать MM точной формальной спецификацией задачи перемножения матриц (т.е. определением функции "перемножение матриц"). Тогда доказательство эквивалентности некоторой программы, предназначенной для решения той же задачи, программе MM становится доказательством корректности такой программы по отношению к спецификации (доказательством того, что эта программа решает именно поставленную задачу, а не какую-нибудь другую).

Доказательство корректности программ по отношению к их (формальной) спецификации часто называют верификацией программ. Таким образом, мы продемонстрируем, в частности, применение алгебры программ для их верификации.

Замечание. Применяемый нами метод верификации принадлежит Бэкусу и не является общепринятым. Чаще, говоря о верификации, имеют в виду спецификации, написанные на логическом языке, а не на функциональном. Мы поговорим и о таких спецификациях, когда займемся так называемой дедуктивной семантикой программ. Заметим, что все последнее время мы занимались по сути денотационной семантикой - ведь программа в модели Б явно представляет свой денотат (реализуемую функцию) в виде комбинации функций.

Конец замечания.

2.5.2.1. Экономная программа R перемножения матриц.

Напомним строение программы MM.

DEF MM :: AA IP * A distl * distr * (1,trans * 2) .

Замечание. Всюду ниже для уменьшения числа скобок будем считать, что операции "A" и "/" имеют высший приоритет по сравнению с "*" и ",". При реализации в модели М этого легко добиться, поместив определения первых операций НИЖЕ в поле определений (почему ?).

Конец замечания.

Рассмотрим программу MM' такую, что

DEF MM' :: AA IP * A distl * distr .

Она "заканчивает" работу MM, начиная уже с пары матриц с транспонированной второй компонентой. Будем оптимизировать именно MM', так как в ней находится

источник неэффективности, с которым мы намерены бороться. Дело в том, что функция AA IP применима только к матрице пар, которая требует памяти объемом mb + ka, где a и b - объем соответственно первой и второй матриц, k - число столбцов второй матрицы, m - число строк первой. Хотелось бы расходовать память экономнее.

Постараемся определить другую программу (назовем ее R) так, чтобы она реализовала ту же функцию, что и MM', но экономнее расходовала память. Основная идея состоит в том, чтобы перемножение матриц представить как последовательное перемножение очередной строки первой матрицы на вторую матрицу. Тогда можно перемножать последующие строки НА МЕСТЕ, освободившемся после завершения перемножения предыдущих строк, так что потребуется лишь объем памяти, сравнимый с размером перемножаемых матриц.

Определим вначале программу mM, выполняющую перемножение первой строки первой матрицы на вторую матрицу.

```
DEF mM :: A IP * distl * (1 * 1,2) .
```

Действительно, программа mM сначала "расписывает" строки второй матрицы первой строкой первой матрицы (функция distl), а затем скалярно перемножает получившиеся пары строк, что и требовалось.

Теперь определим программу R.

```
DEF R :: null * 1 --> const(<>);
            appendl * (mM, MM' * (t1 * 1,2)) .
```

Таким образом, если первая матрица не пустая, то результат функции R получается соединением в один объект (с помощью appendl) результата функции mM (она перемножает первую строку первой матрицы на вторую матрицу) и результата функции MM', которая "хвост" первой матрицы перемножает на вторую матрицу.

Заметим, что если удастся доказать эквивалентность MM' и R, то MM' можно заменить на R и в определении самой R. Так что определение R через MM' можно считать техническим приемом, облегчающим доказательство (не придется заниматься рекурсией). Перепишем определение R без MM'.

```
DEF R :: null * 1 --> const (<>);
           appendl * (mM, R * (t1 * 1, 2)) .
```

Независимо от того, удастся ли доказать эквивалентность R и MM', ясно, что в новом определении R отсутствует двойная общая аппликация и, если вычислять R разумно (как подсказывает внешняя конструкция, т.е. сначала вычислить левый ее операнд, а затем правый), то последовательные строки матрицы-результата можно вычислять на одном и том же рабочем пространстве. Этого нам и хотелось!

2.5.2.2. Верификация программы R

Итак, сосредоточившись на сути задачи, мы выписали ее спецификацию-функцию (т.е. программу MM'), а концентрируясь на экономии памяти, получили оптимизированный вариант программы (т.е. R).

Займемся верификацией программы R, т.е. покажем, что

R = MM'

для всех аргументов, которые нас интересуют, т.е. для всех пар. Есть различные способы такого доказательства. "Изюминка" модели Бэкуса состоит в том, что для доказательства свойств программ и, в частности, их верификации, можно пользоваться общими алгебраическими законами (соотношениями), справедливыми в этой модели, причем для установления и применения этих законов не нужно вводить никакого нового аппарата - все, что нужно, уже определено в модели Б.

Докажем, что верна следующая

Теорема. pair ->-> MM' = R .

Доказательство.

Случай 1: pair & null * 1 ->-> MM' = R .

```
pair & (null * 1) ->-> R = const(<>) .  [По определению R]
pair & (null * 1) ->-> MM' = const(<>) .
              [Так как distr:<<>,X> = <> по определению distr
                     и   A f:<> = <>        по определению A]
```

Следовательно, MM' = R.

Случай 2. (основной).

pair & (not * null * 1) ->-> MM' = R .

Ясно, что в этом случае R = R'', где

```
DEF R'' :: appendl * (mM, MM' * (t1 * 1,2))
```

[по определению формы "условие"].

Расписывая mM, получаем

Формула для R''

```
 R'' = appendl * (A IP * distl * (1 * 1,2),
           ---- f -----    -- g --
AA IP * A distl * distr * (t1 * 1,2)).
----- Af ------   ------- h --------
```

Рис. 2.12

Так как A (A IP * distl) = AA IP * A distl
[по закону (10.7)]

то R'' имеет вид

appendl * (f * g , A f * h)

для указанных под определением R''функций f, g и h.

Поэтому по закону (10.5)

R'' = A f * appendl * (g, h) =

= A f * appendl * ((1 * 1 , 2) , distr * (t1 * 1 , 2))

что по закону (10.6) дает

A f * distr

т.е. MM'.

Теорема доказана.

Каждый из трех использованных в доказательстве общих законов очевиден (следует непосредственно из определения соответствующих функций) и может быть обоснован аналогично закону (10.6).

2.6. Краткий итог по трем моделям

Итак, мы рассмотрели три модели языка программирования (в основном, с технологической позиции; т.е. с точки зрения программиста). Модель Н - самая близкая к языкам программирования, получившим наибольшее распространение (неймановским ЯП). Вместе с тем она проявляет основной источник сложности программирования на этих языках - неразработанность средств функциональной декомпозиции. Модель М демонстрирует один из перспективных подходов к созданию таких средств - она предоставляет мощные средства развития, основанные на выделении двух ключевых абстракций (анализа и синтеза текстов посредством образцов).

Модель Б указывает путь к рациональному программированию, ключевые идеи которого опираются на многовековой опыт применения алгебраических формул. С другой стороны, пока эта модель - из рассмотренных нами самая далекая от современного массового программирования (однако она находится в центре внимания создателей перспективных языков и машин).

Таким образом, ЯП можно строить на весьма различных базисах и различных средствах развития. У каждого - свои достоинства и свои недостатки. В качестве достоинства модели Бэкуса продемонстрирована возможность доказательного программирования на основе принципа концептуальной ясности и законов алгебры программ. Следующий раздел - еще об одном подходе к доказательному программированию, а также о требованиях, которые он предъявляет к ЯП.

3. МЕТОД ХОАРА

На примере метода Бэкуса видно, как подход к доказательному программированию связан со свойствами языка, к которому этот метод применяется (модель Б специально строилась так, чтобы были справедливы достаточно простые законы алгебры программ). Метод Хоара, к изложению которого мы приступаем, также ориентирован на определенный класс языков. Эти языки ближе к традиционным (во всяком случае в них имеются переменные и присваивания). Характерное ограничение состоит в том, что динамическая структура программы в них должна быть хорошо согласована со статической ее структурой. Другими словами, составить представление о процессе выполнения программы должно быть относительно легко по ее тексту. (Еще иначе о том же можно сказать так: в этих языках по структуре знака относительно легко судить о структуре денотата). Указанное согласование и позволяет реализовать основную идею Тони Хоара - ввести так называемую дедуктивную семантику языка, связывающую программные конструкты непосредственно с утверждениями о значениях программных

переменных. Мы рассмотрим метод Хоара на примере конкретного языка и конкретной программы в этом языке.

3.1. Язык Дейк

В качестве представителя рассматриваемого класса языков выберем очень простой язык, который назовем "Дейк" (он очень похож на язык, использованный Дейкстрой в его книге "Дисциплина программирования" - М., Мир, 1978; читать ее доставляет огромное удовольствие).

Начнем с описания синтаксиса языка Дейк с помощью несколько модифицированной БНФ. И сам синтаксис, и применяемую модификацию БНФ (она соответствует предложениям Вирта и принята в качестве английского национального стандарта) удобно объяснять в процессе пошаговой детализации исходного синтаксического понятия "программа". На каждом шаге детализации при этом указываются все допустимые конкретизации промежуточных синтаксических понятий (абстракций). В зависимости от интерпретации применяемые при этом правила конкретизации можно считать или соотношениями, связывающими классы текстов (так называемые формальные языки), или правила вывода в некоторой порождающей системе (а именно, в контекстно-свободной грамматике). Ниже выписаны 16 шагов такой детализации (т.е. 16 правил модифицированной БНФ).

3.1.1. Синтаксис языка Дейк

```
программа    = 'begin' { объявление ';' }{ оператор ';' } 'end'.
объявление   = ( 'var' ! 'arr' ) имя { ',' имя } .
оператор     = присваивание ! развилка ! цикл ! 'null' .
присваивание = переменная ':=' выражение .
переменная   = имя [индекс] .
индекс       = '[' выражение ']' .
выражение    = переменная ! число ! функция .
функция      = имя '(' выражение { ',' выражение } ')' .
развилка     = 'if' { вариант } 'fi' .
цикл         = 'do' { вариант } 'od' .
вариант      = условие '-->' { оператор ';' } .
условие      = выражение (< ! <= ! = ! /= ! >= ! > ) выражение .
имя          = буква { буква ! цифра } .
число        = цифра { цифра } .
буква        = 'a' ! 'b' ! ... ! 'z' .
цифра        = '0' ! '1' ! ... ! '9' .
```

Отличия от оригинальной БНФ сводятся, во-первых, к тому, что выделяются не названия синтаксических понятий (метасимволы), а символы так называемого терминального алфавита (т.е. алфавита, из символов которого строятся программы в описываемом языке). В языке Дейк терминальный алфавит состоит из букв, цифр и символов 'begin', 'end', 'var', 'arr', 'do', 'od', 'if', 'fi', 'null', и некоторых других (скобок, знаков отношений и т.п.). В неясных случаях для выделения терминальных символов применяется апостроф. Во-вторых, применяются круглые, квадратные и фигурные скобки. Круглые - чтобы сгруппировать несколько вариантов конкретизации понятия (несколько альтернатив). Квадратные - чтобы указать на возможность опускать их содержимое. Фигурные - чтобы указать на возможность выписывать их содержимое нуль и более раз подряд в процессе пошаговой детализации (или. как говорят, порождения) конкретной программы.

Некоторый текст признается допустимой программой на языке Дейк тогда и только тогда, когда его можно получить последовательной конкретизацией (т.е. породить) по указанным 16 правилам из исходной абстракции "программа". Такой текст называется выводимым из метасимвола "программа".

Например, текст

```
'begin' 'var' x, i, n ;
   x := M[1] ; i := 1 ;
   'do' i < n --> i := plus(i,1) ;
                 'if' M[i] > x --> x := M[i] ;
                      M[i] <= x --> 'null' ; 'fi' ;
   'od' ;
 'end'
```

допустим в языке Дейк, а если вместо 'plus(i,1) написать "i+1", то получится недопустимый текст (так как "выражение" может быть только "переменной", "числом" или "функцией"). Однако мы позволим себе для наглядности писать "i+1".

3.1.2. Семантика языка Дейк

Поясненим только смысл развилки и цикла. Смысл остальных конструктов традиционен. Для наших целей достаточно интуитивного представления о нем.

Начнем с так называемой операционной семантики развилки и цикла. Другими словами, поясним, как эти конструкты выполняются.

Назовем состоянием некоторое отображение переменных программы в их значения. Это отображение частичное, потому что значения некоторых переменных могут быть неопределенными. Вот пример состояния выписанной выше программы (Она вычисляет максимум M[i])

```
< n --> 5, M --> (2,5,6,8,1), x --> 2, i --> 1 > .
```

Рассмотрим развилку S вида

```
'if'
  P1 --> S1 ,
   ...
  Pn --> Sn
'fi' .
```

Полезно учитывать, что смысл конструктов языка Дейк специально подобран так, чтобы было легче доказывать свойства написанных на нем программ. Как доказывать, рассмотрим в следующем разделе, а пока объясним семантику развилки S.

Пусть S начинает выполняться в некотором состоянии W. Сначала в состоянии W асинхронно (независимо, возможно и параллельно) вычисляются все Pi. Каждое из них либо нормально завершается и дает истину или ложь, либо завершается отказом (в частности, зацикливанием). Если хотя бы одно Pi дает отказ, то S завершается отказом. Если все Pi нормально завершаются (состояние W при этом не меняется!), то случайным образом выбирается Si0 - одно из тех и только тех Si, для которых Pi истинно. Результат выполнения этого Si0 в состоянии W - это и есть результат выполнения всей развилки S. Если же все Pi дают ложь, то S завершается отказом.

Рассмотрим цикл S вида

```
'do'
  P1 --> S1 ,
   ...
  Pn --> Sn
'od' .
```

Выполнение цикла отличается от развилки тем, что если все Pi дают ложь, то S нормально завершается с состоянием W (т.е. его действие в этом случае равносильно пустому оператору). Когда же выбирается Si0, то после его нормального завершения в некотором состоянии Wi0 цикл S снова выполняется в состоянии Wi0. Другими словами, он выполняется до тех пор, пока все Pi не дадут ложь или не возникнет отказ (в последнем случае и весь S завершится отказом).

Вот и все, что нужно знать об (операционной) семантике языка Дейк, чтобы воспринимать еще один подход к доказательному программированию.

Как видите, семантика языка Дейк очень "однородная", она симметрична относительно различных вариантов составных конструктов. Это помогает рассуждать о программах, написанных на таком языке.

3.2. Дедуктивная семантика

Мы уже не раз упоминали о том, что с одними и теми же текстами (сообщениями) можно связывать различный смысл в зависимости от роли, которую они играют при решении конкретных задач. Это справедливо и для текстов программ, написанных на языке программирования. С точки зрения исполнителя, будь в этом качестве человек или машина, смыслом программы естественно считать предписываемую этой программой последовательность действий исполнителя. Правила, сопоставляющие программе последовательность действий (в общем случае - класс последовательностей,

соответствующих классу аргументов) - это "исполнительская" семантика. Обычно ее называют операционной - установившийся, но не слишком удачный термин, "калька" английского operational.

Замечание.. Обычно предполагается, что потенциальный исполнитель программы цели своих действий не знает и не должен знать . Если считать, что уровень интеллекта связан со способностью соотносить планируемые (выполняемые) действия с поставленными целями, то потенциальный исполнитель программы начисто лишен интеллекта, абсолютно туп. Однако эта тупость обусловлена такими его потенциальными достоинствами, как универсальность, определенность и быстродействие. Компьютер способен на все, так как "не ведает, что творит" и делает, что "прикажут"; ему "не приходит в голову" действовать по собственной воле, он не тратит ресурсов на оценку разумности программы, а просто выполняет ее! С такой точки зрения "тупая" операционная семантика вполне оправдана и весьма полезна - она создает прочную основу для взаимодействия авторов, пользователей и реализаторов ЯП.

Было время, когда только операционную семантику и связывали с ЯП. Да и сейчас, когда говорят или пишут о семантике ЯП, чаще всего имеют в виду именно ее (вспомните известные вам описания Алгола, Фортрана, Бейсика и др.). Знакомя с языком Дейк, мы также начали с его операционной семантики.

Конец замечания.

Станем на точку зрения пользователя языка. Нормальный пользователь интеллектом, конечно, обладает и для него естественно соотносить свои действия со своими целями. Поэтому "исполнительской" семантики ему, как правило, недостаточно. Ему нужна более "интеллектуальная" семантика, лучше помогающая судить о той роли, которую программа в состоянии играть при решении его задач. Операционная семантика обычных ЯП уводит в дебри мелких действий вместо того, чтобы предоставить интегральную (цельную) характеристику связи аргументов и результатов программы.

Форм, в которых можно давать такую характеристику, может быть много. С каждой из них связан свой способ приписывать программе смысл, своя семантика. Вы уже знакомы с таким способом, когда программе сопоставляется отображение ее аргументов в результаты - это денотационная семантика. Мы займемся еще одной, дедуктивной семантикой (иногда ее называют аксиоматической или логической).

Если операционная семантика предназначена в основном для того, чтобы четко зафиксировать правила поведения исполнителя (тем самым и выразительные возможности пользователя), то дедуктивная семантика предназначена в основном для того, чтобы четко зафиксировать правила поведения пользователя при доказательстве свойств программ. Наиболее интересное из таких свойств - свойство давать определенные результаты при определенных аргументах.

Уточним несколько туманный смысл слова "определенный". Как и прежде, будем называть состоянием программы отображение переменных программы в их значения. Так что состояние меняется при каждом изменении значения какой-либо переменной.

Говорят, что программа P частично корректна относительно предусловия Q и постусловия R, если для всякого начального состояния, удовлетворяющего условию Q, заключительное состояние удовлетворяет условию R.

Тот факт, что программа P частично корректна, можно записать с помощью специального обозначения - так называемой тройки Хоара

{Q} P {R}

где в скобках явно выписываются соответствующие пред- и постусловия. Корректность называется "частичной" потому, что не гарантируется попадание в заключительное состояние - в общем случае возможен отказ или зацикливание.

Свойство полной корректности записывается обычно тройкой Хоара с квадратными скобками

[Q] P [R]

что означает: "начав с состояния, удовлетворяющего предусловию Q, программа P всегда завершает работу, причем в состоянии, удовлетворяющем постусловию R".

Дедуктивная семантика - это правила сопоставления каждой программе множества ее троек Хоара. Следуя Хоару, эти правила представляют обычно в виде логического исчисления (т.е. совокупности аксиом и правил вывода), в котором кроме общих аксиом и правил вывода (исчисления предикатов первого порядка) имеются и правила вывода троек Хоара (свои для каждого ЯП). Тем самым каждой программе в ЯП ставится в соответствие ее "дедуктивный" смысл - множество формально выводимых в этом исчислении троек Хоара. Если такая дедуктивная семантика согласована с операционной семантикой ЯП так, что выводимость тройки гарантирует ее истинность, то в распоряжении пользователя оказывается аппарат формального доказательства

свойств программ на рассматриваемом ЯП; например, доказательства их частичной или полной корректности.

3.3. Компоненты исчисления Хоара

Посмотрим, что же может понадобиться для построения и практического применения дедуктивной семантики ЯП (на примере языка Дейк и программы вычисления максимального значения элементов одномерного массива, рассмотренной с теми же целями в [1].

Во-первых, нужно уметь выражать условия на состояния программы. Для этой цели нам послужат обычные логические формулы, в которых в качестве предметных переменных допустимы обозначения объектов программы.

Другими словами, условия на состояния мы намерены выражать на логическом языке первого порядка. При такой договоренности становится возможным выразить следующее свойство программы для вычисления максимума (свойство ее частичной корректности)

```
          { n >= 1 }
          x := M[1] ; i := 1 ;
          'do' i < n --> i := i + 1 ;
    (Y1)                    'if' M[i] > x --> x := M[i] ;
                                 M[i] <= x --> 'null'   'fi' ;
               'od' ;
               { x = max (M,n) }      .
```

Оно говорит о том, что если запустить записанную между пред- и постусловиями программу при n >= 1, то после ее завершения будет истинным условие x = max (M,n), т.е. значение переменной x будет равно максимальному элементу массива M при изменении индекса от 1 до n.

При этом отнюдь не утверждается, что программа действительно завершит свою работу (ведь выписано условие частичной, а не полной корректности!).

Во-вторых, нужно от свойств одних фрагментов программы уметь переходить к свойствам других фрагментов (в частности, соседних или вложенных). Ясно, что это удобно делать далеко не для каждого ЯП. Кажется интуитивно понятным, что следить за свойствами состояния программы относительно легко, если динамика ее исполнения согласована со структурой ее текста. Другими словами, если об исполнении программы удобно рассуждать, "читая" ее текст последовательно, без скачков и разрывов. Этот принцип согласования статической и динамической структуры программы положен в основу структурного программирования. Язык Дейк удовлетворяет этому принципу.

В-третьих, чтобы говорить о свойствах фрагментов программы, нужно уметь их обозначать. Поэтому в тройках Хоара следует допустить не только полные программы, но и фрагменты программ. Кроме того, полезно 'разметить' программу, чтобы было удобно ссылаться на отдельные ее точки. Для этого будем пользоваться номерами в круглых скобках. Размеченный фрагмент программы, вычисляющей максимум, примет вид

```
     (1)  x := M[1] ;  (2)  i := 1 ;    (3)
          'do' i < n -->   (4)  i := i + 1 ;  (5)
                     'if' M[i] > x -->  (6)  x := M[i] ;  (7)
                          M[i] <= x --> (8)  'null' (9) 'fi' ;  (10)
               'od' ;  (11)
```

Весь этот фрагмент теперь можно обозначить как Ф(1-11) или даже . просто (1-11). Условие на состояние программы в точке t обозначим через q(t). Так что предусловие для Ф(1-11) получит обозначение q(1), а постусловие q(11). Для обозначения тождества условий будем применять двойное двоеточие. Так что, скажем

q(11) :: { x = max(M,n) }   .

Наконец, в четвертых (самое важное), для каждого языкового конструкта нужно сформулировать правила вывода соответствующих троек Хоара. Эти правила естественно называть дедуктивной семантикой конструкта, а их совокупность для всего языка - дедуктивной семантикой языка. Вскоре мы построим такую семантику для языка Дейк, а пока - несколько предварительных замечаний.

Как уже сказано, для каждого конструкта языка Дейк нужно сформулировать правило вывода допустимых троек Хоара. Тройки Хоара абсолютны в том смысле, что их истинность не зависит от контекста фрагмента, входящего в тройку (почему?). Однако выводить тройки Хоара удобно с помощью условий, характеризующих состояния

программы в отдельных точках. Такие "точечные" условия обычно относительны в том смысле, что их истинность (и выводимость) зависят от других точечных условий.

Процесс вывода тройки Хоара, выражающей свойство корректности некоторой программы P, можно представлять себе как вывод последовательных точечных условий, начиная с крайних - предусловия и постусловия. При этом тройка считается выведенной, если удастся вывести соответствующее точечное условие на противоположном конце языкового конструкта (в частности, всей программы).

Переход от предшествующего точечного условия к последующему, относящемуся к другому концу некоторого фрагмента программы, полезно представлять себе как "логическое преодоление" этого фрагмента. Поэтому правила вывода точечных условий и троек назовем правилами преодоления конструктов.

Содержательно правила преодоления выражают своеобразные законы "символического выполнения" программы. Точнее, это законы последовательного преобразования предикатов (условий), характеризующих состояния программы. Фрагмент программы при таком подходе естественно считать преобразователем предикатов.

Это особенно наглядно при преодолении фрагментов в естественном порядке, слева направо. Однако и движение в противоположном направлении может оказаться плодотворным, особенно если и сама программа, и доказательство ее корректности создаются одновременно. Ведь постусловие выражает цель работы фрагмента программы. Поэтому преодоление завершающего фрагмента программы автоматически формулирует цель для предшествующего фрагмента и т.д., пока не удастся получить условие, выполненное в исходном состоянии. Другими словами, конструирование программы можно отождествить с поиском преобразователя целевого предиката (постусловия) в исходный предикат (предусловие).

В следующем пункте займемся последовательным преодолением уже готового фрагмента программы слева направо (чтобы сосредоточить внимание на сущности и приемах доказательства свойств программ).

3.4. Правила преодоления конструктов ЯП Дейк

Наша цель - научиться выводить постусловие из предусловия, последовательно преодолевая фрагменты программы. При этом нас будут интересовать не любые фрагменты, а только достаточно крупные языковые конструкты, осмысленные с точки зрения дедуктивной семантики.

Замечание. Это означает, в частности, что верить нашему доказательству нужно как раз "по модулю" доверия к правильности связи между дедуктивной и операционной семантиками преодолеваемых конструктов. Обычно предполагается, что эта связь тщательно проверена. Впрочем, польза от доказательства не столько в гарантии правильности (в общем случае, конечно, не обеспечиваемой; почему?), сколько в систематическом исследовании программы с нетривиальной точки зрения. Такое исследование, безусловно, способствует нахождению в программе ошибок.

Конец замечания.

Преодолеть конструкт - это значит отразить в постусловии влияние выполнения этого конструкта на состояние, т.е. на переменные программы. Состояние после выполнения конструкта в общем случае зависит от состояния перед его выполнением, а также от категории и строения конструкта. Поэтому вид постусловия нужно связать с видом предусловия, видом преодолеваемого конструкта и, если нужно, с тройками, характеризующими вложенные конструкты.

3.4.1. Дедуктивная семантика присваивания

Начнем, например, преодолевать Ф(1-2), т.е. по q(1) попытаемся построить разумное q(2). Каждый легко напишет

```
q(2) :: ( n >= 1 & x = M[1] )
```

Но это написано для конкретного предусловия и конкретного оператора присваивания. Как же обобщить и формализовать прием преодоления присваивания, примененный нами только что интуитивно?

Ясно, что мы учли результат выполнения конкретного оператора присваивания Ф(1-2) над конкретными объектами программы x и M[1]. Другими словами, учли операционную семантику присваивания. Результат выполнения состоит в том, что знак x после выполнения оператора присваивания начинает обозначать то же самое, что до его выполнения обозначал знак M[1]. Другими словами, денотат знака M[1] становится денотатом знака x. Итак, если нам до выполнения присваивания что-то известно про

денотат знака M[1], то после выполнения присваивания то же самое можно утверждать и про денотат знака x. Это и есть основная идея описания дедуктивной
семантики оператора присваивания:
всякое утверждение про значение выражения e в операторе вида

v := e

остается верным после выполнения этого оператора по отношению к
значению переменной v.

Осталось придумать, как формализовать эту идею в виде правила преобразования логических формул.

Итак, наша задача - перейти от утверждения про e к утверждению про v, причем первое справедливо до присваивания, второе - после. Присваивание меняет значение v (и значение выражения e, если v в него входит). Поэтому в общем случае предусловие само по себе не годится в качестве постусловия. Нужно, во-первых, найти то, что сохраняется при присваивании (найти его инвариант) и во-вторых, отразить действие присваивания на объекты программы.

Ясно, что таким инвариантом служит всякое утверждение B про старое значение выражения e. Если B истинно до присваивания, то останется истинным и после - ведь старое значение выражения e не меняется. Но форма утверждения B должна быть такой, чтобы и после преодоления присваивания B оставалось утверждением именно про старое значение выражения e. Ведь если записать его просто в форме B(e), то оно может после присваивания стать ложным - у выражения e может оказаться новое значение (каким образом?).

Поэтому обозначим (на метауровне, т.е. в наших рассуждениях о свойствах программы) СТАРОЕ значение выражения e незанятой буквой, скажем Y, и выразим предусловие в форме

(Y = e) => B(Y)

т.е. (Y = e) влечет B(Y). При такой форме записи предусловия в нем явно выделена инвариантная часть B(Y). Для аккуратности потребуем, чтобы переменная v не входила в утверждение B. Теперь можно быть уверенными, что B(Y) действительно не зависит от возможного изменения значения переменной v.

Теперь легко написать, что можно утверждать после присваивания. Ведь старый денотат выражения e стал новым денотатом переменной v! Значит, утверждение B останется истинным, если в него вместо знака Y подставить знак v.

Получаем правило вывода

```
      (Y = e) = > B(Y)
     -----------------
   B( v --> Y )
```

где под горизонтальной чертой изображен результат подстановки знака v в утверждение B вместо всех вхождений знака Y.

Итак, преодоление присваивания состоит из двух шагов, первый из которых содержательный (творческий), а второй - формальный. На первом нужно найти инвариант B, характеризующий старое значение выражения e (причем в него не должен входить знак v !). На втором шаге можно формально применить правило вывода постусловия.

Замечание. Для предусловий, не содержащих v, тривиальное правило преодоления присваивания состоит в простом переписывании предусловия в качестве постусловия. Подобные правила полезно применять при доказательстве таких свойств программ, на которые преодолеваемые операторы не могут повлиять в принципе. Однако интереснее, конечно, правило преодоления, существенно учитывающее операционную семантику преодолеваемого оператора. Такими правилами мы и занимаемся.

Конец замечания.

3.4.1.1. Применение правила преодоления присваивания

Попытаемся двинуться по нашей программе-примеру, стараясь преодолеть оператор (1-2) и получить "интуитивно" написанное нами постусловие q(2) теперь уже формально.

Во-первых, нужно подобрать подходящее B. Как уже было объяснено, это задача творческая. Зная q(2), можно догадаться, что B должно иметь вид

{ n >= 1 & Y = M[1] }.

Замечание. Вот так знание желательного постусловия (по существу, знание цели выполняемых действий) помогает формально преодолевать конструкты программы.

Конец замечания.

Нам нужно вывести из q(1) обычными логическими средствами предусловие для преодоления оператора (1-2). Другими словами, подготовить предусловие для формального преодоления оператора.

Предусловие должно иметь вид

(Y = M[1]) ==> (n >= 1 & Y = M[1])

Оно очевидно следует из n >= 1. Формально следует применить схему аксиом

(A => (C => A & C)

и правило вывода модус поненс

```
A, A => B
---------
    B
```

Подставляя вместо A утверждение (n >= 1), а вместо C - утверждение (Y=M[1]), получаем нужное предусловие. Итак, все готово для формального преодоления фрагмента (1-2) с помощью правила преодоления присваивания.

Совершенно аналогично нетрудно преодолеть и фрагмент (2-3), получив

q(3) :: (n >= 1 & x = M[1] & i = 1).

Замечание. Нетривиальность первого из этапов преодоления оператора присваивания подчеркивает принципиальное отличие дедуктивной семантики от операционной. Дедуктиная семантика не предписывает, а разрешает. Она выделяет законные способы преодоления конструктов, но не фиксирует жестко связь предусловия с постусловием. Зато она позволяет преодолевать один и тот же оператор по-разному, выводя разные постусловия в зависимости от потребностей того, кто выясняет (или доказывает) свойства программы. Можете ли Вы привести пример постусловия для (2-3), отличного от q(3)?

Конец замечания.

Перепишем наше правило преодоления присваивания, обозначив через L предусловие, а через R - постусловие.

```
            L :: (Y = e ) => B(Y)
П(1)       ---------------------
             R :: B( v --> Y)              .
```

Чтобы преодолеть конструкт (3-8), нужно запастись терпением и предварительно разобраться с дедуктивной семантикой остальных конструктов языка Дейк.

3.4.2. Дедуктивная семантика развилки

Рассмотрим развилку S вида

```
"if"
   P1 --> S1   ,
     ...
     ...
   Pn --> Sn
"fi"          .
```

Наша задача - формализовать для нее правила преодоления. Вспомним, что по смыслу (операционной семантике) развилки каждая ее i-ая ветвь Si выполняется только когда истинен соответствующий предохранитель Pi, причем завершение Si означает завершение всей развилки S. Так как по определению постусловия оно должно быть истинным после выполнения любой ветви, получаем следующее естественное правило преодоления, сводящее преодоление S к преодолению ее ветвей.

```
           Ak : {L & Pk} Sk {R}
(П2)       --------------------
                     R
```

где L и R соответственно пред- и постусловия для S.

Таким образом, преодоление развилки следует осуществлять разбором случаев, подбирая такое R, чтобы оно было истинным в каждом из них. Очень часто R представляет собой просто дизъюнкцию постусловий R1v...vRn для операторов S1,...,Sn соответственно. Подчеркнем, что преодоление развилки невозможно, если не выполнено ни одно условие Ri.

3.4.3. Дедуктивная семантика точки

Поскольку наша цель - научиться формально преобразовывать утверждения о программе в соответствии с ее операционной семантикой, то естественно считать допустимой эамену утверждения, привязанного к некоторой точке программы, любым его чисто логическим следствием, привязанным к той же точке. Для единообразия можно считать точку пустым фрагментом (фрагментом нулевой длины), а произвольное чисто

логическое правило вывода - правилом преодоления пустого фрагмента. Применение таких правил очень важно - с их помощью готовят преодоление непустых конструктов программы (мы уже действовали таким способом при преодолении фрагмента (1-2). Таким образом, дедуктивная семантика точки совпадает с дедуктивной семантикой пустого фрагмента. Такова же и дедуктивная семантика оператора "null".

3.4.4. Дедуктивная семантика цикла

Рассмотрим цикл вида

```
"do"
   P1 --> S1,
     ...
     ...
   Pn --> Sn
"od"              .
```

Задача - сформулировать правило его преодоления. Вспомним операционную семантику этого оператора. Он завершает исполнение тогда и только тогда, когда истинно "not" P1 & ... & "not" Pn. Обозначим эту конъюнкцию отрицаний через P и немного порассуждаем о природе циклов.

Циклы - важнейшее средство для описания потенциально неограниченной совокупнисти действий ограниченными по длине предписаниями. Таким средством удается пользоваться в содержательных задачах только за счет того, что у всех повторений цикла обнаруживается некоторое общее свойство, инвариант цикла, не меняющийся от повторения к повторению.

В языке Дейк выполнение циклов состоит только из повторений тела цикла, поэтому инвариант цикла должен характеризовать состояние программы как непосредственно перед началом работы цикла, так и сразу по его завершении.

Обозначим инвариант цикла через I. Естественно, у одного цикла много различных инвариантов (почему?). Тем не менее основную идею цикла, отражающую его роль в конкретной программе, обычно удается выразить достаточно полным инвариантом I и условием завершения P. Условие P отражает достижение цели цикла, а конъюнкция I & P - свойство состояния программы, достигнутого к моменту завершения цикла. Значит, это и есть постусловие для цикла S. А предусловием служит, конечно, инвариант I - ведь он потому так называется, что истинен как непосредственно перед циклом, так и непосредственно после каждого исполнения тела цикла. Осталось выразить сказанное формальным правилом преодоления

```
                         I
(П3)                  -------
                       I & P          .
```

Это изящное правило обладает тем недостатком, что в нем формально не отражена способность утверждения I служить инвариантом цикла. Нужно еще явно потребовать его истинности после каждого исполнения тела (или, что то же самое, после исполнения каждой ветви). Получаем следующее развернутое правило преодоления

```
                 Ak : {I & Pk} Sk {I}
(П4)             --------------------
                       I & P          .
```

Другими словами, если утверждение I служит инвариантом цикла, т.е. для каждого Pk сохраняется при выполнении k-ой ветви цикла, то результатом преодоления всего цикла может служить постусловие I & P.

Скоро мы продолжим движение по нашей программе с использованием инвариантов цикла. Но прежде завершим построение дедуктивной семантики языка Дейк.

3.4.5. От точечных условий к тройкам

Нетрудно заметить, что как в правиле (П2), так и в (П4) предусловиями служат не точечные условия, а тройки Хоара. Поэтому требуется формальное правило перехода от точечных условий к тройкам. Оно довольно очевидно. В сущности именно его мы имели в виду, объясняя саму идею преодоления фрагментов. Зафиксируем некоторый фрагмент Ф и обозначим через L(Ф) некоторое точечное условие для его левого конца, а через R(Ф) - некоторое точечное условие для его правого конца. Через "!==>" обозначим отношение выводимости с помощью наших правил преодоления. Получим

```
              L(Ф) !==> R(Ф)
(П5)          --------------
                {L} Ф {R}          .
```

Замечание. Может показаться, что это правило не совсем естественное и следовало бы ограничиться только правильными языковыми конструктами, а не заниматься любыми фрагментами. Действительно, достаточно применять это правило только для присваиваний, ветвлений, циклов и последовательностей операторов. Но верно оно и в том общем виде, в котором приведено (почему?). При этом недостаточно, чтобы точка привязки утверждения L текстуально предшествовала точке привязки R. Нужна именно выводимость в нашем исчислении (почему?).

Конец замечания.

Итак, мы завершили построение исчисления, фиксирующего дедуктивную семантику языка Дейк.

3.5. Применение дедуктивной семантики

Теперь мы полностью готовы к дальнейшему движению по нашей программе-примеру. Предстоит преодолеть цикл (3-11), исходя из предусловия q(3) и имея целью утверждение q(11). Подчеркнем в очередной раз, как важно понимать цель преодоления конструктов (легко, скажем, преодолеть наш цикл, получив постусловие n >= 1, но нам-то хотелось бы q(11) !).

Правило преодоления цикла требует инварианта. Но нам годится не любой инвариант, а только такой, который позволил бы в конечном итоге вывести q(11). Интуитивно ясно, что он должен быть в некотором смысле оптимальным - с одной стороны, выводимым из q(3), а с другой - позволяющим вывести q(11). Обычная эвристика при поисках такого инварианта - постараться полностью выразить в нем основную содержательную идею рассматриваемого цикла.

Замечание. Важно понимать, что разумные циклы преобразуют хотя бы некоторые объекты программы. Поэтому инвариант должен зависеть от переменных (принимающих, естественно, разные значения в процессе выполнения цикла). Однако должно оставаться неизменным фиксируемое инвариатом соотношение между этими значениями.

Конец замечания.

Внимательно изучая цикл (3-11), можно уловить его идею - при каждом повторении поддерживать x равным max(M,i), чтобы при i = n получить q(11). Выразим этот замысел формально

I1 :: (x = max(M,i)

и попытаемся с помощью такого I1 преодолеть наш цикл.

Замечание. "Вылавливать" идеи циклов из написанных программ - довольно неблагодарная работа. Правильнее было бы формулировать инварианты при проектировании программы, а при доказательстве пользоваться заранее заготовленными инвариантами. Мы лишены возможности так действовать, потому что само понятие инварианта цикла появилось в наших рассуждениях лишь недавно. Однако и у нашего пути есть некоторые преимущества. По крайней мере есть надежда почувствовать сущность оптимального инварианта.

Конец замечания.

Предстоит решить три задачи:

а. Доказать, что I1 - действительно инвариант цикла (3-11).

б. Доказать, что условие q(11) выводимо с помощью I1.

в. Доказать, что из q(3) логически следует I1.

Естественно сначала заняться двумя последними задачами, так как наша цель - подобрать оптимальный инвариант. Если с помощью I1 нельзя, скажем, вывести q(11), то им вообще незачем заниматься. Так как задача (в) тривиальна при i = 1, займемся задачей (б).

Замечание. На самом деле задача (в) тривиальна лишь при условии, что можно пользоваться формальным определением функции max (точнее, определяющей эту функцию системой соотношений-аксиом). Например, такими соотношениями:

Ek : (k >= 1) & (k <= i) & M[k] = max(M,i) .

Ak : (k >= 1) & (k <= i) => M[k] <= max(M,i) .

При i = 1 отсюда следует M[1] = max(M,1). Так что I1 превращается в

(x = M[1])

т.е. просто в одну из конъюнкций q(3).

По сути это замечание привлекает внимание к факту, что при доказательстве правильности программ методом Хоара приходится все используемые в утверждениях понятия описывать на логическом языке первого порядка и непосредственно применять эти (довольно громоздкие) описания в процессе преодоления конструктов. Сравните подход Бэкуса.

Конец замечания.

3.5.1. Первая попытка

Итак, допустим, что I1 - инвариант цикла, и попробуем вывести q(11). По правилу преодоления (П4) в точке (11) выводимо

q(11)a :: x = max(M,i) & "not" (i < n) .

Сразу ясно, что q(11) не выводимо из q(11)a. Легко построить противоречащий пример:

i = 3, n = 2, M = (1,3,10); max (M,3) = 10.

3.5.2. Корректировка инварианта

Как видно, мы не зря сразу занялись задачей (б). Придется внимательнее изучить цикл и понять, что мы упустили, формируя его инвариант.

Контрпример получен при i > n. Ясно, что в цикле (3-11) такое значение i получиться не может, он сохраняет условие i <= n. Но ведь это значит, что обнаружен еще один претендент на роль инварианта цикла! Обозначим его через I2.

I2 :: i <= n .

Нетрудно проверить, что соединяя I1 с I2 в утверждении

I3 :: I1 & I2

можно доказать q(11).

Проведем это доказательство.

Действительно, если I3 окажется инвариантом, то по правилу преодоления цикла выводимо для точки (11)

q(11)b :: I1 & I2 & "not"(i < n) .

Но

q(11)b => (x = max(M,i) & (i = n) => x = max(M,n) .

Что и требовалось.

3.5.3. Правило соединения инвариантов цикла

Уместно отметить, что "пополнять" утверждения, претендующие на роль инварианта, приходится довольно часто в процессе подбора оптимальных инвариантов. Поэтому полезно сформулировать общее правило:

Конъюнкция инвариантов некоторого цикла остается инвариантом этого цикла.

Обратное, естественно, неверно. (Приведите контрпример).

Это правило более общего характера, чем правила преодоления языка Дейк, оно справедливо для любых инвариантов любого преодоления конструктов любого языка.

3.5.4. Инвариантность I1 и I2

Опираясь на правило соединения инвариантов, мы можем теперь решать задачу (а) отдельно для I1 и I2. Займемся сначала доказательством инвариантности I2, как делом более простым.

Напомним, что доказать инвариантность I2 для цикла (3-11) - это значит доказать истинность утверждения

Ak : {I2 & Pk} Sk {I2}

которое в нашем случае сводится к единственной тройке Хоара

{I2 & (i < n)} Ф(4-10) {I2}

так как в цикле (4-11) лишь один вариант.

Чтобы вывести нужную тройку, начнем с утверждения

q(4)a :: I2 & (i < n) :: (i <= n) & (i < n)

как предусловия для Ф(4-11) и постараемся применить правила преодоления сначала присваивания (4-5), а затем развилки (5-10) для вывода утверждения q(10)a :: I2.

Но

q(4)a => (i < n)

и по правилу преодоления присваивания получаем

!=> (i <= n) :: q(5)a .

Аккуратный вывод q(5)a предоставляем читателю (достаточно подготовить для Ф(4-5) предусловие в виде (Y = i+1) => (Y <= n)).

Теперь одного взгляда на фрагмент (5-10) достаточно, чтобы убедиться, что он сохраняет q(5)a - ведь он не изменяет ни i, ни n. Но это соображения содержательные, а при формальном выводе несложно воспользоваться правилами преодоления развилки и вложенных в него операторов (присваивания и пустого).

Оставим это в качестве упражнения и закончим тем самым доказательство инвариантности I2.

3.5.5. Внешний инвариант

Полезно сформулировать в явном виде правила преодоления для утверждений, не зависящих от объектов, изменяемых в преодолеваемых фрагментах. При этом мы, конечно, не получим принципиально новых возможностей преодоления. Однако упрощенные правила бывают особенно удобны при преодолении "по частям", которым мы только что воспользовались (разбив инвариант цикла на части и занимаясь ими по очереди).

Ясно, что упрощенное правило преодоления должно состоять в переписывании предусловия в качестве (конъюнктивного члена) постусловия.

Назовем внешним инвариантом преодолеваемого фрагмента всякое утверждение, к которому применимо такое упрощенное правило. Сформулировать признаки внешних инвариантов для отдельных конструктов языка Дейк - полезное упражнение.

3.5.6. Инвариантность I1

Вернемся к нашей программе-примеру и попытаемся доказать, что I1 - инвариант цикла (3-11).

Нужно доказать утверждение

Ak : {I1 & Pk} Sk {I1}

т.е. в нашем случае

{x = max(M,i) & (i < n)} Ф(4-10) {x = max(M,i)} .

Обозначим I1 & (i < n) через q(4) и рассмотрим его как предусловие для присваивания (4-5).

Преодолевая присваивание, получим

q(5) :: (x = max(M,i-1)) & (i-1 < n) .

Чтобы выполнить это преодоление аккуратно по правилам, нужно сначала применить правило преодоления точки и получить

(Y = i+1) => (x = max(M,Y-1)) & (Y-1,n)

т.е. получить предусловие присваивания в удобной для преодоления форме, а затем получить q(5) непосредственно по правилу (П1).

Теперь нужно преодолеть развилку (5-10).

В соответствии с правилом (П2) постусловие развилки должно быть постусловием каждой ветви развилки. Нам нужно получить в качестве такового I1. Со второй ветвью развилки (5-10) никаких сложностей не возникает:

q(5) & (M[i] <= x) => (x = max(M,i)) :: I1 .

(Применено правило преодоления пустого оператора, т.е. обычное логическое следование).

Займемся первой ветвью.

Ясно, что предусловие

q(6) :: q(5) & (M[i] > x)

непосредственно не пригодно для преодоления присваивания (6-7). Формально потому, что зависит от x. Содержатольно потому, что связь "нового" рассматриваемого значения массива, M[i], с остальными значениями (проявляющаяся в том, что M[i] - максимальное из них) выражена неявно и к тому же через значение x, которое "пропадает" в результате преодоления присваивания. Так что наша ближайшая цель - в процессе подготовки к преодолению проявить эту связь. Именно

q(6) => (M[i] = max(M,i) :: q(6)b .

Обозначив M[i] через Y, нетрудно теперь вывести I1 в качестве постусловия первой ветви развилки (5-10), а следовательно и цикла (3-11).

Осталось убедиться, что I3 логически следует из q(3). Это очевидно.

3.5.7. Исключение переменных

Подчеркнем важность приема, примененного при преодолении присваивания (6-7), его методологическое значение при доказательстве свойств программ на основе дедуктивной семантики. Перед преодолением операторов, содержательно влияющих на предусловие, необходимо вывести из него логическое следствие, не зависящее от изменяемых переменных (т.е. найти инвариант). Назовем этот прием исключением переменных.

Получение подходящих следствий предусловия - творческий акт в преодолении таких операторов. Методологическое значение приема исключения переменных сопоставимо со значением творческого подбора инвариантов цикла. Так что методика

применения дедуктивной семантики для доказательства корректности программ довольно тонко сочетает творческие содержательные действия с чисто формальными.

Подведем итоги. Мы провели доказательство содержательного утверждения о конкретной программе на языке Дейк, пользуясь его дедуктивной семантикой и рядом методологических приемов, опирающихся на понимание сути этой программы. Однако проверить корректность самого доказательства можно теперь чисто формально, не привлекая никаких содержательных (а значит, и подозрительных по достоверности) соображений. Достаточно лишь в каждом случае указывать соответствующее формальное правило преодоления конструкта и проверять корректность его применения. Итак, мы построили дедуктивную семантику языка Дейк и разработали элементы методики ее применения.

Вопрос. Могут ли в программе вычисления максимума остаться ошибки? Если да, то какого характера?

4. ОСНОВНЫЕ АБСТРАКЦИИ

4.1. Введение

Рассмотрев в предыдущих разделах три модели ЯП, мы увидели, как по-разному можно программировать, на каких различных базисах и средствах развития можно строить языки программирования (ЯП). Однако ни одна из наших моделей сама по себе не годится в качестве базового языка индустриального программирования.

Дело в том, что до сих пор мы занимались простыми задачами и решали их простыми средствами. Но реальное индустриальное программирование имеет дело с существенно более сложными задачами. Так, в наших моделях мы совершенно не интересовались тем, как обеспечить выполнение наших программ на разных машинах, как обеспечить взаимодействие многих людей, создающих единый комплекс программ, как обеспечить возможность пользоваться ранее созданными программами или частями программ, написанными на том же или на другом языке. Наконец, хотя мы и объявили надежность выделенной характеристикой качества программного изделия, в наших моделях не уделено должного внимания защите от ошибок при написании программ. Подчеркнув важнейшее свойство базового языка индустриального программирования - возможность строить на его основе свой язык для каждой проблемной области, мы не уделили достаточного внимания средствам развития. Было выделено лишь одно из них - определение новых функций, и совершенно не затронуто развитие структур данных.

[Естественно стремиться к простоте языка, однако до сих пор не удалось найти оптимального соотношения между базисом и средствами развития. Базис реальных ЯП остается громоздким.]

Теперь наша цель - дать возможно более полное представление о современном языке индустриального программирования. Сделаем это, построив нашу четвертую языковую модель - модель А. Для этого выделим ключевые технологические потребности индустриального программирования, соответствующие этим потребностям абстракции и соответствующие этим абстракциям выразительные средства - языковые понятия и конструкты. В качестве представителя современных языков, на примере которого будем демонстрировать излагаемые принципы и понятия, выберем язык Ада. Он хорош для нас тем, что олицетворяет комплексный подход к решению основной проблемы программирования - проблемы сложности, содержит понятия и конструкты, характерные для интересующего нас класса языков, и к тому же имеет немало шансов стать языком массового программирования в обозримом будущем.

Важна для нас и полнота языка в том смысле, что в нем в той или иной форме можно получить ответы на все вопросы современного практического программирования в конкретной проблемной области.

С точки зрения моделей, изученных в первой части язык Ада ближе всего к модели фон Неймана. Именно на ней основаны практически все современные языки индустриального программирования. Две другие модели отстают в этом смысле прежде всего из-за несоответствия архитектуре современных массовых компьютеров. Однако в будущем положение может измениться кардинально.

4.2. Принцип технологичности

Мы исходим из принципа технологичности - всякая языковая абстракция предназначена для удовлетворения технологических потребностей на определенных этапах жизненного цикла комплексного программного продукта. Этот принцип нацеливает на изучение важнейших потребностей в качестве "заказчиков" понятий и конструктов ЯП. С другой стороны, он требует понимания набора потребностей,

обслуживаемых каждым понятием и (или) конструктом. При желании всегда можно увидеть связь этих понятий с общей идеей абстракции-конкретизации. Будем подчеркивать эту связь, когда посчитаем существенной.

Обсуждение абстракций будем сопровождать примерами их воплощения и применения (чаще всего - в языке Ада).

4.3. Общее представление об языке Ада

Бесполезно говорить о технологических потребностях "вообще". Всегда имеется в виду более или менее определенная проблемная область и технология решения задач в этой области. В частности, технология программирования. Чтобы были понятнее рассматриваемые потребности и связанные с ними особенности нашего основного языка примеров (языка Ада) нужно рассказать об особенностях той проблемной области, для которой этот язык в первую очередь и создавался.

Замечание. Это не означает, что его нельзя применять в других областях. Фортран создавался для научных расчетов, а теперь на нем программируют системы для подготовки текстов, пишут трансляторы со специальных языков и т.п. Однако знание первоначальных замыслов авторов помогает понять особенности языка, взглянуть на язык как систему в какой-то мере согласованных решений, почувствовать то, что называют "духом" языка. Иначе язык покажется нагромождением условностей, которые очень трудно запомнить и применять.

Конец замечания.

Язык Ада создан, в основном, в 1978-80 годах в результате грандиозного проекта, предпринятого МО США с целью разработать единый ЯП для так называемых встроенных систем (т.е. систем, представляющих собой части более крупных, объемлющих автоматизированных комплексов). Имелись в виду прежде всего бортовые системы управления военными объектами (кораблями, самолетами, танками, ракетами, снарядами и т.п.). Поэтому решения, принятые авторами Ады, не следует считать универсальными. Их нужно воспринимать в контексте особенностей той конкретной проблемной области, для которой предназначен язык.

В этой проблемной области требуются особо надежные и особо эффективные системы управления, очень важно в максимальной степени использовать проверенные на практике, испытанные программы при развитии систем и переносе их в новые условия эксплуатации. Характерен также колоссальный диапазон сложности систем управления. Наконец - это так называемые системы реального времени (другими словами, такая система должна успевать реагировать на поведение реальных физических объектов, движущихся и изменяющихся с весьма высокими скоростями).

Важно для нас и то, что в этой области распространена так называемая кросс-технология программирования. Отличается она тем, что программы создаются на одной машине (базовой, инструментальной) для выполнения на другой (целевой, объектной) машине. Инструментальная машина обычно обладает существенно более богатыми ресурсами, объектная (часто бортовая, т.е. находящаяся непосредственно на управляемом объекте) - более бедными. Это обстоятельство сильно влияет, в частности, на соотношение возможностей периода компиляции и периода выполнения программ (в кросс-технологии компилятор работает на инструментальной машине).

Таким образом, Ада должна быть ориентирована на тщательный контроль программ при компиляции (надежность встроенных систем и относительно богатые возможности инструментальной машины), оптимизацию объектной программы (эффективность встроенных систем, бедные ресурсы объектной машины и богатые ресурсы инструментальной), должна обеспечивать возможность переносить программы в новые условия эксплуатации при минимальных дополнительных затратах. Итак, определяющие требования к языку Ада - надежность, переносимость, эффективность.

4.4. Пример простой программы на Аде

Чтобы оставить у читателя первое зрительное впечатление об Аде, дадим пример совсем простой (но полной) программы. Основой этого (и некоторых других наших примеров) послужили программы из учебника Янга "Введение в Аду" (S.J.Joung. An Introduction to ADA, Ellis Horwood Ltd, England, 1983).

Напишем программу, которая вводит последовательность символов со стандартного устройства ввода и на стандартном устройстве вывода печатает левые и правые скобки, обнаруженные в этой последовательности.

Вот эта программа:

```
1.  with  текстовый_ввод_вывод;
2.  procedure  печать_скобок is
```

```
3.  ch : символ ;
4.  begin
5.  текстовый_ввод_вывод.получить (ch) ;
6.    while ch /= '.' loop
7.      if ch = '(' or ch = ')' then
8.        текстовый_ввод_вывод.послать (ch) ;
9.      end  if  ;
10.     текстовый_ввод_вывод.получить (ch) ;
11.   end  loop  ;
12. end  печать_скобок ;
```

Итак, в программе двенадцать строчек (номера этих строчек не входят в программу - они нужны нам, чтобы обсуждать ее свойства). Общий вид программы напоминает текст на Алголе 60 или Паскале (некоторые знакомые ключевые слова, характерная ступенчатая запись и т.п.. Это и не удивительно. Ада - из семейства "поздних алголоидов". Точнее, этот язык - развитие Паскаля, в свою очередь созданного (Никлаусом Виртом) на основе Алгола 60.

Вместе с тем даже в этой простой программе заметны характерные особенности поздних языков индустриального программирования.

Во-первых, высокоразвитая модульность. Фраза с ключевым словом "with" (в переводе с английского "с" или "совместно с") говорит о том, что данную процедуру следует читать, понимать и исполнять во вполне определенном контексте. Этот контекст задан модулем-ПАКЕТОМ с именем "текстовый_ввод_вывод". В нем содержатся определения всех ресурсов, необходимых для ввода-вывода текстов (в частности, процедуры "получить" очередной символ со стандартного устройства ввода и "послать" очередной символ на стандартное устройство вывода). Внутри программы, использующей такой пакет, обращаться к его ресурсам следует по составным именам (сначала название пакета, а затем через точку - название ресурса). При необходимости можно, конечно, ввести сокращенные обозначения для часто используемых ресурсов.

Во-вторых, более богатый набор типов данных. В строчке 3 находится объявление переменной ch типа "символ". Это один из предопределенных типов Ады.

[Здесь и далее предопределенные идентификаторы языка Ада переведены на русский язык. В оригинале - тип character. Соображения наглядности более важны, чем строго формальное следование правилам языка; ведь здесь концепции Ады приводятся лишь в качестве примера общих концепций в ЯП. К тому же можно надеяться, что отечественный стандарт Ады будет допускать предопределенные русские названия].

Ни в Алголе 60, ни в Фортране такого символьного типа, равноправного с остальными типами, нет. Один из источников выразительной мощи языка Ада - возможность строить новые типы данных, не предопределенные авторами языка. Такая возможность теперь имеется во всех новых ЯП и мы с ней подробно познакомимся.

В-третьих, ради надежности повышена избыточность, способствующая устранению случайных ошибок. Это и (сколь угодно) длинные названия-идентификаторы, которые можно к тому хе составлять из отдельных слов, соединенных одиночным подчеркиванием. Это и строгая скобочная структура текста - каждый управляющий конструкт снабжен специальным "закрывающим" ключевым словом (цикл в строчках с 6 по 11, условный оператор в строчках 7-9, процедура в строчках 2-12). Надежности, ясности, четкой структуре и избыточности способствуют также и строгие правила ступенчатой записи программ (в Аде она настоятельно рекомендуется в определении языка и отражена в его синтаксисе).

Смысл программы достаточно очевиден. В строчке 5 вводится первый символ обрабатываемой последовательности и помещается в переменную ch. Далее цикл, работающий до тех пор, пока значением переменной ch не станет символ "." ("/=" - это "не равно", "." - это признак конца последовательности обрабатываемых символов). В теле цикла - условный оператор, который посылает на устройство вывода очередной символ, если это открывающая или закрывающая скобка. Затем (строкой 10), вводится в переменную ch очередной символ последовательности и цикл повторяется. Вместе с циклом завершается и процедура печать_скобок.

4.2. Обзор языка Ада

Этот раздел близок по структуре и стилю к разделу 1.4 официального определения языка Ада - национальному стандарту США, принятому в феврале 1983 года. Рассказывая об этом языке и приводя примеры (из различных источников), будем и впредь опираться на это официальное определение.

[Вполне возможно и даже вероятно, что оно без существенных изменений будет принято и в качестве международного стандарта - стандарта ИСО и отечественного ГОСТа. Готовится его перевод на русский язык, отечественные реализации Ады также ориентируются на это определение].

Однако наша цель - не определить язык, а продемонстрировать концепции, характерные (и, как правило, перспективные) для базовых языков индустриального программирования. Поэтому будем стремиться упрощать изложение и избегать несущественных для нашей цели деталей. (Само по себе их обилие в официальном сообщении, к сожалению, также характерно. Оно свидетельствует либо о неразвитости науки и практики языкотворчества, либо о фундаментальных свойствах такого социального явления, как ЯП. Здесь есть над чем думать и работать; поразительный пример лаконичности - определение Никлаусом Виртом языка Модула-2 [5]). Другими словами, нас интересует не столько сам язык Ада, сколько возможность использовать его в качестве источника идей для модели А.

Вместе с тем, приводя примеры адовских программ, будем строго следовать стандарту, чтобы не создавать у читателей лишних затруднений при последующем самостоятельном освоении языка.

Понятия языка Ада (Ада-понятия) будем выделять прописными буквами.

Ада-программа состоит из одного или более программных МОДУЛЕЙ (сегментов), которые можно компилировать раздельно.

Ада-модуль - это ПОДПРОГРАММА (определяет действия - части отдельных ПРОЦЕССОВ) или ПАКЕТ (определяет часть контекста - совокупность объектов, предназначенных для совместного использования) или ЗАДАЧА (определяет асинхронный процесс) или РОДОВОЙ модуль (заготовка пакета или подпрограммы с параметрами периода компиляции).

В каждом модуле обычно две части: внешность или СПЕЦИФИКАЦИЯ (содержит сведения, видимые из других модулей) и внутренность или ТЕЛО (содержит детали реализации, из других модулей невидимые). Разделение спецификации и тела вместе с раздельной компиляцией дает возможность проектировать, писать и проверять программу как набор относительно самостоятельных (слабо зависимых) компонент.

Ада-программа пользуется ТРАНСЛЯЦИОННОЙ БИБЛИОТЕКОЙ. Поэтому в тексте создаваемого модуля следует указывать названия используемых библиотечных модулей.

4.2.1 Модули

Подпрограмма - основной конструкт для определения подпроцессов (ввод данных, обновление значений переменных, вывод данных и т.п.). У подпрограммы могут быть параметры, посредством которых ее связывают с контекстом вызова. Различают две категории подпрограмм - процедуры и функции. Последние отличаются тем, что вырабатывают результат, непосредственно доступный в точке вызова функции. [Поэтому вызов функции всегда входит в некоторое выражение].

Пакет - основной конструкт для определения именованного контекста (иногда говорят "логически связанной" совокупности объектов). Несколько расплывчатое "логически связанной" подразумевает возможность объединить в пакет все то, что автор пожелает видеть единым модулем, названным подходящим именем. Это может быть сделано, во-первых, потому, что все связанные в пакет объекты предполагается использовать совместно; во-вторых, потому, что их необходимо или удобно совместно реализовать; в-третьих, потому, что их невозможно или неудобно раздельно определять из-за адовских ограничений на ВИДИМОСТЬ имен. Возможны и иные причины объединения в один пакет определений отдельных имен. Часть из них может быть при этом скрыта, ЗАЩИЩЕНА от непосредственного использования другими модулями; доступ к таким именам строго регламентирован - только через имена, в спецификации пакета явно предназначенные для внешнего использования.

Задача - основной конструкт для определения асинхронного процесса, способного выполняться параллельно с другими процессами. Процессом называется определенным образом идентифицируемая последовательность действий исполнителя, линейно-упорядоченная во времени. В одном модуле-задаче можно определить один асинхронный процесс или совокупность аналогичных асинхронных процессов (так называемый ЗАДАЧНЫЙ ТИП). [Асинхронность можно обеспечивать как за счет отдельных процессоров для каждого процесса, так и за счет "прерывистого" их выполнения на одном процессоре].

4.2.2. Объявления и операторы

В теле модуля в общем случае две части - ОБЪЯВЛЕНИЯ и ОПЕРАТОРЫ. Объявления вводят новые знаки (ИМЕНА) и связывают их с денотатами (ОБЪЕКТАМИ). Эта связь имени с определенным объектом (знаковая ситуация) сохраняет силу в пределах ОБЛАСТИ ДЕЙСТВИЯ имени. Таким образом, формально объект - это то, что можно именовать. Однако авторы языка стремились к тому, чтобы ада-объектами было удобно непосредственно моделировать содержательные объекты решаемой задачи. Ада-объектами могут быть, в частности, ПОСТОЯННАЯ, ПЕРЕМЕННАЯ, ТИП, ИСКЛЮЧЕНИЕ, ПОДПРОГРАММА, ПАКЕТ, ЗАДАЧА и РОДОВОЙ модуль.

Операторы предписывают действия, которые выполняются в порядке следования операторов в тексте программы (если только операторы ВЫХОДА из конструкта (exit), ВОЗВРАТА (return), ПЕРЕХОДА по метке (go to) или возникновение исключения (исключительной ситуации) не заставят продолжить исполнение с другого места.

Оператор ПРИСВАИВАНИЯ изменяет значение переменной.

ВЫЗОВ ПРОЦЕДУРЫ активизирует исполнение соответствующей процедуры после связывания каждого фактического параметра (АРГУМЕНТА) с соответствующим формальным параметром (ПАРАМЕТРОМ).

УСЛОВНЫЙ (if) и ВЫБИРАЮЩИЙ (case) операторы позволяют выбрать одну из возможных вложенных последовательностей операторов в зависимости от значения УПРАВЛЯЮЩЕГО ВЫРАЖЕНИЯ (условия).

Оператор ЦИКЛА - основной конструкт для описания повторяющихся действий. Он предписывает повторять указанные в его теле действия до тех пор, пока не будет выполнен оператор выхода, явно указанный в теле цикла, или не станет истинным условие окончания цикла.

Блочный оператор (БЛОК) соединяет последовательность операторов с непосредственно предшествующими ей объявлениями в единую ОБЛАСТЬ ЛОКАЛИЗАЦИИ. Объявленные в ней объекты считаются ЛОКАЛЬНЫМИ в этой области.

Несколько операторов обслуживают взаимодействие асинхронных процессов.

При исполнении модуля могут возникать ошибочные ситуации, в которых нельзя нормально продолжать работу. Например, возможно арифметическое переполнение или попытка получить доступ к компоненте массива с несуществующим индексом. Для обработки таких исключительных ситуаций (ИСКЛЮЧЕНИЙ) в конце модулей можно разместить специальные операторы РЕАКЦИИ на исключение (exception). Имеются и явные операторы ВОЗБУЖДЕНИЯ исключений (raise). Они включают в действие аппарат обработки возбужденного исключения.

4.2.3 Типы данных

Среди ада-объектов можно выделить ОБЪЕКТЫ ДАННЫХ (т.е. объекты, которым разрешено играть роль данных по отношению к каким-либо операциям). Каждый объект данных в Аде характеризуется определенным ТИПОМ. Своеобразие этого языка в значительной степени связано именно с системой типов. Для тех, кто работал только с Фортраном, Алголом и Бейсиком, многое в этой системе окажется совершенно незнакомым. В частности, возможность определять новые типы, отражающие особенности решаемой задачи. Для освоивших Паскаль адовские типы привычнее. По сравнению с Паскалем система адовских типов полнее и строже, лучше отвечает своему назначению.

Тип - важнейшая компонента аппарата прогнозирования-контроля. Приписывая объекту данных определенный тип, ограничивают его возможное поведение. С другой стороны, зная тип, получают возможность это поведение контролировать. Наконец, зная ограничения на возможное поведение, можно рационально выделять память и другие ресурсы. С типом в Аде связывают три основных ограничения:

Во-первых, тип ограничивает ОБЛАСТЬ ЗНАЧЕНИЙ объекта; во-вторых, НАБОР ОПЕРАЦИЙ, в которых объекту разрешено фигурировать; в-третьих, набор допустимых для него ролей в этих операциях (скажем, в качестве второго операнда, результата и т.п.).

Имеется четыре категории типов: СКАЛЯРНЫЕ типы (в том числе ПЕРЕЧИСЛЯЕМЫЕ и ЧИСЛОВЫЕ), СОСТАВНЫЕ (в том числе РЕГУЛЯРНЫЕ типы (массивы) и КОМБИНИРОВАННЫЕ (записи, структуры)), ССЫЛОЧНЫЕ типы (указатели) и ПРИВАТНЫЕ типы (представление которых невидимо пользователю).

Дадим краткое введение в каждую из категорий типов.

4.2.3.1. Скалярные типы

Когда определяют перечисляемый тип, явно указывают перечень лексем, которые и составляют область возможных значений объектов вводимого типа. Такой перечень может быть списком дней недели (понедельник, вторник, среда, четверг, пятница, суббота, воскресенье), списком символов некоторого алфавита (`A',`B',...,`Z') и

т.п. Перечисляемые типы избавляют программиста от необходимости кодировать содержательные объекты целыми числами. Перечисляемые типы BOOLEAN (логический) и CHARACTER (символьный) считаются ПРЕДОПРЕДЕЛЕННЫМИ, т.е. встроенными в язык и действующими без предварительного явного объявления в программе. Набор символов типа CHARACTER соответствует алфавиту ASCII - американскому стандарту на коды символов.

Числовые типы обеспечивают точные и приближенные вычисления. В точных вычислениях пользуются ЦЕЛЫМИ типами. Область возможных значений для таких типов - конечный диапазон целых чисел. В приближенных вычислениях пользуются либо АБСОЛЮТНЫМИ типами (для них задается абсолютная допустимая погрешность), либо ОТНОСИТЕЛЬНЫМИ типами (задается относительная погрешность). Абсолютная погрешность задается явно и называется ДЕЛЬТОЙ, относительная погрешность вычисляется по заданному допустимому количеству значащих цифр в представлении числа. Подразумевается, что абсолютные типы будут представлены машинной арифметикой с фиксированной точкой (запятой), а относительные - с плавающей. Типы INTEGER (целый), FLOAT (плавающий), и DURATION (временные задержки для управления задачами) считаются предопределенными.

4.2.3.2. Составные типы

Скалярные типы (и перечисляемые, и числовые) выделяются тем, что объекты этих типов считаются атомарными (не имеющими составляющих). Составные типы, в отличие от скалярных, позволяют определять структурированные объекты (массивы и записи). Массивы служат значениями регулярных типов - компоненты массивов доступны по индексам. "Регулярность" массивов проявляется в том, что все компоненты должны быть одного типа. Записи (структуры) служат значениями комбинированных типов - их компоненты могут быть различных типов; компоненты записей доступны по именам-селекторам. Имена компонент одной и той же записи должны быть различны; компоненты называются также ПОЛЯМИ записи.

Строение записей одного типа может зависеть от значений выделенных полей, называемых ДИСКРИМИНАНТАМИ. Дискриминанты играют роль параметров комбинированного типа - задавая набор дискриминантов, выбирают определенный вариант структуры объектов этого типа. Поэтому типы с дискриминантами называют также ВАРИАНТНЫМИ типами.

4.2.3.3. Ссылочные типы

Если структура объектов составных типов (в случае вариантных типов - все варианты такой структуры) фиксируется статически (т.е. до начала выполнения программы), то ссылочные типы позволяют создавать и связывать объекты динамически (при исполнении программы, точнее, при исполнении ГЕНЕРАТОРОВ). Тем самым появляется возможность динамически создавать сколь угодно сложные конгломераты объектов. Генератор создает объект указанного (статически известного) типа и обеспечивает доступ к вновь созданному объекту через переменную соответствующего ссылочного типа. Передавая (присваивая) ссылки, можно организовать произвольные структуры. Важно, что и элементы, и связи в таких динамических структурах можно менять при исполнении программы.

4.2.3.4. Приватные типы

Доступ к ПРИВАТНЫМ объектам (их называют также абстрактными объектами, а соответствуюшие типы - абстрактными типами данных или АТД) находится под полным контролем автора приватного типа. Такой тип всегда определяется в некотором пакете (который называется ОПРЕДЕЛЯЮЩИМ пакетом для этого типа). Спецификация определяющего пакета фиксирует полный набор операций и тех ролей в этих операциях, в которых могут фигурировать объекты нового типа (в модулях, использующих определяющий пакет). В определяющем пакете фиксируется и реализация приватного типа, однако в использующих модулях она непосредственно недоступна - только через явно перечисленные автором допустимые операции. Поэтому реализацию можно изменять, не заставляя переделывать использующие модули.

Концепция типа в Аде дополнена аппаратом ПОДТИПОВ, (они ограничивают область значений, не затрагивая допустимых операций), а также аппаратом ПРОИЗВОДНЫХ типов (они образуются из уже известных типов, наследуя связанные с ними значения и операции).

4.2.4. Об остальных средствах языка

Посредством СПЕЦИФИКАЦИИ ПРЕДСТАВЛЕНИЯ (УКАЗАНИЯ ПРЕДСТАВЛЕНИЯ) можно уточнить требования к реализации определенных типов на (целевой) машине. Например, можно указать, что объекты такого-то типа следует представить заданным количеством битов, что такие-то поля записи должны располагаться с такого-то адреса. Можно указать и другие детали реализации, вплоть до прямой вставки машинных команд. Ясно, что подробная спецификация представления мешает переносу программы в другую операционную обстановку. С другой стороны, она может оказаться исключительно важной для качества и даже работоспособности реализации. Явная спецификация представления помогает отделять машинно-независимые части модулей от машинно-зависимых. В идеале только спецификацию представления и потребуется менять при переносе программы.

Ввод-вывод (обмен с внешней средой) определен в Аде предопределенными библиотечными пакетами. Имеются средства обмена значениями как предопределенных типов, так и типов, определяемых программистом.

Наконец, имеются средства статической параметризации модулей (действующие до начала исполнения программы, в период компиляции) - аппарат РОДОВЫХ модулей. Родовые параметры могут быть не только объектами некоторого типа данных (как в случае динамических параметров подпрограмм и процедур), но и такими объектами, как типы и подпрограммы (которые в Аде не считаются объектами данных). Так что общие модули, применимые ко всем типам данных определенной категории, в Аде должны быть оформлены как родовые.

На этом закончим краткий обзор конструктов и концепций языка.

Подробнее о возможностях языка Ада поговорим при последовательном изучении основных абстракций программирования.

4.3. Три важнейшие абстракции: данные, операции, связывание

Планируя поведение исполнителя (составляя программу) мы, в конечном итоге, планируем его действия над некоторыми объектами. Чтобы указать, что конкретно следует делать исполнителю, необходимо обозначить конкретный объект, конкретное действие, и обозначить конкретные условия, при которых нужно связать действие с объектом (т.е. указанное действие совершить над указанным объектом). Таким образом, в описании акта исполнителя выделяются составляющие, играющие три различные роли, а именно роли данных, операции и связывания.

Первые две роли выделяются и обсуждаются чаще. Однако значение связывания никак не меньше хотя бы потому, что три названные роли неразрывны в каждом конкретном акте поведения исполнителя (когда указана операция, указаны данные и созрели условия для связывания).

Однако планировать поведение (программировать) было бы невозможно, если бы не удалось разорвать единство этих ролей и рассматривать данные, в нужной степени абстрагируясь от конкретных операций; рассматривать операции, в нужной степени абстрагируясь от конкретных данных, над которыми они выполняются; и , наконец. рассматривать связывание, в нужной степени абстрагируясь от данных и операций, которых оно касается.

Если полученное путем такой абстракции понятие операции отражает активное начало в поведении исполнителя, понятие данного - пассивное начало, то понятие связывания отражает организующее (управляющее) начало - отражает всевозможные виды управления. [С другой стороны, понятие связывания отражает и всевозможные виды конкретизации в программировании].

4.3.1 Принцип единства и относительности трех абстракций

Важно понимать, что указанные три роли полностью различны лишь в рамках конкретного акта исполнителя. Когда же речь идет о каком-либо отдельно взятом объекте, то в зависимости от ситуации или точки зрения он вполне может выступать в различных ролях (иногда - в любой из этих ролей). В этом смысле указанные роли (как и любые другие) относительны. Например, знак "+" чаще всего воспринимается как обозначение операции сложения. Вместе с тем он выступает в роли данного, когда фигурирует как аргумент редукции (вспомните модель Бэкуса) или как элемент формулы, которую переводят в польскую инверсную запись. Но этот же знак и связывает два операнда формулы, между которыми он помещен, предвосхищая их совместное участие в одном акте исполнителя. Этот акт будет выполнен при условии, что управление достигло именно рассматриваемого знака "+".

Относительность и единство (взаимосвязь, взаимозависимость) трех выделенных ролей - глубокая закономерность. Как мы увидим при раздельном изучении каждой из

абстракций (и операций, и данных, и связывания), обслуживающие их конструкты, во-первых, имеют много общего и, во-вторых, для достаточно развитого оформления каждой абстракции привлекаются и две другие. Однако в этом случае они играют подчиненную роль, роль обслуживающего средства, роль инструмента для выделения и (или) реализации основной абстракции.

Укажем еще ряд примеров, подтверждающих принцип относительности и единства выделенных абстракций. Покажем, что он не только объясняет появление и смысл языковых конструктов, но и указывает перспективы их развития.

Так, хороший стиль программирования предполагает, что когда вводят операционную абстракцию (процедуру, функцию, операцию), то явно указывают характеристики данных, к которым ее можно применять (с которыми ее можно связывать). Например, специфицируют все параметры процедуры. Это спецификация операции по данным ИЗВНЕ. Естественное развитие (пока в ЯП отсутствует) - явно специфицировать все глобальные данные, обрабатываемые процедурой, но не передаваемые ей в качестве параметров (или запретить такие данные).

Еще пример. Когда реализуют введенную операционную абстракцию (проще говоря, программируют процедуру или функцию), то также указывают характеристики данных (на этот раз локальных), используемых подчиненными (реализующими) абстракциями. Это спецификация операции по данным ИЗНУТРИ.

Когда вводят абстракцию данных (переменнную, тип переменнных) то указывают класс операций, связываемых с этой абстракцией. Это спецификация данных по операциям ИЗВНЕ. Естественное развитие - из соображений симметрии следовало бы рассматривать и данные (абстракции данных), для реализации которых нужны внутренние (локальные) операции. Это была бы спецификация данных по операциям ИЗНУТРИ.

Вопрос. Зачем могут понадобиться такие спецификации?

Подсказка. Представьте "живые" данные, отражающие текущее состояние процесса. Какие операции потребуются, чтобы их реализовать? Какой должна быть среда, в которую можно перенести такие данные?

Читатель, имеющий даже небольшой опыт программирования, без напряжения приведет и другие примеры операций, данных и связываний. Повидимому, сомнений в рациональности самостоятельного рассмотрения операций и данных у него не возникнет. Выделение абстракции связывания менее традиционно. Поэтому полезно рассмотреть его подробнее и проиллюстрировать плодотворность этого понятия нетривиальными применениями.

4.4. Основные абстракции: связывание

В самом общем понимании связывание неотличимо от установления соответствия, сопоставления, отношения. Однако в применении к программированию нас интересует определенный набор выразительных средств, управляющих связыванием и пригодным для реализации на компьютерах. Поэтому, говоря о связывании здесь, мы подразумеваем аппарат, действующий в рамках существующей или потенциальной системы программирования и предназначенный для организации взаимодействия операций и данных.

Постараемся показать, во-первых, что абстракция связывания помогает с единых позиций понимать кажущиеся совершенно различными сущности и, во-вторых, что выделение такой абстракции помогает увидеть возможные направления развития ЯП.

В качестве составляющих единого аппарата связывания в системе программирования естественно рассматривать редактор связей, загрузчик, оптимизатор, компилятор, интерпретатор и другие системные средства, предназначенные для подготовки программы к выполнению. Ведь все эти средства участвуют в различных этапах окончательного связывания конкретных операций с конкретными операндами.

Итак, можно говорить о связывании загрузочных модулей для последующего совместного использования. Такая работа выполняется редактором связей. Можно говорить о связывании аргументов подпрограммы с ее телом для последующего совместного выполнения. Такая работа выполняется вызовом и заголовком подпрограммы. Вполне разумно говорить и о связывании отдельных компонент объектной программы в процессе ее трансляции с ЯП. Такая работа выполняется транслятором.

Отметим важный аспект. Связывание может распадаться на этапы, выполняемые на различных стадиях подготовки того конкретного акта исполнителя, ради которого это связывание в конечном итоге осуществляется. Например, транслировать можно за

несколько проходов; связать загрузочные модули можно частично редактором связей, частично загрузчиком; при обращении к программе часть работы по связыванию можно выполнить при обращении к подпрограмме, часть - при выполнении ее тела (именно распределением работы по связыванию отличаются различные способы вызова параметров - наименованием, значением, ссылкой и др.).

Повидимому, все эти примеры хорошо известны. Но общая концепция связывания способна привести и к совершенно новому понятию, отсутствующему в традиционных ЯП.

4.4.1. От связывания к пакету (связывание с контекстом, контекст как модуль, пакет)

Из общего курса программирования известно, что такое контекст и что модуль - это программа, рассчитанная на многократное использование в различных контекстах (и для этого соответствующим образом оформленная). В традиционных ЯП контекст задается обычно совокупностью объявлений (описаний) некоторого блока или подпрограммы и связывается с телом блока текстуально, физическим соединением тела и контекста. Но если модуль рассчитан на различные контексты, то и контекст, естественно, может оказаться пригодным для работы с различными модулями. Следовательно, хорошо бы и контекст оформлять по таким правилам, чтобы его не нужно было выписывать каждый раз, а можно было использовать как модуль, связывая с телом блока, например, во время трансляции. Подобной категории модулей ни в Алголе, ни в Фортране, ни в Паскале нет. Впервые такой модуль появился в языке Симула-67 и был назван "классом". В Аде его аналог назван "пакетом". Рассмотрим подробнее путь к пакету на конкретном примере.

В общем курсе программирования при изучении структур данных знакомят с совокупностью понятий, позволяющих работать со строками. Например, определяют представление строк одномерными массивами и предоставляют несколько операций над строками (скажем, в-строку, из строки и подстрока). Спрашивается, каким образом оформить это интеллектуальное богатство так, чтобы им было удобно пользоваться? Алгол 60 позволяет записать соответствующие объявления массивов и процедур, а тем самым сделать их известными многим программистам (этим свойством Алгола и воспользовались авторы курса по программированию [6], из которого взят пример). В сущности совокупность указанных объявлений массивов, переменных и процедур, выписанная в начале блока, позволяет в теле блока работать на языке, расширенном по сравнению с Алголом (понятием строчных переменных и набором операций над такими переменными).

Но вот мы знаем (изучили) эти объявления и хотим ими воспользоваться (скажем, запрограммировать и запустить универсальный нормальный алгоритм Маркова, как нам предлагают авторы того же курса). Алгол заставляет нас переписать в свою программу все нужные объявления. Но это и труд, и ошибки, и время, и место на носителях. На практике, конечно, во всех реализациях Алгола есть возможность обращаться к библиотеке, где можно хранить объявления функций и процедур (но не переменных и массивов). Однако целостного языкового средства, обслуживающего потребность делать доступным расширение языка, однажды спроектированное и полностью подготовленное к использованию, в Алголе нет. Другими словами, не выделена абстракция связывания компонент потенциально полезного контекста. Нет ее ни в Паскале, ни в Фортране, хотя общие объекты последнего - намек на движение в нужном направлении.

Как уже сказано, впервые нужная абстракция была осознана и оформлена соответствующим конструктом в языке Симула-67. Основная идея в том, что совокупность объявлений можно синтаксически оформить (в качестве "класса"), предварив их ключевым словом class и снабдив индивидуальным именем. Так можно получить, например, класс с именем обработка_строк, в котором будут объявлены одномерный массив и процедуры для работы с этим массивом как со строкой символов. Чтобы воспользоваться такими объявлениями (в совокупности!), достаточно перед началом программы указать в качестве приставки имя нужного класса. Объявления из такого класса считаются выписанными в фиктивном блоке, объемлющем создаваемую программу (т.е. доступны в ней). Например, программу нормального алгоритма достаточно предварить приставкой обработка_строк.

В языке Ада аналогичная абстракция воплощена конструктом ПАКЕТ. В первом приближении его основная идея совпадает с идеей класса - это также совокупность объявлений, снабженная именем и пригодная для использования в качестве "приставки". Однако в понятии "пакет" воплощены и другие важнейшие идеи, которыми

займемся позже. А пока подчеркнем, что к новым понятиям нас привела общая концепция связывания.

Вопрос. Чем концепция контекста как модуля отличается от идеи простого копирования контекста? От идеи макроопределений?

Подсказка. Важно, когда происходит связывание, а также чего и с чем. Кроме того, не забывайте о регламентированном доступе к контексту.

4.4.2. Связывание и специализация

Не только отдельные языковые конструкты обязаны своим возникновением тому, что связывание было осознано как самостоятельная абстракция. На его основе возникло целое направление в программировании - так называемое конкретизирующее программирование (помните, мы отмечали, что связывание обобщает основные виды конкретизации в программировании). Когда говорят о конкретизирующем программировании, часто приводят такой пример.

Рассмотрим операцию "**" возведения основания x в степень n. Если понятно самостоятелъное значение связывания, то легко представить себе ситуацию, когда с операцией "**" уже связан один операнд и еще не связан другой. С точки зрения итогового возведения в степень такая ситуация запрещена - еще нельзя совершить запланированный акт поведения (операнды не готовы). Но если понимать связывание как многоэтапный процесс подготовки этого акта, то рассматриваемая ситуация может соответствовать одному из этапов этого процесса. Более того, на аналогичном этапе связывания могут задерживаться целые классы таких процессов. Поэтому такое повторяющееся следует выделить, обозначить и применить (пользуемся одним из важнейших общих принципов абстрагирования - принципом обозначения повторяющегося). Таким образом получается целый ряд одноместных операций ("1**","2**","3**"...) при фиксированном основании и ряд одноместных операций ("**1","**2","**3"...) при фиксированном показателе степени. Но, скажем, операцию "**3" можно реализовать просто как x*x*x, что короче, проще и эффективней программы для "**".

Таким образом и возникает идея универсального конкретизатора, который по параметрической программе (например, "**") и некоторым уже связанным с ней аргументам строит (потенциально более эффективную) конкретизированную программу (например, "**3"). Если такой конкретизатор удастся построить для некоторого класса программ, то возникает надежда обеспечить целую проблемную область эффективными и надежными, "по происхождению" правильными программами. Ведь исходная параметрическая программа предполагается сделанной исключительно тщательно (во всяком случае правильно) - ведь при таком ее широком назначении на нее не жалко усилий.

В настоящее время конкретизирующее программирование интенсивно развивается и у нас, и за рубежом. Конкретизатор в литературе называют иногда специализатором (именно так мы назвали конкретизирующую функцию в модели Бэкуса), а также смешанным вычислителем (за то, что он проводит вычисления и над данными, и над программами).

Отметим, что все языковые конструкты можно при необходимости считать частью реализации аппарата связывания. Ведь с их помощью аргументы программы связываются с ее результатами на той или иной стадии обработки ее текста. Воспользуемся этим наблюдением, чтобы продемонстрировать еще одно применение аппарата связывания - применим его для уточнения терминологии и указания некоторой перспективы в теории трансляции.

4.4.3. Связывание и теория трансляции

Основной результат раздела: такие программы, как компилятор и суперкомпилятор (генератор компиляторов) могут быть формально получены из интерпретатора ЯП с помощью подходящего связывания.

Ключевая идея: следует применить особый вид связывания, обобщающий обычный вызов функции таким образом, что часть параметров функции связывается со своими аргументами, а остальные служат параметрами так называемой остаточной функции. Так называют функцию, вызов которой с недостающими аргументами эквивалентен вызову исходной функции с полным набором аргументов. Такой вид связывания называют специализацией. Выполняется он посредством "универсального специализатора" s, уже знакомого нам по модели Б.

Напомним, s - это форма, которая по произвольной функции двух переменных F(X,Y) и заданному ее аргументу x0 выдает в качестве результата функцию одного

аргумента s(F,x0) такую, что для всех допустимых значений параметра Y справедливо определяющее соотношение

(**) s(F,x0)(Y) = F(x0,Y).

так что s(F,x0) - это и есть остаточная функция.

Покажем, как получить объявленный основной результат. Допустим, что все рассматриваемые функции и формы реализованы подходящими программами. Сохраним для этих программ те же обозначения. Так что s(F,x0) можно теперь считать программой, полученной по исходной программе F с помощью программы s.

Замечание.

Важно понимать, что о качестве получаемых специализированных (остаточных) программ в определении универсального специализатора ничего не сказано. Тривиальное преобразование программ может состоять, скажем, в том, что в остаточной программе просто содержится вызов вида F(x0,Y).

Вопрос. Как оформить тривиальную остаточную программу на одном из известных вам ЯП?

Конец замечания.

Рассмотрим теперь язык программирования L и его интерпретатор i. С одной стороны, i - это такая программа, что для всякой правильной программы p на языке L и исходных данных d

i(p,d) = r,

где r - результат применения программы p к данным d. Другими словами, программа i реализует семантику языка L - ставит в соответствие программе p результат ее выполнения с данными d. С другой стороны, i - это форма от двух аргументов, а именно ограниченная аппликация (ведь она применяет p к d, причем пригодна только для программ из L).

Интерпретатор может быть реализован аппаратно, т.е. быть отдельным устройством, предназначенным для выполнения программ на L. Однако для нас интереснее случай, когда интерпретатор реализован программой. Программа эта написана, конечно, на каком-то языке программирования M. Будем считать, что M отличен от L. Программная реализация интерпретатора интересна именно потому, что в этом случае интерпретатор представлен написанным на языке M текстом-программой и вполне можно ожидать, что в общем случае из этого текста можно систематическими преобразованиями получать другие программы. Например, программы компилятора и суперкомпилятора, также написанные на языке M.

Мы намерены делать это посредством специализатора s. Для определенности будем считать, что программа-специализатор s также написана на языке M, применима к текстам программ, написанным на M и выдает в качестве результатов программы, напмсанные все на том же языке M.

4.4.3.1. Специализация интерпретатора

Посмотрим, что собой представляет s(i,p)? Другими словами, во что специализатор s превращает интерпретатор i после его связывания с конкретной программой p? (Ведь i - форма от двух аргументов, так что s к ней применим; при этом в соответствии со смыслом s с i связывается первый аргумент интерпретатора - p, а второй остается свободным параметром). Применяя (**), получим

s(i,p)(d) = i(p,d) = r.

[Обратите внимание, чтобы выписать результат специализатора, нужно "передвинуть" функциональные скобки на позицию вправо и опустить символ специализатора].

Другими словами, s(i,p) - это такая программа p', которая после применения к данным d дает результат r. Следовательно, p' эквивалентна программе p. Но p' написана уже на языке M, а не на L! Следовательно, p' - это перевод программы p на язык M. Итак, связав интерпретатор (написанный на языке M) с исходной программой на языке L, получили ее перевод на M.

Кратко это можно выразить так: специализация интерпретатора по программе дает ее перевод.

[Подумайте, что в общем случае можно сказать о качестве полученного перевода - скорости работы, объеме памяти; а что - о скорости перевода?]

4.4.3.2. Специализация специализатора

Итак, при различных i специализатор дает переводы с разных языков. Нетрудно теперь догадаться, что при фиксированном i специализатор s представляет собой компилятор с языка L на язык M. Ведь, как мы видели, в этом случае он по заданной p получает ее перевод p'. Действительно, посмотрим, что такое s(s,i)? Вновь применяя (**), получим

s(s,i)(p) = s(i,p).

Но ведь s(i,p) это p', перевод программы p на язык M! Так что s(s,i) - (написанный на M) это компилятор KLM с языка L на язык M.

Кратко выразим это так: специализация специализатора по интерпретатору дает компилятор. Или еще короче: автоспециализация по интерпретатору дает компилятор.

[Снова есть повод подумать о возможном качестве компилятора и затратах на его получение в общем случае].

4.4.3.3. Двойная автоспециализация

Теперь нетрудно догадаться, что специализатор может выступать и в роли суперкомпилятора. Ведь по заданному интерпретатору i (который можно считать описанием языка L), специализатор выдает компилятор с языка L язык M. Действительно, посмотрим, что такое s(s,s)? Опять применяя (**), получим

s(s,s)(i) = s(s,i).

Но ведь s(s,i) = KLM! Так что s(s,s) - это действительно суперкомпилятор над языком M (в свою очередь написанный на M).

Кратко выразим это так: двойная автоспециализация дает суперкомпилятор.

[Нельзя ли получить что-либо интересное тройной автоспециализацией?

Подсказка. А что если подставлять различные воплощения специализатора s?]

Три последовательных применения специализатора удобно наглядно выразить следующей серией соотношений

s(s,s)(i)(p)(d) = s(s,i)(p)(d) = s(i,p)(d) = i(p,d) = r.

Другими словами, s(s,s) воспринимает описание языка L (т.е. i) и выдает компилятор s(s,i), который в свою очередь воспринимает исходную программу p на языке L и выдает ее перевод s(i,p), который уже воспринимает исходные данные d и выдает результат r.

Таким образом, мы убедились, что абстракция связывания (точнее, частичное связывание) позволяет с единых позиций рассмотреть важнейшие понятия теории трансляции и вывести полезные закономерности. Именно, связав i с p, получили перевод; связав s с i, получили компилятор; связав s с s - суперкомпилятор.

[Строго говоря, мы имеем здесь дело не с суперкомпилятором, а с более универсальной программой.

Вопрос. В чем это проявляется?].

Замечание о сущности трансляционных понятий. Хотя непосредственное практическое значение этих закономерностей пока проблематично, они помогают увидеть заманчивые перспективы, а также четче выделять понятия.

Действительно, обычно отличие, скажем, интерпретации от компиляции формулируют несколько расплывчато. Говорят, что интерпретатор воспринимает исходную программу вместе с исходными данными и выполняет ее последовательно "шаг за шагом", в соответствии с операционной семантикой языка L.

[Написание интерпретаторов на машинных или ранее реализованных языках - хорошо известный, естественный и для многих целей удобный способ реализации ЯП. Для некоторых из них (Лисп, Апл, Бейсик) - в некотором смысле единственный способ полной реализации. Это справедливо для всех языков, в которых программа может меняться в процессе исполнения - только "шаг за шагом" и можно уследить за таким изменением].

Когда говорят о компиляции, подразумевают перевод всей программы как целого, без учета конкретных исходных данных, с исходного языка L на объектный язык M. С конкретными исходными данными исполняется уже результат такого перевода.

Такого рода содержательные различия, конечно, существенны, однако значительная их часть улавливается на формальном уровне, нам теперь вполне доступном. Ведь интерпретатор - это форма с двумя аргументами, а компилятор - с одним. Интерпретатор - это (ограниченный) аппликатор, а компилятор - это преобразователь программ (сохраняющий их смысл).

Вот пример пользы от рассмотрения языковых концепций (связывания) с математической позиции.

Важно понимать, что формальные преобразования специализатора в компилятор и суперкомпилятор не отражают некоторых содержательных аспектов этих понятий. Обычно компилятор применяют ради повышения скорости работы переведенных программ по сравнению с интерпретацией. Специализатор же в общем случае может выдать остаточную программу, состоящую в сущности из интерпретатора и обращения к нему. В таком случае неоткуда ждать выигрыша в скорости. При попытках "оптимизировать" такую программу за счет раскрытия циклов и т.п. она может стать непомерно длинной. Аналогичные соображения касаются и суперкомпилятора. Тем не менее в указанном направлении получены обнадеживающие результаты для частных видов специализаторов.

Не до конца улавливается приведенными соотношениями и сущность компиляции. Она в переводе на другой язык, на котором может оказаться вовсе невозможно или очень невыгодно писать интерпретатор исходного языка L (например, это невозможно делать на небольшой встроенной бортовой машине). А ведь в наших соотношениях все программы (кроме p) написаны на объектном языке M. Сказанное не означает, что в подобных случаях непригодна математическая позиция. Просто нужны и другие математические модели компиляции. Например, проекционная, где компилятор рассматривается как реализация проекции (отображения языка L на M), а не как специализация написанного на M интерпретатора (последнего может и не существовать).

Конец замечания.

На этом закончим обсуждение связывания как самостоятельной абстракции. Как видим, оно оказалось весьма емким, глубоким понятием, взаимодействующим со многими концепциями программирования.

4.5. Принцип цельности (согласования абстракций)

Считая достаточно обоснованной самостоятельную ценность каждой из трех выделенных абстракций, продемонстрируем на их примере один весьма общий принцип проектирования, который назовем принципом цельности (согласования абстракций). Проявляется он при проектировании самых различных объектов, но нас интересует прежде всего как принцип проектирования программ и ЯП. Точнее, нас интересует скорее проявление общего принципа цельности, который в программировании чаще всего выглядит как принцип согласования создаваемых абстракций. Поэтому будем называть этот принцип и принципом цельности, и принципом согласования (абстракций) в зависимости от контекста.

Называют его также принципом концептуальной целостности (хотя лучше говорить "принцип цельности").

Суть его в том, что детали проекта в идеале должны быть следствием относительно небольшого числа базисных, ключевых решений. Другими словами, в цельном проекте большинство деталей можно предсказать, зная базисные решения. Содержательно это означает, что проект выполнен на основе цельной концепции, единого замысла, а не представляет собой нагромождение случайностей.

Покажем, как принцип цельности проявляется на трех уровнях рассмотрения программного проекта, два из которых - языковые.

4.5.1. Первый уровень - собственно программа

Сначала - несколько совсем общих соображений. Проектирование - это сочетание абстракции и конкретизации (принимая конкретное проектировочное решение, тем самым одновременно вводят абстракции нижнего уровня, предназначенные для реализации принятого решения). Поэтому цельная концепция в идеале должна воплощаться согласованными абстракциями, а отсутствие таковой проявляется в их несогласованности.

Согласованность (абстракций) понимается как удобство их совместного использования для удовлетворения определяющих потребностей.

Возвратимся к трем выделенным абстракциям. [Иметь с ними дело приходится независимо от того, насколько сознательно они выделяются в технологическом цикле проектирования программ]. Покажем, как принцип согласования позволяет выработать естественные критерии качества языковых конструктов, обеспечивающих работу с данными, операциями и связыванием.

В соответствии с принципом технологичности (который справедлив не только для ЯП, но и для создаваемых с их помощью программ), выделяемые абстракции призваны обслуживать определенные технологические потребности. Рассмотрим известный технологический прием проектирования - метод пошаговой детализации (который мы уже не раз применяли) и попытаемся выявить критерии соответствия данных, операций и

связывания потребностям этого метода. Тем самым определяющими технологическими потребностями послужат потребности пошаговой детализации. Другими словами, цельность концепции, единство замысла будет оцениваться соответствием языковых конструктов потребностям выбранного технологического метода.

[Этот метод лежит в основе многих технологий программирования, ориентированных на создание ясных, надежных и эффективных программ, обладающих четкой структурой. Именно такую цель ставит перед собой структурный подход к программированию [7], одной из компонент которого служит пошаговая детализация].

Напомним: применяя этот метод, мы исходили из достаточно ясно сформулированной, хорошо понятной постановки задачи. Однако хорошо известно, что человек способен удерживать в памяти для совместного анализа обычно не более семи независимых объектов - это знаменитое число Ингве. Приходим к выводу, что в понятной постановке задачи компонент мало. Следовательно, они содержательные, крупные, имеющие непосредственную связь с сутью решаемой задачи. Итак, операций в постановке задачи мало, они крупные и содержательные. Но тогда и операнды у этих операций обладают теми же свойствами. И их связывание должно опираться на естественные средства доступа к таким емким операциям и данным. По мере детализации операций менее емкими становятся и операнды, и средства связывания.

Итак, в процессе пошаговой детализации свойства данных, операций и связывания должны на каждом шаге быть взаимно согласованными.

Следовательно, чтобы обеспечить технологические потребности пошаговой детализации, языковые конструкты должны обслуживать согласование указанных свойств на каждом шаге детализации.

4.5.2. Второй уровень - средства программирования

Мы пришли к важному критерию качества языковых конструктов: в хорошем ЯП конструкты, обслуживающие определение и использование каждой из упомянутых абстракций, должны быть согласованы (ведь хороший язык должен поддерживать метод пошаговой детализации).

4.5.3. Третий уровень - средства развития

Подчеркнем важный момент. Содержательные абстракции на каждом шаге детализации зависят от решаемой задачи. Как уже говорилось, для работы в конкретных прикладных областях создаются проблемно-ориентированные языки (ПОЯ). Это и есть задача, решаемая с помощью базового языка. Решать ее также естественно методом пошаговой детализации (иногда его называют методом абстрактных машин).

Следовательно, в базовых языках должны быть согласованными между собой и средства развития всех трех разновидностей абстракций. Другими словами, в ЯП, претендующем на универсальность, принцип цельности в идеале призван работать как при проектировании базиса ЯП, так и средств его развития.

Подчеркнем, что принцип цельности носит неформальный, почти эстетический характер. Способность оценивать уровень цельности языков и программ приходит только с опытом. Однако понятие об этом трудно постигаемом принципе исключительно полезно в программистской деятельности. Поэтому разговор о нем здесь представляется уместным.

4.5.4. Принцип согласования и нормальные алгоритмы

Чтобы лучше прочувствовать изложенный принцип, попробуем оценить на его основе исходную (немодифицированную) модель Маркова (язык нормальных алгоритмов). В этой модели всякая программа представляет собой линейную последовательность однородных операций (подстановок) над линейной последовательностью однородных данных (символов). Связывание также однородно - просматривается последовательность операций и данных, связывается конкретная операция-подстановка с подходящей последовательностью данных, выполняется эта операция и затем - новый цикл связывания.

Отметим очевидную согласованность всех трех абстракций. Но эта согласованность обслуживает не пошаговую детализацию, а простоту исследования свойств нормальных алгоритмов (именно к этому и стремился их изобретатель). Вместе с тем очевидно, что с требованиями пошаговой детализации как основные абстракции, так и средства развития в этой модели не согласованы (В чем это проявляется?).

Таким образом, понятие согласованности абстракций не абсолютно, а относительно, зависит от того, какие технологические потребности признаются определяющими.

Замечание. Известный тезис нормализации утверждает, что всякий алгоритм можно заменить эквивалентным нормальным алгоритмом. Но важно хорошо понимать смысл слова "можно" в этом тезисе. Можно заменить, если принять абстракцию потенциальной осуществимости, отвлечься от таких "несущественных деталей", как необходимые для этого ресурсы. Любой, кто писал нормальные алгоритмы, прекрасно понимает, что ни одной реальной программы непосредственно в исходной модели Маркова нельзя даже написать - она практически наверняка будет неправильной и отладить ее будет невозможно в обозримое время (даже если предположить сколь угодно высокую скорость выполнения самих марковских подстановок). Ведь все выделяемые при программировании абстракции, как данных, так операций и связывания, нужно подразумевать или хранить вне программы. Поэтому единственный разумный путь к практической осуществимости программирования на языке нормальных алгоритмов - моделировать на этом языке другой, более совершенный в технологическом отношении язык.

Конец замечания.

4.5.5. Принцип цельности и модель Бэкуса

Посмотрим с точки зрения принципа цельности на модель Бэкуса. Вспомним: рассматривая структуру операций на каждом шаге детализации, мы были вынуждены отдельно рисовать (внеязыковыми средствами) структуры данных, соответствующие этому шагу (например, матрицы в примере с матричным умножением). В этой модели извлечь сведения о структуре обрабаьываемых данных из текста программы - нетривиальная задача. Связывание конкретных данных с конкретными операциями также закодировано в структуре операций. Аналогичными недостатками с точки зрения согласования абстракций с потребностями пошаговой детализации обладают, конечно, все классические ЯП с неразвитыми средствами определения структур данных - и Алгол, и Фортран, и Апл (в этих ЯП единственная структура данных - массив, посредством которой и приходится моделировать все остальные структуры).

В более современных ЯП принцип согласования абстракций (как между собой, так и с важнейшими технологическими потребностями) осознан и учтен в гораздо большей степени. Продемонстрируем это положение на примере языка Ада. Одновременно продолжим строить модель "максимального" базового языка индустриального программирования (модель А).

4.6. Пример: пошаговая детализация средствами Ады

Рассмотрим следующую задачу.

Содержательная постановка. Необходимо предоставить пользователю комплекс услуг, позволяющих ему моделировать сеть связи. Пользователь должен иметь возможность изменять сеть (добавлять и убирать узлы и линии связи), а также получать информацию о текущем состоянии сети.

Требования к реализации. Представление сети должно быть скрыто от пользователя, во-первых, чтобы он не мог нарушить целостность сети (чтобы не было линий, ведущих "никуда" и (или) "ниоткуда", а также изолированных узлов). Во-вторых, предполагается развивать возможности моделирования (скажем, отражать стоимость связей). Важно, чтобы при этом пользователю не потребовалось изменять готовые программы.

[Если не регламентировать доступ пользователя к представлению сети, то изменить это представление при развитии комплекса услуг окажется почти невозможным - ведь пользователю придется менять те программы, в которых отражены особенности старого представления].

4.6.1. Первый шаг детализации: уточнение постановки задачи в терминах языка Ада

Так как речь идет не об алгоритме, а о предоставлении пользователю комплекса услуг, в Ада-терминах естественно отобразить этот комплекс на совокупность "логически связанных" объектов, в данном случае - связанных по меньшей мере совместным использованием. Другими словами, первое наше решение состоит в том, чтобы создавать ПАКЕТ, а не подпрограмму или задачу. Вспоминая, что разделение спецификации и тела пакета позволит скрыть от пользователей пакета детали реализации (в частности, представление сети, в полном соответствии с требованиями), получаем еще одно подтверждение, что решение правильное.

Итак, создаем ПАКЕТ. Нужно придумать ему название, выражающее назначение предоставляемого комплекса услуг. Попробуем "сеть". Нехорошо. Повидимому, так лучше называть тот объект, который будет моделироваться, и чье представление нужно

скрыть в теле нашего пакета. Попробуем "моделирование сети связи". Лучше, но слишком конкретно. Хотя в постановке задачи и требованиях речь идет именно о моделировании сети связи, однако специфика связи (кроме самой сети) ни в чем не отражена (нет и речи о пропускной способности каналов, классификации сообщений и т.п.), да и специфика моделирования не затронута (никаких моделей отправителей, получателей и т.п.). Скорее мы собираемся предоставить лишь комплекс услуг по управлению сетью. Так и назовем пакет: "управление_сетью".

[Точное название настраивает на то, чтобы в пакете не было лишнего, а пользователю помогает применять наш комплекс и в других областях, не обязательно для моделирования сетей связи].

4.6.2. Второй шаг детализации

Теперь нужно написать спецификацию пакета, объявив все объекты, с которыми сможет работать пользователь.

```
0.  with параметры_сети; use параметры_сети;
1.  package управление_сетью is
2.    type имя_узла is new INTEGER range 1 .. макс_узлов;
3.    type число_связей is new INTEGER range 0..макс_связей ;
4.    type индекс_узла is new INTEGER range 1..макс_связей;
5.    type перечень_связей is array (индекс_узла) of имя_узла;
6.    type связи is
7.      record
8.        число : число_связей := 0;
9.         узлы : перечень_связей;
10.     end record ;
11.   --операции над сетью
12.   procedure вставить (узел : in имя_узла);
13.   procedure удалить (узел : in имя_узла);
14.   procedure связать (А_узел, В_узел : in имя_узла);
15.   --сведения о текущем состоянии сети
16.   function узел_есть (узел : имя_узла) return boolean;
17.   function все_связи (узел : имя_узла) return связи;
18. end управление_сетью;
```

4.6.3. Подробности второго шага

Текст спецификации пакета с названием "управление_сетью" при первоначальном знакомстве может показаться непонятным. Во всяком случае не верится, что он получен одним шагом детализации. Действительно, величина шага зависит как от привычки проектировщика, так и от свойств языка. Ведь в общем случае далеко не каждый мелкий шаг поддерживается подходящим законченным языковым конструктом. Например, в Аде шаги нельзя дробить сколь угодно мелко хотя бы потому, что действует

Правило последовательного определения : при очередном определении можно использовать только предопределенные, внешние или ранее объявленные имена.

Но при пошаговой детализации нельзя заранее объявить те имена, которые понадобятся - они попросту неизвестны. Когда проектируют совокупность модулей, это не помеха (порядок модулей не существенен). А вот внутри модулей правило последовательного определения мешает пошаговой детализации [особенно внутри пакетов; почему?] . Приходится либо применять средства, выходящие за рамки Ады (например, псевдокод [7]), либо записывать пакет "с конца к началу" - этот порядок с учетом правила последовательного определения лучше отражает последовательность появления имен при пошаговой детализации.

[С точки зрения принципа технологичности любые несоответствия языка потребностям пошаговой детализации служат источником "точек роста", намечают направление развития либо самого языка, либо других связанных с ним технологических инструментов. Для Ады, в частности, разрабатываются специальные средства поддержки пошагового конструирования программ].

Упражнение. Укажите внешний эффект (исходные данные и результаты) хотя бы одного из таких средств.

Упражнение (повышенной сложности). Разработайте проект хотя бы одного такого средства; проект комплекса таких средств. Реализуйте проект для ограниченного набора допустимых адовских конструктов.

Итак, проявим более мелкие шаги проектирования нашего пакета.

Шаг 2.1 (строка 17). Объявляем функцию с названием "все_связи", формальным параметром с названием "узел" (значениям этого параметра приписан тип с названием "имя_узла") и результатом, которому приписан тип с названием "связи".

[Ниже будем писать короче: функцию "все_связи" с параметром "узел" типа "имя_узла" и результатом типа "связи"].

Эта функция дает возможность узнавать о текущем состоянии сети (точнее, о связях одного узла). Обратите внимание, пока совершенно неясно, что такое "имя_узла" и "связи". Это лишь названия, отражающие роль в создаваемом комплексе услуг тех объектов, которые еще предстоит воплотить в программе.

Шаг 2.2 (строка 16). Нехорошо запрашивать связи узла, не зная, имеется ли он в сети. Поэтому (продолжая предоставлять средства узнавать о состоянии сети) объявляем функцию узел_есть с параметром "узел" типа имя_узла и результатом логического типа (BOOLEAN).

Замечание. Обратите внимание, мы записываем только формальные СПЕЦИФИКАЦИИ (заголовки) функций. Содержащихся в них сведений достаточно, чтобы можно было (столь же формально) написать вызов такой функции. Но, во-первых, рано или поздно придется написать ТЕЛО функции (сделаем это в ТЕЛЕ пакета). Во-вторых, нужно как-то сообщить пользователю, что же содержательно делает объявляемая функция.

Например, из объявления в строке 16 пользователь поймет, что, задав функции узел_есть аргумент типа имя_узла, он получит в качестве результата истину или ложь. Но откуда ему узнать, что истина соответствует случаю, когда узел с указанным именем есть в сети, а ложь - когда его нет. Название функции лишь намекает на такое истолкование. Конечно, названия должны быть мнемоничными и помогать запоминать смысл программных объектов, но они не могут заменить точных сведений.

Ада не предоставляет специальных средств для полного и точного описания внешнего эффекта модуля. Ведь адовские спецификации рассчитаны прежде всего на исполнителя, на компьютер, а отнюдь не на пользователя. Поэтому, как и в случае с другими ЯП, проектирование Ада-модуля следует сопровождать проектированием точного описания его внешнего эффекта (применяя при необходимости средства, выходящие за рамки ЯП). [Некоторые экспериментальные языки предоставляют встроенные средства соответствующего назначения].

Часто бывает необходимо параллельно создавать и описание внешнего эффекта, специально ориентированное на пользователей. Эта так называемая пользовательская документация принципиально отличается от описаний, рассчитанных на автомат-транслятор (именно таковы описания на ЯП) или человека-реализатора (отличается по структуре документов, стилю изложения, выделяемым свойствам объектов и т.п.).

С точки зрения пользовательской документации на программное изделие, ЯП всегда выступает в роли инструмента реализации. Он тем лучше, чем проще объяснить пользователю назначение выделенных программных компонент и чем ими самими удобнее и дешевле пользоваться. Назовем соответствующий критерий качества языка критерием выделимости.

По выделимости Ада превосходит, скажем, Алгол 60 или Бейсик, так как позволяет адекватно оформлять не только компоненты-функции, но и компоненты-данные, и компоненты-задачи, и компоненты более "тонкого" назначения. Другими словами, Ада выигрывает в выделимости потому, что предоставляет более развитые средства абстракции и конкретизации.

[Причем здесь абстракция-конкретизация?].

Конец замечания.

Шаг 2.3 (строки 12-14). Предоставляя средства для изменения сети, определяем три процедуры: вставить, удалить и связать (параметры у них типа имя_узла).

С одной стороны, после этого шага мы можем быть довольны - внешние требования к проектируемому комплексу услуг в первом приближении выполнены. С другой стороны, появилась необходимость определить упомянутые на предыдущих шагах типы данных.

[Так всегда - завершив абстракцию и конкретизацию верхнего уровня, создаем почву для аналогичной работы на нижнем уровне и наоборот].

Шаг 2.4 (строка 2). Определяем тип имя_узла. Этот тип уже частично нами охарактеризован (где?) - данные этого типа могут служить аргументами всех процедур и функций, объявленных в нашем пакете. Другими словами, этот тип уже охарактеризован по фактору (применимых) операций. Выписывая его явное определение,

мы характеризуем данные этого типа по фактору изменчивости - указываем, что диапазон (range) их возможных значений - целые числа от 1 до числа макс_узлов (пока еще не определенного). Одновременно мы относим объявляемый тип к категории целых числовых типов и тем самым завершаем его характеристику по фактору применимых операций (в Аде для целых типов предопределены обычные операции целой арифметики - сложение "+", вычитание "-", умножение "*" и др.).

Шаг 2.5 (строки 6-10). Определяем тип "связи" результата функции все_связи. Замысел в том, чтобы эта функция сообщала число связей указанного узла и перечень связанных с ним узлов. В Алголе или Фортране не могло быть функций, которые в качестве результата выдают составной объект. В Аде можно ввести составной тип, объекты которого либо состоят из однотипных подобъектов - являются массивами, либо из разнотипных - являются записями. Результат задуманной нами функции все_связи - пара разнотипных объектов (число и узлы). Другими словами, это запись, первое поле которой называется "число", а второе - "узлы". Тип значений первого поля назван "число_связей", второго - "перечень_связей".

В этом же объявлении указано, что при создании объекта типа "связи" его поле "число" получает начальное значение 0. Это так называемая ИНИЦИАЛИЗАЦИЯ объектов, которой нет, скажем, в Алголе, но для знающих Фортран - дело привычное (вспомните объявление начальных данных DATA).

Итак, на шаге 2.5 снова кое-что определилось, но опять появились новые имена - число_связей и перечень_связей.

Шаг 2.6 (строка 5). Перечень_связей определяем как регулярный тип одномерных массивов, составленных из объектов типа имя_узла, доступ к которым - по индексам типа индекс_узла.

Шаг 2.7 (строка 4). Индекс_узла определяем как тип объектов, значения которых лежат в диапазоне целых чисел от 1 до макс_связей (максимального допустимого числа связей у узла в сети - оно пока не определено).

Шаг 2.8 (строка 3). Число_связей определяем как тип объектов, значения которых лежат в диапазоне целых чисел от 0 до макс_связей . Как видите, этот тип похож на предыдущий, но отличается своей ролью и диапазоном значений.

Остались неопределенными только имена макс_узлов и макс_связей. Их неудобно фиксировать в том же модуле - ведь они могут изменяться в зависимости от потребностей пользователя и наличных ресурсов. Поэтому будем считать, что эти имена определены во внешнем для нашего модуля контексте, а именно в пакете с именем "параметры_сети". Доступ к этому контексту из модуля управление_сетью обеспечивается его нулевой строкой.

Это так называемое УКАЗАНИЕ КОНТЕКСТА. После ключевого слова with в нем перечисляются пакеты, объявления из которых считаются имеющими силу в модуле, непосредственно следующем за таким указанием.

Пакет параметры_сети можно определить, скажем, так:

```
1. package параметры_сети is
2.   макс_узлов : constant INTEGER := 100;
3.   макс_связей: constant INTEGER := 8;
4. end параметры_сети;
```

Тем самым макс_узлов определено в качестве ПОСТОЯННОЙ целого типа со значением 100, а макс_связей - в качестве постоянной того же типа со значением 8. Значения постоянных нельзя менять при исполнении программы (вот еще один элемент прогнозирования и контроля в Аде).

На этом завершим первоначальное знакомство со спецификацией пакета.

4.6.4. Замечание о принципе цельности (согласования абстракций)

Взглянем на шаги с 2.1 по 2.8 с точки зрения потребности согласовывать абстракции.

Выделив на шаге 2.1 операционную абстракцию - функцию все_связи, мы немедленно ощутили потребность обозначить классы возможных аргументов и результатов этой функции, не занимаясь их детальной проработкой. Если бы приходилось работать на Фортране, Алголе или Бейсике, сделать это оказалось бы невозможным - непосредственно подходящих предопределенных типов данных в этих ЯП нет, а возможность строить новые типы также отсутствует. Скорее всего пришлось бы нарушить естественный порядок детализации и сначала придумать способ представлять "связи" некоторым массивом, а затем учесть, что в этих ЯП функции не вырабатывают

результаты-массивы (только скаляры) и представить нужную абстракцию не функцией, а процедурой. Важно понимать, что такого рода отклонения запутывают логику программы, провоцируют ошибки, затрудняют отладку и т.п.

Лучшее, что можно сделать в этом случае - выйти за рамки используемого ЯП и фиксировать шаги детализации на частично-формализованном псевдокоде. О применении такого псевдокода в структурном подходе к программированию на классических ЯП можно прочитать, например, в уже упоминавшейся книге [7].

В Аде мы смогли провести шаг детализации полностью в рамках языка. Причем, вводя операционную абстракцию, были вынуждены воспользоваться средствами определения абстракций другого рода - абстракций данных. Точнее говоря, на шаге 2.1 мы воспользовались лишь тем, что в Аде можно вводить новые абстракции данных и можно вводить для этих абстракций подходящие названия.

Обратите внимание, мы ввели не названия отдельных объектов данных (только так и можно в классических ЯП), а именно названия целых классов (типов) обрабатываемых объектов.

Определения типов мы также имели возможность вводить по шагам, вполне аналогично тому, как в классических ЯП вводят операционные абстракции, выделяя нужные процедуры. На шаге 2.1 обозначили новый тип "связи"; на шаге 2.5 уточнили его строение, но потребовались названия типов число_связей и перечень_связей. На шагах 2.6 и 2.8 уточнили строение этих типов, но остались неопределенными макс_узлов и макс_связей. Наконец, уточнили их характеристики и даже значения.

Четкость пошаговой детализации поддерживалась языковыми средствами связывания. Можно было не заботиться о реализации процедур и функций - ее можно определить позже, в теле пакета - средства связывания обеспечат согласованное использование спецификаций и тел процедур. В классических ЯП так поступить нельзя, пришлось бы опять выходить за рамки языка или нарушать порядок детализации и выписывать тела процедур (а это не всегда можно сделать, еще не зная структуру используемых данных).

Итак, должно быть видно, как принцип согласования основных абстракций (между собой и с потребностями пошаговой детализации) воплощен в Аде. Во-первых, согласованность основных абстракций действительно требовалась, и ,во-вторых, Ада необходимые выразительные средства предоставляет.

Принцип цельности дает основания ввести критерий технической оценки языка, который можно назвать критерием цельности: язык тем лучше, чем ближе он к идеалу с точки зрения принципа согласования абстракций. Ясно, что с точки зрения технологии пошаговой детализации Ада превосходит классические ЯП и по этому критерию.

Упражнение. Пользуясь критерием цельности, оцените другие известные ЯП (Паскаль, Ассемблер, рефал).

4.6.5. Замечания о конструктах

Пройдемся по написанным фрагментам программы еще раз. Теперь поговорим о строении, смысле и назначении использованных конструктов.

В целом мы написали две СПЕЦИФИКАЦИИ ПАКЕТА. Отличительный признак этого конструкта - ключевое слово package (пакет). Спецификация пакета содержит объявления имен, которые становятся доступными при использовании пакета посредством указания контекста (например, объявления из пакета параметры_сети становятся доступны в пакете управление_сетью за счет указания контекста with параметры_сети).

Спецификацию пакета можно оттранслировать и поместить в ТРАНСЛЯЦИОННУЮ БИБЛИОТЕКУ. Получится модуль, пригодный для связывания (посредством указаний контекста) с другими (использующими его) модулями в процессе их трансляции и загрузки.

Пакет может состоять из одной спецификации или из спецификации и тела. Например, для пакета параметры_сети тело не требуется, в отличие от пакета управление_сетью (как Вы думаете, почему?).

Если пакет состоит из двух частей (спецификации и тела), то выполнять программу, в которой отсутствует одна из них, нельзя. Однако для трансляции использующих модулей достаточно одной только спецификации используемого пакета. Итак, создавать и транслировать спецификации пакетов можно отдельно от их тел, но исполнять - только совместно с телами пакетов.

В спецификацию пакета входит совокупность ОБЪЯВЛЕНИЙ.

В каждой из строк 2-3 спецификации пакета параметры_сети находится ОБЪЯВЛЕНИЕ ПОСТОЯННОЙ, точнее, ОБЪЯВЛЕНИЕ ЧИСЛА. Это одна из разновидностей

ОБЪЯВЛЕНИЯ ОБЪЕКТА. Назначение всякого объявления объекта - связать имя с характеристиками поведения объекта, названного этим именем. Поэтому обязательными компонентами объявления служат само вводимое имя, ключевые слова, отличающие разновидность объявления и тем самым характеризующие поведение объявляемого объекта в целом, и компоненты-параметры, уточняющие характеристики поведения.

Так, в строке 2 объявляемое имя - макс_узлов, уточняющие параметры - имя типа (INTEGER) и константа 100 (изображение целого числа). Полное объявление связывает с именем объекта макс_узлов тип INTEGER и константу 100 как характеристику поведения объекта. Попросту говоря, имя макс_узлов начинает обозначать константу 100 типа INTEGER.

[Чтобы понять, зачем нужно обозначать константы именами, достаточно представить себе программу, где константа 100 используется в десяти местах, и допустить, что нужно изменить ее значение на 200. Тогда в нашей спецификации достаточно изменить одну цифру в строке 2, а иначе пришлось бы изменять десять мест с риском где-нибудь заменить не ту константу (или не на то значение). Так объявления постоянных способствуют надежности ада-программ].

Перейдем к спецификации пакета управление_сетью. В каждой из ее строк 2,3 и 4 мы написали ОБЪЯВЛЕНИЕ ТИПА. В нем всегда указывают, как совокупность значений объявляемого типа образуется из совокупности значений ранее известных типов (предопределенных или ранее объявленных). В нашем случае в строке 2 указано, что новый тип имя_узла образован из предопределенного типа INTEGER (является типом, ПРОИЗВОДНЫМ от типа INTEGER), причем данные типа имя_узла могут обозначать только целые из диапазона от 1 до макс_узлов. В строке 3 и 4 аналогичные сведения сообщаются о типах число_связей и индекс_узла, только здесь указаны другие диапазоны.

Зачем нужны объявления типов, Вы уже частично знаете. В том модуле, где будет использоваться пакет управление_сетью, можно будет объявить переменную (скажем, А) типа имя_узла и переменную (скажем, В) типа число_связей. Так вот переменную А можно указать в качестве аргумента процедуры "вставить" или "связать", а переменную В - нельзя. Это ошибка, обнаруживаемая при трансляции, В сущности, ради такого контроля и нужны объявления типов, прогнозирующие поведение (возможные роли) соответствующих данных.

В строке 5 - объявление типа, но на этот раз не скалярного (как в строках 2-4), а СОСТАВНОГО, точнее РЕГУЛЯРНОГО. Указано, как значения нового типа перечень_связей образуются из значений типов имя_узла и индекс_узла. Именно, значения типа перечень_связей - это одномерные (так как указан лишь один диапазон индексов) МАССИВЫ, компонентами которых служат значения типа имя_узла, а доступ к этим компонентам - по индексам типа индекс_узла.

В строках 6-10 - также объявление составного типа, но на этот раз - КОМБИНИРОВАННОГО. Указано, что значениями нового типа "связи" могут быть любые ЗАПИСИ с двумя полями. Первое поле с именем "число" и допустимыми значениями типа "число_связей" (при создании записи этому полю присваивается начальное значение 0). Второе поле с именем "узлы" типа перечень_связей.

Если в модуле, использующем наш пакет, объявлена переменная, скажем, X типа "связи" и I типа индекс_узла, то через

X.узлы(I)

обозначается значение типа имя_узла, которое служит I-ой компонентой поля "узлы" переменной X.

Строки 11 и 15 - это примечания, не влияющие на смысл модуля. Примечанием считается остаток любой строки, начинающийся с двух минусов.

В строках 12-14 - ОБЪЯВЛЕНИЯ ПРОЦЕДУР. В скобках указаны имена (названия) формальных параметров, их типы и РЕЖИМ использования (in - только для чтения - ВХОДНЫЕ параметры; out - только для записи - ВЫХОДНЫЕ; in out - и для чтения, и для записи - ОБНОВЛЯЕМЫЕ). Режим in напоминает вызов параметров значением в Алголе, in out - вызов ссылкой в Фортране, out - точного аналога в этих ЯП не имеет.

В строках 16-17 - ОБЪЯВЛЕНИЯ ФУНКЦИЙ. Отличаются от процедур ключевым словом function (а не procedure) и указанием типа результата (после return). Режим параметров не указывается, потому что для функций всегда подразумевается режим in (все параметры функций - только входные, т.е. функции не могут менять значения своих аргументов).

Обратите внимание, в спецификации пакета указаны лишь спецификации (заголовки) процедур и функций. В таком случае их тела следует поместить в ТЕЛО ПАКЕТА, о котором пойдет речь в следующем разделе.

На этом закончим предварительное знакомство с использованными Ада-конструктами, а заодно и раздел о трех основных абстракциях программирования.

4.6.6. Как пользоваться пакетом управление_сетью

В предыдущем подразделе мы спроектировали пакет управление_сетью, на примере которого начали изучать принцип раздельного определения, реализации и использования абстракций. Посмотрим, как пользоваться спроектированным комплексом услуг.

Пусть нужно построить сеть из пяти узлов (13, 33, 25, 50, 90) и шести дуг (13, 33), (33, 25), (33, 50), (33, 90), (13, 50) и (25, 90). [Нарисуйте такую сеть].

Это можно сделать следующей процедурой построение_сети.

```
1.  with управление_сетью ;
2.  use управление_сетью ;
3.  procedure построение_сети is
4.  begin
5.     вставить(узел => 33) ;
6.     вставить(узел => 13) ;
7.     связать(А_узел => 33, В_узел => 13) ;
8.     вставить(25) ; связать(33,25) ;
9.     вставить(50) ; вставить(90) ; связать(33,50) ;
10.    связать(33,90) ; связать(13,50) ; связать(25,90) ;
11. end построение_сети ;
```

Первые две строки позволяют в процедуре построение_сети пользоваться услугами, предоставляемыми пакетом управление_сетью так, как будто все услуги объявлены непосредственно перед третьей строкой.

Как уже сказано, строка с ключевым словом with называется УКАЗАНИЕМ КОНТЕКСТА (with clause). Указание контекста делает видимыми (доступными по ПОЛНЫМ именам) все услуги, объявленные в пакетах, перечисленных вслед за with. Например, к процедуре "вставить" можно было бы обратиться так:

```
управление_сетью.вставить(...),
```

а объявить переменную типа "связи" можно так:

```
А : управление_ сетью.связи ;
```

Строка с ключевым словом use называется УКАЗАНИЕМ СОКРАЩЕНИЙ (use clause). Это указание позволяет пользоваться видимыми именами, не предваряя их именем пакета. Так мы и поступили в процедуре построение_сети. Подчеркнем, что указание сокращений действует только для уже видимых имен. Его обязательно нужно предварять указанием контекста.

В первых трех операторах (строчки 5-7) явно указаны имена формальных параметров, которым соответствуют аргументы - это так называемый вызов по ключу. Его удобно применять при большом количестве параметров, а также когда не все параметры нужно задавать (когда значения остальных параметров выбираются ПО УМОЛЧАНИЮ, задаются заранее; например, при объявлении процедуры). К тому же вызов по ключу помогает чтению.

[Вопрос. Всегда ли вызов по ключу повышает надежность программы?]

Если теперь нужно напечатать сведения о построенной сети, то это можно сделать следующими операторами (будем считать, что доступны процедуры новая_строка (переход на новую строку при печати) и "печать" (целого числа или массива)).

```
новая_строка ;
for i in имя_узла loop
  if узел_есть(i) then
   печать(i) ;
   печать(все_связи(i).узлы) ;
  end if;
  новая_строка;
end loop;
```

Будет напечатано в нашем случае:

```
13  33  50
25  33  90
33  13  25  50  90
50  33  13
90  33  25
```

Обратите, внимание: тип имя_узла используется для указания диапазона изменения значений переменной цикла. В нашем случае тело цикла выполнится 100 раз.

4.6.7. Третий шаг детализации - тело пакета

До сих пор мы смотрели на наш комплекс услуг с точки зрения потенциального пользователя. Теперь настало время реализовать те услуги, которые мы объявили в спецификации пакета. В терминах Ады это означает, что нужно спроектировать ТЕЛО ПАКЕТА управление_сетью.

Создавать тело пакета будем также пошаговой детализацией.

Шаг 3.1. Неважно, с детализации какой процедуры или функции начинать - ведь ни одну из них нельзя написать прежде, чем не станет понятно, как представлена сама сеть, с которой нужно работать. Поэтому начнем с проектирования представления данных. Займемся представлением сети.

Есть много вариантов такого представления (таблица, список, перемешанная таблица и т.п.). Выберем представление сети массивом.

```
сеть : array (имя_узла) of запись_об_узле ;
```

Мы написали ОБЪЯВЛЕНИЕ ОБЪЕКТА. Как всякое объявление объекта, оно связывает имя ("сеть") с характеристиками того объекта данных, который в дальнейшем будет значением (денотатом) объявленного имени. В нашем случае этот объект - одномерный массив с коипонентами типа запись_об_узле, доступными по индексам типа имя_узла.

Шаг 3.2. Следует заняться уточнением того, как устроен объект типа запись_об_узле. Естественно считать, что это некоторая структура данных, куда вносятся сведения о том, включен ли узел в сеть и если да, то какие узлы с ним связаны. Объявим тип запись_об_узле.

```
type запись_об_узле is
  record
    включен : BOOLEAN := false;
    связан  : связи ;
  end record;
```

Итак, каждая запись об узле состоит из двух полей: поле с именем "включен" с начальным значанием false служит признаком включения узла в сеть, а поле с именем "связан" содержит все связи узла.

Шаг 3.3. Теперь все готово, чтобы заняться операциями над сетью. Начнем с функции узел_есть.

Уточним ее внешний эффект: она должна быть применима к любому объекту типа имя_узла и должна выдавать результат true, если узел с таким именем есть в сети и false в противном случае.

Мы сформулировали ее содержательный эффект. Такого рода сведения о функции узел_есть должны быть в пользовательской документации. Это необходимое для пользователя дополнение к спецификации (заголовку функции), укаанной в спецификации пакета в строке 18. Но сейчас нас интересует реализация функции. Поэтому следует обеспечить ее содержательный эффект в терминах реализационных, в частности, через представление сети (которое пользователю недоступно и даже может оказаться неизвестным). Было бы естественным выдавать в качестве результата просто значение поля "включен" записи об узле. Но для этого на всю остальную реализации пакета необходимо наложить единое требование (если угодно, определить дисциплину работы с этим полем): его значением в любой компоненте массива "сеть" после выполнения любого действия должно быть true, если узел есть в сети, и false в противном случае. При выполнении этого требования необходимый содержательный внешний эффект функции узел_есть обеспечивается следующим объявлением (определением):

```
function узел_есть(узел : имя_узла) return BOOLEAN is
begin
  return сеть(узел).включен ;
end узел_есть ;
```

[Обратите внимание: в объявлении функции повторена ее спецификация.]

ОПЕРАТОР ВОЗВРАТА (return) завершает исполнение тела функции, доставляя в качестве ее результата значение указанного выражения. В нашем случае это ВЫБОРКА (поля "включен" из записи, находящейся в массиве "сеть" по индексу, указываемому значением формального параметра "узел").

Шаг 3.4. Займемся реализацией функции все_связи. Содержательный внешний эффект - проявление связей узла. При соответствующей дисциплине работы с сетью ее реализация могла бы быть такой:

```
function все_связи(узел : имя_узла) return связи is
begin
  return сеть(узел).связан ;
end все_связи ;
```

[Вопрос. В чем должна состоять требуемая дисциплина?]

К такой функции можно обращаться лишь тогда, когда известно, что узел в сети есть, иначе можно выбрать неопределенные значения в полях "связан" и "узлы".

Шаг 3.5. Реализация процедуры "вставить" (с очевидным содержательным эффектом) может выглядеть так:

```
procedure вставить(узел : in имя_узла) is
begin
  сеть(узел).включен := true ;
  сеть(узел).связан.число := 0 ;
end вставить ;
```

Теперь займемся процедурами "удалить" и "связать". Они чуть сложнее за счет того, что нужно вносить изменения в несколько компонент массива "сеть".

Шаг 3.6. Содержательный эффект процедуры "удалить" очевиден: узел с указанным именем должен быть удален из сети и, с учетом требования поддерживать целостность сети, все связи, в которых он участвовал, должны быть ликвидированы.

Такого содержательного эффекта можно достичь многими способами. Здесь естественно воспользоваться тем, что пользователи лишены возможности непосредственно изменять "сеть" (скажем, явными присваиваниями этому массиву), они могут к нему добираться только посредством объявленных в спецификации пакета процедур и функций. Поэтому задача реализатора пакета - обеспечить согласованный внешний эффект объявленных услуг (при этом внутренний эффект процедур и функций можно варьировать).

Другими словами, действие процедуры "удалить" на массив "сеть" должно быть таким, чтобы функции узел_есть и все_связи выдали результаты, согласованные с содержательным представлением об отсутствии узла в сети. Один вариант реализации - присвоить false соответствующему полю "включен" и подправить поле "связан" во всех узлах, с которыми был связан удаляемый узел. Другой вариант - в этой процедуре поле "связан" не подправлять, но изменить реализацию функции все_связи так, чтобы перед выдачей своего результата она приводила поле "связан" в соответствие с полем "включен".

[Это и есть варианты упоминавшихся выше дисциплин работы с сетью.]

Рациональность выбора одного из вариантов не очевидна. Если часто удаляют узлы и редко просят их перечень, может оказаться выгодным второй вариант, иначе - первый. Оценить частоту запросов может оказаться делом непростым. Поэтому возможность менять реализацию (подстраиваясь к условиям эксплуатации), не меняя внешнего эффекта, может оказаться очень важной.

[Обратите внимание: вновь в действии принцип цельности. Не спроектировав представления данных, мы не могли начать проектировать процедуры. А теперь видим, что проектирование данных может зависеть от дисциплины взаимодействия операций.]

Выберем первый вариант реализации.

```
procedure удалить(узел : in имя_узла) is
begin
   сеть(узел).включен := false;
   for i in 1..сеть(узел).связан.число loop
     чистить(узел,сеть(узел).связан.узлы(i)) ;
   end loop ;
end ;
```

Понадобилась процедура "чистить", которая должна убрать в узле, указанном вторым параметром, связь с узлом, указанным первым параметром.

```
procedure чистить(связь : имя_узла, в_узле : имя_узла)  is
begin
  for i in 1..сеть(в_узле).связан.число loop
    if сеть(в_узле).связан.узлы(i) = связь then
      переписать(в_узле, после => i) ;
    end if ;
  end loop ;
end чистить ;
```

Осталось спроектировать процедуру "переписать" - она должна переписать связи в указанном узле, начиная с номера "после", и уменьшить на единицу общее число связей этого узла.

```
procedure переписать(в_узле : in имя_узла, после : in индекс_узла)
  is
  запись:связи renames сеть(в_узле).связан ;
begin
  запись.число := запись.число - 1 ;
  for j in после..запись.число loop
    запись.узлы(j) := запись.узлы(j+1) ;
  end loop ;
end переписать ;
```

Здесь мы впервые воспользовались ОБЪЯВЛЕНИЕМ ПЕРЕИМЕНОВАНИЯ, чтобы сократить имена и сделать их более наглядными. Этот же прием можно было применять и раньше. Напомним, что о диагностике ошибок мы пока не заботимся (предполагается, что перед применением процедуры "удалить" всегда применяется функция есть_узел, чтобы не применять удаление несуществующего узла).

"Запись" - это имя объекта типа "связи" [а именно, объекта сеть(в_узле).связан], локальное для процедуры "переписать". Общий вид объявления процедуры:

```
<спецификация процедуры> is
<локальные объявления> ;
begin
  <операторы> ;
end процедуры ;
```

Оборот for j in <диапазон> - это ОБЪЯВЛЕНИЕ УПРАВЛЯЮЩЕЙ ПЕРЕМЕННОЙ ЦИКЛА, область действия которой - до конца цикла. Внутри БАЗИСНОГО ЦИКЛА (от loop до end loop) j считается постоянной. Если диапазон пуст (это бывает, когда его правая граница меньше левой), базисный цикл не выполняется ни разу. Иначе он выполняется при всех последовательных значениях j из указанного диапазона, если только выполнение всего оператора цикла не будет досрочно завершено оператором выхода (exit).

В нашем случае все имена узлов из массива "узлы" с индексами от "после+1" до "число" перемещаются на позиции с предыдущим индексом. В результате массив "узлы" содержит все старые связи, кроме вычеркнутой, а их общее количество предварительно скорректировано (уменьшено на 1).

Шаг 3.8. Содержательный эффект процедуры "связать" также очевиден: она применима к включенным в сеть узлам; после ее применения узлы считаются связанными.

Снова можно было бы реализовать такой эффект по-разному. Выберем следующий способ, учитывающий конкретные реализации остальных наших процедур: в запись о каждом из аргументов процедуры "связать" будем добавлять указание о связи с другим аргументом.

По-прежнему не будем заботиться о диагностике ошибок, когда связей оказывается слишком много (больше макс_связей). Но если два узла просят связать вторично, то будем такой запрос игнорировать. Следует учесть также, что требование связать узел с самим собой вполне законно.

```
procedure связать(А-узел, В-узел : in имя_узла) is
begin
```

```
    if not есть_связь(A_узел, B_узел) then
      установить_связь(A_узел, B_узел) ;
      if A_узел /= B_узел then
        установить_связь(B_узел, A_узел) ;
      end if ;
    end if ;
  end связать ;
```

Мы ввели вспомогательную функцию есть_связь с очевидным эффектом (возможно, ее полезно и пользователю предоставить) и вспомогательную процедуру установить_связь, которая призвана вносить изменения в массив "узлы" своего первого аргумента. Ключевое слово not - это знак отрицания (унарная логическая операция). Продолжим их детализацию.

```
  function есть_связь(A_узел, B_узел : имя_узла) return BOOLEAN is
    запись : связи renames сеть(A_узел).связан ;
  begin
    for i in 1..запись.число loop
      if запись.узлы(i) = B_узел then
        return true ;
      end if ;
    end loop ;
    return false ;
  end есть_связь ;
```

Оператор return завершает выполнение тела функции.

```
  procedure установить_связь(откуда, куда : in имя_узла) is
    запись : связи renames  сеть(откуда).связан ;
  begin
    запись.число := запись.число+1 ;
    запись.узлы(запись.число) := куда ;
  end установить_связь ;
```

Таким образом, количество связей увеличивается на единицу м в качестве последней связи записывается имя узла "куда".

Вопрос. Нельзя ли переименование указать вне процедур и функций, чтобы не повторять его?

Итак, все услуги реализованы. Осталось выписать полное тело пакета. Для экономии места и времени позволим себе не выписывать объявления процедур и функций полностью, обозначая пропуски многоточием. Обратите внимание на порядок следования объявлений. Он существенен, соответствует правилу последовательного определения. Но оно касается лишь вспомогательных объявлений. Имена из спецификации пакета считаются объявленными ранее.

```
package body управление_сетью is
  type запись_об_узле is
    record
      включен : BOOLEAN : = false;
      связан  : связи;
    end record;
сеть : array(имя_узла) of запись_об_узле;
 function узел_есть(узел : имя_узла) return BOOLEAN is
  ........
 function все_связи(узел : имя_узла) return связи is
  ........
 procedure вставить(узел : in имя_узла) is
  ........
 procedure переписать(в_узле : in имя_узла, после : in индекс_узла)
  ........
 procedure чистить(связь : имя_узла, в_узле:имя_узла)  is
  ........
 procedure удалить(узел : in имя_узла) is
  ........
 function есть_связь(A_узел, B_узел : имя_узла) return BOOLEAN is
  ........
```

```
 procedure установить_связь(откуда, куда : in имя_узла) is
  ........
 procedure связать(А-узел, В-узел : in имя_узла) is
  ........
end управление_сетью;
```

Подчеркнем, что тип запись_об_узле, объект "сеть", процедуры "переписать", "чистить", "установить_связь", функция "есть_связь" недоступны пользователю, так как объявлены в теле, а не в спецификации пакета.

Третий шаг детализации завершен. Осталась прокомментировать полученный результат.

4.6.8. Принцип раздельного определения, реализации и использования услуг (принцип РОРИУС). Комментарий к примеру управление_сетью

Итак, мы написали три сегмента: спецификацию пакета управление_сетью, процедуру построение_сети и тело пакета управление_сетью. Важно понимать роли этих сегментов в жизненном цикле программы.

В этих сегментах воплощен принцип раздельного определения, реализации и использования услуг (РОРИУС).

По существу это рациональное применение абстракции на различных этапах проектирования. Проектируя определение пакета, отвлекаемся от деталей его возможного использования и вариантов реализации. Проектируя использование пакета, отвлекаемся от деталей определения и, тем более, реализации. Проектируя реализацию, отвлекаемся от несущественного (с точки зрения реализации) в определении и использовании.

[Упражнение. Приведите конкретные примеры. Хотя бы из нашего пакета.]

Более того, каждая названная абстракция представлена своим материальным воплощением - отдельным модулем. Каждый из трех модулей-сегментов - законченный продукт интеллектуальной деятельности в том смысле, что его можно записать в библиотеку и (при наличии документации) использовать без помощи автора и без жесткой связи с остальными модулями.

Три наших модуля, однако, не являются полностью независимыми. Центральным служит, конечно, модуль определения, т.е. спецификация пакета. Оставляя спецификацию неизменной, можно выбирать варианты реализации (тело пакета), не заставляя изменять использование (процедуры, аналогичные процедуре "построение_сети"). И это только благодаря тому, что реализация защищена от использования (принцип защиты абстракций) - из процедуры построение_сети нельзя непосредственно добраться, скажем, до массива "сеть" и нарушить дисциплину его эксплуатации операциями пакета. С другой стороны, никакое изменение реализации (согласованное со спецификацией и содержательным внешним эффектом объявленных услуг) не в состоянии повлиять на использование, иначе как по части ресурсоемкости (расходование времени, памяти и других ресурсов).

Таким образом, рассмотренные разновидности модулей, воплощающие абстракции определения, использования и реализации, обслуживают важнейшую технологическую потребность - проектировать, испытывать и хранить программы по частям, отдельными модулями. Эти же абстракции, как было показано, хорошо поддерживают метод пошаговой детализации, выбранный нами как показательный метод проектирования. Изучив принципы цельности и РОРИУС, мы достигли цели раздела. В следующем разделе продолжим изучение основных абстракций.

4.7. Основные абстракции: данные

Замечание об относительной пассивности данных. Рассматривая три выделенные роли в акте исполнителя, мы подчеркивали, что с ролью данных ассоциируется пассивное начало. Здесь уместно добавить (вспомнив обсуждение принципа единства и относительности ролей), что это не означает и не требует полной пассивности объекта, выступающего в роли данного, а лишь его относительной пассивности с точки зрения рассматриваемого акта того исполнителя, планирование поведения которого нас интересует. Тот же объект с другой точки зрения может быть активным и даже сам выступать в роли исполнителя. В качестве примера можно привести обработку данных в реальном времени (их активность проявляется в том, что они могут изменяться, отражая свойства некоторого асинхронно работающего процесса) или задачу управления асинхронными процессами, когда осуществляющий управление исполнитель вправе рассматривать эти (активные) процессы как данные, на которые направлено его управляющее воздействие.

Часто проводят параллель между данными и памятью (с одной стороны) и операциями и процессором (с другой стороны). Уместно напомнить, что память может быть устроена на активных элементах (и тем самым подчеркнуть идею относительности). Резюмируя, скажем, что данными обычно считают любые обрабатываемые объекты, независимо от их внутренней природы. Так что быть данным - качество внешнее по отношению к сущности объекта, выступающего в этой роли.

В частности, для одной и той же категории объектов в одном ЯП может оказаться естественным выступать в роли данных, а в другом это может быть запрещено. Так, процедуры могут быть данными в Паскале и Алголе-68 (их можно передавать в качестве значений, присваивать компонентам других объектов), но в Аде.

Конец замечания.

4.7.1. Классификация данных

Обрабатываемые объекты-данные различаются по многим признакам.

Во-первых, данные - это знаки содержательных денотатов, играющих определенные роли в решаемой задаче.

Например, в программе начисления зарплаты одна переменная может быть предназначена для хранения числа детей сотрудника, а другая переменная - для хранения количества рабочих дней. Очевидно, что складывать или перемножать значения таких переменных бессмысленно, это содержательная ошибка.

Таким образом, данные можно классифицировать по содержательным ролям. Очень заманчиво было бы уметь явно отражать в программе результаты такой содержательной классификации с тем, чтобы сделать ее доступной как исполнителю, так и читателю программы. По существу такая классификация - это прогнозирование поведения определенных объектов данных (например, прогноз о том, что такие-то переменные никогда не могут быть операндами одного и того же сложения). Прогноз такого рода облегчает понимание программы и создает предпосылки для автоматического содержательного контроля - прямой путь к повышению надежности программ. Ясно также, что такого рода классификация данных по содержательным ролям непосредственно вытекает из пошаговой детализации (на каждом шаге имеем дело с содержательной задачей; разлагая ее решение на составляющие, одновременно выделяем их роли в этом решении).

Возможность отражать содержательную классификацию отсутствует в большинстве классических языков. В Аде сделаны шаги в нужном направлении.

Во-вторых, данные различаются по своему внутреннему строению, структуре; характеру связей своих составдяющих.

Например, массивы, таблицы, списки, очереди. С этой точки зрения обычно особенно важен способ доступа к составляющим данных. Классификация данных по способу доступа к составляющим обычно имеется в виду, когда говорят о структурах данных (а также, обычно с точки зрения больших совокупностей данных, о базах данных, банках данных и т.п.). Классификация по структуре есть в том или ином варианте почти во всех ЯП, но возможности сильно зависят от языка. В современных ЯП чаще всего это массивы и записи. И то, и другое можно считать способом представления таблиц. Последние в явном виде имеются, например, в МАСОНе [10] и его последующих модификациях.

В-третьих, данные различаются по своим способностям к изменению (изменчивости). Например, в некоторых случаях известен диапазон возможных изменений или известно, что данное вообще не должно меняться. Это еще одна сторона прогнозирования поведения данного. Например, из того, что данное обозначает количество детей у сотрудницы, следует, что его значение может быть лишь целым числом в диапазоне от 0 до 100. Прогнозирование поведения такого рода встречается только в относительно новых языках, начиная с Паскаля.

В-четвертых, данные могут различаться по способу своего определения. Их свойства могут быть предопределены в языке (т.е. определены автором языка) или же определены программистом с помощью языковых средств. В последнем случае в идеале это такие средства, которые позволяют программисту по существу определить новый язык.

Замечание о принципе зашиты абстракций. Как правило, предопределенные свойства не могут быть изменены программистом и в этом смысле надежно защищены от искажений. В идеале при определении данных с новыми свойствами (при определении нового языка) у программиста должна быть возможность принять меры к тому, чтобы

при использовании эти вновь введенные свойства данных (абстракции) были неотличимы от предопределенных.

Этот принцип, который мы назовем принципом защиты абстракций, еще послужит для нас предметом изучения. Если он обеспечен, то на каждом шаге детализации есть полная возможность действовать так, как будто в распоряжении программиста появился виртуальный исполнитель для сконструированного уровня абстракции. Подчеркнем, что при этом детализация осуществляется от задачи к реализации (сверху - вниз), а создание виртуальных машин - в общем случае, от реальной машины к задаче - (снизу-вверх). Аппарат для определения данных, ориентированный на принцип защиты абстракций, имеется только в новейших языках, в частности, в Аде.

Конец замечания.

В-пятых, данные могут различаться по своему представлению на более низком уровне абстракции (на реализующей виртуальной машине, в терминах реализующей структуры данных, по классу необходимых для реализации ресурсов, по объему и дисциплине использования памяти и т.п.). Например, для чисел может требоваться одно слово, два или несколько в зависимости от нужной точности вычислений, память для данных одной категории может выделяться в некотором стеке (например, для локальных данных блоков) или в так называемой куче (для элементов динамически изменяемых списковых структур), для некоторых данных разумно выделять самую быструю память (например, быстрые регистры для переменной цикла). Это еще одна форма прогнозирования поведения объектов - запрос для них подходящих ресурсов. Она - из самых старых; классификация с точки зрения представления встречается практически во всех ЯП, ориентированных на эффективное использование ресурсов машины.

Наконец, выделим еще шестой фактор классификации. По важности его можно указать и первым. Речь идет о классификации по внешним (абстрактным, не зависящим от реализации или содержательной роли) свойствам (по применимым операциям), определяющим возможности данного играть определенные роли или вступать в определенные отношения с другими объектами программы.

Например, данное может быть числом, символом, указателем, задачей, очередью, стеком. В каждом из этих случаев к нему применим определенный набор операций, у него имеется определенный набор атрибутов, характерных для данных именно этого класса и т.п. Чтобы не было путаницы с первым, ролевым фактором классификации, подчеркнем, что переменная для хранения числа апельсинов может отличаться от переменной для хранения числа яблок по содержательной роли, но не отличаться по внешним свойствам.

В заключении подчеркнем, что указанные факторы ортогональны, т.е. данное можно характеризовать перечнем классов по всем шести факторам. Классификация не претендует на полноту, но позволит нам ориентироваться, в частности, в системе управления данными в Аде.

4.7.2. Типы данных

Классификация данных присутствует в каждом ЯП. В примитивном виде мы имели с ней дело и в рассмотренных нами моделях. В Алголе она отражена в системе классов и типов (классы - процедуры, метки, простые переменные, массивы, переключатели; типы - целый, вещественный, логический), в Фортране - также в системе классов типов (классы имен - массив, переменная, внутренняя функция, встроенная функция, внешняя функция, подпрограмма, переменная и общий блок; типы - целый, вещественный, двойной точности, комплексный, логический, текстовый). В более современных языках имеется тенденция более полно отражать классификацию данных в системе типов, в связи с чем само понятие типа данных меняется от языка к языку. Поэтому к понятию типа данных будем относиться как к рабочему понятию, отражающему определенные аспекты нашей классификации (и возможно, кое-что еще) в каждом языке по-своему и ориентированном, прежде всего, на упрощение прогнозирования и контроля.

4.7.3. Система типов в Аде (элементы теории типов)

Система типов в ЯП - это всегда система классификации денотатов (в общем случае, и данных, и операций, и связываний; возможно, и других сущностей). Задачи, которые ставятся перед такой классификацией, зависят от назначения языка, а также от других причин (отражающих, в частности, специфику ЯП как творения человеческого интеллекта и вместе с тем явления не только научно-технического, но и социального).

Так, система типов в Аде отражает, конечно, специфику встроенных систем программного обеспечения (в особенности, требование повышенной надежности, эффективности объектной программы и относительное богатство ресурсов базовой (транслирующей) машины). Но, с другой стороны, некоторые свойства этой системы явно предписаны техническими требованиями заказчика и не могли быть изменены авторами языка.

Таким образом, излагаемые ниже принципы построения системы типов в Аде нужно воспринимать как интересный и в целом дееспособный вариант классификации обрабатываемых данных, но отнюдь не как окончательное единственно верное решение.

4.7.3.1. Динамические, статические и относительно-статические ЯП. (Назначение системы типов данных в Аде)

Обрабатываемый объект, как мы уже подчеркивали в нашей классификации данных, можно характеризовать по-разному. Некоторые его свойства и связи с другими объектами остаются неизменными при любом исполнении его области действия (участка программы, где этот объект считается существующим). Такие свойства и связи называются статическими.

Например, в Алголе тип объекта (целый, вещественный, логический) - одно из статических свойств. Сама область действия объекта - по определению статическое свойство. Связь двух обьктов по свойству принадлежать одной области действия - статическая связь. Свойство объекта при любом исполнении области действия принимать значения только из фиксированной совокупности значений - статическое свойство. Исчерпывающий перечень применимых к объекту операций - статическое свойство.

Другие свойства и связи изменяются в процессе исполнения области действия. Их называют динамическими.

Например, конкретное значение переменной - динамическое свойство. Связь формального параметра с конкретным фактическим в результате вызова процедуры - динамическая связь. Размер конкретного массива с переменными границами - динамическое свойство.

Часто статические и динамические характеристики называют соответственно характеристиками периода компиляции (периода трансляции) и периода выполнения, подчеркивая то обстоятельство, что в период компиляции исходные данные программы недоступны и, следовательно, динамические характеристики известны быть не могут. Известны лишь характерисмки, извлекаемые непосредственно из текста программы и тем самым относящиеся к любому ее исполнению (т.е. статические характеристики).

Однако деление на статические и динамические характеристики иногда оказывается слишком грубым. Например, размер массива в Алголе может в общем случае изменяться при различных исполнениях его области действия, однако при каждом конкретном исполнении этот размер зафиксирован при обработке объявления (описания) массива и в процессе исполнения области действия изменяться не может. Так что это и не статическая характеристика, и вместе с тем не столь свободно изменяемая, как скажем, значение компоненты массива в Алголе, которое можно изменить любым оператором присваивания. Такие характеристики, которые могут меняться от исполнения к исполнению, но остаются постоянными в течение одного исполнения области действия объекта, будем называть относительно статическими.

Иногда пользуются и еще более тонкой классификацией характеристик по фактору изменчивости. Например, связывают изменчивость не с областью действия объекта, а с периодом постоянства других его избранных характеристик (скажем, выделяемого объекту пространства или связи с другими объектами (каналами ввода-вывода и т.п.)).

Уровень изменчивости характеристик допустимых в языке денотатов - одно из важнейших свойств языка. Одна крайняя позиция представлена концепцией неограниченного (образно говоря "разнузданного") динамизма, когда по существу любая характеристика обрабатываемого объекта может быть изменена при выполнении программы. Такая концепция не исключает прогнозирования и контроля, но не связывает их жестко со структурой текста программы.

Неограниченный динамизм присущ не только практически всем машинным языкам, но и многим языкам программирования достаточно высокого уровня. Эта концепция в разной степени воплощена в таких динамических ЯП, как Бейсик, Апл, Лисп, отечественные ИНФ и Эль-76 [11]. Идеология и следствмя динамизма заслуживают отдельного изучения.

Другая крайняя позиция выражена в стремлении затруднить программисту всякое изменение характеристик денотатов. Вводя знак, нужно объявить характеристики денотата и использование знака должно соответствовать объявленным характеристикам. Конечно, "неограниченной" статики в программировании добиться невозможно, она противоречит сути управления поведением исполнителя. Так что всегда разрешается менять, скажем, значения объявленных обрабатываемых объектов (значения данных).

Зато остальные характеристики в таких статических языках изменить трудно. Обычно стремление к статике объясняют надеждой на повышение надежности программ (за счет дополнительной избыточности при обязательном объявлении характеристик возникает возможность дополнительного контроля) и надеждой на повышение скорости объектных программ (больше связываний можно выполнить при трансляции и не тратить на это времени в период исполнения).

Вместе с тем сама по себе идея объявления характеристик (прогнозирования поведения) и контроля за их инвариантностью требует создания, истолкования и реализации соответствующего языкового аппарата. Поэтому статические языки, как правило, сложнее динамических, их описания объемнее, реализации тяжеловеснее. К тому же надежды на положительный эффект от статики далеко не всегда оправдываются. Тем не менее среди языков индустриального программирования преобладают статические языки Фортран, ПЛ/1, Кобол, Алгол, Паскаль. Раньше это частично можно было объяснить трудностями эффективной реализации динамических языков. Сейчас на первое место выходит фактор надежности, и с этой точки зрения "старые" статические языки оказываются "недостаточно_статическими" - аппарат прогнозирования и контроля у них связан скорее с требуемым распределением памяти, чем с другими характеристиками поведения, существенными для обеспечения надежности (содержательными ролями, изменчивостью значений, применимыми операциями и т.п.).

Ада принадлежит скорее к статическим, чем к динамическим языкам, хотя по некоторым признакам ее лучше было бы назвать языком относительно-статическим с развитым аппаратом прогнозирования-контроля. Концепция типа в Аде предназначена в основном для прогнозирования-контроля статических характеристик. Ее дополняет концепция подтипа, предназначенная для прогнозирования-контроля относительно-статических характеристик. В дальнейшем мы будем рассматривать эти концепции вместе, считая концепцию подтипа составной частью концепции типа.

4.7.3.2. Концепция типа данных как знаковой системы

Постановка задачи. Вспомним нашу классификацию данных. Мы выделили шесть факторов, характеризующих данные (роль в программе, строение, изменчивость, способ определения, представление, связанные операции). Язык программирования как знаковая система, предназначенная для планирования поведения исполнителя (в частности, для планирования его манипуляций с данными) может как иметь, так и не иметь специальных понятий и конструктов, позволяющих программисту характеризовать данные.

Крайняя позиция, с которой мы сталкивались в наших моделях - полное или почти полное отсутствие таких средств. И в модели Неймана, и в модели Бэкуса, и в модели Маркова нельзя было охарактеризовать данные ни по одному из шести факторов. Конечно, эти факторы существенны независимо от применяемого языка. Просто при проектировании программы на недостаточно развитом языке программист вольно или невольно использует для характеристики данных внеязыковые средства (держит "в уме", отражает в проектной документации и т.п.).

Кое-что из такой внешней классификации данных находит воплощение и в программе - вспомните распознавание строения (структуры) выражений перед применением к ним того или иного правила перевода в ПОЛИЗ) - но далеко не все. А если находит, то в такой форме, что программиста не может проконтролировать не только исполнитель, но и коллеги-программисты (да и он сам делает это, как правило, с трудом).

Указанная крайняя позиция игнорирует потребность прогнозирования-контроля. Она характерна для ранних машинных языков и в чистом виде в современном программировании не встречается. В Фортране и Алголе можно (а иногда - обязательно нужно) предварительно описать строение данного и охарактеризовать набор его допустимых значений, чтобы получить доступ к этому данному в соответствии с описанием. Еще более развиты средства описания данных в таких языках, как Кобол и ПЛ/1.

Вспоминая пример с пошаговой детализацией и принцип согласования абстракций, можно почувствовать, что тенденция к развитым средствам описания данных -

естественная тенденция в современных языках, ориентированных на создание надежных и эффективных программ с ясной структурой.

Допустим, что эта тенденция осознана. Возникает следующая задача - какие ключевые концепции должны быть положены в основу средств описания данных? Если угодно, как построить "язык в языке", знаковую систему для характеристики данных в языке программирования?

Первый вариант: перечни атрибутов. Повидимому, первое, что приходит в голову каждому, знакомому с нашими шестью факторами - это ввести свои средства для каждого фактора и сопровождать каждый объект перечнем по меньшей мере из шести характеристик. И это не надуманный вариант. Примерно так сделано в ПЛ/1. Объявляя переменную, в этом языке можно перечислить ее характеристики (так называемые "атрибуты") по многим факторам (основание системы счисления, способ представления, способ выделения памяти, структура и способ доступа к компонентам, потребуется ли печатать значения и т.п.).

Такая знаковая система, как показывает опыт, вполне дееспособна, однако с точки зрения ясности и надежности программ (да и эффективности) оставляет желать лучшего.

Во-первых, довольно утомительно задавать длинные перечни атрибутов при объявлении данных. Из-за этого в ПЛ/1 придуманы даже специальные "правила умолчания" атрибутов (это одна из самых неудачных и опасных с точки зрения надежности концепция ПЛ/1; к тому же этих правил много, они сложны; так что запомнить их невозможно).

Во-вторых, допустим, что один перечень атрибутов - у данного, а другой (тоже достаточно длинный) перечень - у формального параметра процедуры. Может ли такое данное быть фактическим параметром? Чтобы это понять, нужно сравнить оба перечня, потратив время и рискуя ошибиться (исполнитель ошибиться не рискует, но ему это тоже дорого обходится). И дело не только (и не столько) в самом переборе атрибутов, сколько в сложности правил, определяющих применимость такой процедуры к такому данному.

Итак, запомнив, что у проблемы два аспекта - прогнозирование (описание) и контроль (проверка допустимости поведения данных), поищем другие решения. (Отметим, что по отношению к языку мы сейчас колеблемся между технологической и авторской позициями, задевая семиотическую).

Второй вариант: именованные объявляемые типы. Вспомним принцип обозначения повторяющегося. Заметили, что приходится иметь дело с длинными перечнями атрибутов? Отлично, значит, нужно обозначить это повторяющееся именем и упоминать как единое целое при повторном использовании. Следовательно, нужен языковый конструкт для объявления таких имен.

Естественно считать, что имя обозначает не только перечень атрибутов, но и класс данных, обладающих объявленными характеристиками. Мы пришли к понятию типа данных как класса объектов-данных, обладающих известными атрибутами. А искомый языковой конструкт - это обеявление типа данных. Его назначение - связывать объявляемое имя с указанным перечнем атрибутов данных.

Теперь можно не умалчивать атрибуты, как в ПЛ/1, а коротко и ясно обозначать их одним именем. Кажется, совсем просто, но это шаг от ПЛ/1 к Алголу-68, в котором очень близкая к изложенной концепция типа данных.

С проблемой прогнозирования мы справились, а как с проблемой контроля? Нужно сравнить перечни атрибутов (возможно, обозначенных именами). Другими словами, здесь мы никак не продвинулись. Те же длинные перечни и те же сложные правила совместимости объявленных характеристик с фактическими.

Как справиться с проблемой? Ведь кажется совершенно естественным, что один объект данных в общем случае может принадлежать нескольким объявленным типам данных. Так что для контроля поведения данных (скажем, контроля совместимости объявленного типа формального и объявленного типа фактического параметра) недостаточно знать имена типов: если имена совпадают, то, конечно, совместимы, а вот когда не совпадают, нужно проверять совместимость характеристик типов.

Проблема совместимости (иногда говорят, проблема эквивалентности типов, потому что простейшее правило совместимости - эквивалентность атрибутов) в общем случае очень сложна. Характеристиками могут быть тексты рекурсивной природы (например, характеристика в объявлении ссылочного типа А может указывать на то, что некоторая компонента значения этого объекта может быть только ссылкой типа А).

Так что проблема эквивалентности типов может оказаться даже алгоритмически неразрешимой.

Третий вариант: концепция уникальности типа (строгая типизация). Поистинне блестящее решение состоит в том, чтобы полностью игнорировать эту проблему, "чуть-чуть" подправив концепцию типа, сделав центральным понятием концепции не атрибуты, а имя типа. По определению языка будем считать разными и в общем случае несовместимыми типы с разными именами независимо от прочих характеристик, связанных с именем. Если теперь потребовать, чтобы каждый объект данных был связан ровно с одним типом (с одним именем типа), то и прогнозировать, и проверять - одно удовольствие! Нужно сказать, что объект будет вести себя так-то, обозначаем стиль (характер) его поведения именем типа из имеющегося набора типов. Нет подходящего - объявляем новый. Нужно проверить совместимость - сравниваем имена типов. Просто, ясно и быстро.

Но наше "чуть-чуть" - в историческом плане шаг от Алгола-68 к Аде. Через Паскаль, где эта идея присутствует, но реализована недостаточно четко. Априорная несовместимость типов, названных разными именами, вместе с идеей "каждому объекту данных - ровно один тип" - ключевая концепция аппарата типов в Аде. Назовем ее концепцией уникальности типа. Языки с концепцией уникальности типа иногда называют строго типизированными языками.

Сформулируем правила (аксиомы), характеризующие эту концепцию.

1. Каждому объекту данных сопоставлен один и только один тип.
2. Каждому типу сопоставлено одно и только одно имя (явное или неявное). Типы с неявными именами называются анонимными и все считаются различными.
3. При объявлении каждой операции должны быть явно указаны (специфицированы) имена типов формальных параметров (и результата, если он есть).
4. Кроме специально оговоренных случаев, различные типы считаются несовместимыми по присваиванию и любым другим операциям.

4.7.3.3. Критичные проблемы, связанные с типами

Остался "маленький" вопрос. Прогнозировать - легко, проверять - легко. А легко ли планировать поведение исполнителя - программировать? Как увязать несовместимость типов, обозначенных разными именами, с естественным желанием иметь достаточно мощные операции, применимые к объектам разных типов (полиморфные операции).

Ведь если указать в качестве характеристики формальных параметров такой операции какой-то тип, то к другим типам операция окажется неприменимой. Назовем отмеченную проблему проблемой полиморфизма операций. Напомним, что полиморфизм операций широко распространен в математике, жизни, программировании. Операция "+" применима к целым числам, вещественным числам, матрицам, комплексным числам, метрам, секундам и т.п.

Проблема полиморфизма возникла, как только мы взглянули на концепцию уникальности типа с позиции операций. Но неприятности поджидают нас и со стороны данных. Помните, мы назвали естественной ситуацию, когда в процессе работы программы один и тот же объект данных выступает в разных ролях, обладает различными характеристиками. Например, число яблок может превратиться в число людей, взявших по яблоку; буфер, работавший в режиме очереди, может начать функционировать в режиме стека; у массива может измениться размерность и т.п.

Каким должен быть единственный тип такого объекта? Если он отражает сразу много ролей, то не придем ли мы к отсутствию контроля (ведь пока буфер работает как очередь, к нему нужно запретить обращаться как к стеку и наоборот)? Если же только одну роль, то как ему попасть в другую (ведь разные типы несовместимы)? Назовем выделенную проблему янус-проблемой (объект ведет себя, как двуликий Янус, даже "многоликий").

Сказанное можно кратко отразить следующей табличкой.

```
----------------------------------------------------
| разные типы - одна операция -) полиморфизм       |
| разные типы - один объект данных -) янус-проблема|
----------------------------------------------------
```

Рис. 4.1

4.7.3.4. Критичные потребности и критичные языковые проблемы

Несколько драматизируем ситуацию и временно представим, что найти решение проблемы полиморфизма не удалось. И вот мы в роли программиста, который старается следовать хорошему стилю создания программ.

Допустим, что к его услугам тип "целый", тип "комплексный", тип "матрица". Ему нужно, в частности, предоставить пользователю возможность складывать объекты любого из названных типов. Хороший стиль программирования требует ввести операционную абстракцию, позволяющую пользователю игнорировать несущественные детали (в данном случае - особенности каждого из трех типов данных) и действовать независимо от них. Попросту говоря, нужно ввести единую (полиморфную) операцию сложения. Если такой возможности язык не предоставит, то у программиста может оказаться единственный разумный вариант - избегать пользоваться таким ЯП. Аналогичным ситуациям нет числа. Скажем есть возможность рассчитать потребности взвода, роты, батальона, полка. Требуется ввести абстракцию - "рассчитать потребности подразделения" и т.п.

Итак, мы пришли к понятию критичной технологической потребности. Если язык не предоставляет средств для удовлетворения этой потребности, он просто не выживет в качестве инструмента для решения задач рассматриваемой проблемной области. Проблему удовлетворения критичной потребности мы называем критичной языковой проблемой.

4.7.3.5. Проблема полиморфизма

Допустим, что критичность для Ады проблемы полиморфизма осознана. Как же ее решать?

Повидимому, самым естественным было бы ввести "объединяющий" тип (скажем, "операнд_сложения" или "боевое_подразделение"), частными случаями которого были бы исходные типы. И определить нужную операцию для объединяющего типа. Но что значит "частными случаями"? Ведь в соответствии с концепцией уникальности каждый объект принадлежит только одному типу. Если это "взвод", то не "боевое_подразделение"! Так что в чистом виде эта идея не проходит.

Нечто подобное можно реализовать с помощью так называемых ВАРИАНТНЫХ типов, но каждый вариант должен быть выделен соответствуюшим значением дискриминанта, причем заботиться об этом должен пользователь такой "квазиполиморфной" операции. Поэтому такое решение можно рассматривать лишь как суррогат. Пока мы его в деталях изучать не будем.

Вот если бы к услугам программиста уже был тип "боевое_подразделение", а ему понадобилось ввести новые операции именно для взводов, то (при условии, что у объектов "боевое_подразделение" есть поле "вид") можно было бы объявить, скажем,

```
type T - взвод is new боевое_подразделение (вид => взвод);
```

Теперь "взвод" - это так называемый ПРОИЗВОДНЫЙ тип с РОДИТЕЛЬСКИМ типом боевое_подразделение. К его объектам применимы все операции, применимые к боевому_подразделению. Но можно теперь объявить новые операции, применимые только к "взводам", т.е. к "боевым_подразделениям", в поле "вид" которых - значение "взвод". Итак, это решение для полиморфизма "сверху-вниз" - нужно заранее предусмотреть частные случаи нужного типа.

Основной вариант решения проблеммы полиморфизма, предлагаемый Адой, это так называемое ПЕРЕКРЫТИЕ операций. Идея состоит в том, чтобы распознавать денотат знака, привлекая на помощь контекст этого знака (определенным образом ограниченный). По отношению к операциям идея перекрытия проявляется в том, что связь между вызовом операции и ее объявлением устанавливается не только по имени (т.е. знаку) операции, но и с учетом типов операндов, типа результата и даже имен формальных параметров (если вызов по ключу). Это и есть вспомогательный контекст.

Например, в одной и той же области действия можно объявить две функции

```
function потребности (подразделение:взвод) return расчет;
function потребности (подразделение:рота) return расчет;
```

Для каждой функции нужно написать свое тело. Если теперь объявить объекты

```
A:взвод;
B:рота;
```

то вызов потребности(A) означает выполнение тела первой функции, а вызов потребности(B) - второй. Для пользователя же "видна" единственная операционная абстракция "потребности", применимая к объектам и типа "взвод", и типа "рота", т.е. полиморфная функция.

[Упражнение. Укажите дополнительный контекст знака функции в приведенном примере.]

Конечно, такое решение требует своего заголовка и своего тела для каждого варианта допустимых типов параметров, но это и есть плата за полиморфизм. К тому же все не так страшно, как может показаться. Ведь самое главное, что пользователь получает в точности то, что нужно - полиморфную операцию, сохраняя полный контроль над использованием объектов в соответствии с их типами во всех остальных случаях. Это полиморфизм "снизу-вверх", когда частные случаи операции можно добавлять (причем делать это несложно).

[Вопрос. Как это сделать?]

4.7.3.6. Янус-проблема

Вспомним, как возникла янус-проблема. Мы хотели легко контролировать поведение и пришли к концепции уникальности. И потеряли возможность иметь объекты, играющие одновременно разные роли. Но ведь это совершенно естественная ситуация и она, конечно, требует выразительных средств. В конечном итоге система типов в каждой программе - это некоторая классификация данных. Одна из простейших и самая распространенная классификация - иерархическая. Хорошо известный пример такой классификации - классификация животных и растений по типам, классам, отрядам, семействам, родам и видам. При этом каждое животное (классифицируемый объект) играет сразу несколько ролей (и как представитель вида, и как представитель рода, и как представитель семейства, и т.п.). К характеристикам типа (общим для всех животных этого типа) добавляются специфические характеристики класса (общие только для выбранного класса, а не всего типа), затем добавляются характеристики отряда и т.д., вплоть до характеристик вида.

Еще сложней ситуация, когда классификация не иерархическая. Человек - одновременно сотрудник лаборатории, отдела, института и т.п., жилец в квартире, доме, микрорайоне и т.п., подписчик газеты, муж, брат, сват, любитель бега и т.п. Если нужно написать на Аде пакет моделирование_человека, то как уложиться в концепцию уникальности!

Напомним, что проблема возникла именно потому, что мы хотим прогнозировать и контролировать различные роли объектов. Если игнорировать проблему прогнозирования-контроля, то исчезнет и янус-проблема. Как в Алголе - везде массивы целых, и представляй их себе в любой роли (ведь эти представления - вне программы!).

Полного и изящного решения янус-проблемы Ада не предлагает - этого пока нет ни в одном ЯП.

Но можно выделить три основные свойства Ады, направленные на решение янус-проблемы. Каждое из них по-своему корректирует концепцию уникальности, а вместе они образуют практически приемлемое решение.

Эти средства - ПРОИЗВОДНЫЕ ТИПЫ + ПРЕОБРАЗОВАНИЯ типов + понятие ОБЪЕКТА.

Производные типы. Мы уже видели, что объявление производного типа указывает родительский тип и определяет, что объекты производного типа могут принимать лишь подмножество значений, допустимых для объектов родительского типа. Вместе с тем для объектов производного типа можно определить новые операции, неприменимые в общем случае к объектам родительского типа. Но ведь это знакомая нам связь старшей и младшей категории в иерархической классификации.

Если, скажем, определен тип "млекопитающие", то его производным может стать тип "хищные", его производным тип "кошки", его производными типы "сибирские_кошки" и "сиамские_кошки". При этом все уменьшается совокупность допустимых значений (в "хищные" не попадают "коровы", в "кошки" - "собаки", в "сибирские_кошки" - "львы") и добавляются операции и свойства (млекопитающие - операция "кормить молоком"; хищные - "съедать животное"; кошки - "влезть на дерево"; сибирскик кошки - "иметь пушистый хвост"), причем всегда сохраняются операции и свойства всех родительских типов, начиная с так называемого БАЗОВОГО ТИПА - не имеющего родительского типа.

Итак, решение для полиморфизма "сверху-вниз" - это одновременно решение янус-проблемы.

[Вопрос. Почему это решение названо решением сверху-вниз? Сравните с решением снизу-вверх.]

Чтобы все сказанное увязать с концепцией уникальности, нужно поговорить еще и о преобразованиях и объектах данных.

Преобразования типов. Считается, что каждое объявление производного типа вводит и ПРЕОБРАЗОВАНИЕ ОБЪЕКТОВ из родительского типа в производный и обратно. При этом перейти от родительского типа к производному можно только при выполнении объявленных ограничений на значения объекта, а обратно - всегда. Теперь можно написать процедуру, скажем, "кормить_молоком" и применять ее и к "телятам", и к "котятам", и к "львятам". Считается, что в позиции фактического параметра применяется подразумеваемое преобразование типа и процедура применяется к "млекопитающему". Но нельзя применять процедуру "влезть_на_дерево" к "корове" - только к "кошке".

Возможны и явные преобразования типа. В них нет ничего удивительного - это просто функции, аргументы которых одного типа, а результаты - другого. Их естественно считать преобразованием типа, если они сохраняют значение объекта в некотором содержательном смысле. Так, можно написать преобразование из типа "очередь" в тип "стек", сохраняющее содержимое объекта. К результату такого преобразования можно применять операции "втолкнуть", "вытолкнуть", определенные для стеков, а к аргументу нельзя. Подчеркнем, что написать преобразование можно далеко не в каждом контексте - нужно иметь возможность "достать" содержимое аргумента и "создать" содержимое результата.

Объекты. Из трех выделенных средств осталось уточнить понятие "объект".

Напомним: концепция уникальности требует, чтобы типы в программе образовывали разбиение объектов, т.е. каждый объект попадает в точности в один тип. Другими словами, типы не пересекаются и объединение объектов всех типов - это и есть множество всех объектов программы.

Но это значит, что к объектам нужно относить только сущности, с которыми связан тип (в описании языка или явно в программе). Таковыми в Аде служат, во-первых, изображения предопределенных значений (изображения чисел, символов и логических значений); во-вторых, переменные, постоянные, формальные параметры, выражения. Не относятся к объектам в Аде процедуры, функции, типы, сегменты, подтипы. Имена у них есть, а типов нет. Их нельзя присваивать и передавать в качестве параметров процедур и функций - это их основное содержательное отличие от объектов.

4.7.3.7. Неформальные теоремы о критичных проблемах

В заключении этого раздела обратим внимание на способ решения критичных технологических проблем. Нам каждый раз требовалась пара - чисто языковые средства плюс методика применения этих средств для решения проблемы.

Например, для проблемы полиморфизма - это языковое средство (перекрытие операций) плюс методика определения серии операции с одним названием. Для янус-проблемы - это снова языковое средство (производные типы) плюс методика определения системы производных типов для реализации нужной классификации данных. Сочетание средства с методикой служит доказательством существования решения критичной проблемы, доказательством жизнеспособности языка. Вместе с тем не исключено, что будут найдены другие, более удачные решения средствами того же языка. Однако с авторской позиции (т.е. для авторов языка) исключительно важно иметь доказательства существования решения всех известных критичных проблем (другими словами, автор должен уметь доказывать неформальные теоремы существовования решения критичных проблем).

4.8. Регламентированный доступ

Начав с общей потребности прогнозировать и контролировать, мы пришли к приемлемому решению (пока весьма абстрактному) - к концепции типа, ориентированной на имена. Теперь нужно научиться описывать строение, изменчивость, представление и т.п.. Другими словами, нужно разобраться со средствами, позволяющими связать с именем типа технологически значимые характеристики данных.

В соответствии с принципом технологичности, эти средства обслуживают определенные потребности жизненного цикла комплексного программного продукта. Чтобы лучше почувствовать, что это за потребности, продолжим нашу серию примеров.

4.8.1. Задача моделирования многих сетей

Постановка задачи. Допустим, что пользователям понравился пакет управление_сетью и они заказывают более развитые услуги. Одной сети теперь мало.

Нужно сделать так, чтобы пользователи могли создать столько сетей, сколько им потребуется, и по отношению к каждой могли воспользоваться любой из уже привычных операций (вставить, связать и т.п.). Вместе с тем больше всего пользователи оценили именно надежность наших программных услуг, гарантию целостности сети. Это важнейшее свойство необходимо сохранить.

Итак, пользователи предъявляют внешнее требование надежности услуг, а возможно (если они достаточно подготовлены) и целостности создаваемых сетей. Наша роль, как реализаторов комплекса услуг, перевести внешние требования (потребности) на язык реализационных возможностей.

Эти потребности и возможности относительны на каждом этапе проектирования. Скажем, потребность в надежности реализуется возможностью поддерживать целостность. Потребность в целостности реализуется, например, возможностью обеспечить (строго) регламентированный доступ к создаваемым сетям. Так что под технологическими потребностями будем понимать потребности, возникающие, возможно, на достаточно детальных этапах проектирования.

Итак, будем считать, что технологическая потребность пользователя в том, чтобы иметь в распоряжении класс данных "сети", иметь возможность создавать сети в нужном количестве и получать регламентированный доступ к каждой созданной сети.

Варианты и противоречия. После нашего разговора о типах данных, казалось бы, несложно такую потребность удовлетворить - достаточно сделать доступным для пользователя соответствующий тип данных (скажем, "сети"). Давайте так и поступим.

Допустим, что структура нашего объекта "сеть" вполне удовлетворяла пользователей. Объявим регулярный тип "сети", все объекты которого устроены аналогично массиву "сеть".

```
(а)       type сети is array  (имя_узла) of запись_об_узле;
```

Теперь объект "сеть" можно было бы объявить так:

```
(б)    сеть: сети;
```

Так же может поступить и пользователь, если ему понадобятся, скажем, две сети.

```
(в)    сеть1, сеть2: сети;
```

И к его услугам два объекта из нужного класса.

Но возникает несколько вопросов.

Во-первых, мы подчеркивали, что объект "сеть" недоступен пользователю непосредственно (это и гарантировало целостность). Другими словами, объект "сеть" не был видим пользователю (ему было не доступно непосредственно имя этого объекта). А чтобы писать объявления вида (в), нужно явно выписать имя "сети". Другими словами, объявление типа "сети" должно быть видимым пользователю!

Но видимые извне пакета объявления должны находиться в его спецификации. Куда же именно в спецификации пакета следует поместить объявление (а)? Вспомним правило последовательного определения. В (а) использованы имена типов имя_узла и запись_об_узле. Но последний пока не объявлен в спецификации нашего пакета. Значит, нужно объявить и этот тип (после типа "связи", т.е. после строки 11), затем поместить (а).

Во-вторых, нужно подправить определения и реализацию операций. Ведь пока они работают с одним и тем же объектом "сеть". Нужно сделать так, чтобы пользователь мог указать интересующую его сеть в качестве фактического параметра нужной операции. К тому же грамотный программист всегда стремится сохранить преемственность со старой версией программы, стремится обеспечить, чтобы ранее написанные программы могли работать без каких-либо изменений. Поэтому совсем хорошо, если бы можно было работать и по-старому, с одной и той же сетью, когда параметр не задан, и по-новому, когда задан.

Поэтому перепишем строки с 13 по 18 спецификации пакета.

```
(13') procedure вставить (узел : in имя_улла, в_сеть : in out сети );
```

Обратите внимание: режим второго параметра in out ! Указанная им сеть служит обновляемым параметром (результатом работы процедуры "вставить" служит сеть со вставленным узлом).

```
(14') procedure удалить (узел : in имя_улла, из_сети : in out сети );
(15') procedure связать (А_узел, В_узел : in имя_узла,
                             в_сети : in out сети );
(17') function  узел_есть (узел : имя_узла, в_сети : сети) return BOOLEAN;
(18') function  все_связи (узел : имя_узла, в_сети : сети)
                            return связи;
```

Так как нужно обеспечить и возможность работать по-старому, с одной сетью, то в спецификации пакета следует оставить и строки 13-18, и 13'-18'. Тогда в соответствии с идеей перекрытия (совмещения) операций в зависимости от заданного количества параметров будет вызываться нужная спецификация (и, конечно, нужное тело) операции. Например, написав

удалить (узел => 33);

вызовем строку 14 (и соответствующее тело для этой процедуры), а написав

удалить (узел => 33, из_сети => сеть1);

или

удалить (33, сеть1);

вызовем строку 14' (и еще не созданное нами тело для этой процедуры).

И вот тут приходится вспомнить о принципиальной проблеме, которая, возможно, уже давно мучает вдумчивого читателя. Ведь мы теперь не можем обеспечить целостность сетей! Для того мы и скрывали объект "сеть" в теле пакета, чтобы пользователь не мог написать, скажем,

сеть (33).связан.число := 7;

и нарушить тем самым дисциплину работы с сетью так, что последующее выполнение процедуры

удалить (33);

(в которой есть цикл по массиву связей узла 33) может привести к непредсказуемым последствиям.

Введя объявление (в), пользователь может нарушить целостность объекта сеть1 оператором

сеть1 (33).связан.число := 7;

Теперь читателю должно быть полностью понятна принципиально важная концепция регламентированного (а отнюдь не свободного) доступа к объектам класса "сети", упомянутая при постановке задачи. Нужно позволить пользователю создавать новые сети и работать с ними посредством объявленных операций, но нужно защитить объекты этого класса от нежелательного (несанкционированного) доступа.

Обратите внимание: раньше нежелательный доступ к объекту "сеть" был невозможен для пользователя потому, что объект был невидим, скрыт в теле пакета. Объявив тип "сети" в спецификации, мы сделали видимыми для пользователя и имя типа, и имена (селекторы) полей объектов этого типа. Так что скрыть объявление типа в теле пакета нельзя - ведь нужно оставить видимым имя типа (чтобы объявлять новые сети), а оставить полностью открытым также нельзя - можно нарушить целостность.

Вот если бы ввести тип так, чтобы его имя было видимо, а строение объектов - невидимо (т.е. разделить его спецификацию и реализацию)! Тогда технологическая потребность в регламентированном доступе была бы полностью удовлетворена.

Эта красивая и естественная идея в Аде воплощена в концепции ПРИВАТНЫХ типов данных.

4.8.2. Реализация регламентированного доступа. Приватные типы данных

Подумаем о выразительных средствах, реализующих концепцию регламентированного доступа.

По существу нужно определить некоторую абстракцию данных - проявить то, что существенно (для пользователя - имя типа и операции с объектами этого типа) и скрыть то, что несущественно (и даже вредно знать пользователю - строение объектов). Но ведь аналогичная задача для операционных абстракций решается легко: в спецификацию пакета помещается то, что должно быть видимым (спецификация операции), а в тело - то, что следует скрыть (полное определение операции).

Так что было бы естественным аналогично оформить абстракцию данных - поместить в спецификацию пакета то, что должно быть видимым (что может играть роль минимальной "спецификации типа"), а в тело пакета упрятать полное определение типа. В Аде минимальная "спецификация типа" воплощена конструктом "объявление приватного типа", например

(а')   type сети is private ;

(а также перечнем спецификаций применимых операций).

Полное определение приватного типа кажется естественным, по аналогии с определениями операций, поместить в тело пакета.

Почти так и нужно поступать в Аде. Но помещать полное объявление приватного типа приходится не в тело пакета, а в "полузакрытую" (ПРИВАТНУЮ) часть спецификации пакета, отделяемую от открытой части ключевым словом private .

Спецификация обновленного пакета, который теперь естественно назвать, скажем, "управление_сетями", может выглядеть следующим образом.

```
          package управление_сетями is
       ...   -- как и раньше; строки 2-11.
    type сети is private ;
          ... -- операции над сетями
          ... -- строки 13-18.
     ... -- строки 13'-18'.
          private
                type запись_об_узле is
                      record
                            включен : bollean := false ;
                            связан  : связи ;
                       end record ;
                type сети is array (имя_узла) of запись_об_узле ;
          end управление_сетями ;
```

В общем случае спецификация пакета имеет вид

```
                package имя_пакета is
                      объявления_видимой_части

          [     private
                      объявления_приватной_части  ]
                end имя_пакета ;
```

Квадратные скобки указывают, что приватной части может и не быть (как, например, в пакете управление_сетью). Зачем же в Аде понадобилась приватная часть? Почему нет полной аналогии между операционными абстракциями и абстракцией данных? (Кстати, в языке Модула-2 [5] эта аналогия выдержана полностью). Об этом поговорим позже, а пока завершим разговор о нашем примере.

Семантика приватной части проста. Эту часть можно считать "резидентом тела пакета" в его спецификации. В теле пакета непосредственно доступны все имена, объявленные в спецификации пакета, в том числе и в приватной части. С другой стороны, в использующих этот пакет сегментах видимы только объявления открытой части спецификации.

Продолжая пример с моделированием многих сетей, напишем теперь использующий сегмент - процедуру две_сети.

```
          with управление_сетями ;
          use   управление_сетями ;
          procedure две_сети is
                сеть1, сеть2 : сети ;
          begin
                вставить (узел => 13, в_сеть => сеть1 ) ;
                вставить (узел => 33, в_сеть => сеть1 ) ;
                связать (А_узел => 13, В => узел 33, в_сети => сеть1) ;
                сеть2 := сеть1 ; -- присваивание полных объектов !
          ...
     end две_сети ;
```

Когда управление дойдет до места, отмеченного многоточием, будут созданы две сети: "сеть1" и "сеть2", с узлами 33 и 13, причем эти узлы окажутся связанными между собой.

Таким образом, пользователи могут создавать сети и работать с ними с полной гарантией целостности - все требуемые услуги предоставлены нашим обновленным пакетом. Конечно, для этого нужно подправить тело пакета, поместив туда определения новых операций. Оставим это в качестве упражнения.

Подведем некоторые итоги. Мы познакомились с концепцией регламентированного доступа и с его реализацией (воплощением) посредством приватных типов данных. Доступ к "приватным" данным возможен лишь посредством операций, объявленных в ОПРЕДЕЛЯЮЩЕМ ПАКЕТЕ (таковым для типов "сети", "имя_узла", "связи" и др. служит пакет управление_сетями). Невозможен доступ, основанный на знании строения объектов (т.е. выборкой или индексацией) - оно скрыто в приватной части и в

использующих сегментах неизвестно (в теле определяющего пакета объекты приватных типов ничем не отличаются от любых других - их строение известно, выборка и индексация разрешены!.

4.8.3. Строго регламентированный доступ. Ограниченные приватные типы

В общем случае к объектам приватных типов применимы также операции присваивания и сравнения на равенство и неравенство (полных объектов, как в процедуре две_сети!). Хотя это и удобно (такие операции часто нужны и неразумно заставлять программистов определять их для каждого приватного типа), все-таки концепция строго регламентированного доступа в таких типах не выдержана до конца. Она точно воплощена лишь в так называемых ОГРАНИЧЕННЫХ приватных типах. К объектам таких типов неприменимы никакие предопределенные операции, в том числе присваивания и сравнения - все нужно явно определять (в определяющем пакете). Объявления ограниченных риватных типов выделяются ключевыми словами limited private , например

```
type  ключ is limited private ;
```

Применение к объектам типа "ключ" операций присваивания или сравнения вызовет сообщение об ошибке, если только в определяющем пакете для этого типа не определены свои собственные операции, обозначаемые через ":=", "=" или "/=".

Так как ограниченные приватные типы точно воплощают идею строго регламентированного доступа, интересно понять, в каких же задачах столь строгие ограничения могут быть существенными.

Присваивания. Вспомним, что бывают ЯП вовсе без присваивания. Таковы Базисный Рефал (и наша модель М), чистый Лисп, модель Б. Чтобы соответствовать роли базового языка, Ада должна предоставлять средства развития, позволяющие моделировать и такие языки, причем моделировать в точности, с гарантией целостности создаваемых абстракций. Так что (ограниченные) приватные типы вместе со средством их определения (пакетом) - важнейшее средство развития в Аде, позволяющее моделировать по существу любой новый язык. Итак, мы не только указали класс задач, для которых важны такие типы, но и осознали роль приватных типов как характерной принадлежности базового языка.

Упражнение (повышенной сложности). Создайте определяющий пакет для каждой из наших моделей (Н,М и Б).

Примечание. Могут понадобиться ссылочные типы, которыми мы еще подробно не занимались.

Замечание. Подтверждением принципа цельности (согласованности данных, операций, связывания) служит наблюдение Дейкстры, что присваивание нужно лишь в тех языках, где есть повторения (циклы). Именно в циклах нужны переменные, которым нужно присваивать значения (постоянные относительно очередного исполнения тела цикла) .

Когда циклов нет и потенциальная бесконечность обслуживается рекурсией (как в моделях М и Б), достаточна ИНИЦИАЛИЗАЦИЯ (частный случай присваивания - присваивание начальных значений). Так что наличие циклов должно быть согласовано не только с другими операциями (присваиванием), но и с данными (появляются переменные), а также со связыванием (областью действия каждой переменной в таких языках в идеале должен быть некоторый цикл).

Вопрос. Что можно сказать в этих условиях об исходных данных и результатах?

Конец замечания.

Еще пример. Допустим, что нужно моделировать выпуск изделий с уникальными номерами (вспомним заводские номера двигателей, автомобилей, самолетов и т.п.). Естественно считать, что объект приватного типа "изделие" - результат базовой для этого типа функции "создать", генерирующей очередное "изделие". Если позволить присваивать объекты этого типа переменным, то станет возможным их дублировать и уникальность будет нарушена. Поэтому тип "изделие" должен быть ограниченным приватным.

Сравнения. Во-первых, эти операции для объектов некоторых типов могут попросту не иметь смысла, т.е. объекты могут быть несравнимыми. Из соображений надежности попытку их сравнения следует квалифицировать как ошибку (т.е. сравнение должно быть запрешено). Так, при первоначальном знакомстве с Адой упоминались задачные типы, объекты которых - параллельно исполняемые задачи. Сравнение таких

объектов на равенство (при отсутствии присваивания) бессмысленно просто потому, что любой из них уникален по определению. Да и содержательно трудно приписать какой либо естественный смысл сравнению на равенство (а не, скажем, на подобие) двух независимо исполняемых задач. Ведь они в постоянном изменении, причем асинхронном, а всякое реальное сравнение занимает время.

Поэтому в Аде задачные типы - ограниченные приватные по определению и, следовательно, всякая попытка сравнить задачи на равенство квалифицируется как ошибка. Кстати, любые составные типы с компонентами ограниченного типа считаются ограниченными.

Во-вторых, нежелание допускать сравнение на равенство может быть связано с защитой от нежелательного доступа, с обеспечением секретности. Так, и пользователи, и файлы в файловой системе могут быыть снабжены атрибутами типа "ключ". Однако неразумно разрешать пользователю сравнивать ключи с тем, чтобы решать, пользоваться файлом или нет. Право сравнивать ключи и разрешать доступ должно быть только у самой файловой системы. Поэтому для пользователя тип "ключ" должен быть ограниченным - он сможет его, скажем, передать другому, но не сможет "подделать" или "подобрать".

4.8.4. Инкапсуляция

Определение приватных типов данных - один из примеров инкапсуляции - заключения определяемого в "защитную оболочку", предохраняющую от разрушения. Мы видели, как недоступность (защита) строения объектов приватных типов от произвольного доступа со стороны пользователя гарантирует их целостность в некотором содержательном смысле.

Понятие инкапсуляции занимает по общности промежуточное положение между концепцией регламентированного доступа и ее конкретной реализацией приватными типами данных.

Приватные типы Ады с точки зрения пользователя - это инкапсулированные (защищенные) типы данных. Однако с точки зрения реализатора тела определяющего пакета - это обычные незащищенные типы. В общем случае инкапсулировать можно не только типы, но и отдельные объекты или системы объектов (таковы объекты, объявленные в теле пакета или блока).

Итак, задача о моделировании сетей помогла проявить технологическую потребность в инкапсуляции и дала повод поработать с конструктами, удовлетворяющими эту потребность - с объявлениями приватного типа и пакетами.

Накоплено достаточно фактического материала, чтобы продолжить анализ общей концепции типа данных.

4.9. Характеристики, связанные с типом. Класс значений, базовый набор операций

Анализируя концепцию типа, мы концентрировали внимание на способе привязки характеристик к объектам данных и пришли к концепции типа, ориентированной на имена (с однозначным соответствием

объект --> тип ,

определенным на всех объектах). Сутью характеристик мы при этом не интересовались.

И вот еще один пример пользы разумной абстракции. Абстракция от сути характеристик привела к концепции уникальности типа. Концепция уникальности - к простоте прогнозирования-контроля. А простота прогнозирования-контроля позволяет по-новому взглянуть на саму концепцию типа.

Действительно, в ранних ЯП под характеристиками данных обычно понимались характеристики их значений (уже упоминавшиеся разрядность, точность, вид выделяемой памяти и т.п.). Конечно, неявно всегда подразумевалось, что к значениям с одними характеристиками применимы одни операции, а к значениям с другими характеристиками - другие. Но вполне могло оказаться (да так и оказывается), что к объектам с разными характеристиками применимы "одни и те же" операции. Пример - та же операция сложения или процедура печати. И раньше чувствовалось, что совокупность применимых операций - существенная содержательная характеристика класса данных (ведь это естественно следует из понятия данного как абстракции от применяемой операции). Но пока для определения класса данных, к которым применима операция, нужно было (как, скажем, в Алголе 68) производить нетривиальные расчеты эквивалентности характеристик, было трудно воплотить эту идею в точных языковых конструктах.

Концепция типа, ориентированная на имена, упростила связывание с типом любых характеристик. Теперь легко узнать типы данных, связанных с любой операцией - их имена явно указаны в спецификациях параметров операции. Может показаться, что нетрудно проделать и обратное - указать для каждого типа все применимые операции.

Но представим себе, что пакет управление_сетями предоставлен в распоряжение большого коллектива пользователей. Каждый из них волен написать сегменты, использующие этот пакет, и в этих сегментах ввести свои операции над объектами типа "сети". Например, один ввел процедуру выделения связных подсетей, другой - процедуру подсчета хроматического числа сети, третий - вычисления минимального маршрута между двумя узлами. Следует ли считать характеристикой типа "сети" набор из всех этих операций? И каков содержательный смысл в такой характеристике? Ведь в разных контекстах доступны разные элементы этого набора .

Лучше считать характеристикой типа относительно постоянное его свойство, связанное с ним в любых контекстах. Таковы КЛАСС ЗНАЧЕНИЙ и БАЗОВЫЙ НАБОР ОПЕРАЦИЙ. В Аде они связываются с типом соответственно его ОБЪЯВЛЕНИЕМ и ОПРЕДЕЛЯЮЩИМ ПАКЕТОМ.

4.10. Воплощение концепции уникальности. Определение и использование типа в Аде

Концепция типа воплощена в Аде, в основном, четырьмя конструктами: объявлением типа, пакетом, объявлением подтипа и объявлением объекта. В совокупности они и позволяют считать, что тип в Аде обладает, кроме имени, еще двумя важнейшими характеристиками - классом значений и набором применимых операций. С каждым из названных четырех конструктов мы уже встречались. Поговорим подробнее об их строении, смысле и взаимодействии.

Объявление типа вводит имя нового типа и связывает это имя с конструктором типа. Последний служит для создания нового типа из уже известных. Примеры конструкторов типа нам уже встречались - они располагаются в объявлениях типов после ключевого слова is.

Создать (объявить) новый тип в Аде - это значит определить класс допустимых значений объектов этого типа и набор базовых операций, связанных с создаваемым типом.

В Аде предопределены некоторые типы данных (обслуживающие наиболее общие потребности проблемной области - универсальный_целый. универсальный_вещественный и другие), а также класс допустимых значений перечисляемых типов. Кроме того, предопределены операции, применимые к объектам целых категорий типов. Например, со всеми регулярными типами связана операция получения указателя на компоненту массива (индексация), со всеми комбинированными типами связана операция получения указателя на компоненту записи - выборка (СЕЛЕКЦИЯ), со всеми так называемыми ДИСКРЕТНЫМИ типами - получение по заданному значению последующего или предыдущего. Если явно не оговорено обратное, то к объектам любого типа можно применять сравнение на равенство и неравенство, извлечение и присваивание значения.

Предопределенные типы, классы значений и операции служат исходным материалом для нескольких категорий конструкторов типа - у каждой категории типов свой конструктор. Подробнее мы рассмотрим их позже, каждый в связи с соответствующей категорией типов.

Объявление типа (а, следовательно, и конструктор типа) служит для того, чтобы полностью и окончательно определить класс допустимых значений и в общем случае лишь частично определить базовые операции. Полный и окончательный набор базовых операций фиксируется определяющим пакетом, в спецификации которого вместе с объявлением нового типа присутствуют и объявления новых операций, связанных с ним.

Например, конструктор ПРОИЗВОДНОГО типа, создавая тип имя_узла, определяет для него класс (в данном случае - диапазон) значений и связывает с типом имя_узла обычные операции над целыми числами (наследуемые у родительского типа INTEGER). Остальные базовые операции для объектов типа имя_узла (вставить, удалить, связать и др.) определены в конце спецификации пакета управление_сетью, который и служит для этого типа определяющим пакетом.

Еще пример. Конструктор КОМБИНИРОВАННОГО типа, создавая тип "связи", определяет для него, во-первых, класс значений (записи с двумя полями - первое с СЕЛЕКТОРОМ "число" типа число_связей, второе с селектором "узлы" типа "перечень-связей"). Во-вторых, этот конструктор определяет обычные для всех комбинированных типов операции доступа как ко всему значению объекта (по имени объекта), так и к

его компонентам по составным именам с использованием селекторов. Еще одна (и последняя) базовая операция для этого типа определена в пакете управление_сетью - это операция все_связи. Доступ к полному значению объекта типа "связи" использован в реализации функции все_связи (в операторе возврата), а доступ к одному полю по селектору - в реализации операций вставить, удалить, чистить, переписать.

Конструктор ПРИВАТНОГО типа , создавая тип "сети", определяет в качестве базовых операций только присваивание и сравнение на равенство и неравенство. Определяющий пакет управление_сетями добавляет базовые операции вставить, удалить и др.

[Обратите внимание: одни и те же операции могут быть базовыми для различных типов! За счет чего?].

Класс значений для типа "сети" определен полным объявлением в приватной части, однако пользователю этот класс остается "неизвестным" (непосредственно недоступным).

Запас предопределенных типов, значений и операций - это базис языка, а конструкторы типа - характерный пример средств развития [Вспомните: в наших моделях ЯП были только конструкторы операций]. Процесс развития языка (с помощью любых конструкторов) начинается с применения конструкторов к базису. На очередном шаге развития конструкторы применяются к любым уже определенным сущностям (в том числе и к базисным). [Применение конструкторов к базису имеет в Аде свои особенности, которыми мы заниматься не будем].

Об объявлениях подтипа и объекта подробнее поговорим после знакомства с перечисляемыми типами.

4.11. Конкретные категории типов

4.11.1. Перечисляемые типы. "Морская задача"

Займемся технологической потребностью, которая часто встречается в так называемых дискретных задачах (в дискретной математике вообще). Речь идет о потребности работать с относительно небольшими конечными множествами.

Замечание. Когда конечные множества очень большие, то с точки зрения программирования они могут оказаться неотличимыми от множеств бесконечных. Специфика конечного сказывается тогда, когда мощность множества удается явно учесть в программе. При этом, в соответствии с принципом согласования абстракций, средства объявления конечных множеств должны быть согласованы со средствами манипулирования как множествами в целом, так и отдельными их элементами.

Конец замечания.

Рассмотрим очень упрощенную задачу управления маневрами корабля.

Содержательная постановка. Корабль движется по определенному курсу. Поступает приказ лечь на новый курс. Требуется рассчитать команду, которую следует отдать рулевому, чтобы направить корабль по новому курсу (совершить нужный маневр).

Модель задачи. Будем считать, что возможных курсов только четыре: север, восток, юг и запад. Можно отдать лишь одну из четырех команд: прямо, налево, направо и назад. Исполнение команд "налево" и "направо" изменяет курс на 90 градусов. Требуется рассчитать новую команду по заданным старому и новому курсам. Например, для старого курса "восток" и нового курса "север" нужный маневр выполняется по команде "налево".

Программирование (полная формализация задачи). Попытаемся, как уже не раз делали, применить пошаговую детализацию.

Шаг 1. Нужно "рассчитать" команду. Это уже не комплекс услуг, а одна операция. Представим ее функцией, для которой старый и новый курсы служат аргументами, а команда рулевому - результатом. Чтобы сразу написать спецификацию такой функции на Аде, нужно верить, что на последующих шагах можно будет ввести подходящие типы аргументов и результата. Поверим, не думая пока об особенностях этих типов. Тогда нужную операционную абстракцию можно воплотить следующей спецификацией.

```
function  маневр (старый, новый : курс) return команда ;
```

Шаг 2. Как формализовать понятие "команда"?

С точки зрения языка программирования это должен быть тип данных - ведь имя "команда" использовано в спецификации формальных параметров функции.

Следоватедьно, нужно определить связанные с этим типом значения и базовые операции.

С содержательной точки зрения ясно, что нужны только названия конкретных команд. Другими словами, нам требуется лишь различать отдельные команды. Что с ними можно делать - неясно, это выходит за рамки модели нашей задачи (наша модель неполна в том отношении, что в ней отсутствует модель устройства корабля и модель рулевого как части корабля). Поэтому, формализуя понятие "команда", удовлетворимся тем, что проявим список существенных команд и отразим в названиях команд их роль в управлении кораблем.

Существенных команд всего четыре: прямо, налево, направо, назад. Таким образом, нужно объявить тип данных с четырьмя перечисленными значениями. Ада позволяет это сделать с помощью следующего объявления ПЕРЕЧИСЛЯЕМОГО типа.

```
type команда is (прямо, налево, направо, назад) ;
```

Не зря эта категория типов названа "перечисляемыми типами" - все допустимые значения явно перечислены.

Замечание. Такие типы называют иногда "перечислимыми". Однако этот термин в математике занят, относится тоже к множествам и имеет другой смысл (перечислимые множества вполне могут быть бесконечными). К тому же специфика рассматриваемых типов в том и состоит, что их значения явно перечисляют при определении такого типа.

Конец замечания.

Перечисляемые типы придуманы Н.Виртом и впервые появились в созданном им языке Паскаль. До сих пор мы имели дело с такими типами, для которых имело смысл говорить об описании множества значений, о создании нового или разрушении старого значения. Но никогда не шла речь о явном перечислении в программе всех значений некоторого типа. Другими словами, определения типов были всегда интенсиональными и никогда - экстенсиональными. Это было и не нужно и, как правило, невозможно - шла ли речь о целых или вещественных числах, о массивах или записях. "Не нужно" означает, что такова была дисциплина применения этих типов. "Невозможно" связано с их практической бесконечностью.

В нашем случае и можно, и нужно давать экстенсиональное определение типа, явно перечислив все значения типа "команда". Можно, потому что их всего четыре, и мы их уже перечислили. А зачем это нужно?

Нужно потому, что всякое интенсиональное определение опирается на существенные индивидуальные свойства элементов определяемого множества. А в нашем случае таких свойств нет! Годятся любые различные имена для команд (желательно мнемоничные, отражающие содержательную роль команд в нашей задаче). Захотим, будет {прямо, налево, направо, назад}, не понравится - сделаем {вперед, влево, вправо, обратно} или {так_держать, лево_руля, право_руля, задний_ход} и т.п. Поэтому нет иного способа догадаться о принадлежности конкретного имени к типу "команда", кроме как увидеть его в соответствующем списке.

Итак, мы познакомились с объявлением перечисляемых типов, значения которых - мнемонические имена компонент модели решаемой задачи (в нашем случае - имена команд). До изобретения Вирта программисты явно кодировали такие компоненты (обычно целыми числами). В сущности, Вирт предложил полезную абстракцию от конкретной кодировки, резко повышающую надежность, понятность и модифицируемость программ без заметной потери эффективности.

Шаг 3. Как формализовать понятие "курс"?

Создавая подходящую абстракцию, выделим то, что важно для решения задачи. Важно, во-первых, различать курсы, во-вторых, поворачивать налево, направо и назад.

Итак, следует представить абстракцию "курс" типом данных, объекты которого в качестве значений могут принимать названия конкретных курсов (север, восток, юг, запад), причем к ним можно применять операции "налево", "направо", "назад". Мы уже умеем вводить типы данных с подходящими базовыми операциями. Для этого нужно написать определяющий пакет, где и дать объявления типа "курс" и операций поворота.

Нетрудно догадаться, что "курс" должен быть перечисляемым типом.

```
type  курс is (север, восток, юг, запад) ;
```

Ведь снова, как и для типа "команда", с точки зрения решаемой задачи абсолютно несущественно внутреннее строение этих значений. Поэтому невозможно вводить "курс" каким-либо конструктором составного типа. Действительно, из чего

состоит "север" или "восток"? Важно лишь, что это разные сущности, связанные между собой только тем, что направо от севера - восток, а налево от востока - север.

Таким образом, значения типа "курс", также как и типа "команда", следует считать просто именами компонент модели задачи (точнее, той модели внешнего мира, на которой мы решаем нашу содержательную задачу). Существенные связи этих имен - непосредственное отражение содержательных связей между именуемыми компонентами внешней модели. Мы пришли еще к одной причине, по которой нам нужны в программе все такие имена-значения - иначе не запрограммировать базовые функции (ведь нет никаких внутренних зависимостей между значениями, только внешние, а их-то и нужно отражать).

Когда мы программировали базовую функцию "связать" в пакете управление_сетью, нам, наоборот, были абсолютно безразличны индивидуальные имена узлов - можно было программировать, опираясь на внутреннее строение именуемых объектов (на строение записей_об_узле в массиве "сеть"). Это внутреннее строение создавалось пользователем, работающим с пакетом, посредством других базовых операций. Когда же будем программировать, скажем, операцию "налево", никакое внутреннее строение не подскажет нам, что налево от юга находится восток. Это следует только из модели внешнего мира, создаваемой нами самими, т.е. в данном случае создателем пакета, а не пользователем. Поэтому мы обязаны явно сопоставить восток - югу, север - востоку, юг - западу.

Замечание.

Конечно, часто прибегают к такому кодированию внешних объектов, чтобы внутренние свойства кодов отображали закономерности (связи) внешнего мира. В нашем случае, закодировав страны света (курсы) последовательными целыми числами от 0 до 3, можно отразить связи между соседними курсами (особенно если вычислять по модулю 4). Не отвергая такого приема в принципе (тем более что в Аде можно управлять представлением перечисляемых типов), заметим, что нетрудно усложнить задачу так, чтобы простое кодирование ничего хорошего не дало - достаточно рассмотреть повороты в пространстве, а не на плоскости. К тому же, когда учитывают полную стоимость программных услуг, выгоды от "хитрого" кодирования часто оказываются призрачными.

Конец замечания.

Замечание о природе значений перечисляемых типов.

Итак, начав с особенностей значений типа "курс", мы отметили и характерную особенность его базовых операций - для их реализации не требуется знать внутреннее строение значений этого типа.

Таким образом, создавая абстракцию "курс", мы попадаем в ситуацию, когда внутренняя природа значений нас не интересует абсолютно. Ни с точки зрения пользователя абстракции "курс", ни с точки зрения программиста, реализующего эту абстракцию. Первому важно иметь возможность указать (идентифицировать) нужное значение и воспользоваться связью различных значений, воплощенной в базовых операциях. Второму также важно и идентифицировать значения, и иметь возможность запрограммировать базовые операции. Но последние, как мы выяснили, касаются только связей различных значений, так что и для реализатора внутренняя природа значений безразлична.

С ситуацией, когда строение значений скрыто от пользователя, но отнюдь не безразлично для реализатора, мы уже встречались, когда изучали приватные типы данных. Теперь строение не скрыто, но существенно лишь постольку, поскольку обеспечивает идентификацию значений. В остальном можно считать, что его просто нет - перед нами список имен объектов внешнего мира, играющих определенные роли в решаемой задаче.

Перечисляемые типы похожи на приватные тем, что также создают для пользователя определенный уровень абстракции - пользователь вынужден работать только посредством операций, явно введенных для этих типов. Однако если операции приватных типов обычно обеспечивают доступ к скрытым компонентам содержательных "приватных" объектов, то операции перечисляемых типов отражают связи между теми содержательными объектами, которые названы явно перечисленными в объявлении типа именами [нужно понимать, что приватный тип вполне может оказаться реализованным некоторым перечисляемым типом].

Конец замечания.

Пора вспомнить о нашей задаче. Завершая шаг детализации, определим перечисляемый тип "курс" вместе с базовыми операциями-поворотами. Сделаем это с помощью спецификации пакета "движение".

```
package движение is
      type курс is (север, восток, юг, запад) ;
      function налево (старый : курс) return курс ;
      function направо (старый : курс) return курс ;
      function назад  (старый : курс) return курс ;
end движение ;
```

Итак, определили тип "курс" с четырьмя именами-значениями и тремя базовыми операциями.

Шаг 4. Тело функции "маневр".

Идея в том, чтобы понять, каким поворотом можно добиться движения в нужном направлении, и выдать соответствующую команду.

```
function маневр (старый, новый : курс) return команда ;
begin
      if новый = старый then return прямо ;
      elsif новый = налево(старый) then return налево ;
      elsif новый = направо(старый) then return направо ;
      else return назад ;
 end if ;
end маневр ;
```

Мы свободно пользовались сравнением имен-значений на равенство. Условный оператор с ключевым словом elsif можно считать сокращением обычного условного оператора. Например,

```
if B1 then S1 ;
elsif B2 then S2 ;
elsif B3 then S3 ;
end if ;
```

эквивалентно оператору

```
if B1 then S1 else if B2 then S2
       else if B3 then S3
                  end if ; end if ; end if ;
```

Такое сркращение удобно, когда нужно проверять несколько условий последовательно.

Тем самым программирование функции "маневр" завершено.

Если считать, что "маневр" - лишь одна из предоставляемых пользователю услуг, можно включить ее в пакет со следующей спецификацией.

```
package услуги is
      type команда is (прямо, налево, направо, назад) ;
            package движение is
      type курс is (север, восток, юг, запад) ;
      function налево (старый : курс) return курс ;
      function направо (старый : курс) return курс ;
      function назад  (старый : курс) return курс ;
end движение ;
            use движение ;
            function маневр (старый, новый : курс) return
команда ;
      end услуги ;
```

Замечания о конструктах.

Во-первых, видно, что пакет ("движение") может быть объявлен в другом пакете. Чтобы воспользоваться объявленными во внутреннем пакете именами при объявлении функции "маневр", указание контекста (with) не нужно. Но указание сокращений (use) обязательно, иначе пришлось бы писать

```
function маневр (старый, новый : движение.курс) return команда ;
```

Во-вторых, имена функций совпадают с именами команд (обратите внимание на тело функции "маневр"). Это допустимо. Даже если бы возникла коллизия наименований, имена функций всегда можно употребить с префиксом - именем пакета. Например, движение.налево, движение.назад, а имена команд - употребить с так называемым КВАЛИФИКАТОРОМ. Например, команда(налево), команда(направо), команда(назад). На самом деле в нашем случае ни префиксы, ни квалификаторы не

нужны, так как успешно действуют правила перекрытия - по контексту понятно, где имена команд, а где функции.

В-третьих, функция "маневр" применима к типу данных "курс", но не входит в набор его базовых операций. Ведь определяющий пакет для этого типа - "движение". Зато для типа "команда" функция "маневр" - базовая операция.

Конец замечания.

Осталось выполнить

Шаг 5. Функции пакета "движение".

```
function налево (старый : курс) return курс is
      begin
            case старый of
                  when север => return запад ;
                  when восток => return север ;
  when юг  => return восток ;
  when запад => return юг ;
 end case ;
end налево ;
```

Замечание о согласовании абстракций.

Перед нами - наглядное подтверждение принципа цельности. Раз в Аде есть способ явно описывать "малые" множества (вводить перечисляемые типы), то должно быть и средство, позволяющее непосредственно сопоставить определенное действие каждому элементу такого множества. Таким средством и служит ВЫБИРАЮЩИЙ ОПЕРАТОР (case). Между ключевыми словами case и of записывается УПРАВЛЯЮЩЕЕ ВЫРАЖЕНИЕ некоторого перечисляемого типа (точнее, любого ДИСКРЕТНОГО типа - к последним относятся и перечисляемые, и целые типы с ограниченным диапазоном значений). Между of и end case записываются так называемые ВАРИАНТЫ. Непосредственно после then (когда) записывается одно значение, несколько значений или диапазон значений указанного типа, а после "=>" - последовательность операторов, которую нужно выполнить тогда и только тогда, когда значение управляющего выражения равно указанному значению (или попадает в указанный диапазон).

Выбирающий оператор заменяет условный оператор вида

```
if старый = север then return запад ;
         elsif старый = восток then return север ;
         elsif старый = юг     then return восток;
         elsif старый = запад  then return юг    ;
         end if ;
```

Замечание. Условный и выбирающий операторы - частные случаи реализации развилки - одной из трех основных управляющих структур (последовательность, развилка, цикл), используемых в структурном программировании.

Вопрос: Чем выбирающий оператор Ады отличается от развилки в языке Дейк?

Конец замечания.

По сравнению с условным, выбирающий оператор, во-первых, компактнее (не нужно повторять выражение); во-вторых, надежнее - и это главное.

Дело в том, что варианты обязаны охватывать все допустимые значения анализируемого типа и никакое значение не должно соответствовать двум вариантам. Все задаваемые после when значения (и диапазоны) должны быть вычисляемы статически (т.е. не должны зависеть от исходных данных программы, с тем чтобы компилятор мог их вычислить). Так что компилятор в состоянии проверить указанные требования к вариантам выбирающего оператора и обнаружить ошибки.

Наконец, статическая вычислимость обеспечивает и третье преимущество выбирающего оператора - его можно эффективно реализовать (значение выбирающего выражения может служить смещением относительно начала вариантов).

Теперь читатель в состоянии самостоятельно запрограммировать функции "направо" и "назад", завершив тем самым решение нашей "морской" задачи.

Позже мы еще попрактикуемся в определении и применении перечисляемых типов. А сейчас посмотрим на "морскую" задачу из другой языковой среды.

4.11.1.1. Морская задача и Алгол.

Представим, что в нашем распоряжении не Ада, а Алгол.

Технология. Уже на первом шаге детализации нам не удалось бы ввести подходящую операционную абстракцию. Помните, нам была нужна уверенность в возможности определить подходящие типы для понятий "курс" и "команда". В Алголе вообще нет возможности определять типы, в частности, перечисляемые. Поэтому пришлось бы "закодировать" курсы и команды целыми числами. Скажем, север - 1, восток - 2, юг - 3, запад - 4; команда "прямо" - 1, "налево" - 2, "направо" - 3, "назад" - 4. Заголовок функции "маневр" выглядел бы, скажем, так:

```
integer procedure маневр (старый, новый) ;
  integer старый, новый ; value старый, новый ;
```

Приступая к проектированию тела функции, мы не имели бы случая предварительно создать абстрактный тип "курс" с операциями поворота. Но ведь именно с операциями поворота связана основная идея реализации функции "маневр" на Аде! Вспомните, чтобы подобрать подходящую команду, мы проверяли возможность получить новый курс из старого определенным поворотом. Если бы применяемая технология программирования не требовала вводить абстрактный тип "курс", то и идея реализации функции "маневр" вполне могла оказаться совсем другой. Не было бы удивительным, если ее тело было запрограммировано "в лоб", скажем, так:

```
   begin маневр :=
     if старый = новый then 1
else if старый = 1 & новый = 4 v старый = 2 & новый = 1
  v старый = 3 & новый = 2 v старый = 4 & новый = 3 then 2
     else if старый = 1 & новый = 2 v старый = 2 & новый = 3
  v старый = 3 & новый = 4 v старый = 4 & новый = 1 then 3
     else if старый = 1 & новый = 3 v старый = 2 & новый = 4
  v старый = 3 & новый = 1 v старый = 4 & новый = 2 then 4 ;
   end маневр ;
```

Конечно, "классические" программисты постарались бы "подогнать" кодировку курсов и команд с тем, чтобы заменить прямой перебор "вычислением" команды. Однако такой прием неустойчив по отношению к изменению условий задачи и в общем случае с большой вероятностью может оказаться ошибочным. К тому же оно менее понятно по сравнению с решением на Аде. Нужно "держать в голове" кодировку, чтобы понимать программу.

Таким образом, отсутствие средств, поддерживающих нужные абстракции (в частности, в процессе пошаговой детализации), вполне может помешать и наиболее творческим моментам в программировании, помешать увидеть изящное, надежное, понятное и эффективное решение.

Надежность. Внимательнее сравним программы на Аде и Алголе с точки зрения надежности предоставляемой услуги. Чтобы воспользоваться операцией "маневр", на Аде можно написать, скажем,

```
маневр (север, восток) ;
```

а на Алголе -

```
маневр (1, 2) ;
```

Ясно, что первое - нагляднее, понятнее (а, значит, и надежнее). Но высокий уровень надежности гарантируется не только наглядностью, но и контролем при трансляции. На Аде нельзя написать

```
маневр (1, 2)
```

- транслятор обнаружит несоответствие типов формальных и фактических параметров! На Алголе в нашей реализации можно написать

```
маневр (25, 30) ;
```

и получить 4. А чтобы получить тот же уровень контроля, который автоматически обеспечивает Ада-компилятор, нужно добавить в программу явные проверки диапазона целых значений и обращение к соответствующим диагностическим процедурам. И все это будет работать динамически, а в Аде - статически. Так что надежность при программировании на Алголе может быть обеспечена усилиями только самого программиста и только за счет снижения эффективности объектной программы.

Можно постараться добиться большей наглядности, введя переменные "север", "восток", "юг" и "запад" (постоянных в Алголе нет!). Им придется присвоить

значения (1, 2, 3, 4) - также во время работы объектной программы, но зато окажется возможным писать столь же понятно, как и на Аде:

маневр (север, восток) ;

Однако в отличие от имен-значений перечисляемого типа в Аде, которые по определению - константы, эти переменные не защищены от случайных присваиваний. Не говоря уже о защите от применения к ним других операций (в Аде к значениям определенного перечисляемого типа применимы, конечно, только операции, формальные параметры которых соответственно специфицированы).

Заключение. Итак, мы выделили технологическую потребность определять небольшие множества имен и работать с ними на таком уровне абстракции, когда указываются лишь связи этих имен между собой и с другими программными объектами. Эта потребность и удовлетворяется в Аде перечисляемыми типами данных. Важно, что удовлетворяется она комплексно, в соответствии с важнейшими общими принципами (такими, как принцип согласованности абстракций) и специфическими требованиями к языку (надежность, понятность и эффективность программ). Должно быть ясно также, что перечисляемые типы не могут быть полностью заменены аппаратом классических языков.

На самом деле возможности, предоставляемые перечисляемыми типами в Аде, еще богаче. Мы познакомимся с ними в рамках более широкой категории типов.

4.11.2. Дискретные типы

Перечисляемые типы - частный случай так называемых ДИСКРЕТНЫХ типов. Дискретным называется тип, класс значений которого образует ДИСКРЕТНЫЙ ДИАПАЗОН, т.е. конечное линейно-упорядоченное множество. Это значит, что в базовый набор операций для дискретных типов входит, во-первых, операция сравнения "меньше", обозначаемая обычно через "<"; во-вторых, функции "первый" и "последний", вырабатывающие в качестве результатов, соответственно, минимальный и максимальный элементы диапазона и, в_третьих, функции "предыдущий" и "последующий" с очевидным смыслом. Эти операции для всех дискретных типов предопределены в языке Ада.

Кроме перечисляемых, дискретными в Аде являются еще и ЦЕЛЫЕ типы. Класс значений любого целого типа считается конечным. Для предопределенного типа INTEGER он фиксируется реализацией языка (т.е. различные компиляторы могут обеспечивать различный диапазон предопределенных целых; этот диапазон должен быть указан в документации на компилятор; кроме того, его границы доставляются (АТРИБУТНЫМИ) функциями "первый" и "последний"). Для определяемых целых типов границы диапазона значений явно указываются в объявлении целого типа (см. объявление типа имя_узла).

Любые дискретные типы можно использовать для индексации и управления циклами. Мы уже встречались и с тем, и с другим в пакете управление_сетями.

В "морской" задаче нам не потребовались предопределенные базовые операции для типа "курс". Но в соответствии с его объявлением

```
                                        север < восток < юг < запад
    причем
    последующий (север) = восток;
    предыдущий (восток) = север ;
    курс'первый = север ;
    курс'последний = запад ;
```

Так что функцию "налево" можно было реализовать и так:

```
    function налево (старый: курс) return курс is
            begin
                case старый of
                        when север => return запад ;
                        when others  => return предыдущий (старый) ;
                end case ;
    end налево ;
```

Обратите внимание, функция "предыдущий" неприменима к первому элементу диапазона (как и функция "последующий" - к последнему элементу).

Вариант when others в выбирающем операторе работает тогда, когда значение выбирающего выражения (в нашем случае - значение формального параметра "старый") не соответствует никакому другому варианту. Выбирающее выражение должно быть дискретного типа (вот еще одно применение дискретных типов, кроме индексации и управления циклами) и каждое допустимое значение должно соответствовать одному и только одному варианту. Такое жесткое правило было бы очень обременительным без

оборота when others. С его помощью можно использовать выбирающий оператор и тогда, когда границы диапазона изменяются при выполнении программы, т.е. являются относительно-статическими. Конечно, при этом изменяется не тип выбирающего выражение, а лишь его подтип - динамические границы не могут выходить за рамки статических границ, определяемых типом выражения.

Вообще, если D - некоторый дискретный тип, то справедливы следующие соотношения. Пусть X и Y - некоторые значения типа D. Тогда

последующий (предыдущий (X)) = X, если X /= D'первый;
предыдущий (последующий (X)) = X, если X /= D'последний;
предыдущий (X) < X, если X /= D'первый.

Для дискретных типов предопределены также операции "<=" (меньше или равно), ">", ">=", "=" и "/=" с очевидным смыслом.

Вот еще несколько примеров дискретных типов. Предопределены дискретные типы BOOLEAN, CHARACTER , INTEGER . При этом считается, что тип BOOLEAN введен объявлением вида

```
type BOOLEAN is (true, false) ;
```

так что true < false.

Для типа CHARACTER в определении языка явно перечислены 128 значений-символов, соответствующих стандарту ASCII, среди которых первые 32 - управляющие телеграфные символы, вторые 32 - это пробел, за которым следуют !"#$%&'()*+,-./0123456789:;<=>?, третьи 32 - это коммерческое at (@), за которым идут большие латинские буквы, затем [\]^_; наконец, последние 32 - знак ударения '; затем малые латинские буквы, затем {|}, затем тильда ~ и символ вычеркивания.

Границы диапазона для целого типа INTEGER считаются зависящими от реализации. Их можно узнать с помощью атрибутных функций INTEGER'первый и INTEGER'последний.

Для типов BOOLEAN, CHARACTER и INTEGER предопределены обычные операции для дискретных типов. [Мы привели не все такие операции]. Кроме того, для типа BOOLEAN предопределены обычные логические операции and, or, xor и not с обычным смыслом (xor - исключительное "или").

Для типа INTEGER предопределены также унарные операции "+", "-", "abs" и бинарные "+", "-", "*", "/", "**" (возведение в степень) и другие.

Вот несколько примеров определяемых дискретных типов.

```
type день_недели is (пн, вт, ср, чт, пт, сб, вс) ;
type цвет is (красный, желтый, зеленый) ;
type код_операции is (сложить, вычесть, умножить, разделить) ;
type месяц is (январь, февраль, март, апрель, май, июнь, июль,
     август, сентябрь, октябрь, ноябрь, декабрь) ;
type год is new INTEGER range 0..2099 ;
type этаж is new INTEGER range 1..100 ;
```

4.11.3. Ограничения и подтипы

Проанализируем еще одну технологическую потребность - потребность ограничивать множество значений объектов по сравнению с полным классом значений соответствующего типа. Рассмотрим фрагмент программы, меняющей знак каждого из десяти элементов вектора A.

```
for j in 1..10 loop
  A(j) := -A(j) ;
end loop ;
```

Перед нами фрагмент, который будет правильно работать только при условии, что вектор A состоит ровно из десяти элементов. Иначе либо некоторые элементы останутся со старыми знаками, либо индекс выйдет за границу массива. К тому же такой фрагмент способен работать только с вектором A и не применим к вектору B. Другими словами, это очень конкретный фрагмент, приспособленный для работы только в специфическом контексте, плохо защищенный от некорректного использования. [В чем это проявляется?].

Допустим, что потребность менять знак у элементов вектора возникает достаточно часто. Вместо того, чтобы каждый раз писать аналогичные фрагменты, хотелось бы воспользоваться принципом обозначения повторяющегося и ввести подходящую подпрограмму, надежную и пригодную для работы с любыми векторами. Другими словами, мы хотим обозначить нечто общее, характерное для многих конкретных действий, т.е. ввести операционную абстракцию.

От чего хотелось бы отвлечься? Повидимому, и от конкретного имени вектора, и от конкретной его длины. Возможно, от конкретного порядка обработки компонент или от конкретной размерности массива. Можно ли это сделать и целесообразно ли, зависит от многих причин. Но прежде всего - от возможностей применяемого ЯП. Точнее, от возможностей встроенного в него аппарата развития (аппарата абстракции-конкретизации).

Здесь уместно сформулировать весьма общий принцип проектирования (в частности, языкотворчества и программирования). Будем называть его принципом реальности абстракций.

4.11.3.1. Принцип реальности абстракций

Назовем реальной такую абстракцию, которая пригодна для конкретизации в используемой программной среде.

Тогда принцип реальности можно сформулировать так:

В программировании непосредственно применимы лишь реальные абстракции.

Иначе говоря, создавая абстракцию, не забудь о конкретизации.

Другими словами, следует (создавать возможность) абстрагироваться только от таких характеристик, которые применяемый (создаваемый) аппарат конкретизации позволяет указывать в качестве параметров настройки.

В нашем примере попытаемся построить ряд все более мощных абстракций, следуя за особенностями средств развития в Аде. Скажем сразу - Ада позволяет явно построить первую из намеченных четырех абстракций (мы ведь собрались отвлечься от имени, от длины, от порядка и от размерности); со второй придется потрудиться (здесь-то и понадобятся ограничения и подтипы); третья потребует задачного типа и может быть построена лишь частично, а четвертая вообще не по силам базисному аппарату развития Ады (т.е. для Ады - это нереальная абстракция).

Абстракция от имени. Аппарат развития, подходящий для первой абстракции, нам известен. Чтобы абстрагироваться от имени вектора (и, конечно, настраиваться на конкретное имя в момент связывания) достаточно ввести функцию с параметром и результатом нужного типа.

Что значит "нужного типа"? Пока мы абстрагируемся только от имени вектора, сохраняя все остальные его конкретные характеристики. Поэтому нужен тип, класс значений которого - десятиэлементные векторы. Объявим его.

```
type вектор is array (1..10) of INTEGER ;
```

Теперь нетрудно объявить нужную функцию.

```
function "-" (X : вектор) return вектор is
  Z: вектор ;
  begin
    for j in (1..10) loop
      Z(j) := -X(j) ;
    end loop ;
    return Z ;
  end "-" ;
```

Обратите внимание: такая функция перекрывает предопределенную операцию "-". Становится вполне допустимым оператор присваивания вида

```
A := -A ;
```

где A - объект типа "вектор", а знак "-" в данном случае идентифицирует не предопределенную операцию над числами, а определенную нами операцию над векторами.

Замечание.

В Аде новые знаки операций вводить нельзя. Это сделано для того, чтобы синтаксический анализ текста программы не зависел от ее смысла (в частности, от результатов контекстного анализа). Скажем, знак "[" нельзя применять для обозначения новой операции, а знак "-" - можно.

Именно для того, чтобы продемонстрировать перекрытие знака "-", мы ввели функцию, а не процедуру, хотя в данном случае последнее было бы эффективнее. Действительно, ведь наш исходный фрагмент программы создает массив-результат непосредственно в массиве-аргументе. А функция "-" создает новый массив, сохраняя аргумент неизменным. Так что более точной и эффективной была бы абстракция-процедура следующего вида:

```
procedure минус (X : in out вектор) is
begin
  for j in (1..10) loop
```

```
    X(j) := -X(j) ;
  end loop ;
end минус ;
```

Конец замечания.

Уже при такой слабой абстракции (только от имени вектора) мы оказались перед необходимостью согласовывать операционную абстракцию и абстракцию данных (обратите внимание на диапазон 1..10 и в объявлении функции, и в объявлении типа). Так что принцип согласования абстракций работает и при создании языковых конструктов (при создании языка, на метауровне), и на уровне их применения. При этом согласование на метауровне призвано всячески облегчать согласование на уровне применения.

Вопрос. Нельзя ли упростить последнее в нашем случае?

За счет согласованного объявления управляющей переменной цикла и диапазона допустимых индексов мы повысили надежность программы. Применяя функцию "-", невозможно выйти за границы массива - ведь ее аргументами могут быть только десятиэлементные векторы.

Абстракция от длины вектора (начало). Пойдем дальше по пути абстракции. Как написать функцию, применимую к вектору любой длины? В соответствии с концепцией уникальности типа каждый формальный параметр должен иметь определенный тип. Поэтому возникают два согласованных вопроса (снова действует принцип согласования абстракций!):

Как объявить нужный тип?

Как написать тело процедуры, работающей с массивом произвольной длины?

Здесь полезно на время оторваться от нашего примера и вспомнить об общем контексте, в котором эти вопросы возникли.

4.11.4. Квазистатический контроль

Мы продолжаем заниматься в основном данными - одной из трех важнейших абстракций программирования. Исходя из потребности прогнозирования и контроля поведения объектов (в свою очередь выводимой из более общей потребности писать надежные и эффективные программы), мы пришли к концепции типа данных, ориентированной на имена. Исходя из назначения системы типов, выделили динамические, статические и относительно-статические (говоря короче, квазистатические) языки в зависимости от степени гибкости прогнозирования-контроля. Отметили, что в Аде концепция собственно типа ориентирована на прогнозирование-контроль статических характеристик поведения объектов, а концепция подтипа - на прогнозирование-контроль квазистатических (или, если угодно, квазидинамических) характеристик. Наша ближайшая цель - обосновать полезность концепции подтипа.

Вспомним классификацию данных и в ней - фактор изменчивости. Он играет особую роль, так как касается самого динамического атрибута объекта - его значения.

Поэтому с фактором изменчивости в наибольшей степени связано противоречие между потребностью гибко изменять поведение объектов и потребностью прогнозировать это поведение. Другими словами, это противоречие между потребностью в мощных операционных абстракциях, применимых к весьма разнообразным данным, и потребностью ограничить их применение, чтобы достичь надежности и эффективности.

Первая потребность требует свободы, вторая - дисциплины. Чем точнее удается прогнозировать изменчивость, чем в более жесткие рамки "зажимается" возможное поведение, тем надежнее контроль, больше возможностей экономить ресурсы. Но вместе с этим снижается общность, растут затраты на аппарат прогнозирования-контроля, растут затраты на создание близких по назначению программ, на их освоение.

Мы уже сталкивались с этим противоречием, когда занимались проблемой полиморфизма. Но тогда речь шла о небольшом наборе разновидностей (типов) объектов и противоречие удалось разрешить за счет конечного набора частных операционных абстракций с одним и тем же именем (каждая для своего набора типов объектов). Теперь нас интересует такая ситуация, когда разновидностей неограниченно много. Именно такая ситуация создалась при попытке абстрагироваться от длины вектора в программе "минус".

Абстракция от длины вектора (продолжение). В чем конкретно противоречие?

С одной стороны, нужна как можно более мощная операционная абстракция, применимая к векторам любой длины. Для этого нужно иметь возможность ввести согласованный с такой абстракцией обобщенный тип, значениями которого могут быть

массивы произвольной длины. Иначе возникнет противоречие с концепцией уникальности типа (какое?).

Такой тип в Аде объявить можно. Например, так:

```
type вектор_любой_длины is array (INTEGER range <>) of INTEGER ;
```

Вместо конкретного диапазона индексов применен оборот вида

```
тип range  <>
```

который и указывает на то, что объявлен так называемый НЕОГРАНИЧЕННЫЙ регулярный тип, значениями которого могут быть массивы с любыми диапазонами индексов указанного типа (в нашем случае - целого).

Аналогичный неограниченный регулярный тип с диапазонами индексов перечисляемого типа вводит объявление

```
type таблица is array (буква range <>) of INTEGER ;
```

Значения такого типа могут служить, скажем, для перекодировки букв в целые числа.

Упражнение. Напишите соответствующую программу перекодировки.

Вернемся к типу вектор_любой_длины. Как объявлять конкретные объекты такого типа? Ведь объявление вида

```
Y : вектор_любой_длины ;
```

не прогнозирует очень важной характеристики объекта Y - его возможной длины (другими словами, прогнозу не хватает точности с точки зрения распределения ресурсов программы).

Поэтому само по себе такое объявление не позволяет ни обеспечить эффективность (нельзя распределить память при трансляции), ни настроить функцию на конкретный аргумент такого типа. И, конечно, раз длина вектора не объявлена, то нет оснований контролировать ее (например, в процессе присваивания). [Нарушен ли здесь принцип реальности абстракций?]

Не вдаваясь в обоснования, примем в качестве внешнего требования к языку следующую реализационную аксиому: размеры массивов не должны изменяться .

Ясно, что в рамках статического прогнозирования-контроля, построенного на концепции уникальности типа, реализационная аксиома не согласуется либо с потребностью в надежном контроле, либо с потребностью в мощных абстракциях.

Упражнение. Докажите эту (неформальную) теорему.

4.11.5. Подтипы

Чтобы разрешить указанное противоречие, авторы Ады были вынуждены ввести концепцию ПОДТИПА (специально для квазистатического прогнозирования-контроля изменчивости объектов).

Подтип представляет собой сочетание ТИПА и ОГРАНИЧЕНИЯ на допустимые значения этого типа. Значения, принадлежащие типу, должны, во-первых, принадлежать классу значений ограничиваемого типа и, во-вторых, удовлетворять соответствующему ОГРАНИЧЕНИЮ.

Подтип можно указывать при объявлении объектов. Например,

```
A : вектор_любой_длины (1..10) ;
```

объявляет десятиэлементный вектор A (причем использовано так называемое ОГРАНИЧЕНИЕ ИНДЕКСОВ);

```
выходной : день_недели range сб..вс ;
```

объявляет объект типа день_недели, принимающий значение либо "сб", либо "вс" (причем применяется так называемое ОГРАНИЧЕНИЕ ДИАПАЗОНА).

Бывают и другие виды ограничений (для вещественных и вариантных комбинированных типов).

Раньше мы говорили, что объявление объекта связывает с ним некоторый тип. На самом деле правильнее сказать, что оно связывает с ним некоторый подтип. Когда ограничение отсутствует, то все значения типа считаются удовлетворяющими подтипу.

Подтип, указанный в объявлении объекта, характеризует его во всей области действия объекта, в течение всего периода его существования. Поэтому становится возможным, во-первых, учитывать подтип при распределении памяти для объекта (например, для массива A выделить ровно десять квантов памяти); во-вторых, контролировать принадлежность подтипу при присваивании. Последнее приходится иногда делать динамически (поэтому и идет речь о "квазистатическом контроле"). Это может замедлять работу программы, зато повышает надежность.

Пусть, например, объявлены объекты

```
A,B : вектор_любой_длины (1..10) ;
выходной : день_недели range сб..вс ;
праздник : день_недели ;
```

```
  день_рождения : день_недели ;
  C,D : вектор_любой_длины (1..11) ;
  будний_день :  день_недели range пн..пт ;
  учебный_день : день_недели range пн..сб ;
```

Тогда присваивания

```
  A := B; B := A; праздник := день_рождения;
  день_рождения := будний_день;
  праздник := выходной;
  C := D; D := A;
```

не потребуют никакой дополнительной динамической проверки, так как допустимые значения выражений в правой части присваивания всегда удовлетворяют подтипам объектов из левой части.

Присваивания

```
  A := C; C := A; A := D; B := D; D := A; D := B;
  будний_день := выходной;
  выходной := будний_день;
```

также не требуют дополнительной динамической проверки - они всегда недопустимы, и обнаружить это можно статически, при трансляции.

А вот присваивания

```
     будний_день := учебный_день;
     будний_день := праздник;
     учебный_день := выходной;
     учебный_день := праздник;
```

нуждаются в динамической проверке (почему?).

4.11.6. Принцип целостности объектов

Вернемся к нашей процедуре "минус", вооруженные концепцией подтипа. Допустим, что ее параметр типа вектор_любой_длины. Как обеспечить настройку на конкретный вектор-аргумент? Другими словами, абстракцию мы обеспечили (есть обобщенный тип), а вот реальна ли она (чем обеспечена конкретизация)?

Вспомним, как это делается в Алголе 60 или Фортране. Границы конкретного массива-аргумента нужно передавать обрабатывающей процедуре в качестве дополнительных аргументов. Это и неудобно, и ненадежно (где гарантия, что будут переданы числа, совпадающие именно с границами нужного массива?).

Другими словами, перед нами пример нарушения целостности объекта. Состоит оно в том, что цельный объект-массив при подготовке к передаче в качестве параметра приходится разбивать на части (имя - отдельно, границы - отдельно), а в теле процедуры эти части "собирать" (к тому же при полном отсутствии контроля - ведь транслятор лишен информации о связи между границами и именем массива; знает о ней лишь программист).

Создатели более современных языков руководствуются принципом (сохранения) целостности объектов. Суть его в том, что язык должен обеспечивать возможность работать с любым объектом как с единым целым (не требуя дублировать характеристики объекта и тем самым устраняя источник ошибок). Более точно этот принцип можно сформулировать так :

Вся необходимая информация об объекте должна быть доступна через его имя.

Соблюдение принципа целостности требует включения в базис соответствующих средств доступа к характеристикам объектов. В Аде, где этот принцип положен в основу языка, предопределены универсальные функции, позволяющие узнавать атрибуты конкретных объектов по именам этих объектов.

Они так и называются - АТРИБУТНЫЕ ФУНКЦИИ. Тот или иной набор атрибутных функций связывается с объектом в зависимости от его типа. В частности, для объектов регулярного типа определены атрибутные функции нигр(k) и вегр(k), сообщающие нижнюю и верхнюю границы диапазона индексов по к-му измерению. Например,

```
A'нигр(1) = 1,   B'нигр(1) = 1,
A'вегр(1) = 10,  C'нигр(1) = 1,
D'вегр(1) = 11.
```

Абстракция от длины вектора (окончание). Теперь совершенно ясно, как объявить процедуру "минус", применимую к любому массиву типа вектор_любой_длины.

```
procedure минус (X : in out вектор_любой_длины) is
begin
  for j in (X'нигр(1)..X'вегр(1)) loop
    X(j) := -X(j) ;
```

```
   end loop ;
      end минус ;
```

[Для одномерных массивов вместо нигр(k) и вегр(k) можно писать короче - нигр и вегр, так что заголовок цикла может выглядеть красивей
```
    for j in (X'нигр..X'вегр) loop
```
].

Итак, мы полностью справились с нашей второй абстракцией. При этом воспользовались принципом целостности, чтобы обеспечить реальность абстракции. Со стороны данных для этого понадобилось ввести неограниченные типы и ограничения-подтипы, а со стороны операций - атрибутные функции (опять потребовалось согласовывать абстракции!). В ближайших разделах поговорим об этих понятиях подробнее.

Замечание. Сочетание относительно "свободных" типов с постепенным ограничением изменчивости вплоть до полной фиксации значений объектов (когда они становятся константами) широко применяется при прогнозировании-контроле поведения объектов в Аде. В качестве одного из самых общих ограничений изменчивости можно указать также на требование ко всем компонентам регулярного типа иметь один и тот же подтип. Оно позволяет выделять для них одинаковый объем памяти и легко обеспечивать доступ к компонентам по индексу.

Конец замечания.

4.11.7. Объявление подтипа

Подтип, вводимый в объявлении объекта, является анонимным. Но можно объявлять именованные подтипы. Естественно делать это тогда, когда одни и те же ограничения нужно использовать для объявления многих различных объектов. (Обратите внимание, подтип играет роль, аналогичную той роли, которую играл сам тип в ранних ЯП - ведь подтип затрагивает только совокупность значений объектов, не касаясь применимых к ним операций. Кроме присваивания, как было показано совсем недавно).

Вот несколько примеров именованных подтипов.

```
subtype рабочий_день is день_недели range пн..пт ;
subtype операция_типа_сложения is
                            код_операции range сложить..вычесть ;
subtype операция_типа_умножения is
                            код_операции range умножить..разделить ;
subtype натуральный is INTEGER range 0..INTEGER'последний ;
subtype положительный is INTEGER range 1..INTEGER'последний ;
subtype буква is CHARACTER range `A'..`Z' ;
subtype цифра is CHARACTER range `0'..`9' ;
```

В качестве простого упражнения объявите подтип весенний_месяц, выходной_день, восьмеричная_цифра и т.п.

По внешнему виду объявление подтипа похоже на объявление производного типа. Однако это конструкты совершенно разного назначения. Разберемся с этим подробнее.

4.11.8. Подтипы и производные типы. Преобразования типа

Формально отличие производных типов от подтипов должно быть уже известно. Производный тип - это именно новый тип, характеризуемый классом значений и набором базовых операций.

По сравнению с родительским типом класс значений производного типа может быть сужен (за счет ограничения при объявлении производного типа), а набор базовых операций - расширен (за счет объявлений базовых операций в определяющем производный тип пакете).

А для подтипа по сравнению с базовым типом набор базовых операций может быть только сужен (за счет ограничения допустимых значений).

Кроме того, производный тип в общем случае несовместим по присваиванию и передаче параметров как с родительским типом, так и с другими производными типами одного родительского типа. Например

```
                 type год is new INTEGER range 0..2099 ;
    type этаж is new INTEGER range 1..100 ;
                               A: год ;
                               B: этаж ;
                                    ...
                               A := B ; -- недопустимо! Несовместимость типов, хотя
                                                          --  значения  заведомо
попадут в нужный
```

-- диапазон.

Имя подтипа служит сокращением для сочетания ограничиваемого типа (назовем его БАЗОВЫМ ТИПОМ) и ограничения. Когда такое имя используется при объявлении объекта, считается, что объявлен соответственно ограниченный объект базового типа. Когда такое имя применяется в спецификации параметра процедуры или функции, то аргументом может быть любой объект базового типа, удовлетворяющий соответствующему ограничению. Присваивания объектам различных подтипов одного и того же базового типа не подвергаются никакому дополнительному контролю, кроме принадлежности присваиваемого значения подклассу значений, выделяемого подтипом левой части.

Содержательные роли, которые призваны играть в программе объекты различных подтипов одного базового типа (при квалифицированном использовании языка) должны быть аналогичными, взаимозаменяемыми (с точностью до ограничений на значения).

Другое дело - содержательные роли производных типов одного и того же родительского типа. Производные типы вводятся именно для того, чтобы можно было контролировать, применяются ли они точно по назначению. Поэтому объекты различных производных типов в общем случае не считаются взаимозаменяемыми и по присваиванию (а также сравнениям) несовместимы (это явно контролируется компилятором).

Ведь различное назначение (содержательная роль) различных производных типов одного и того же родительского типа выражается в том, что им соответствуют совершенно различные наборы операций даже тогда, когда объекты этих типов имеют "одинаковые" значения (точнее, значения, соответствующие одинаковым значениям родительского типа).

Вместе с тем при необходимости между такими (родственными) типами допустимы явные преобразования типа.

4.11.8.1. Лес типов

Назовем лесом типов ориентированный граф, вершинами которого служат типы, а дуги соединяют родительский тип с производным. Вот пример леса типов

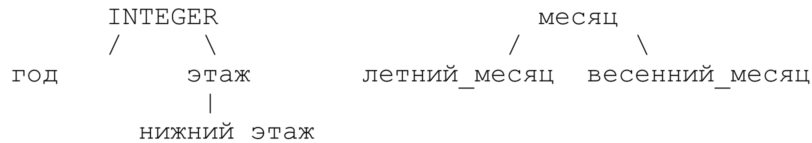


Рис. 4.2

где

```
type нижний_этаж is new этаж range 1..3 ;
type летний_месяц is new месяц range июнь..август ;
type весенний_месяц is new месяц range март..май ;
```

Как видите, лес типов в программе может состоять из отдельных деревьев.

4.11.8.2. Родственные типы и преобразования между ними

Типы из одного дерева в лесу типов называются РОДСТВЕННЫМИ. В Аде допустимы явные преобразования между родственными типами, которые указываются с помощью имени так называемого целевого типа, т.е. типа, к которому следует преобразовать данное. Каждое определение производного типа автоматически (по умолчанию) вводит и операции преобразования родственных типов (но применять эти операции нужно явно!). Например, можно написать

```
A := год(B) ;
```

а также

```
B := этах(A);
```

Обе эти операции (и "год", и "этаж") введены по умолчанию указанными выше объявлениями производных типов, но применены явно, когда потребовалось присвоить объект одного родственного типа другому. Во втором присваивании потребуется проверить попадание в диапазон допустимых значений.

Преобразования родственных типов (с учетом различных содержательных ролей объектов разных типов) далеко не всегда оправданы. Именно поэтому и вводятся явные преобразования, которые программист должен указать сознательно. Так надежность согласуется с гибкостью. По существу в таких случаях программист явно говорит, что полученное значение должно в дальнейшем играть другую содержательную роль.

Например, в задаче моделирования учреждения может появиться тип "сотрудник", характеризуемый целой анкетой атрибутов и набором базовых операций ("загружен_ли", "выполнить_задание", "включить_в_группу" и т.п.). Есть и

```
type рук_группы is new сотрудник ;
```

со своими базовыми операциями ("дать_задание", "подготовить_план_работы", "где_сотрудник" и т.п.).

Пусть объявлены объекты

```
A: сотрудник ;
B: рук_группы ;
```

Тогда присваивание

```
B := рук_группы(A) ;
```

содержательно может означать "повышение" сотрудника A. Ясно, что "автоматически" такое преобразование не делается!

Итак, мы разобрались с основными особенностями подтипов и производных типов. В следующем разделе подведем промежуточный итог знакомства с системой типов в Аде, сопоставив ее возможности с нашей классификацией данных.

Замечание о мобильности программ.

В связи с целыми типами интересно поговорить о мобильности программ. Мобильность - это величина, обратно пропорциональная затратам на перенос программы (программного изделия) из одной программной среды в другую. Например, с БЭСМ-6 на машины серии ЕС или СМ ЭВМ, из операционной системы ОС РВ в РАФОС или UNIX и т.п. По мере развития вычислительной техники мобильность становится одним из важнейших показателей качества программных продуктов. С ней связан уровень дублирования разработок, тиражность программных изделий, возможность пользоваться накопленным запасом программ в новой программной среде и т.п.

Какая связь между мобильностью и целыми типами?

Рассмотрим, чем отличаются объявления типа

```
type косвенный is new INTEGER range -32768.. +32767;
type прямой is range -32768.. +32767;
```

Если предопределенный (реализацией) тип INTEGER имеет диапазон, скажем, -4096.. +4096, то для такой реализации объявление типа "косвенный" придется заменять на другой. Например,

```
type длинный is new LONG_INTEGER;
```

А для типа "прямой" транслятор сам сможет подобрать подходящее представление, подходящий родительский тип, опираясь на то, что подразумеваемый здесь "по умолчанию" предопределенный родительский тип "универсальный_целый" - потенциально не ограниченного диапазона.

Конец замечания.

4.11.9. Ссылочные типы (динамические объекты)

До сих пор в наших примерах, а также в Алголе 60 и Фортране, нам приходилось встречаться лишь с объектами, которые с точки зрения времени их существования естественно называть статическими или квазистатическими. Время существования статических объектов совпадает с полным временем выполнения всей программы. Время существования квазистатических объектов согласуется с временем очередного исполнения их статически определимой области действия.

Область действия объекта - это часть текста программы, при выполнении которой объект считается доступным (например, для чтения или записи его значения, для передачи объекта в качестве параметра и т.п.).

Например, в Алголе 60 областью действия переменной служит тот блок, в котором эта переменная объявлена (за вычетом тех блоков, где объявлен объект с тем же идентификатором). Так что переменная в Алголе - типичный квазистатический объект. Статическими в Алголе могут считаться лишь переменные самого внешнего блока или процедуры.

Так как квазистатические объекты создаются и исчезают синхронно с исполнением фиксированных компонент текста программы (областей действия этих объектов), то, во-первых, доступ к ним возможен только внутри этих компонент и, во-вторых, в каждой из них легко явно назвать все квазистатические объекты индивидуальными именами, фигурирующими непосредственно в тексте программы (т.е. также квазистатическими). Это по существу еще один характеристический признак

квазистатического объекта - такие и только такие объекты могут иметь квазистатические имена.

Но имена содержательных обрабатываемых объектов далеко не всегда удобно фиксировать в тексте программы. Ведь эти объекты могут находиться во внешней для программы среде или возникать динамически как во внешней среде, так и в самой программе при обработке других объектов.

Когда ЯП не предоставляет адекватных выразительных средств, существенно динамические объекты приходится моделировать с помощью квазистатических объектов или их частей. Например, создавая на Алголе процедуры для работы с такими динамическими объектами, как элементы очередей, списков, таблиц, стеков, приходится моделировать их совокупностью компонент статических массивов.

Обычный прием, обеспечивающий доступ к динамическим объектам, состоит в том, что ссылки на такие объекты служат значениями квазистатических объектов, явно фигурирующих в программе. Таким образом, возникает технологическая потребность в категории так называемых ССЫЛОЧНЫХ ТИПОВ, т.е. типов данных, класс значений которых - ссылки на динамические объекты.

Динамические объекты отличаются от статических или квазистатических, во-первых, тем, что создаются при выполнении так называемых ГЕНЕРАТОРОВ, а не при обработке объявлений; во-вторых, тем, что доступ к ним осуществляется через объекты ссылочных типов.

Поэтому и время существования динамических объектов в общем случае связано не с местом их первого упоминания в программе , а с временем существования ссылки. Ссылку можно передавать от одного ссылочного объекта другому, сохраняя динамический объект даже при исчезновении ранее ссылавшихся на него квазистатических объектов.

Чтобы сохранить возможность квазистатического прогнозирования-контроля поведения динамических объектов, необходимо классифицировать ссылочные объекты в соответствии с типом динамических объектов, на которые им разрешено ссылаться.

С точки зрения класса значений и применимых операций динамический объект не отличается от квазистатического (кроме операции создания и, возможно, уничтожения).

Поэтому концепция типа не обязана различать динамические и квазистатические объекты. Именно так и сделано в Аде, где объекты одного и того же типа могут быть как статическими, так и динамическими в зависимости от того, объявлены они или созданы генератором.

Чтобы лучше почувствовать технологическую потребность в динамических объектах, а также ближе познакомиться с соответствующими конструктами Ады, запрограммируем управление очередью.

4.11.9.1. Пример. Задача управления очередью

Постановка задачи. Требуется создать пакет, предоставляющий две услуги. Во-первых, нужно ставить в очередь на обслуживание объекты известного типа, поступающие из внешней среды. Во-вторых, по запросу нужно отправлять на обслуживание объект, стоящий в очереди первым, удалив его из очереди. Дисциплина в очереди - первый пришел, первый ушел.

Решение задачи. Аналогичная задача рассматривается в учебнике по общему курсу программирования [6]. В приведенном там решении находящиеся в очереди объекты хранятся в статическом массиве фиксированной длины. Это и неудобно (какая длина нужна?) и ненадежно (массив можно испортить - очередь не является целостным объектом). Приходится полностью моделировать выделение и освобождение памяти для объектов из очереди.

Ада позволяет, во-первых, защитить очередь от нежелательного доступе, сделать ее инкапсулированным объектом; во-вторых, легко справиться с неограниченностью очереди.

Все это можно сделать с помощью подходящего пакета с двумя операциями "в_очередь" и "из_очереди".

Напишем сначала спецификацию пакета.

```
package  очередь is
type  клиент is  array (1..200) of  CHARACTER;
- - пусть для определенности нужно "обслуживать" тексты
- - длиной 200 символов; позже покажем, как можно
- - настраивать пакет на нужный тип клиента.
procedure  в_очередь(x : in клиент);
```

```
procedure  из_очереди (x : out  клиент);
end очередь;
```

Здесь пока знакомо все. Ключевой момент - реализация нашего пакета, а именно представление очереди. Так как количество стоящих в ней клиентов может изменяться в широких пределах, будем представлять очередь списком записей типа

```
type очередник is
        record
      тело: клиент;
      связь: на_очередника;
    end record ;
```

Тип на_очередника можно ввести объявлением

```
type на_очередника is  access очередник;
```

Ключевое слово access (доступ) указывает, что это именно ссылочный тип.

"Очередник" будет у нас динамическим объектом, создаваемым для каждого нового клиента по мере его прибытия.

В Аде динамические объекты, как обычно, создаются с помощью ГЕНЕРАТОРОВ, а доступ к вновь созданным динамическим объектам обеспечивается взаимодействием квазистатических и динамических объектов ССЫЛОЧНЫХ ТИПОВ.

Основная идея состоит в том, что можно объявить тип P ссылок на объекты другого типа S. После этого становится возможно не только обычным образом объявлять объекты типа S (т.е. создавать их квазистатическими, при обработке объявлений), но и создавать их динамически при выполнении соответствующего ГЕНЕРАТОРА. Генератор создает пару объектов - динамический объект s^ типа S и динамический объект p^ типа P. Это можно изобразить картинкой

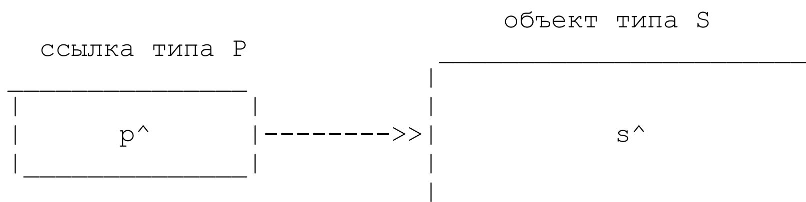


Рис. 4.3

Теперь можно присваивать p^ другим объектам типа P (в частности, квазистатическим, т.е. обычным переменным), привязывая вновь созданные объекты к ранее построенной структуре данных.

Генератор представляет собой выражение new S. Формально - это указатель функции с побочным эффектом. Значение этого выражения - объект p^ ссылочного типа, а побочный эффект состоит в создании нового объекта s^ типа S. При этом p^ ссылается на s^.

Имеется универсальный ссылочный объект null, который считается объектом любого ссылочного типа и не ссылается при этом ни на какой объект.

Наконец, осталось сказать, что отдельные поля объектов, на которые ссылаются переменные ссылочного типа, доступны с помощью обычных выборок или индексации (как будто переменные имеют соответствующий комбинированный или регулярный тип). Например, если T - переменная типа на_очередника, то T.тело указывает на поле с селектором "тело", а T.связь - на поле с селектором "связь".

Путаницы не возникает, потому что у объектов типа на_очередника собственных полей нет. А вот чтобы указать на весь объект типа "очередник", нужно специальное соглашение. (Ведь T обозначает ссылку на очередника, а не запись из двух полей). В Аде для этого используется обозначение T.all.

Например,

```
T.all  := (тело => клиент, связь => null);
```

В этом присваивании использовано еще два соглашения Ады. В скобках изображено так называемое "составное значение" (агрегат), т.е. изображение конкретного значения составного типа, сразу все его поля (в данном случае - два поля). [В Фортране аналогично выглядит объявление начальных данных].

Чтобы в нижеследующем теле пакета все было понятным, осталось напомнить о правиле последовательного определения. Обратите внимание, тип "очередник" содержит поле "на_очередника", а тип "на_очередника" использует имя типа "очередник". Но ведь правило последовательного определения запрещает использовать имена до их объявления. Заметим, что для ссылочных типов приведенная ситуация естественна, ведь они и нужны, чтобы организовать соединения (структуры) динамических объектов.

Чтобы не нарушить принцип последовательного определения, в Аде применяют так называемое ПРЕДВАРИТЕЛЬНОЕ ОБЪЯВЛЕНИЕ имен, где не сообщают их характеристик. В результате на эти имена становится возможным ссылаться в последующих объявлениях.

```
package body очередь is
   type очередник; - - предварительное объявление
   type  на_очередника is  access очередник;
   type очередник is
      record
         тело: клиент;
         связь: на_очередника;
      end record ;
   первый, последний, новый: на_очередника;
   нет_клиентов : exeption ;
  procedure в_очередь (x: in клиент) is
   begin
      новый := new очередник; - - создается динамический объект
      новый.all := (тело => x, связь => null);
      if первый = null then
         первый := новый;
      else
         последний.связь := новый;
      end if ;
      последний := новый;
   end в_очередь;

   procedure из_очереди (x: out клиент) is
   begin
      if первый /= null then
         x := первый.тело;
         первый := первый.связь;
      else raise нет_клиентов;
      end if ;
   end из_очереди;
   function есть_клиенты return BOOLEAN is
   begin
      return первый /= null;
   end есть_клиенты;
end очередь;
```

Функция есть_клиенты нужна, чтобы можно было узнать о наличии клиентов (чтобы не возникало ИСКЛЮЧИТЕЛЬНОЙ СИТУАЦИИ "нет_клиентов"). Однако даже если бы мы сами не позаботились о проверке наличия клиентов (в процедуре "из_очереди"), то при попытке взять компоненту "тело" по ссылке null возникла бы предопределенная исключительная ситуация "неправильная_ссылка".

Обратите внимание: генератор "new очередник" работает в сущности с перекрытием. Создается объект типа "очередник" и ссылка на него именно того типа, который нужен по контексту! Ведь по самому генератору нельзя узнать, какого типа ссылка требуется (в принципе на объекты типа "очередник" могут ссылаться объекты разных ссылочных типов). Можно представить себе ЯП, где в генераторе указывается ссылочный тип (в нашем случае было бы "new на_очередника"). Тогда привлечение контекста не потребовалось бы. Но в таком случае при изменении левой части присваивания пришлось бы менять и генератор. Это понижает надежность при модификациях программ.

Кроме того, вторая трактовка хуже соответствует сути генератора. Ведь его задача - создать объект типа "очередник", а ссылка нужна только для присваивания. Тем не менее, скажем, в Модуле-2 принята именно вторая трактовка прежде всего из-за простоты ее реализации.

Таким образом, мы познакомились со ссылочными типами в действии, когда они применяются для создания целостных динамических объектов (обратите внимание, таким объектом служит динамически создаваемая и уничтожаемая очередь клиентов, состоящая также из динамических объектов/.

Вопрос. Через какое имя доступны атрибуты нашего динамического объекта?

Ответ. Через имя пакета "очередь".

Этот пример можно развивать в различных направлениях. Если нужно вводить много очередей (возможно, с клиентами других типов), то легко осознать потребность

в параметрическом типе "очередь". Это даст нам повод поговорить о РОДОВЫХ параметрах (см. р.13.). С другой стороны, наш пакет рассчитан на последовательное использование процедур "в_очередь" и "из_очереди". Ддругими словами, нельзя обращаться к процедуре "в_очередь", пока не закончит работу процедура "из_очереди" и наоборот (почему?).

Между тем, часто желательно позволить подобным действиям выполняться асинхронно (параллельно). Ведь из жизненной практики мы знаем, что очередь применяется именно как средство организовать взаимодействие асинхронно протекающих процессов (например, взаимодействие "живых" клиентов и предприятий обслуживания). Этот вариант развития нашего примера даст нам повод заняться программированием асинхронных процессов.

4.12. Типы ка объекты высшего порядка (атрибутные функции)

Как только тип данных становится обозначаемым (легко обозримым, определяемым), появляется потребность обращаться с ним самим как с данным, употреблять его в качестве операнда различных операций без каких-либо априорных ограничений. Точнее говоря, одно априорное ограничение имеется.

4.12.1. Статическая определимость типа

Это ограничение можно назвать статической определимостью типа в языке Ада. Состоит оно в том, что тип каждого объекта должен быть определим статически (по тексту программы). Отсюда следует, что тип не может быть параметром (а значит, и аргументом) или результатом какой-либо операции, определяемой пользователем (так как она работает в период исполнения).

4.12.2. Почему высшего порядка?

Статическая определимость типа - необходимое условие статического контроля типов (почему?). Но контролировать типы - значит применять к ним "контролирующие" операции (при статическом контроле - в период компиляции)! Так что с точки зрения потребности в контроле тип следует считать объектом высшего порядка. Аналогия с функциями высшего порядка довольно очевидна. Функция высшего порядка (форма) служит мета-функцией в том смысле, что характеризует связь функций низшего порядка (своих аргументов или результатов). Объект высшего порядка (тип) служит мета-объектом в том же смысле - характеризует связь объектов низшего порядка (а именно их принадлежность этому конкретному типу).

4.12.3. Действия с типами

Что же можно "делать" с таким объектом высшего порядка, как тип данных?

Во-первых, можно (и нужно) получать из одних типов другие, т.е. преобразовывать типы (не объекты данных, а именно типы). Преобразователями типов служат конструкторы типов, с которыми мы уже знакомы. Так, конструктор производного типа преобразует родительский тип в новый тип, в общем случае ограничивая класс значений и сохраняя набор базовых операций. Конструктор регулярного и комбинированного типа преобразует типы компонент регулярных и комбинированных объектов в новый составной тип. Определяющий пакет представляет собой преобразователь объявленных в нем типов, дополняющий набор их базовых операций. Обратите внимание, все названные преобразователи типов (иногда говорят "типовые функции" с ударением на первом слоге) в язык Ада встроены. Они не только предопределены создателями языка, но и по виду своего вызова отличаются от обычных функций. По смыслу это, конечно, функции, аргументами и результатами которых служат типы. Но синтаксически они функциями не считаются и работают не в период исполнения, как остальные функции, а в период трансляции (т.е. это "статические" функции).

Таким образом, в преобразователях типов используются все три компоненты типа. Имя - для идентификации; множество значений и базовых операций - как аргументы для получения новых множеств значений и операций. Вместе с тем статическая (т.е. ограниченная) семантика преобразователей типа подчеркивает ограниченность статической концепции типа в языке Ада (и согласуется с ней). Такая ограниченность характерна для всех (статических) языков, где тип играет роль основного средства статического прогнозирования-контроля).

Во-вторых, с помощью типов можно управлять. Вы уже видели, как тип управляет прогнозированием-контролем, когда используется в объявлении объектов и спецификации параметров. Тип непосредственно управляет связыванием спецификации и тела процедуры, когда работает принцип перекрытия. Тип управляет выполнением

цикла, когда непосредственно определяет диапазон изменения управляющей переменной цикла. В перечисленных примерах тип в Аде по-прежнему выступает в качестве аргумента встроенных в язык, предопределенных конструктов. Пользователь лишен возможности ввести, скажем, новый принцип прогнозирования-контроля, новый принцип перекрытия или новую разновидность циклов.

В-третьих, тип может служить аргументом функций, вычисляющих отдельные значения того же самого или другого типа. Это и есть так называемые АТРИБУТНЫЕ функции. Например, функция "первый" вычисляет наименьшее значение заданного дискретного типа, функция "длина" вычисляет целочисленное значение - длину диапазона заданного дискретного типа. На самом деле аргументом таких функций служит ПОДТИП (ведь тип - частный случай подтипа). А подтип, как мы видели в примере с процедурой "минус", может быть связан с объявленным объектом. Поэтому в качестве аргумента атрибутной функции может выступать не только тип, но и объект данных, который в таком случае и идентифицирует соответствующий подтип. Так что уже использованные нами функции нигр и вегр считаются атрибутными - их аргументами может быть любой регулярный подтип. Обозначения всех атрибутных функций предопределены в Аде. Чтобы отличать их от обычных функций (точнее, чтобы объекты высшего порядка - подтипы не оказывались в синтаксической позиции фактических параметров) и применяется специфическая запись вызова атрибутных функций - аргумент-подтип отделяют апострофом.

Еще одна, четвертая возможность использовать тип как аргумент - настройка РОДОВЫХ СЕГМЕНТОВ.

4.13. Родовые сегменты

Мы уже отмечали, что тип не может быть динамическим параметром, управляющим исполнением процедур и функций. Иначе был бы невозможен статический контроль типа объектов. Однако статическая концепция прогнозирования-контроля типов не противоречит статической же обработке заготовок пакетов и процедур с тем, чтобы настраивать их (в период трансляции) на конкретные значения так называемых РОДОВЫХ ПАРАМЕТРОВ (СТАТИЧЕСКИХ ПАРАМЕТРОВ).

Статическими параметрами заготовок пакетов и процедур (РОДОВЫХ СЕГМЕНТОВ) могут поэтому быть и типы, и процедуры. Определяя родовой сегмент, можно ввести абстракцию, пригодную для использования в различных конкретных контекстах.

Например, в нашем пакете "очередь" весьма конкретный тип "клиент" явно не соответствует степени общности пакета в целом. Можно создать существенно более мощную абстракцию, переписав спецификацию этого пакета следующим образом.

```
generic
type клиент is private
package очередь is
      procedure в_очередь (x: in клиент);
      procedure из_очереди (x: out клиент);
      function есть_клиенты return BOOLEAN;
end очередь;
```

Тело пакета можно оставить без изменений. Теперь в нашем распоряжении родовой пакет с родовым параметром "клиент". Указывать (категорию и некоторые другие характеристики) родовых параметров (в данном случае private), нужно для контроля за использованием параметра внутри родового сегмента. В нашем случае, например, внутри этого сегмента недопустимы какие-либо операции с объектами типа "клиент", кроме предопределенных для всех приватных типов. С другой стороны, в качестве типов-аргументов в данном случае допустимы любые типы (кроме ограниченных). Таким образом, спецификация private относится скорее к внутренности родового сегмента, чем к контексту его использования. В общем случае в качестве спецификации родового параметра можно указывать категории перечисляемых типов, целых, вещественных, ссылочных, регулярных, но не комбинированных (иначе пришлось бы в родовом сегменте фиксировать названия и типы полей; кстати, чем это плохо?).

Итак, абстракцию мы создали. А как с конкретизацией? Теперь нельзя непосредственно использовать пакет "очередь" с помощью указателя пакета, пока не задан конкретный тип-аргумент. Родовой пакет "очередь" - это только заготовка. Ее можно использовать при объявлении "настоящего" пакета, скажем, "добрая-очередь", если предварительно объявлен тип "добрый-клиент".

```
type добрый_клиент is ...;  -- какой-то тип.
package добрая_очередь is new очередь (добрый_клиент);
```

Теперь можно написать

```
with добрая_очередь; use добрая_очередь;
```

После чего применить объявления вида

```
   A: добрый_клиент;
   B: добрый_клиент;
   в_очередь (A); в_очередь (B);
```

В результате в очереди окажется сначала значение объекта А (первым), затем В (последним).

Можно объявить

```
      type хитрый_клиент is ...; -- другой-тип;
      package хитрая_очередь is new очередь (хитрый_клиент);
         .........................
      with хитрая_очередь; use хитрая_очередь;
   C: хитрый_клиент;
   Д: добрый_клиент;
  .........................
   в_очередь (C); из_очереди (Д);
```

при этом за счет перекрытия (ведь В и Д - разных типов) срабатывают правильные операции над разными очередями.

Таким образом, в Аде имеется мощный аппарат статической абстракции-конкретизации - аппарат родовых сегментов. Чтобы создать более полное представление о его возможностях, приведем без особых пояснений еще один пример родового пакета с более впечатляющим набором родовых параметров.

```
      generic
            type  элемент is private ; -- это уже понятно
            type  индекс is (<>); -- допустим любой дискретный тип
   type  вектор is array (индекс) of элемент;
             -- любой регулярный тип,
             -- но имя типа индексов указывать обязательно нужно!!
   with function сумма (X, У: элемент) return элемент;
   --закончился список из трех формальных
   --родовых параметров, последний -
   --формальная функция, применимая к объектам
   --формального типа "элемент"
      package на_векторах is
            function сумма (A, B: вектор) return вектор;
            function сигма (A: вектор) return элемент;
      end на векторах;
```

Обратите внимание, здесь одна функция "сумма" - формальный родовой параметр, другая - обычная функция, объявленная в спецификации пакета. Как писать тело такого пакета, должно быть понятно:

```
package body  на_векторах is
            function сумма (A, B: вектор) return вектор is
     Z:вектор;
            begin
for j in вектор'нигр .. вектор'вегр loop
      Z(j) := сумма (A(j), B(j));
end  loop;
return  Z;
            end сумма;
function сигма (A: вектор) return элемент is
Z: элемент := A (вектор'нигр);
     for j in вектор'нигр + 1 .. вектор'вегр loop
       Z := сумма (Z, A(j));
end loop ;
     return Z ;
end сигма;
end на_векторах;
```

Вот возможная конкретизация этого пакета:

```
      package на_целых_векторах is
            new на_векторах (INTEGER, день, ведомость,'+');
```

Здесь тип "ведомость" считается введенным объявлением

```
            type ведомость is array (день range < >) of INTEGER;
```

а '+' - предопределенная операция для целых.

Так что если

```
   T: ведомость (Вт..Пт) := (25,35,10,20);
   R: ведомость (Вт..Пт) := (10,25,35,15);
то в соответствующем контексте
        сумма(T,R) = (35,60,45,35);
        сигма(T) = 90; сигма(R) = 85;
```

Обратите внимание на применение агрегатов, а также на то, как в них используется линейный порядок на дискретных типах.

Родовые аргументы должны строго соответствовать спецификации родовых параметров. За этим ведется строгий контроль. Так, функция '+' подошла, а, скажем "or" или тем более "not" - не подойдет (почему?).

Замечание. Обратите внимание на нарушение принципа целостности объектов в аппарате родовых сегментов.

Вопрос. В чем это проявляется?

На этом закончим разговор о родовых сегментах.

4.14. Асинхронные процессы

4.14.1. Зачем они нужны

Мы уже рассматривали несколько ситуаций, когда возникает потребность в асинхронных процессах. Еще когда речь шла о модели Бэкуса, отмечалась возможность ускорить перемножение матриц, если независимые скалярные произведения вычислять на параллельно (асинхронно) работающих процессорах. Отмечали мы и возможную пользу от независимого вычисления отрицаний в процедуре "минус". Но это были примеры оптимизации, когда асинхронные вычисления полезны, но не являются необходимыми. Хотелось бы понять, в каких ситуациях нельзя обойтись без асинхронных процессов.

Классический пример - так называемые задачи реального времени, когда программа должна успевать за темпом внешних событий. Тогда очевидно, что асинхронное исполнение может стать совершенно необходимым (если применяются не слишком быстрые компьютеры).

4.14.2. Пример. Система регистрации и обработки сообщений

Постановка задачи. Пусть, скажем, на вход системы регистрации и обработки сообщений поступают сообщения из внешней среды. Минимальный интервал между появлением сообщений - r. Полная обработка каждого сообщения состоит из двух последовательных этапов - этапа анализа А и этапа синтеза С. На анализ в худшем случае требуется времени

r - e,

где e пренебрежимо мало, а на синтез в худшем случае может потребоваться времени до 3r (хотя это случается редко). Обычно синтез также укладывается в r-e. Система обязана успевать реагировать на каждое сообщение по возможности быстро, выдавая синтезированный ответ во внешнюю среду.

Обсуждение. Как проектировать такую систему?

Ясно, что если для каждого сообщения вынолнять сначала А, потом С и только затем принимать новое сообщение, то можно опоздать - ведь времени потребуется до 4r, а минимальный интервал - всего r.

Поэтому совершенно необходимо применить асинхронное выполнение этапов А и С, своего рода поточную линию, состоящую в данном случае из "станков" А и С. Станок А воспринимает очередное сообщение, анализирует его и передает результат станку С, а сам принимает и обрабатывает следующее сообщение. Система станет успевать воспринимать каждое сообщение (ведь А работает меньше r). Но возможно отставание из-за С. Теперь станок С может не успевать получать результаты анализа от А.

Итак, нужно "развязать" тесную взаимосвязь между А и С. Обычный прием для этого - организовать очередь к станку С. Она будет играть роль "принимающего кармана" у станка на поточной линии, роль буфера, позволяющего станку А выдавать результат анализа (разбор), не дожидаясь готовности С воспринять его, а станку С - работать в доступном ему темпе, последовательно обрабатывая разборы. При этом очередь может расти. Однако так как длинный синтез бывает редко, будем считать, что обычно ее длина не будет превышать некоторой разумной величины.

Итак, допустим, что необходимость асинхронной работы сегментов А и С, а также необходимость очереди разборов обоснована. Спрашивается, как оформить сегменты А,С и "очередь"?

4.14.3. Управление асинхронными процессами в Аде (фрагмент обзора языка)

В Аде взаимодействие асинхронных процессов обслуживают несколько операторов.

Оператор ЗАДЕРЖКИ (delay) приостанавливает исполнение задачи, в которой он находится, на указанный в нем период (реального, астрономического) времени.

Вызов ВХОДА R, находяшийся в задаче K, аналогичен вызову процедуры, но в общем случае не исполняется немедленно, а лишь "заказывает РАНДЕВУ" категории R. Это значит, что задача K (назовем ее пассивной или обслуживаемой по входу R, короче клиентом), готова к рандеву (взаимодействию, встрече) с другой задачей M, в которой вход R объявлен. Задачу M назовем активной или обслуживающей вход R (короче - мастером). Она оказывается готовой обслужить заказ задачи K лишь тогда, когда достигнет оператора ПРИЕМА (accept) входа R. Оператор приема предписывает действия, выполняемые в момент рандеву. Когда эти действия завершаются, рандеву считается состоявшимся и обе задачи могут продолжать асинхронно работать (до следующего взаимодействия-рандеву). Если задача M достигает оператора приема входа R раньше, чем его закажет какая-либо обслуживаемая задача, то задача M приостанавливается и ждет появления заказов (ждет рандеву).

Таким образом, рандеву происходит тогда (и только тогда), когда и клиент, и мастер оказываются к нему готовыми (задача K дошла до вызова входа и заказала рандеву категории R, а задача M дошла до оператора приема и готова выполнить заказ).

Собственно рандеву состоит в том, что аргументы вызова входа R (из задачи-клиента) связываются с параметрами оператора приема (из задачи-мастера) и выполняется тело оператора приема.

Все происходит так, как будто из задачи K обращаются к процедуре R, объявленной в задаче M. Выполнение оператора приема в задаче M означает тем самым и выполнение оператора вызова в задаче K (и тем самым завершение рандеву категории R). Другими словами, задачи K и M как бы сливаются на время рандеву, а затем продолжают работать независимо до следующего возможного рандеву.

Оператор отбора входов (ОТБИРАЮЩИЙ оператор, оператор select) позволяет мастеру ожидать сразу нескольких рандеву и отбирать (из заказанных!) те рандеву, которые удовлетворяют указанным в этом операторе УСЛОВИЯМ ОТБОРА.

4.14.4. Продолжение примера. Разделяемые ресурсы. Режим взаимного исключения с развязкой

Из обзора языка Ада ясно, что A и C должны быть отдельными задачами. А вот что такое "очередь"?

Мы уже говорили о том, что нельзя допускать одновременное исполнение процедур "в_очередь" и "из_очереди". А если оставить "очередь" пакетом, доступным как задаче A, так и задаче C, то именно так и получится - пока задача A будет исполнять "в_очередь", передавая в буфер приготовленный разбор, задача C может обратиться к процедуре "из_очереди", доставая разбор для синтеза ответа. Читателю предлагается самому разобраться с тем, почему может нарушиться работа очереди.

Очередь представляет в нашем случае так называемый РАЗДЕЛЯЕМЫЙ РЕСУРС. Доступ к такого рода ресурсу из параллельно работающих задач следует организовывать в так называемом РЕЖИМЕ ВЗАИМНОГО ИСКЛЮЧЕНИЯ. Это значит, что когда доступ к очереди имеет задача A, то его не может иметь задача C, и наоборот. С другой стороны, нужно обеспечить РАЗВЯЗКУ доступа к ресурсу, т.е. обеспечить, чтобы количество последовательных обращений к очереди из задачи A не было жестко связано с количеством последовательных обращений из задачи C. Иначе очередь не сможет играть роль буфера, позволяющего A и C при необходимости работать с разной скоростью.

Итак, нужно организовать доступ к разделяемому ресурсу в режиме взаимного исключения с развязкой. Это один из важнейших видов взаимодействия асинхронных процессов (задач).

[Еще один вид взаимодействия процессов - параллельное и независимое вычисление компонент некоторого составного объекта. Например, компонент вектора в нашей операции "минус", если бы мы продолжили свою серию абстракций. Такое взаимодействие асинхронных процессов естественно назвать взаимодействием по результату. Бывают, конечно, и другие виды взаимодействия процессов. Взаимодействие по результату (и аргументу) в Аде не поддержано подходящими выразительными средствами.]

4.14.4.1. Концепция внутренней дисциплины. Монитор Хоара-Хансена.

Определенной дисциплины доступа к разделяемому ресурсу можно добиться по-разному. Например, можно требовать, чтобы эта дисциплина была запрограммированна в самих задачах A и C. Такая концепция внешней дисциплины не очень удобна. Во-

первых, потому, что внешнюю дисциплину тяжело отлаживать и изменять (она оказывается "размазанной" по реализации А и С); во-вторых, потому, что обязательно нужны специфические атрибуты, характеризующие разделяемый ресурс. Специфика таких атрибутов проявляется в том, что они должны быть доступны из параллельных задач и не требовать при этом взаимного исключения. Несмотря на описанные неудобства, первые средства параллельного программирования, появившиеся в ЯП, были основанны именно на таких атрибутах и требовали соответствующей внешней дисциплины доступа к разделяемым ресурсам. Это и так называемый аппарат событий в языке ПЛ/1, и аппарат семафоров в языке Алгол 68. Важно понимать, что концепция внешней дисциплины доступа подразумевает полную пассивность самого разделяемого ресурса между сеансами доступа.

Тони Хоару принадлежит идея активного разделяемого ресурса, самостоятельно управляющего доступом - идея так называемого монитора Хоара. В противовес внешней дисциплине соответствующую монитору Хоара общую концепцию можно называть концепцией внутренней дисциплины.

Идея состоит в том, что единственное, что могут сделать внешние, обслуживаемые монитором задачи - это стать в очередь к одному из его входов. В остальном дисциплина доступа (обслуживания) полностью определяется внутри монитора, который сам является асинхронно работающей задачей.

Программа именно этой задачи и определяет дисциплину доступа к разделяемому ресурсу.

Концепция внутренней дисциплины лучше соответствует концепции абстракции данных (в данном случае, концепции разделяемого ресурса) - ведь все особенности данного сосредоточены в мониторе, а не "размазаны" по внешним задачам.

Аппарат паралельного программирования в Аде ориентирован именно на концепцию внутренней дисциплины управления доступом к разделяемым ресурсам.

Поэтому в нашем примере естественно организовать очередь разборов в виде монитора Хоара, реализованного в виде обслуживающей задачи.

Оформим все три задачи - А, С и "очередь" в одном пакете. Однако так как "очередь" - обслуживающая, вспомогательная задача, мы скроем ее от пользователя пакета, поместив ее объявление в тело пакета.

```
package анализ_синтез is
        type сообщение is array (1..200) of CHARACTER;
        type ответ is new сообщение;
        task A is
              entry прими (X: in сообщение);
        end A;
        task C is
              entry выдай (X: out ответ);
        end C;
  end анализ_синтез;
```

Как уже говорилось, каждый вход (entry) с точки зрения использования не отличается от процедуры. Так что вход "прими" в задаче А служит для приема сообщений извне (к нему обращаются из контекста, использующего пакет анализ_синтез). Соответственно вход "выдай" в задаче С служит для выдачи ответов. Задачу "очередь" нельзя объявить в спецификации пакета. Она должна быть инкапсулирована, чтобы обеспечить целостность ресурсапосредника между задачами А и С.

```
          package body анализ_синтез is
      type разбор is new сообщение;
                task буфер is -- это вместо "очереди".
                      entry стань_в_очередь (X: in разбор);
                      entry выйди_из_очереди (X: out разбор);
                end буфер;
          --как видите, спецификация задачи тоже отделена от реализации.
                task body A is
                      раб: сообщение;
                      готов: разбор;
                begin
                   loop
                      accept прими (X: in сообщение) do
                        раб := X; -- работает во время рандеву
                      end прими;
                      ...    -- операторы анализа сообщений
```

```
                    ...    -- работают вне рандеву, присваивают
                    ...    -- значение переменной "готов"
                    стань_в_очередь (готов);  -- работает при рандеву
                    end loop ;--цикл, пока задачу не остановят извне
                end A;
```

[A пишется без всякого учета своей параллельности с C!]

Поясним уже использованные конструкты, а затем продолжим писать тело пакета анализ_синтез.

4.14.5. Замечания о конструктах (к пакету анализ_синтез)

Итак, в пакете анализ_синтез объявлены два типа, "сообщение" и "ответ", причем второй - производный от первого, и две задачи A ("анализ") и C ("синтез"). Формально объявления этих типов нужны, чтобы можно было специфицировать параметры входов "прими" и "выдай" соответственно задач A и C. Различать типы сообщений и ответов полезно для повышения надежности программирования в контексте, использующем наш пакет.

Формально спецификация задачи - это объявление объекта анонимного задачного типа. Оно связывает имя задачи с объектом, который представляет асинхронный процесс, определяемый телом соответствующей задачи. Таким образом данные задачных типов - активные данные. Объект задачного типа, т.е. асинхронный процесс, запускается (т.е. начинает работать) в момент, когда заканчивается обработка объявления объекта (в нашем случае, объявления задачи).

В языке можно объявить именованный задачный тип. Например,

```
        task type анализ is
              entry прими (X: in сообщение );
        end анализ;
```

Такое объявление связывает имя "анализ" с задачным типом, класс значений которого - асинхронные процессы со входом "прими", определяемые телом задачи с именем "анализ". В нашем случае, чтобы определить не одну анализирующую задачу, а целый задачный тип, достаточно заменить спецификацию задачи A на объявление задачного типа "анализ", а в теле задачи A (в теле пакета) заменить первую строку на

```
                    task begin анализ is
```

В контексте, где доступен задачный тип "анализ", можно объявить индивидуальную задачу этого типа, скажем

A: анализ; -- т.е. обычное объявление объекта.

При обработке такого объявления запускается новый асинхронный процесс типа "анализ" (т.е. создается новый объект задачного типа "анализ") и с ним связывается имя A. Если нужно, можно запустить и другие процессы этого типа объявлениями

```
                    A1: анализ;
                    A2: анализ; -- и т.д.
```

При этом доступ к входу "прими" нужного процесса обеспечивает составное имя вида A1.прими, A2.прими и т.п.

Когда же, как в нашем случае, имеется единственный процесс со входом "прими", имя задачи можно не указывать и пользоваться простым именем входа.

Обратите внимание, что входы задач можно рассматривать как аналоги селекторов в объектах комбинированных типов. Обращение ко входу по составному имени напоминает выборку значений поля. В определяющем пакете для задачного типа могут быть объявлены подходящие базовые операции. Все сказанное и позволяет считать задачные типы полноценными (ограниченными!) типами данных, причем данных активных, а не пассивных.

4.14.5.1. Отступление о задачах и подпрограммах

Может возникнуть естественный вопрос. Почему подпрограммы (процедуры и функции) не считаются в Аде данными, а "более мощные" задачи - считаются?

Коротко говоря, дело в том, что концепция "подпрограммного" типа не соответствует общей концепции типа в Аде, а концепция "задачного" типа при определенных ограничениях - соответствует вполне.

Действительно, объектами задачного типа служат различные запуски одного и того же тела задачи, определенного статически соответствующим объявлением (задачного типа или индивидуальной задачи). В Аде у запуска задачи не может быть динамических параметров. Поэтому, во-первых, внутри тела задачи легко выполнить полный статический контроль типов. Во-вторых, если задача передается в качестве

динамического параметра некоторой подпрограмме, то внутри последнего доступ к задаче-параметру возможен только по статически определенным входам, зафиксированным в объявлении задачного типа, указанного в спецификации параметра. Так что и здесь никаких трудностей для статического контроля типов не возникает.

Итак, концепция задачного типа без динамических параметров запуска вполне согласуется с концепцией уникальности типа объектов.

По аналогии, объектами "подпрограммного" типа в Аде следовало бы считать различные запуски одной и той же подпрограммы. Рассмотрим сначала, почему это не слишком привлекательно, а потом поговорим, почему нехорошо относить к одному типу запуски различных подпрограмм.

Совершенно понятно, зачем нужно много асинхронных процессов с поведением одного типа (т.е. объектов одного задачного типа). Они могут, скажем, моделировать звено самолетов-истребителей или участок однотипных станков. Такие процессы могут естественно взаимодействовать между собой. А запуски одной и той же подпрограммы при последовательном исполнении могут "взаимодействовать" только в случае рекурсии (иначе они не могут сосуществовать) и для этого есть известный механизм локальных переменных. При параллельном исполнении запуски "одной и той же" подпрограммы становятся частями различных асинхронных процессов (и тогда для организации их взаимодействия естественно воспользоваться аппаратом взаимодействия задач). Итак, будем считать понятным, почему невыгодно считать объектами "подпрограммного" типа запуски одной и той же подпрограммы.

Если же считать относящимися к одному типу запуски различных подпрограмм, то в общем случае легко прийти к противоречию с концепцией уникальности типа.

Действительно, объекты-подпрограммы немедленно могут стать динамическими параметрами других подпрограмм (запуск подпрограмм без динамических параметров в общем случае - бессмыслица, в отличие от запуска задач; ведь динамический доступ к задачам происходит по входам, а к подпрограммам - только при запуске, в соответствии с их природой). Но в общем случае разные аргументы-подпрограммы требуют в свою очередь аргументов разных типов (и, возможно, вырабатывают результаты разных типов, если это функция). Поэтому внутри тел подпрограмм становится невозможным статически контролировать типы. Так что единственная приемлеиая альтернатива - допускать в качестве параметров подпрограмм только подпрограммы с фиксированными типами, с фиксированным числом аргументов и результатов. Однако для Ады с ее концепцией имени типа как характеристики содержательной роли объекта эта альтернатива близка к уже рассмотренной альтернативе (подпрограммный тип как совокупность запусков одной и той же подпрограммы).

Добавим, что динамически возникающие подпрограммы-параметры потребовали бы (при соблюдении адовского уровня защиты) реализовать практически всю контролирующую мощь компилятора на целевой машине. Это неприемлемо для бедных бортовых машин.

Тем самым неформальная теорема "О неприемлемости для языка категории Ады "подпрограммных" типов" доказана.

Конец отступления.

Вернемся к задачным типам.

Добавим, что объекты задачных типов могут служить компонентами объектов составных типов.

Например, можно объявить иассив из десяти "анализаторов".

```
A: array (1..10) of анализ;
```

и обращаться к соответствующим входам с помощью индексации

```
A(1).прими ...; ...; A(10).прими ...
```

Задачные объекты могут, естественно, быть и динамическими. Например, можно ввести ссылочный тип

```
type P is access анализ;
```

и переменную R типа P

```
R : P;
```

Теперь понятно действие оператора

```
R := new анализ;
```

А именно, создается новый асинхронный процесс типа "анализ" и ссылка на него помещается в R. К соответствующему входу можно теперь обращаться через

```
R.прими ...
```

Подчеркнем, что у динамических задачных объектов не может быть динамических параметров, так что все сказанное про соответствие задачных типов концепции уникальности сохраняет силу и для динамических задачных объектов.

4.14.6. Тело пакета анализ-синтез (продолжение)

```
               task body C is
                     раб : разбор;
                     готов : ответ;
               begin loop
                     выйди_из_очереди (раб);  - - работает при рандеву
                       - - операторы синтеза ответа,
                    ...      - - работают вне рандеву, присваивают
                    ...      - - значение переменной "готов"
     accept выдай(X : out ответ) do
  X := готов;                - - работает во время рандеву
     end выдай;
               end C;
               task body буфер is          - - простейший вариант для нас
     package наша_очередь is new очередь(разбор);
              -- становятся доступны, в частности,
              -- наши процедуры в-очередь и из-очереди
     use наша_очередь;
               begin
    loop
      select
        accept стань_в_очередь (X: in разбор) do
          в_очередь(X);
        end стань_в_очередь;
      or
        when есть_клиенты & стань_в_очередь'счетчик = 0 ==>
          accept выйди_из_очереди (X: out разбор) do
             из_очереди(X);
          end выйди_из_очереди;
      end select ;
    end loop ;
               end буфер ;
 end анализ_синтез ;
```

4.14.7. Замечания о конструктах (продолжение)

Итак, в теле пакета анализ_синтез объявлен производный тип "разбор". Он предназначен для использования только в теле пакета для взаимодействия между задачами А и С. Поэтому от пользователя скрыт.

Затем объявлена задача "буфер". Это и есть монитор Хоара, регулирующий доступ к очереди разборов как ресурсу, разделяемому между задачами А и С. Но, как вы увидите, в самом мониторе никак ни отражен тот факт, что он имеет дело именно с задачами А и С. Он пригоден для буферизации объектов типа "разбор" между любыми задачами, лишь бы они обращались к его входам стань_в_очередь и выйди_из_очереди. (Как видно по названиям, эти входы монитора соответствуют операциям над очередью).

Далее следует тело задачи А. Формально тело задачи отличается от тела процедуры лишь тем, что в первом допустимы специальные "задачные" операторы, недопустимые в теле обычной процедуры. Один из таких "задачных" операторов - оператор приема (accept). По форме он похож на объявление процедуры. Во всяком случае, содержит название соответствующего входа, список формальных параметров и тело - последовательность операторов между ключевыми словами do и end названием входа.

Задача А. После обработки объявления задачи А тело ее запускается. Это значит, что обрабатываются объявления переменных "раб" и "готов" (т.е. создаются объекты типов "сообщение" и "разбор" соответственно и связываются с именами "раб" и "готов"). Первая предназначена служить промежуточным хранилищем сообщений, а вторая - для готового "разбора". Затем начинается "бесконечный" цикл, внутри которого находится уже упоминавшийся оператор приема входа "прими" и не выписанная явно последовательность операторов анализа принятого сообщения и подготовки "разбора" в переменной "готов". Завершает внутренность цикла оператор вызова входа стань_в_очередь в задаче "буфер".

"Бесконечный" цикл выглядит непривычно для людей, впервые знакомящихся с взаимодействием асинхронных процессов. В последовательном программировании он всегда свидетельствовал об ошибке. Здесь же он естественен - ему действительно

следует работать всегда, пока существует задача А. В языке имеются средства извне "убить" (т.е. завершить выполнение) любой задачи.

Начав цикл, задача А доходит до оператора приема и, в общем случае, приостанавливается, ждет вызова входа "прими" извне (из контекста, использующего пакет анализ_синтез). Как уже было сказано, если вызовы появляются до момента, пока задача А доходит до оператора приема входа "прими", эти вызовы выстраиваются в очередь ко входу "прими". В момент, когда эта очередь не пуста, а задача А дошла до оператора приема, происходит РАНДЕВУ с той обслуживаемой задачей, чей вызов входа "прими" - первый в очереди. Это значит, что выполняется тело оператора приема - входа "прими" (не зря он похож на объявление процедуры "прими") после установления связи формального и фактического параметра. Другими словами, переменной "раб" присваивается значение "сообщения" переданного в качестве параметра при вызове входа. На этом рандеву завершается и остальные операторы внутри цикла выполняются асинхронно с операторами обслуживаемой задачи.

Когда управление попадает на вызов входа стань_в_очередь, задача А приостанавливается снова, на этот раз потому, что требуется рандеву с задачей "буфер" для передачи готового разбора в очередь разборов. После этого рандеву снова работает оператор приема входа "прими" и т.д.

Таким образом, задача А, работая асинхронно с С и с обслуживающей задачей "буфер", участвуя попеременно в рандеву, получает "сообщения" и передает готовые "разборы".

Задача С. Теперь должно быть понятно тело задачи С как формально, так и содержательно. Здесь "бесконечный" цикл начинается с рандеву с обслуживающей задачей-монитором "буфер", чтобы получить очередной разбор. Затем вне рандеву синтезируется "ответ" в переменной "готов", который и "забирается" (обслуживаемыми задачами; из контекста, в котором используется наш пакет) при выполнении оператора приема входа "выдай". После рандеву с обслуживаемой задачей готовый ответ становится значением выходного параметра - аргумента вызова входа "выдай", т.е. действительно передается обслуживаемой задаче.

Таким образом, задача С, работая асинхронно с обслуживаемыми внешними задачами и монитором "буфер", также участвуя попеременно в рандеву, получает из очереди "разборы" и выдает готовые "ответы".

Задача "буфер". Осталось самое интересное - задача "буфер". В ее теле объявлен пакет "наша_очередь" с помощью конкретизации родового пакета "очередь" для родового аргумента - типа "разбор". После указателя сокращений (use) становится возможным пользоваться в теле задачи буфер процедурами работы с очередью, реализованной на динамических объектах типа "очередник", где имя типа "клиент" обозначает теперь тип "разбор".

Наша цель - добиться того, чтобы к этим процедурам можно было обращаться только в режиме взаимного исключения с развязкой. Именно это и гарантирует выполняемый в "бесконечном" цикле отбирающий оператор.

4.14.7.1. Оператор отбора входов (select)

Общий вид этого оператора

```
select
[ when условие ====> ]  отбираемая_альтернатива
     последовательность_операторов
or
        . . .
or
[ when условие ====> ]  отбираемая_альтернатива
                последовательность_операторов
[ else последовательность_операторов ]
end select ;
```

Отбираемой альтернативой может быть ОПЕРАТОР ПРИЕМА, ОПЕРАТОР ЗАДЕРЖКИ или ОПЕРАТОР ЗАВЕРШЕНИЯ задачи. Когда управление в задаче достигает оператора отбора, то, во-первых, вычисляются все условия и те альтернативы, для которых условия истинны, считаются ОТКРЫТЫМИ. Затем среди открытых альтернатив рассматриваются ОПЕРАТОРЫ ПРИЕМА, для которых очередь вызовов соответствующих входов непуста. Если такие найдутся, то произвольным образом выбирается одна из таких альтернатив, происходит соответствующее рандеву, затем выполняется последовательность операторов, расположенная за этой отобранной альтернативой, и оператор отбора считается выполненным. Если же среди открытых альтернатив нет операторов приема,

готовых к рандеву, то выполняется ОПЕРАТОР ЗАДЕРЖКИ на указанное количество секунд (если за это время возникает готовность к рандеву у открытых операторов приема, то отбирается альтернатива, готовая к рандеву и оператор отбора завершается как обычно). После задержки и выполнения соответствующей выбранной альтернативе последовательности операторов оператор отбора считается выполненным. Если одна из открытых альтернатив - оператор завершения (terminate), то (если нет готовых к рандеву операторов приема), при определенных дополнительных условиях задача может быть завершена (до этого должны, в частности, завершиться так называемые подчиненные задачи, запущенные завершаемой задачей).

Альтернатива else может быть выбрана, если нет открытых операторов приема, готовых к рандеву.

В нашем случае - две альтернативы. Одна всегда открыта, вторая - только когда в очереди есть клиенты. Так что ставить в очередь можно всегда, а брать можно, естественно, когда есть что брать.

Если обращения ко входам стань_в_очередь и выйди_из_очереди возникают одновременно, то оператор отбора случайным образом выбирает только один оператор приема. Таким способом обеспечивается взаимное исключение доступа к очереди из задач А и С. С другой стороны, обслужив рандеву, оператор отбора, находясь в цикле, снова получает управление. Поэтому если, скажем, много вызовов из задачи А идут подряд, то они будут обслужены при последовательных исполнениях оператора отбора. То же при последовательных вызовах из С, если клиенты есть.

Может быть и так, что задача С ждет рандеву, и очередь непуста, но все время поступают новые разборы и монитор обслуживает задачу А. Чтобы такого не случалось, цикл в задаче "буфер" должен выполняться по возможности быстро. Тогда "буфер" будет всегда успевать и принимать разборы, и выдавать их задаче С. Возможна такая ситуация, когда задача С хдет рандеву с "буфером", а "буфер" ждет рандеву с задачей А - нет клиентов. Тогда ожидание будет до тех пор, пока не появится вызов входа стань_в_очередь. Выберется соответствующая альтернатива и на следующем выполнении цикла альтернатива для задачи С окажется открытой и сможет конкурировать с первой альтернативой.

Итак, задача-монитор полностью воплощает в себе дисциплину обслуживания. В данном случае - относительно равноправный доступ к очереди со стороны А и С. Монитор полностью гарантирует взаимное исключение с развязкой.

Замечание.

Осталось пояснить, зачем нужно дополнительное условие отбора во второй альтернативе.

Действительно, если постановка в очередь не будет обслуживаться достаточно долго, то в результате входное сообщение окажется пропущенным. [Как это может случиться?]

Чтобы избежать такой неприятности, нужно гарантировать выбор первой альтернативы, как только появляется вызов входа "стань_в_очередь". В соответствии с семантикой оператора отбора это можно сделать, защитив вторую альтернативу дополнительным условием "нет вызова входа стань_в_очередь".

Но для этого нужно иметь возможность проверить, имеется ли такой вызов! Другими словами, нужно иметь доступ к подходящему атрибуту этого входа. Для подобных целей в Аде предусмотрена атрибутная функция вида

P'счетчик,

где Р - имя входа некоторой задачи. Ее результат - количество вызовов, ожидающих обслуживания этим входом. Так что условие перед второй альтернативой следует записать в виде

```
when есть_клиенты and стань_в_очередь'счетчик = 0
```

что и сделано.

Теперь вторая альтернатива окажется закрытой, если есть вызовы для первой альтернативы, которая и будет беспрепятственно выбираться.

В Аде есть и другие атрибутные функции, связанные с управлением асинхронными процессами.

[Рациональной структуризацией управления асинхронными процессами много и плодотворно занимался Бринч-Хансен. Ему принадлежит и очень близкое к изложенному понятие монитора. Интересующегося читателя отсылаем к [12].]

На этом закончим разговор об асинхронных процессах и их программировании на языке Ада.

4.15. Числовые типы (модель числовых расчетов)

4.15.1. Суть проблемы

Рассматривая основные технологические потребности, невозможно обойти потребность вести числовые расчеты. Эта потребность, как известно, в свое время предопределила само возникновение ЭВМ. Хотя сейчас потребность в числовых расчетах - далеко не самая главная в развитии компьютеров и ЯП, абсолютная потребность в объеме, точности и надежности числовых расчетов продолжает расти. Так что ни в одном базовом языке индустриального программирования ее игнорировать нельзя.

Парадоксально, но факт, что так называемые машинно-независимые языки для научных расчетов (Фортран, Алгол и их диалекты) не предоставили удовлетворительной модели числовых расчетов, в достаточной степени независимой от машины.

В этом их аспекте особенно сказалась несбалансированность средств абстракции и конкретизации. Абстрагироваться от особенностей ЭВМ можно, а настроиться на конкретную машину - нельзя. Обеспечить надежность при переходе с машины на машину - проблема.

Суть в том, что ни в Фортране, ни в Алголе нет возможности явно управлять диапазоном и точностью представления числовых данных. Можно лишь указать, что требуется "двойная точность" (в некоторых диалектах градаций больше), но какова эта точность, зависит от реализации. Таким образом, пользователь "машинно-независимого" языка оказывается в полной зависимости от конкретной машины, если ему нужно гарантировать надежность расчетов.

Одна из причин такой ситуации в том, что в начале "эры ЭВМ" скорости числовых расчетов придавалось исключительно большое значение. Поэтому считалось практически нереальным применять какое-либо представление чисел и какие-либо базовые операции, отличные от непосредственно встроенных в машину. Поскольку такие встроенные числовые типы различны на различных ЭВМ, считалось невозможным эффективно решить проблему выбора представления в соответствии с машинно-независимыми указаниями пользователя.

Поэтому проблема обеспечения надежности числовых расчетов традиционно оставалась вне рамок "машинно-независимых" языков.

По мере накопления пакетов программ и осознания того факта, что зависимость от конкретных представлений числовых типов - одно из важнейших препятствий при переносе программ из одной вычислительной среды в другую, рос интерес к созданию достаточно универсальной (машинно-независимой) схемы управления числовыми расчетами.

К моменту создания Ады проблема надежности программного обеспечения (в частности, при переносе из одной вычислительной среды в другую) была осознана как важнейшая, потеснившая по своей значимости проблему скорости расчетов. К тому же доля числовых расчетов в общем времени исполнения программ существенно сократилась. Появились и методы относительно эффективной реализации вычислений с любой заданной точностью (за счет микропрограммирования, например).

Все это сделало актуальной и реальной попытку разработать гибкую схему машинно-независимого управления числовыми расчетами, пригодную для широкого класса машин. Одна из таких схем воплощена в Аде.

[Указывать представление числовых данных можно и в языке ПЛ/1, и в КОБОЛе. Однако в этих языках отсутствует явная связь представления данных с гарантией надежности расчетов]

4.15.2. Основная цель

Главная цель состоит в том, чтобы при работе с числовыми типами программист мог гарантировать надежность расчетов независимо от объектной машины (если только эта машина пригодна для их выполнения). Гарантировать надежность означает, в частности, гарантировать как необходимую точность расчетов, так и отсутствие незапланированных исключительных ситуаций (переполнений, например) при допустимых исходных данных.

Очевидно, что достичь такой цели можно, только предоставив программисту возможность объявлять нужные ему диапазон и точность представления чисел.

Искусство авторов языка (создателей схемы управления расчетами) проявляется в том, чтобы найти разумный компромисс между единообразием управления и эффективностью расчетов при гарантированной надежности.

Начать естественно с разумной классификации числовых данных в зависимости от характера расчетов.

В Аде три категории числовых типов: целые, вещественные плавающие и вещественные фиксированные. Для каждой категории типов - свои средства объявления.

Целые типы предназначенны для точных расчетов, вещественные - для приближенных. При этом плавающие типы воплощают идею действий с нормализованными числами, представленными с относительной погрешностью, зависящей от количества значащих цифр, а фиксированные - идею действий с ненормализованными числами, представленными с абсолютной погрешностью, зависящей от положения десятичной точки (запятой).

[Работать с ненормализованными числами приходится потому, что во встроенных системах (на которые в первую очередь ориентированна Ада) часто применяются относительно дешевые компьютеры с машинным словом небольшой длинны и без встроенной плавающей арифметики.

Чтобы достичь на них высокой скорости и приемлемой точности расчетов, приходится вникать в детали выполнения каждого арифметического действия, экономя на каждом сдвиге положения десятичной точки. Здесь и оказываются полезными фиксированные типы.]

Плавающие типы чаще встречаются в языках программирования для числовых расчетов. Поэтому ограничимся демонстрацией основной идеи управления расчетами на примере плавающих типов Ады.

4.15.3. Зачем объявлять диапазон и точность

Оставим пока в стороне синтаксис и семантику соответствующих объявлений. Ведь объявить диапазон и точность - дело относительно нехитрое. А вот что делать, когда разрядная сетка машины или принятое в ней представление чисел не соответствует объявлению типа.

Важно понимать, что объявление типа остается ососбенно полезным именно в такой ситуации.

Действительно, в худшем случае исполнитель получает всю информацию, чтобы признать (и информировать пользователя), что он непригоден для такой программы.

Если объявленные требования к диапазону и точности критичны для предлагаемых расчетов, то такой отказ, безусловно, предпочтительнее, чем трата времени и усилий на заведомо ошибочные расчеты, да еще с риском оставить пользователя в "счастливом" неведении. К тому же легко автоматически или визуально выделить фрагменты программы, вызвавшие непригодность исполнителя. Это помогает либо подобрать подходящий исполнитель (автоматически, если доступна, скажем, неоднородная сеть машин), либо изменить выделенные компоненты программы.

Но в конкретной реализации можно пойти и на моделирование нужного числового типа (программным, микропрограммным или иным современным способом), гарантируя надежность расчетов, возможно, в ущерб скорости.

Разумность такого шага каждый раз должна взвешиваться, пользователь должен быть информирован о возможных затратах, но в целом такое решение подготовлено уже упомянутыми факторами (снижение относительной доли расчетов, повышение требований к надежности, повышение эффективности моделирования).

4.15.4. Единая модель числовых расчетов

Чтобы гарантировать надежность расчетов на любой объектной машине, т.е. управлять диапазоном и точностью на достаточно высоком уровне абстракции, пользователь должен опираться на модель расчетов, единую для всех объектных машин. Однако, стремясь в первую очередь к переносимости (машинной независимости, абстракции) программ, нельзя забывать про эффективность конкретизации (вспомните принцип реальности абстракций). Т.е. единая модель расчетов должна быть гибкой (требовать от конкретных реализаций минимума свойств и действий).

В Аде абстракция (единство) обеспечивается понятием модельных чисел, а конкретизация (гибкость) - понятием безопасных и допустимых чисел.

Совокупность модельных чисел каждого подтипа полностью фиксируется на машинно-независимом уровне (т.е. семантикой языка и применяемым программистом объявлением).

Безопасные числа - это некоторое надмножество модельных чисел, а допустимые числа - надмножество безопасных. Оба эти надмножества зависят от конкретной реализации.

Гибкость модели проявляется в том, что расчеты программируются и оцениваются с помощью модельных чисел, а выполняются с допустимыми числами.

Понятие безопасных чисел, отличных от модельных, нужно только для определения исключительных ситуаций (например, переполнения). Пока результаты операций попадают в диапазон безопасных чисел, исключительные ситуации возникать не должны. Таким образом, при переносе в другую вычислительную среду эффект

выполнения расчетов может быть другим за счет исключительных ситуаций, если безопасные числа в этих средах отличаются.

Итак, с одной стороны, все реализации ориентированы на единую "систему координат" - модельные числа; с другой стороны каждая реализация работает со своими допустимыми числами.

4.15.5. Правила программирования набора модельных чисел. Пример

Пусть, например, объявлен тип

```
type скорость is digits 8;
```

После ключевого слова digits указан спецификатор D (равный в нашем случае восьми), определяющий набор модельных чисел типа "скорость" по следующим (единым для всех реализаций) правилам.

Во-первых, вычисляется количество B двоичных цифр, обеспечивающих ту же относительную погрешность, что и D десятичных цифр.

В общем случае

B = [D * log(10)/log(2) + 1]

(т.е. D - это способ указать B).

В нашем случае B = [8 * 3,3 + 1] = 27.

Набор (фактически - диапазон) модельных чисел определяется как совокупность всех чисел, представимых в виде

знак * мантисса * (2 ** порядок)

где знак - это +1 или -1, мантисса - правильная двоичная дробь, записанная ровно B двоичными цифрами, первая из которых - 1 (т.е. нормализованная дробь), порядок - целое число между -4*B и +4*B.

Таким образом, с каждой спецификацией D связывается конечный набор вещественных модельных чисел. При этом фиксируется не только количество цифр в мантиссе, но и максимальный порядок. В нашем случае он равен

27 * 4 = 108.

[Чем больше порядок, тем "реже" встречаются модельные числа - точность представления относительна.]

Таким образом, в нашем случае в качестве безопасных чисел могут фигурировать, скажем, числа, представимые 27-разрядной мантиссой и семиразрядным порядком (2**7 = 128 > 108), а в качестве допустимых - с 40-разрядной мантиссой и 7-разрядным порядком. Так что для представления такого диапазона допустимых чисел подойдет, скажем, 48-разрядное машинное слово.

[Уточнить диапазон и точность при объявлении производного типа, числового подтипа или объекта можно, как обычно, с помощью ограничения. Например,

```
type высота is new скорость range 0.0 .. 1.0E5 ;
```

(высота может меняться от нуля до десяти тысяч).

```
subtype высота_здания is высота range 0.0 .. 1.0E3 ;
высота_полета : высота digits 5 ;
```

(для переменной высота_полета допустима меньшая точность, чем указано в типе "высота").]

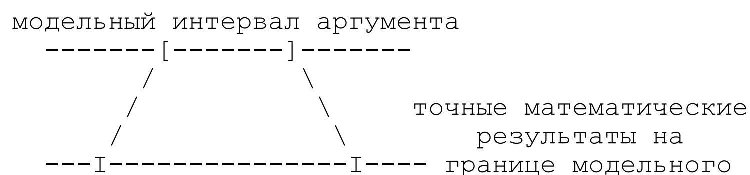
4.15.6. Управление операциями

Результаты всех предопределенных операций над допустимыми числами определяются с помощью модельных чисел следующим единым способом.

Модельным интервалом называется любой интервал между модельными числами. С каждым допустимым числом ассоциируется так называемый связанный модельный интервал - это минимальный модельный интервал, к которому принадлежит рассматриваемое число (для модельных чисел связанным интервалом считается само модельное число).

Основное требование к точности реализации предопределенных операций (модельное ограничение) состоит в том, что результат операций над допустимыми числами должен попадать в минимальный модельный интервал, содержащий точные математические результаты рассматриваемой операции при изменении аргументов операции в пределах их связанных интервалов.

Наглядно это можно представить следующей диаграммой



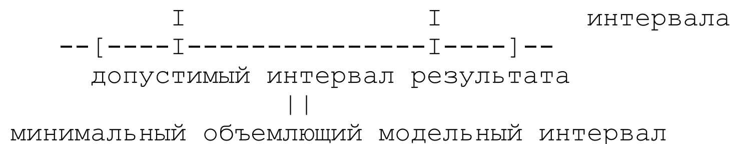


Рис. 4.4

Таким образом, значениями операций вещественных типов в конкретных реализациях могут быть любые числа, обладающие модельным интервалом (или аналогично определяемым "безопасным" интервалом), но отклонения результатов операций регламентированы модельным ограничением (аналогичным "безопасным" ограничением, когда учитываются безопасные числа при "машинно-зависимом" программировании).

Искусство программиста состоит теперь в том, чтобы гарантировать надежность расчетов, подбирая диапазон и точность в соответствии с условиями задачи.

При этом он не должен забывать, что при реализации излишний диапазон или излишняя точность могут стоить дорого и по времени, и по памяти (а могут и вовсе выйти за возможности конкретной реализации языка). Короче говоря, нужно программировать как можно ближе к нуждам решаемой задачи.

В Аде предусмотрено, что реализация может предоставлять несколько предопределенных (именованных или анонимных) плавающих типов с различными диапазонами и точностью расчетов.

Искусство автора компилятора проявляется в том, чтобы компилятор был в состоянии подобрать подходящий предопределенный тип (в качестве допустимых чисел) для любого (или почти любого) объявления типа (с оптимальными затратами на расчеты при полной гарантии соответствия модельному ограничению).

Обратите внимание, в Аде отдано явное предпочтение двоичным машинам (ведь в соответствии с модельным ограничением результат операции над модельным числом должен быть точным, если математический результат оказывается модельным числом). К тому же не любым двоичным машинам, а с достаточно большим порядком во встроенном представлении плавающих чисел (ведь далеко не во всех машинах допустимы порядки, вчетверо превышающие длину мантиссы; во всяком случае, в БЭСМ-6 это не так).

На этом закончим знакомство с единой системой числовых расчетов, воплощенной в концепции числовых типов Ады.

4.16. Управление представлением

4.16.1. Суть проблемы

Единая модель числовых расчетов, как мы видели, позволяет программисту непосредственно управлять представлением данных числовых типов в объектной машине. Например, потребовать, чтобы число было представлено не менее чем 36 разрядами. Но потребность управлять представлением возникает не только для числовых типов и не только для данных. Причем требуется управлять в некотором смысле окончательным (абсолютным) представлением, а не промежуточным (относительным, как это бывало, когда мы в теле пакета представляли данные приватного типа данными комбинированного типа). Другими словами, требуется управлять представлением объектов в терминах понятий, в общем случае выходящих за рамки машинно-независимого ЯП и определенных только на конкретной объектной машине.

Такого рода потребность особенно часто возникает в системных программах, вынужденных взаимодействовать, скажем, с нестандартными устройствами ввода-вывода или со средствами управления прерываниями целевой (объектной) машины. Например, требуется явно указать адрес, с которого располагается реализация реакции на конкретное прерывание.

Конечно, программы, использующие управление абсолютным представлением, перестают быть машинно-независимыми. Однако это та мера зависимости от машины (та мера конкретизации), без которой программа остается неработоспособной.

Вместе с тем это зависимость только от целевой машины. Сами средства управления представлением данных могут оставаться нормальными конструктами машинно-независимого языка, где они играют роль компонент общего аппарата связывания. А именно связывания абстрактной спецификации данных с их конкретным абсолютным представлением. Такое связывание выполняется при трансляции и может быть выполнено на любой транслирующей (инструментальной) машине.

С другой стороны, если связывание такого рода выделено как достаточно общее (абстрактное) языковое понятие и ему соответствует легко идентифицируемый конструкт, то настройка программы (ее перенос) на другую целевую машину сводится к изменению только таких конструктов.

Именно такой идеей и руководствовались авторы языка Ада. Управление абсолютным представлением выделено в конструкт, который называется УКАЗАНИЕМ ПРЕДСТАВЛЕНИЯ (спецификацией представления, representation clauses). Указание представления должно следовать за объявлением тех сущностей, представление которых в нем конкретизируется. В Аде можно указывать представление для типов, объектов данных, подпрограмм, пакетов и задач, а также для входов.

[Высшим уровнем оформления абстракции представления было бы выделение специального программного сегмента ("модуля представления")].

4.16.2. Примеры управления представлением

Рассмотрим несколько примеров одного из самых нужных указаний представления - УКАЗАНИЯ АДРЕСА.

Обычно указание адреса применяют для того, чтобы связать объект данных, подпрограмму или вход задачи с той ячейкой памяти целевой машины, которая играет какую-то особую роль. Это может быть регистр определенного периферийного устройства; адрес, по которому передается управление при определенном прерывании; ячейка, состояние которой определяет режим работы машины и т.п.

Чтобы можно было записать указание адреса, должен быть доступен тип "адрес". Содержательно значения этого типа играют роль адресов памяти целевой машины, формально это один из предопределенных типов. В Аде его определяющим пакетом считается предопределенный пакет "система", характеризующий целевую машину. Следовательно, тип "адрес" становится доступным с помощью указания контекста вида

```
with система; use система;
```

Вот примеры указания адреса с очевидным назначением.

```
for управл_ячейка use at 16#0020# ;
```

после at записана шестнадцатеричная константа типа "адрес". Такое указание адреса должно быть помещено среди объявлений блока, пакета или задачи после объявления объекта управл_ячейка.

```
task обработка_прерывания is
                              entry выполнить ;
                              for выполнить use at 16#40# ;
                              -- вызвать вход  "выполнить" - это значит
                              -- передать управление в ячейку 16#40#
                   end обработка_прерывания ;
```

Еще примеры использования указаний представления.

```
      слово : constant  := 4 ; -- элемент памяти - байт,
                          -- "слово" - из четырех байтов.
      type состояние is  (A,M,W,P) ; -- четыре возможных состояния
   -- код символов, внутренние прерывания, ожидание, супервизор.
      type режим is  (фикс, десят, плав, сообщ) ;
      type маска_байта is  array  (0 .. 7) of BOOLEAN ;
      type маска_состояния is array  (состояние) of BOOLEAN ;
      type маска_режима is array (режим) of BOOLEAN ;
      type слово_состояние_программы is
            record
                   маска_системы ; маска_байта ;
     ключ_защиты : INTEGER range 0 .. 3 ;
     состояние_машины : маска_состояния ;
     причина_прерывания : код_прерывания ;
     код_длины_команды : INTEGER range 0 .. 3 ;
     признак_результата : INTEGER range 0 .. 3 ;
     маска_программы : маска_режима ;
     адрес_команды : адрес ;
   end record ;
  -- ниже следует указание представления для этого типа
  for слово_состояния_программы use
    record at mod 8 ; -- адрес должен быть нулем по модулю 8, т.е.
                 -- адресом двойного слова.
```

```
            маска_системы at 0 * слово range 0 .. 7 ; -- маска системы
               -- расположена в первом байте слова.
   ключ_защиты at 0 * слово range 10 .. 11 ;
               -- разряды 8 и 9 не используются.
   состояние_машины at 0 * слово range 12 .. 15 ;
   причина__прерывания at 0 * слово range 16 .. 31 ;
   код_длины_команды at 1 * слово range 0 .. 1 ;
   признак_результата at 1 * слово range 2 .. 3 ;
   маска_программы at 1 * слово range 4 .. 7 ;
   адрес_команды at 1 * слово range 8 .. 31 ;
 end record ;
```

Здесь применено так называемое УКАЗАНИЕ ПРЕДСТАВЛЕНИЯ ЗАПИСИ. Запись типа слово_состояние_программы располагается в двойном слове, т.е. по адресам, кратным 8, причем для каждого поля указано точное расположение относительно начала записи.

На этом закончим знакомство с управлением представлением, а одновременно и с системой типов в Аде. В следующем разделе сопоставим ее с нашей исходной классификацией данных.

4.17. Классификация данных и система типов Ады (прогнозирование-контроль поведения объектов в Аде)

Рассматривая данные как одну из основных абстракций программирования, мы выделили шесть факторов их классификации. Опираясь на нашу классификацию, дадим теперь краткий обзор средств управления данными в Аде. Напомним, что классификация эта возникла из анализа технологических потребностей (прежде всего, из потребностей прогнозирования и контроля, обеспечивающих надежность программирования). Так что по сути дела нас будет интересовать, в какой степени рассматриваемые понятия и конструкты Ады обслуживают эти потребности.

1. Содержательные роли данных. На первый взгляд возможность явно отражать в программе содержательные роли данных так, чтобы можно было автоматически контролировать связывания, кажется почти фантастической. А идея очень проста. Во-первых, нужно, чтобы содержательная роль объекта получила имя, отличающее ее от других ролей. Во-вторых, проектируя объект данных, нужно связывать с ним имя той роли, в которой он должен выступать при выполнении программы. Имя роли естественно указывать при объявлении объекта. В-третьих, проектируя (объявляя) содержательные действия, нужно явно учитывать (указывать при объявлении действий) имена ролей объектов, участвующих в этих действиях.

При таком прогнозировании контроль за соответствием поведения (использования) объектов объявленным их ролям становится легко формализуемым. Его обеспечивает концепция типа, ориентированная на имена. В частности, реализованная в Аде концепция уникальности типа (строгая типизация).

Упражнение. Предложите иные концепции, обеспечивающие прогнозирование-контроль содержательных ролей данных.

2. Строение данных. Как уже отмечалось, классификация по структуре данных (по способу доступа к составляющим) имеется практически во всех ЯП. В Аде по этому фактору различаются скалярные, регулярные, комбинированные, ссылочные, задачные и приватные данные. Соответственно выделяются и категории типов данных. Правила объявления типов данных в Аде таковы, что к одному типу можно относить только данные, формально "близкие" по своему строению. При необходимости подчеркнуть, что данные разного строения играют одинаковую содержательную роль, их можно объявить в качестве вариантов так называемого вариантного комбинированного типа.

3. Изменчивость данных. Характеристика изменчивости связана с общей концепцией типа в Аде, однако стоит по отношению к ней несколько особняком. Точнее говоря, при объявлении типа данных в Аде всегда сообщается максимально допустимая изменчивость данных этого типа. Но когда объявляют отдельный объект или целый класс объектов (подтип), изменчивость можно ограничить.

Возникают два вопроса. Первый - откуда брать исходные совокупности значений, определяющие максимальную изменчивость. Второй - как управлять изменчивостью.

Для каждой предопределенной категории типов имеются свои исходные совокупности значений и способы управления изменчивостью. Для целых типов - определяемый реализацией диапазон целых чисел, для вещественных типов - определяемый реализацией диапазон вещественных чисел, для символьных - стандартный алфавит. Для перечисляемых скалярных типов исходная совокупность значений задается явным их перечислением при объявлении типа.

Для составных типов максимальная исходная совокупность определяется прямым произведением совокупностей значений составляющих типов. Для ссылочных - это всевозможные ссылки на объекты квалифицирующего типа (т.е. типа, на который позволено ссылаться объектам создаваемого типа). Для приватных - на уровне использования конкретные значения обычно вообще не известны, а на уровне реализации - это в конечном итоге всегда один из типов других категорий.

Изменчивостью управляют посредством явных ограничений - указывая диапазон изменения значений, диапазон индексов, фиксируя значения отдельных полей или даже всего объекта. В последнем случае получается постоянная. Концепция типа в Аде в целом обеспечивает квазистатическое управление изменчивостью.

4. Способ определения. Различаются предопределенные типы и объявляемые пользователем. С этой точки зрения в Аде особенно интересны приватные типы. На уровне использования они инкапсулированы и могут быть сделаны неотличимыми от предопределенных. В Аде это одно из важнейших средств развития, средство определения нового языка. Точнее, это основное средство определения нового языка с новыми абстракциями данных. Без приватных типов можно вводить новые операционные абстракции, но не абстракции данных. Именно поэтому приватные типы незаменимы при создании надежных пакетов прикладных программ.

5. Представление. Ада позволяет управлять, во-первых, относительным представлением данных, когда речь идет о представлении приватных типов на уровне их реализации типами иных категорий; во-вторых, абсолютным представлением, когда речь идет о представлении любых типов на целевой машине (посредством указаний представления).

6. Внешние свойства. Набором применимых операций в Аде управляют объявлением типа и определяющим пакетом.

Итак, система типов языка Ада хорошо согласуется с нашей классификацией. С другой стороны, эта классификация указывает направления развития адовских средств управления данными.

Упражнение. Предложите такие средства.

Замечание. Наша классификация данных отражает характеристики данных, обычно охватываемые концепцией типа. Но данные различаются и по другим факторам. Один из них - отношение данного и модуля программы. Очень четко такое отношение отражено в языке Том [13] понятием класса данного. Выделены глобальные данные, параметры, локальные и синхропараметры. Аналогичные понятия имеются, конечно, и в других ЯП.

Вопрос. Как Вы думаете, разумно ли объединить понятие класса и типа?

Подсказка. Не забудьте, в частности, о концепции уникальности типа.

4.18. Предварительный итог по основным абстракциям и модели А.

Итак, выделив три основные абстракции - данные, операции и связывание, мы углубились в основном в изучение первой из них, одновременно получая представление о мощном языке индустриального программирования (фактически мы строим "максимальную" модель такого языка, модель А).

В отличие от традиционных ЯП (Фортрана, Алгола, Бейсика и др.), язык Ада ориентирован скорее на данные, чем на операции. В нем в первую очередь поддерживается такой стиль программирования, когда проектируется не столько программа, сколько комплекс программных услуг, опирающийся на ключевую структуру данных.

Решая наши модельные задачи в таком стиле, мы одновременно вникали в суть целой серии основных абстракций программирования (чисел, дискретных диапазонов, асинхронных процессов, массивов и записей, ссылок и динамических объектов).

С другой стороны, параллельно мы знакомились с различными видами операций (для каждой категории типов - свои) и с различными видами связывания (статическое, динамическое, квазистатическое, подразумеваемое по умолчанию, выбираемое компилятором, указываемое программистом).

Однако вне поля нашего зрения осталось еще несколько важных абстракций, которыми мы теперь и займемся, одновременно завершая знакомство с языком Ада (и построение нашей четвертой модели, модели А).

Упражнение. Приведите примеры перечисленных видов связываний.

Вопрос. Как Вы думаете, чем отличается модель А от языка Ада?

Литература

1. С.С.Лавров. Основные понятия и конструкции языков программирования. - М.: Финансы и статистика, 1982.

2. Базисный рефал и его реализация на вычислительных машинах (методические рекомендации). - ЦНИПИАСС, Госстрой СССР, 1977.

3. J.Backus. Can Programming Be Liberated from von Neumann Style? A Functional Style and Its Algebra of Programs. - CACM, 1978, v. 21, n.8, 613-641.

4. Д.Грис. Наука программирования.-М.: Мир, 1984.

5. Н.Вирт. Модула-2. (Пер. с английского Л.А.Захарова.- В кн.: Языки программирования.- М.:Наука,1985. (Алгоритмы и алгоритмические языки).

6. Э.З.Любимский, В.В.Мартынюк, Н.П.Трифонов. Программирование. - М.:Наука, 1980.

7. Дж.Хьюз, Дж.Мичтом. Структурный подход к программироованию. - М.: Мир, 1980.

8. С.Янг. Алгоритмические языки реального времени. Конструирование и разработка. - М.: Мир, 1985.

9. Ф. Брукс. Как проектируются и создаются программные комплексы.

10. В.Л.Темов. Метаалгоритмическая система общего назначения МАСОН. - сб. Прикладная математика в биологии., Л.: ИНФ им. Павлова, 1972.

11. В.М.Пентковский. Автокод Эльбрус. - М.: Наука, 1982.

12. Бринч-Хансен. Методы проектирования операционных систем.

13. В.Л. Темов. Язык и система программирования Том. (В печати).

14. Т. Пратт. Языки программирования. Разработка и реализация. (появилось второе американское издание 1983 года).

15. Д.Грис "Конструирование компиляторов для цифровых вычислительных машин" - М.:Мир,1975.

16. Р.Хантер "Проектирование и конструирование компиляторов" - М."Финансы и статистика",1984.

17. Пайл Я. "Ада - язык встроенных систем". М.: Финансы и статистика, 1984, главы 9 и 10.

18. П.Вегнер. Программирование на языке Ада. М.: Мир, 1983.

19. The Programming Language Ada Reference Manual. American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983. - Lecture Notes in Computer Science, 155, 1983.

20. Н. Вирт. Алгоритмы + структуры данных = программы. - М.: Мир, 1985.

4.19. Раздельная компиляция

Раздельная трансляция (компиляция) - одна из критичных технологических потребностей индустриального программирования.

ТРАНСЛЯЦИОННЫЙ МОДУЛЬ - это программный сегмент, пригодный для раздельной трансляции. Это фрагмент текста, который можно физически отделить от контекста и применять посредством трансляционной библиотеки.

Мы затронули одну из важнейших концепций ЯП - концепцию модульности. Здесь ограничимся следующим определением.

Модуль - это (материализованная) абстракция, доведенная до воплощения в отдельном физическом объекте.

4.19.1. Аппарат связывания трансляционных модулей

Трансляционные модули связываются для того, чтобы взаимодействовать как части единой программы. Для этого приходится называть имена партнеров по связыванию.

Выделим два основных вида связывания, которые назовем односторонним и двусторонним соответственно. При одностороннем связывании лишь один из двух связываемых модулей называет имя своего партнера. При двустороннем - оба. В Алголе, Фортране и Бейсике применяется только одностороннее связывание (в процедуре не указывается вызов, при объявлениях объекта не указываются все его использования и т. д.). В Аде для трансляционных модулей применяются оба вида связывания.

Трансляционный модуль в Аде - это такой программный сегмент, все внешние связи которого оформлены как связи с трансляционной библиотекой.

Как же оформить эти связи?

С односторонним связыванием мы уже фактически познакомились, когда применяли указание контекста (with).

Например, трансляционными модулями были: спецификация пакета управление_сетью (в ней вообще нет внешних связей, кроме предопределенных имен типов), процедура построение_сети (она связана только с пакетом управление_сетью и эта связь оформлена односторонним связыванием - с помощью указания контекста), родовой сегмент "очередь" - также трансляционный модуль. Все это примеры так называемых первичных или открытых (library) модулей. Название "открытых" отражает факт, что в силу односторонней связи этими модуями можно пользоваться открыто, в любом месте программы, для чего достаточно употребить соответствующее указание контекста.

Тело пакета управление_сетью и вообще тела подпрограмм и пакетов без внешних связей (кроме связи со "своей" спецификацией) служат примерами так называемых вторичных модулей. Их имена можно трактовать как указание двусторонней связи с соответствующими спецификациями.

[Ведь на тела можно указывать (ссылаться) извне только через спецификации. При этом имя пакета или подпрограммы в заголовке спецификации трактуется как указание на связь с соответствующим телом, а то же имя в заголовке тела трактуется как указание на связь с соответсвующей спецификацией. Так что связь действительно двусторонняя.

Когда же сами спецификации - не отдельные модули, а внутренние компоненты других модулей, то по-прежнему можно оформлять тела пакетов, процедур и задач как вторичные модули, но для этого нужно явно указать соответствующую двустороннюю связь. Именно: в том модуле, где находится спецификация, применяют так называемую заглушку, указывающую на вторичный модуль, а в заголовке вторичного модуля явно указывают имя того модуля, где стоит заглушка. Признаком двусторонней связи служит ключевое слово separate .

Например, можно оформить как вторичный модуль тело любой процедуры из пакета управление_сетью. Выберем процедуру "вставить" и функцию "перечень_связей". Тогда в теле пакета вместо объявления этих подпрограмм нужно написать заголовки вида

```
function перечень_связей (узел: имя_узла)
                return BOOLEAN is separate;
```

```
            procedure вставить (узел: in имя_узла) is separate;
    -- перед нами две ссылки на вторичные модули.
```

Соответствующие вторичные модули нужно оформить так:

```
            separate (управление_сетью) -- указано местонахождение заглушки
            function перечень_связей (узел: имя_узла) return BOOLEAN is
                                        . . . -- тело как обычно
            end перечень_связей;

      separate (управление_сетью)
            procedure вставить (узел: in  имя_узла) is
                                        . . . -- тело как обычно
            end вставить;
```

Теперь вторичные модули можно транслировать отдельно. В Аде предписан определенный (частичный) порядок раздельной трансляции. В частности, все вторичные после модулей с соответствующими заглушками (старших модулей, которые могут быть как первичными, так и вторичными).

Итак, к вторичному модулю можно "добраться" только через его партнера. Поэтому в отличие от открытых библиотечных модулей их естественно называть закрытыми. Свойство закрытости обеспечено применением явной двусторонней связи.

Подчеркнем, что закрытые модули появились в Аде не случайно. Они, с одной стороны, естественное развитие разделения спецификации и реализации до уровня модульности (вспомните, модуль - материальное воплощение абстракции). С другой стороны, они обслуживают очень важный принцип конструирования языка Ада, который можно было бы назвать принципом защиты авторского права (языковыми средствами).

Дело в том, что тексты вторичных модулей можно вовсе не предоставлять пользователю при продаже програмных изделий, созданных на Аде. Ему передаются только спецификации - открытые модули и оттранслированные вторичные. Тогда пользователь никак не сможет незаконно добраться до вторичных модулей. При соответствующей защите от несанкционированного доступа он не только не сможет неправильно пользоваться ими, но не сможет и скопировать или употребить для построения других систем (отдельно от закупленных).

Замечание.

В общем случае возникает соблазн рассматривать раздельно транслируемый модуль как программу, управляющую транслятором в процессе раздельной трансляции. Результатом выполнения такой программы может быть модуль загрузки, перечень обнаруженных ошибок и (или) изменения в трансляционной библиотеке. При таком общем взгляде на раздельную трансляцию управление ею становится столь же разнообразным и сложным, как программирование в целом. Такой взгляд может быть полезным, например, при конкретизирующем программировании, когда универсальный модуль рассматривается как генератор разнообразных версий программы, учитывающих особенности конкретных применений.

Разнообразными становятся и способы извлечения из внешней среды информации об указанных особенностях (параметры, анализ среды, диалог с программистом и т.п.). Так что в общем случае создание универсальных модулей в корне отличается от написания частей конкретных программ. Такие модули становятся метапрограммами, описывающими процесс или результат создания частей конкретных программ. Поэтому язык для написания универсальных модулей и управления их связыванием в общем случае может сильно отличаться от исходного языка программирования. Так что рассмотренный общий взгляд по существу выводит за рамки собственно раздельной трансляции.

Настройка "универсальных" модулей на конкретное применение в Аде есть - ее обслуживает аппарат родовых сегментов. Но конкретизация родовых сегментов выполняется не при связывании собственно модулей, а позже, в рамках трансляции сегментов.

Конец замечания.

4.20. Основные абстракции: нотация (представление знаков)

4.20.1. Проблема знака в языке программирования

Вспомним, что ЯП - знаковая система для планирования поведения исполнителя. Но знаковая ситуация возникает лишь тогда, когда знак может быть передан отправителем и получен адресатом. Как отправителем, так и адресатом может оказаться и человек, и компьютер. Язык программирования должен быть, во-первых, средством мышления людей, создающих программы, во-вторых, средством их общения между собой по поводу создания программ; в-третьих, средством общения людей с

компьютерами и ,в-четвертых, компьютеров между собой. Для людей важно, чтобы знаки были и выразительны, и надежны, и лаконичны, и удобны для письма и чтения. Необходимость общаться с компьютерами предъявляет к знакам особые требования. Достаточно вспомнить, что знаки языка программирования нужно вводить устройствами ввода и выводить устройствами вывода.

Указанные требования весьма разнообразны и порой противоречивы. Вспомним, что в своих моделях мы "для удобства" пользовались знаками, которых часто нет на устройствах ввода-вывода. Может оказаться невозможным использовать буквы кириллицы, некоторые привычные символы операций и т.п. Мы уже привыкли, что нельзя опускать и поднимать индексы, вводить рукописный текст и т.п. (хотя в будущем это наверняка будет возможно).

Итак, даже если в знаковой ситуации, соответствующей ЯП, сконцентрировать внимание исключительно на выборе знаков, по возможности абстрагируясь от проблемы смысла, то найти решение, удовлетворяющее в разумной мере пользователей языка и производителей оборудования, бывает очень не просто. Когда же необходимо искать решение с учетом массового применения языка в национальном (тем более, мировом) масштабе, то возникает самостоятельная серьезная проблема - проблема знака.

В этом разделе мы сконцентрируемся лишь на части этой большой проблемы, проблеме представления знаков (проблеме нотации).

4.20.2. Определяющая потребность

Выделим технологическую потребность, определяющую в настоящее время решение проблемы нотации - потребность записывать программу так, чтобы ее можно было ввести в любой компьютер без особых затрат и риска внести ошибки. Назовем ее потребностью совместимости по вводу. Эта потребность - определяющая в том смысле, что ради ее удовлетворения в современной ситуации с индустриальным программированием можно в значительной степени пренебречь, скажем, пожеланиями некоторых категорий пользователей (разнообразие шрифтов, управление цветом, нелинейная запись и т.п.).

Другими словами пишуший программу (отправитель знака) должен иметь возможность абстрагироваться от особенностей устройств ввода у адресата. С другой стороны, нужно обеспечить возможность "каждому желающему" конкретному исполнителю выступить в роли адресата, возможность воспринять написанное отправителем.

4.20.3. Основная абстракция

Абстракция, обслуживающая потребность совместимости по вводу, хорошо известна - это абстрактный (эталонный) текст. Понятие эталонного текста определено в каждом ЯП. (Эталонный) текст - это конечная последовательность (эталонных) символов. Набор символов в языке обычно конечен и линейно упорядочен. Он называется алфавитом язка. Потребность в совместимости удовлетворяется за счет того, что на определенном уровне абстракции именно эталонный текст является знаком программы. Именно он подразумевается, когда работают на конкретном устройстве ввода-вывода.

Но на конкретном устройстве свой алфавит. Так что приходится придумывать способ обозначать эталонные символы конкретными символами, доступными на устройстве, а эталонный текст в целом - конкретным текстом (составленным из конкретных символов) . Так, эталонные иероглифы Алгола (begin, end и т.п.) обозначаются иногда "BEGIN", "END", иногда _begin_, _end_, иногда `НАЧАЛО', `КОНЕЦ' и т.п.

Таким образом, конкретный текст обозначает эталонный, а тот, в свою очередь, обозначает программу (план для исполнителя).

Итак, основная абстракция осознана - это эталонный текст. Но, в соответствии с принципом реальности абстракций, для каждой абстракции необходимы средства конкретизации (средства учета тех частностей от которых абстрагировались, создавая абстракцию). Проблема нотации дает пример, когда средства конкретизации по необходимости выходят за рамки языка, создавая внешнюю проблему конкретизации эталонного текста.

4.20.4. Проблема конкретизации эталонного текста

Обычно в ЯП отсутствуют средства управления связью конкретных и абстрактных текстов. Дело в том, что средства управления сами должны быть обозначены некоторыми текстами, их также нужно вводить и выводить. Короче, для них возникнут те же проблемы, что и для языка в целом. Так что решать проблему конкретизации приходится вне ЯП.

Тем более важно принять рациональные решения, определяющие правила конкретизации абстрактных текстов, так как они принимаются "раз и навсегда".

Важность решений, о которых идет речь, можно показать на примере Алгола-60. В свое время его авторы по существу игнорировали проблему конкретизации. Они ввели три уровня языка - эталонный, для публикаций и конкретные представления. Первый был ориентирован "исключительно на взаимопонимание", второй - на "типографские особенности", третий - на устройства ввода-вывода. Что касается проблемы конкретизации, то авторы ограничились оговоркой, что каждая реализация должна иметь "правила для перевода конкретных представлений в эталонные".

Именно "правила". Не программы, не программные изделия для перевода, а только правила. Затраты ресурсов на такой перевод и риск внести ошибки не оценивались и не контролировались. На практике это привело к несовместимости различных трансляторов с Алгола. Так как реализаторы не только не подкрепляли "правила" конкретными программными изделиями, но и не всегда четко осознавали "правила". Проблема несовместимости реализаций, в свою очередь, сыграла не последнюю роль в том, что Алгол не сумел выдержать конкуренцию с Фортраном в качестве языка массового программирования для научных расчетов.

Итак, допустим, что важность проблемы конкретизации осознана. Как рационально решить эту проблему?

4.20.5. Стандартизация алфавита

Игнорировать проблему нельзя, управлять конкретизацией невозможно. Остается по существу единственный путь стандартизация алфавита (или определение ЯП со стандартным алфавитом).

Ключевая идея состоит в том, что проблема выносится за рамки рассматриваемого языка и выбирается опорный стандарт на цифровые коды символов (достаточно распространенный, лучше всего - международный). Эталонный алфавит языка определяется через опорный стандарт (по существу, эталонным алфавитом становится некоторое подмножество цифровых кодов символов из опорного стандарта, а связанные с этими кодами видимые (графические) и (или) управляющие символы образуют допустимые конкретные алфавиты). Тем самым определяются и допустимые вариации конкретных алфавитов (рамками того же опорного стандарта).

Теперь становится, с одной стороны, невозможным "изобретать" в эталонном алфавите ЯП такие символы, как иероглифы Алгола (которые по определению "не имеют никакого отношения к составляющим их символам") или наши сдвоенные стрелочки (-->-->), так как их нет в международных стандартах, для них нет цифровых кодов. Другими словами, авторам ЯП предлагается выбирать из стандартного набора символов. С другой стороны производители оборудования и систем программирования вынуждены считаться с действующими стандартами и обеспечивать, во-первых, наличие на клавиатуре устройств минимального набора знаков и ,во-вторых, их правильное, определяемое стандартом, соответствие цифровым кодам (например, А - 101, В - 102, 0 (нуль) - 60, 1 - 61 и т.п.). Таким образом, на некотором этапе обработки текст обязательно представлен стандартной последовательностью числовых кодов. Ее и следует считать "эталонным" текстом. Именно такой эталонный текст и обеспечивает практическую совместимость по вводу.

Стандартизация алфавита требует коллективных усилий международного сообщества, самоограничения и дисциплины авторов ЯП, производителей компьютеров и периферийных устройств. Но и уровень совместимости по исходным программам в языке со стандартизированным алфавитом зависит не от распространенности конкретной реализации языка, а от распространенности опорного стандарта.

Благодаря целенаправленной деятельности национальных и международных организаций по стандартизации в настоящее время существуют достаточно авторитетные стандарты на символы (семибитный и восьмибитный международные стандарты ИСО и соответствующие национальные стандарты, в том числе и отечественный ГОСТ). Так что создана приемлемая база для разработки ЯП со стандартным алфавитом.

Рост технических возможностей и ,соответственно, потребностей пользователей, может привести к пересмотру стандартов на коды символов (например, чтобы можно было работать с цветом или с различными шрифтами). Тогда появится больше возможностей и у авторов ЯП. Вместе с тем потери от несовместимости обычно несопоставимы с выигрышем от нарушения стандарта, так что известная доля консерватизма в решении проблемы нотации вполне естественна.

Для ЯП со стандартным алфавитом нет особого смысла различать эталонные и конкретные тексты. Другими словами, абстракция представления в этом случае почти вырождается за счет стандартизации конкретных представлений.

Первым языком со стандартным алфавитом был Фортран. В настоящее время этот путь решения проблемы представления знака для вновь создаваемых ЯП можно считать общепринятым.

4.20.6. Основное подмножество алфавита

Еще одна заслуживающая внимания идея (позволяющая работать на "бедных" устройствах, не соответствующих полному опорному стандарту на коды символов) состоит в выделении так называемого основного подмножества алфавита. При этом в определении ЯП фиксируются правила изображения остальных символов алфавита с помощью комбинаций символов из основного подмножества. Написанный по этим правилам текст обозначает нужный текст в полном алфавите, а передавать и воспринимать его можно на "бедных" устройствах.

В любой "богатой" реализации языка можно (и нужно) иметь средства для кодирования и декодирования "бедных" текстов по упомянутым правилам, так что идея основного подмножества практически не мешает "богатым" пользователям и существенно помогает "бедным".

4.20.7. Пример: алфавит языка Ада

Текст исходной программы в Аде - это последовательность символов. Символы делятся на графические и управляющие. Каждому символу однозначно соответствует семибитный код ИСО. Вариации графических символов возможны только в рамках, допустимых стандартом ИСО для национальных стандартов (например, знак доллара можно заменить знаком фунта стерлингов). Управляющие символы графического представления не имеют, они предназначены для форматирования текста (горизонтальная табуляция, вертикальная табуляция, возврат каретки, перевод строки, перевод страницы).

Среди графических символов выделено основное множество (большие латинские буквы, цифры, пробел и спец-символы # & ' ( ) * + , - . : ; < = > _ |).

Кроме того, в алфавит входят малые латинские буквы и дополнительные символы (! $ % ? @ [\ ] ' ` { } ^).

Правила, позволяющие обозначить произвольную программу с помощью только основного множества, таковы. Во-первых, в качестве обязательных элементов программы (ключевые слова, ограничители и разделители) используются только символы из основного множества. Во-вторых, малые и большие буквы эквивалентны всюду, кроме строк и символьных констант. (Так что и идентификаторы можно представлять в основном множестве). А строки обозначаются с помощью символа & так, что "явное" изображение строки эквивалентно "косвенному", использующему название нужной подстроки. Например, если ASCII.DOLLAR - это название строки "$", то обозначение "A $ C" эквивалентно "A" & ASCII.DOLLAR & "C".

Подобные названия для всех дополнительных символов и малых латинских букв предопределены в языке Ада. Это и позволяет записать любую программу с помощью одного только основного множества. (Еще пример: "AвC" эквивалентно "A" & ASCII.LC_B & "C").

4.20.8. Лексемы

Понятие эталонного текста как последовательности символов (литер) позволяет абстрагироваться от особенностей устройств ввода-вывода. Однако символ - слишком мелкая единица с точки зрения тех сущностей, которые необходимо обозначатьть в языке. Их намного больше, чем элементов в алфавите. Удобно, когда эти сущности имеют индивидуальные обозначения, подобные словам естественного языка, а текст оказывается последовательностью таких "слов", называемых лексемами. Мы пришли к еще одному (промежуточному) уровню абстракции - уровню лексем. (Можно считать, что этот уровень удовлетворяет потребность в рациональной микроструктуре текста - приближает размеры "неделимого" знака к размеру "неделимого" денотата).

Когда этот уровень абстракции выделен явно, и при письме, и при чтении можно оперировать достаточно крупными единицами (лексемами), абстрагируясь (когда это нужно) от конкретного способа представления лексем символами алфавита. Становится проще манипулировать с текстом, увеличивается надежность, растет скорость создания и восприятия текста.

Между тем в ЯП уровень лексем выделяется далеко не всегда. Неудачная идея игнорировать пробелы как естественные разделители возникла на заре развития ЯП (ср.Фортран и Алгол), повидимому как отрицательная реакция на необходимость "считать пробелы" в первых позиционных автокодах. В результате была временно утеряна отлично зарекомендовавшая себя традиция естественных языков - выделять

слова пробелами. В Алголе к тому же игнорируются все управляющие символы, а в Фортране - переход на новую строку внутри оператора. В естественных языках подобные особенности текста обычно используются как разделители слов. В последние годы идея явного выделения уровня лексем становится общепризнанной и при конструировании ЯП.

[Интересно отметить, что "возвращение пробела" как значащего символа связано и пониманием "ключевых слов" просто как зарезервированных слов (а не иероглифов, как в Алголе), ничем другим от остальных слов-лексем не отличающихся. Но тогда естественно запретить сокращать ключевые слова (иначе их можно спутать теперь уже не только с другими ключевыми словами, но и с идентификаторами). Это в целом полезное ограничение, так как способствует надежности программирования, помогая чтению за счет некоторой дисциплины письма (что вполне в духе индустриального программирования). Кстати, не очевидно, что напечатать "procedure" труднее, чем `proc', с учетом переключения внимания на спецзнаки. К тому же современные системы подготовки текстов позволяют легко вводить словари сокращений (так что и чтения не затрудняют, и печатать удобно)].

4.20.9. Пример: лексемы в Аде

Лексемы в Аде аналогичны словам естественного языка. Они делятся на шесть классов: ограничители (знаки препинания), идентификаторы (среди которых - зарезервированные ключевые слова), числа, обозначения символов, строки и примечания. В некоторых случаях, когда невозможно иначе однозначно выделить лексему, требуется явный разделитель между смежными лексемами. В качестве разделителя выступает или пробел, или управляющий символ, или конец строчки. Пробел, естественно, не действует как разделитель в строках, примечаниях и в обозначении пробела (` '). Управляющие символы (кроме, возможно, горизонтальной табуляции, эквивалентной нескольким пробелам) всегда служат разделителями лексем. Между лексемами (а также до первой и после последней лексемы) текста, допустимо несколько разделителей. Заметим, что каждая лексема должна располагаться на одной строке (ведь конец строки - разделитель).

Со списком ключевых слов Ады мы познакомились по ходу изложения. Многие из них привычны (procedure, begin, do и т.д.). Сокращать ключывые слова недопустимо.

Ниже следует описание классов лексем.

Ограничитель. Это одиночный символ

```
       & ' ( ) * + , - . / : ; < = >
```

и пара символов

```
=> .. ** := /= >= <= << >> <>
```

При этом символ может играть роль ограничителя только, когда он не входит в более длинную лексему (парный ограничитель, примечание, строку).

Идентификатор. Отличается от алгольного идентификатора только тем, что внутри его допускается одиночное подчеркивание. Большие и малые буквы считаются эквивалентными. Идентификаторы считаются различными, если отличаются хотя бы одним символом (в том числе и подчеркиванием).

Примеры: 'A', '*', ''', ' ' и т.п.

Строка. Это последовательность графических символов, взятая в двойные кавычки. Внутри строки двойная кавычка изображается повторением двойной кавычки ("").

Примеры. "Message of the day".

Примечание. Начинается двумя минусами и заканчивается концом строки.

Число.

Вот примеры целых чисел:

```
  65_536  ,   10.000
  2#1111_1111#   ,  16#FF#  ,  016#0FF#
            -- целые константы, равные 255
  16#E#E1  ,  2#1110_0000#
            --  это 222
```

Вот примеры вещественных чисел

```
16#F.FF#E+2  ,     2#1.1111_1111_111#E11
            -- 4095.0
```

(Пробелы внутри не допускаются - ведь они разделители).

4.21. Исключения (исключительные ситуации)

В самом начале курса отмечалось, что планировать поведение исполнителя означает, в частности, согласовывать его модель мира с моделью решаемой задачи.

Чем лучше эти модели согласованы априори, тем проще планировать поведение, проще достичь взаимопонимания. В сущности, изученные нами абстракции включают в ЯП именно для того, чтобы приблизить мир исполнителя к миру решаемых задач. Однако до сих пор у нас не было случая обратить внимание на то, что "мир исполнителя" не следует ограничивать операциями и данными (возможно, весьма мощными и разнообразными).

Сколь мощной ни была бы операция, ее нужно явно указать в программе. Между тем одна из самых общих технологических потребностей, потребность писать надежные программы, непосредственно приводит к абстракции новой категории, к понятию об априорных правилах поведения исполнителя (т.е. правилах поведения, предопределенных в ЯП и не связанных непосредственно с какими-либо указаниями в программе).

Будем говорить только о правилах поведения исполнителя в так называемых исключительных ситуациях.

[Хотя именно априорные правила поведения, а не специфические данные и операции, характеризуют современные развитые ЯП. С этой точки зрения удобно рассматривать и взаимодействие задач, и так называемый перебор с возвратом (backtracking), и вызов процедур по образцу, и поведение так называемых экспертных систем, и вообще программирование достаточно высоко интеллектуальных исполнителей (в том числе программирование человеческой деятельности)].

Связь с этим аспектом мира исполнителя не обязательно явно указывать в программе - он вступает в действие при исполнении любой программы, как только наступает исключительная ситуация. Вместе с тем в развитом ЯП могут быть и специальные средства, позволяющие управлять поведением исполнителя в исключительных ситуациях.

4.21.1. Технологическая потребность

Чтобы почувствовать потребность в априорных правилах поведения, представим себе заводского технолога, планирующего последовательность операций по изготовлению некоторой детали (аналогия с программированием довольно очевидна). Описание технологии (технологическая карта) предусматривает, скажем, отливку, затем фрезеровку поверхностей и расточку отверстий. Каждый из этапов расписывается в технологической карте довольно подробно. Однако все подробности касаются особенностей создания именно нужной детали. Общие (априорные) правила поведения исполнителей подразумеваются соответствующими их квалификации.

В технологической карте, конечно, не сказано, что должен делать фрезеровщик, если выйдет из строя фреза, если возникнет пожар, землетрясение, нападение противника. Если бы технолог был вынужден планировать поведение исполнителя в любых ситуациях, то он никогда не закончил бы работу над такими "технологическими картами".

В сущности, специализация в человеческой деятельности основана на способности выделить небольшой класс ситуаций, считающихся существенными для этого вида деятельности, а от всех остальных абстрагироваться, считать необычными, исключительными, требующими переключения в другой режим, в другую сферу деятельности. Примерам нет числа. Кулинарный рецепт не описывает, естественно, поведения хозяйки, когда в процессе приготовления блюда зазвонит телефон, физические модели (законы) применимы при определенных ограничениях и ничего не говорят о поведении соответствующих объектов при нарушении ограничений (скажем, законы Ньютона о поведении объектов при релятивистских скоростях).

Программист, вынужденный создавать программу для "тупого" исполнителя, по необходимости попадает в положение заводского технолога, которого заставляют писать инструкции по гражданской обороне или поведению во время пожара. Ведь надежная программа должна вести себя разумно в любых ситуациях.

Важно понимать, что без априорных правил поведения исполнителя не обойтись. Ведь если программист захочет все предусмотреть, он должен планировать проверку соответствующих условий. Но исключительные ситуации могут возникнуть в процессе исполнения программы проверки! Так что чисто программным путем задачу не решить.

Нужно предусмотреть общие правила поведения исполнителя при исключительных ситуациях (дающие возможность и здесь разумно сочетать абстракцию с конкретизацией). А именно, с одной стороны, концентрироваться на специфике планируемого процесса, абстрагируясь от редких, необычных или относящихся к другой сфере ситуаций; с другой стороны, при необходимости управлять поведением исполнителя в таких ситуациях, конкретизируя его поведение по сравнению с общими правилами с учетом реальной ситуации.

В жизни примерами "общих правил" служат обязанность доложить начальству, вызвать пожарную команду по телефону 01, милицию - по 02, скорую помощь - по 03, а также инструкции по оказанию неотложной помощи пострадавшим, мобилизационное предписание и т.п..

Пример конкретизации - подробный план эвакуации людей из помещения при пожаре, план мобилизационных и эвакуационных мероприятий на случай войны, перечень занятий дежурного при отсутствии телефонных звонков (для него обычная работа - отвечать на телефонные звонки, а их отсутствие - "необычная ситуация"; такой перечень "планирует его поведение" в необычной ситуации).

4.21.2. Языковые конструкты

Вернемся к ЯП и рассмотрим аппарат, обслуживающий описанную технологическую потребность (управлять поведением исполнителя в необычных ситуациях). Аналогом житейской необычной ситуации служит исключение (исключительная ситуация, exception). Концепция исключения в ЯП содержательно имеет много общего с концепцией аппаратного внутреннего прерывания, однако могут быть и существенные отличия. Ближе всего к понятию прерывания трактовка исключений в языке ПЛ/1. Об исключениях в Аде коротко и ясно написанно в учебнике Вегнера.

4.21.2.1. Исключения

Рассмотрим концепцию исключения, ориентируясь на Аду, стараясь больше уделять внимания "авторской позиции", т.е. объяснять, почему при проектировании языка были приняты излагаемые решения.

Все потенциальные исключения в Аде имеют индивидуальные имена и известны статически. Они либо предопределены, либо объявлены программистом.

Предопределенные исключения касаются, естественно, самых общих ситуаций. Например, при нарушении ограничений, связанных с типом (ограничений допустимого диапазона значений, диапазона индексов и т.п.) возникает исключение нарушение_ограничения (constraint_error); при ошибках в числовых расчетах (переполнение, деление на нуль, исчезновение и т.п.) - исключение численная_ошибка (numeric_error); при неправильной компоновке программы (отсутствие тела нужного программного сегмента и т.п.) - исключение нет_сегмента (program_error); при нехватке памяти для размещения динамических объектов - исключение нет_памяти; при нарушении во взаимодействии асинхронных процессов (аварийное или нормальное завершение процесса, содержащего вызываемый вход и т.п.) - исключение ошибка_взаимодействия (tasking_error).

Если, скажем, объявить

```
A:array (1 .. 10) of INTEGER ;
```

то при I = 11 или I = 0 в момент вычисления выражения A(I) возникает предопределенная исключительная ситуация нарушение_ограничения.

Определяемые исключения явно вводятся программистом посредством объявления исключения. Например, объявление

```
объект_пуст, ошибка_в_данных : exception ;
```

вводит два исключения (исключительные ситуации). Возникают такие исключения в момент, явно указываемый программистом посредством оператора исключения (raise). Например, результатом исполнения оператора

```
raise ошибка_в_данных;
```

служит возникновение исключительной ситуации ошибка_в_данных.

Факт возникновения исключительной ситуации переводит исполнителя в новый режим, режим обработки исключения. В этом режиме и действуют упоминавшиеся "априорные правила поведения исполнителя". Рассмотрим их.

4.21.2.2. Обработка исключений

В режиме обработки исключения существенно используется динамическая структура программы. Другими словами, конкретные действия исполнителя зависят от динамической цепочки вызовов, ведущей к программному конструкту, в котором возникло исключение. Дело в том, что с каждым исключением может быть связана серия так называемых реакций на исключение, расположенных в различных программных конструктах. В современных ЯП принят принцип динамического выбора реакции - всегда выбирается реакция на возникшее исключение, динамически ближайшая к месту "происшествия".

Например, рассмотрим программу вида

```
procedure P is
       ошибка : exception ;
```

```
      procedure R ; - - предварительное определение процедуры R
      procedure Q is
      begin
        R ; - - вызов процедуры R ;
        . . .
        . . . - - (1)
      exception
        . . .
        when ошибка => PUT("ОШИБКА в Q") ; - - реакция на
                                           - - исключение "ошибка"
        . . .
      end Q ;
      procedure R is
      begin
        . . . - - (2)
      end R ;
    begin
        . . . - - (3)
        Q ;   - - вызов процедуры Q
        . . .
    exception
        . . .
      when ошибка => PUT("ОШИБКА в P") ; - - еще одна реакция
                                         - - на то же исключение
    end P ;
```

Если исключение "ошибка" возникает на месте (3), то сработает реакция на это исключение в теле P и будет напечатано "ошибка в P". Если это же исключение возникнет при вызове процедуры Q, то будет напечатано "ошибка в Q".

Пока подбор реакции как по динамической цепочке вызовов, так и по статической вложенности конструктов дал одинаковые результаты.

А вот когда исключение "ошибка" возникает на месте 2 (при вызове процедуры R в теле процедуры Q), отличие принципа динамического выбора от статического проявляется наглядно. Статический выбрал бы реакцию в теле P, а динамический выберет реакцию в теле Q. Будет напечатано "ошибка в Q", что существенно точнее отражает суть случившегося. Именно для того, чтобы можно было точнее, конкретнее реагировать на исключение, и принят (практически во всех языках) принцип динамического выбора реакции.

Обратите внимание, объявляются исключения статически, подобно переменным и процедурам, а реакции на них выбираются динамически из статически определенного множества возможных реакций.

[Если разрешить вводить новые имена исключений динамически, то следовало бы создавать динамически и реакции на них , т.е. динамически создавать программу. Такого рода возможности противоречат концепции статического контроля и в современных языках индустриального программирования практически не встречаются].

Итак, при обработке исключений динамическая структура программы нужна для точного и гибкого управления поведением исполнителя (гибкой конкретизации его поведения).

[Вместе с тем в современных языках поведение исполнителя в режиме обработки исключений довольно жестко регламентировано. Нет прямых аналогов, скажем, такой житейской возможности, как сообщить начальству и ждать распоряжений "на месте происшествия". Или позвонить сразу и в милицию, и в скорую помощь, и в пожарную охрану, одновременно принимая собственные меры. Конечно, всякое поведение можно моделировать, но, скажем, несколько исключений в одном месте возникнуть не могут].

Принятая в языке стратегия ("правила поведения") обработки исключений прямо связана со взглядом на сущность исключений. Этот взгляд, в свою очередь, зависит от важнейших требований, определивших авторскую позицию при создании языка. Хотя в конечном итоге различия поведения могут показаться не такими уж значительными, рассмотреть их обоснование и поучительно, и интересно. Выберем для определенности два языка, ПЛ/1 и Аду. Скажем заранее, что различия касаются лишь продолжения работы после обработки исключения.

4.21.2.3. Принцип ремонта на месте

Сначала о ПЛ/1. Несколько упрощая, можно сказать, что один из основных принципов конструирования языка ПЛ/1 - "предпочитать такие истолкования

конструкций, которые позволяют оправдать их дальнейшее исполнение". В соответствии с этим принципом введены многочисленные правила подразумеваемых преобразований данных, допустимы различного рода сокращения и т.п.

В согласии с этим принципом исключение трактуется как относительно редкое, но в целом естественное для выполняемого конструкта событие. При его обработке следует направить усилия на скорейшее возобновление прерванного процесса. Эти усилия можно наглядно представить себе как "ремонт" процесса в том месте, где возникло исключение. Естественная модель поведения - прервать исполняемый процесс, вызвать "ремонтную бригаду" на место происшествия, после окончания ремонта продолжить прерванный процесс. Обратите внимание на три составных поведения - прервать, вызвать (это значит, понять, кого вызвать - ремонтная бригада подбирается динамически) и продолжить.

Подобный взгляд полезен, скажем, при нехватке памяти - нужно вызвать подпрограмму сборки мусора или подпрограмму динамического выделения памяти, а затем попытаться продолжить работу.

Описанное отношение к сущности исключения можно назвать принципом ремонта на месте.

Однако, где гарантии, что "отремонтированный" процесс сможет нормально работать? Если, скажем, исключение связано с окончанием файла или нарушением ограничений, то довольно бессмысленно продолжать работу прерванного процесса. В ПЛ/1 в таких случаях в реакции на исключение (после ремонта) применяют передачу управления туда, откуда признано разумным продолжать работу.

По многим причинам это далеко не лучшее решение. Одна из основных в том, что динамическая структура программы оказывается слабо связанной с ее статической структурой. Чтобы разобраться в программе, приходится "прокручивать" каждый оператор. Короче говоря, решение с передачей управления в общем случае затрудняет чтение и отладку программ. По "неструктурированности" это решение можно сравнить с выходом из подпрограммы не по возврату, а по передаче управления. Что при этом происходит с динамической цепочкой вызовов? Остается только гадать или определять, руководствуясь "тонкими" правилами!

Итак, будем считать обоснованным, что решение с передачей управления противоречит концепции "структурного программирования". А без передачи управления принцип ремонта не позволяет адекватно обрабатывать многие виды исключений.

4.21.2.4. Принцип динамической ловушки

Теперь об Аде. Одно из ключевых требований к Аде - способствовать надежному программированию. Другими словами, следует стремиться к минимуму отказов из-за ошибок в программе и в данных. Если же отказ неизбежен, то следует обеспечить по меньшей мере осмысленную диагностику.

Требование надежности оправдывает трактовку исключения как свидетельства непригодности аварийного "процесса" (того процесса, где возникло исключение) к нормальной работе в создавшихся условиях. Стремясь к минимуму отказов, следует не "ремонтировать" аварийный процесс, а нацелить обработку исключения на локализацию последствий "аварии", на создание возможности продолжать работу тех (связанных с аварийным) процессов, которых авария пока не коснулась. Саму обработку исключения в "аварийном" процессе обычно разумно рассматривать скорее не как "ремонт", а как попытку сохранить как можно больше сведений для анализа ситуации на уровне иерархии, принимающем решения о последующих действиях.

Именно такой принцип действует в языке Ада (ведь надежность - одна из основных целей этого языка), а также в языке Эль-76 на мащинах серии Эльбрус.

При такой цели естественная стратегия - последовательно признавать аварийными вложенные процессы (начиная с самого внутреннего) до тех пор, пока среди них не найдется процесс, в котором приготовлена реакция на возникшее исключение. При этом аварийные процессы, в которых нет нужной реакции, последовательно завершаются аварийным способом. Найденная в конечном итоге реакция на исключение призвана обеспечить нормальное продолжение работы уцелевших процессов (и, возможно, выдачу сообщения об ошибке).

Итак, никакого возврата к аварийному процессу при такой стратегии нет, а значит, нет и опасности вызвать "лавину" исключений и сообщений об авариях. Если ведущий процесс сочтет возможным, он может снова запустить (в новых условиях) и бывший аварийный процесс. Но решение об этом не встроено в семантику языка, а программируется на уровне иерархии, высшем по отношению к аварийному процессу.

Назовем описанный принцип обработки исключений принципом динамической ловушки. Название связано с тем, что возникшее исключение распространяется по

динамически объемлющим процессам до тех пор, пока не попадет в ловушку, заготовленную в одном из процессов (т.е. не достигает реакции на исключение).

Именно этот принцип поведения исполнителя в исключительных ситуациях воплощен в Аде (как целиком отвечающий требованиям к языку и, в частности, способствующий надежному и структурному программированию).

Осталось лишь объяснить, как именно устроена ловушка (реакция на исключение), а также рассказать о некоторых дополнительных возможностях языка.

4.21.2.5. Ловушка (реакция на исключение)

Как было видно из примера с процедурами P, Q и R, в общем случае тела подпрограмм, тела пакетов, тела задач, а также блоки содержат в конце обычной последовательности операторов еще часть, определяемую ключевым словом exception. Это и есть ловушка исключений. Она устроена аналогично оператору выбора, но вместо значений перечисляемого типа после ключевого слова when фигурируют имена исключений.

Например,

```
  begin
    ... - - последовательность операторов
  exception -- ловушка исключенийt
    when плохо_обусловленная | численная ошибка =>
         PUT("матрица плохо обусловлена");
    when others =>
         PUT("фатальная ошибка");
         raise ошибка;
  end ;
```

Альтернатива others, как обычно, выбирается в том случае, когда не выбраны остальные. В нашем примере при возникновении исключения плохо_обусловленная или численная_ошибка (первое - объявляемое, второе - предопределенное) печатается матрица "плохо обусловленная матрица" и обработка исключения завершается (затем продолжает нормально работать динамический объемлющий процесс). Любые другие исключения будут пойманы альтернативой others, будет напечатано "фатальная ошибка" и возникнет новое исключение "ошибка" как результат работы оператора исключения (raise).

Это исключение будет распространяться в динамически объемлющих процессах, пока не попадет в ловушку (для предопределенных исключений ловушки предусмотрены в предопределенном пакете "система"). Если бы второй альтернативы не было, то любое исключение, отличное от двух указанных в первой альтернативе нашей ловушки, распространялось бы по динамически объемлющим процессам до "своей" ловушки.

4.21.3. Дополнительные особенности обработки исключений

Для исключений также действует принцип разделения спецификации и реализации. Объявление исключения можно считать его "спецификацией", а ловушку - его "телом". Так что ловушка также считается объявлением.

Вопрос. Что естественно считать "использованием" исключения?

Когда исключение возникает в объявлениях некоторого конструкта, то оно немедленно распространяется на динамически объемлющие конструкты. Собственная ловушка конструкта рассчитана только на исключения, возникшие среди операторов конструкта. Это правильно, так как в ловушке могут использоваться объекты, которые из-за "недообработки" объявлений (ведь в них возникла авария) окажутся несуществующими.

А приведенный выше пример показывает, что есть и еше одна причина - можно попасть в бесконечный цикл, если при возникновении исключения "ошибка" искать реакцию в той же ловушке.

Интересно рассмотреть особенности распространения исключений на взаимодействующие задачи. У тела задачи нет динамически объемлющего процесса - соответствующий телу задачи процесс работает асинхронно, с ним могут быть динамически связаны посредством рандеву много удругих "равноправных" процессов. С другой стороны, каждый асинхронный процесс запускается при обработке некоторого фиксированного объявления в определенном конструкте.

Именно поэтому, когда исключение возникает среди объявлений задачи, то задача аварийно завершается и на месте запустившего ее объявления (задачи) возникает предопределенное исключение ошибка_взаиммодействия.

Когда же исключение возникает в теле задачи и в своем распространении доходит до самой внешней в этом теле ловушки, то задача завершается аварийно (вне

зависимости от того, "поймано" ли исключение), а в любом другом асинхронном процессе при попытке исполнить оператор вызова входа аврийно завершенной задачи в этом операторе возникает предопределенное исключение ошибка_взаимодействия. Так что при аварии в теле задачи исключение (возможно, не то же, а другое) распространяется "во все стороны".

Подавление исключений (оптимизирующие указания). Для полноты представления об обработке исключений осталось добавить, что в случае, когда некоторые проверки ограничений, потенциально приводящие к возникновению исключений, считаются дорогими (неприемлемо ресурсоемкими), их можно отменить с помощью так называемых (оптимизирующих) указаний вида

pragma подавить (проверка_индексов, на => таблица);

Если реализация способна реагировать на такие указания (они не обязательны для исполнения), то в нашем случае проверка индексов будет отменена для всех объектов типа "таблица"). Конечно, подобными указаниями нужно пользоваться очень осторожно. В Аде есть и другие оптимизирующие указания. Все они не обязательны.

Итак, аппарат исключений позволяет отделить программирование содержательной функции программного сегмента от программирования его взаимодействия с другими сегментами в чрезвычайных (аварийных) обстоятельствах.

Программируя содержательную функцию, можно абстрагироваться от необычных ситуаций, а программируя взаимодействие в необычных ситуациях, в значительной степени абстрагироваться от содержательной функции сегмента. Для особо важных необычных ситуаций можно заранее заготовить названия и ловушки в библиотечных модулях. Таким образом, концепция исключения - одна из компонент общего аппарата абстракции-конкретизации.

Ее можно было бы довести и до уровня модульности, физически отделив ловушки от тел сегментов. Повидимому, такая возможность появится в ЯП будущего.

На этом закончим знакомство с концепцией исключения.

4.22. Библиотека

До сих пор мы избегали подробного описания языковых свойств, ограничиваясь сведениями, достаточными для демонстрации рассматриваемых концепций и принципов. Однако в ближайшем будущем мы намерены существенно затронуть авторскую позицию, для которой, конечно, важны все тонкости языка (иначе они бы в нем не появились). Более того, мы намерены изложить принципы, в определенном смысле управляющие сложностью создаваемого языка. Для их понимания необходимо, чтобы читатель был в состоянии в деталях сопоставить решения, принятые авторами различных языков. Поэтому в ближайших разделах, завершая знакомство с основными языковыми абстракциями, мы достаточно подробно остановимся на избранных аспектах Ады, а именно на раздельной компиляции, управлении видимостью идентификаторов и обмене с внешней средой. Изложение не претендует на полноту описания языковых свойств. Основная цель упоминания подробностей - продемонстрировать сложность языка и возникающие в этой связи проблемы. Заинтересованного читателя отсылаем к руководствам по Аде [17-19]. В этом разделе использованы материалы, предоставленные автору С.И.Рыбиным.

Итак, мы рассмотрели виды связывания раздельно транслируемых модулей в Аде. Рассмотрим ту же самую проблему немного с другой стороны - обсудим устройство программной (трансляционной) библиотеки. Заметим, что на устройство, способ формирования и модификации библиотеки Ады накладывается ряд ограничений и требований (что естественно), причем эти ограничения зафиксированны в определении языка (вот это особенно важно) и транслятор обязан следить за их выполнением.

[Ада-библиотеку полезно рассмотреть хотя бы потому, что Ада - первый ЯП, в котором вопросы раздельной компиляции и устройство библиотеки тщательно проработаны и зафиксированы в определении языка (в виде требований к реализации (исполнителю)!). В этом отношении полезно сравнить Аду с Фортраном.]

4.22.1. Состав программной (трансляционной) библиотеки.

4.22.2 Компилируемый модуль

Компилятор получает "на вход" компилируемый модуль, который состоит из (возможно пустой) спецификации контекста и собственно текста модуля.

Спецификация контекста содержит указатель контекста (with) и указатель сокращений (use). Займемся первым.

Уже было сказано, что with - средство явного указания односторонней связи : в использующем модуле перечисляютт имена необходимых ему библиотечных модулей.

Таким образом, любое имя, используемое любым способом в данном модуле, должно быть

либо объявлено в самом этом модуле, или в связанных с ним (при помощи двусторонней связи!) библиотечных или родительских модулях,

либо объявлено в пакете STANDARD,

либо предопределено,

либо явно перечислено в указателе контекста (with).

Итак, "пространство имен" модуля ограничено и явно описано.

Упражнение. Сравните с EXTERNAL в Фортране. В чем отличия?

[Использование в модуле имени другого библиотечного модуля, присутствующего в библиотеке, но не названного в указателе контекста первого - нарушение правил языка.

Перечисление имен используемых библиотечных модулей в указателе контекста не является необходимым с точки зрения идентификации используемых в модуле имен. (Пока мы вынуждены ограничиться интуитивным пониманием того, что такое имя, и что такое идентификация имени.) Так, например, если имя не удается идентифицировать никаким другим образом, можно просто перебирать имена всех библиотечных модулей и искать его среди этих имен (тем более, что имена всех библиотечных модулей различны). Однако указатель контекста необходим не только для описания внешних связей модуля - фактически он описывает зависимость модулей, определяющую порядок их компиляции (а также порядок создания и изменения программной библиотеки)].

В указателе контекста необходимо перечислять лишь непосредственно используемые имена библиотечных модулей, т.е. те имена, которые явным образом присутствуют в тексте модуля. Так, например, если (библиотечная) процедура P использует (библиотечную) процедуру Q, а та, в свою очередь - библиотечный пакет R, то соответствующие компилируемые модули должны иметь вид

```
with R;                           with Q;
     procedure Q is                    --  with R писать не надо!
     ...      ...                      procedure P is
     begin                             ...      ...
        ...                            begin
        R.P1;  --  вызов процедуры,       ...
               --  описанной в R;         Q;  --  вызов Q;
        ...                               ...
     end Q;                            end R;
```

Таким образом, в указателе контекста каждого модуля должы быть перечислены лишь имена его "непосредственных контрагентов". (Правила языка не запрещают указать имя "лишнего" модуля, однако, как мы далее увидим, это не просто бессмысленно, но и опасно).

[Заметим, что с содержательной точки зрения косвенная зависимость модулей друг от друга (через третий модуль) может быть сколь угодно сложной. Ведь в видимой части (библиотечных) пакетов можно объявлять не только типы и их операции, но и объекты (переменные), "разделяемые" несколькими процедурами. Рассмотрим простой пример.

```
package P is
        CONTROL_I:INTEGER;
        CONTROL_K:INTEGER;
     end P;
with P; use P;
     procedure Q1 is
        ...   --  определение функций F1,...,F10
        ...
     begin
        ...
        case CONTROL_I is
           when 1 =>
              CONTROL_J:=F1(CONTROL_I);
            ...   ...
           when 10 =>
```

```
                CONTROL_J:=F10(CONTROL_I);
             when others => null;
          end case
       end Q1;
  with P; use P;
       procedure Q2 is
          ...   --  определение функций G1,...,G10;
     ...
       begin
          ...
          case CONTROL_J is
             when 1 =>
                CONTROL_I:=G1(CONTROL_J);
               ...   ...
             when 10 =>
                CONTROL_I:=G10(CONTROL_J);
             when others => null;
          end case ;
          ...
       end Q2;
```

Процедуры Q1 и Q2 формально независимы (ни одна из них не упоминает другую в своей спецификации контекста), однако их содержательная зависимость через пакет P очевидна.

Окажется ли эта особенность Ады хорошей или плохой с точки зрения "надежного" языка индустриального программирования, покажет опыт применения].

4.22.3. Порядок компиляции и перекомпиляции (создания и модификации программной библиотеки)

Очевидно, что этот порядок не может быть абсолютно произвольным. Нам уже известно все, чтобы сформулировать требования к нему. Выделим две группы требований, обусловленные двусторонним и односторонним связыванием соответственно. Напомним, что в исходном состоянии библиотека содержит лишь предопределенные библиотечные модули.

4.22.3.1. Двусторонние связи

1). Тело следует компилировать после спецификации.

Следствия: После перекомпиляции спецификации необходимо перекомпилировать тело. Перекомпиляция тела не требует перекомпиляции спецификации.

2). Вторичный модуль следует компилировать позже соответствующего родительского модуля.

Следствие. Перекомпиляция родительского модуля влечет перекомпиляцию всех его вторичных модулей.

4.22.3.2. Односторонние связи

Использующий модуль следует компилировать позже используемого (т.е. модуль можно компилировать только после компиляции (спецификаций, не тел!) модулей, перечисленных в его указателе контекста.

Следствия:

1). После перекомпиляции библиотечного модуля (спецификации, а не тела) необходимо перекомпилировать все использующие модули (упоминающие данный в указателе контекста).

2). Перечислять "лишние" модули в указателе контекста действительно вредно!

Вопрос. А как в ФОРТРАНе? (компиляция модулей независимая).

[Реализации дано право квалифицированно "разбираться в ситуации" и выявлять (с целью оптимизации) те перекомпиляции, которые фактически не обязательны.

Вопрос. За счет чего?

4.22.4. Резюме : логическая и физическая структура программы

Подводя итог, скажем, что в Аде естественно различать физическую и логическую структуры программы. Эти понятия тесно связаны, но не эквивалентны. Логическая структура - это абстракция физической структуры, а именно абстракция от конкретного способа разбиения на (физические) модули.

Во всех случаях, когда важен смысл, а не особенности жизненного цикла программы (создание, компиляция, хранение, модификация), программа нас интересует

с точки зрения именно своей логической структуры. Однако в остальных случаях приходится учитывать (и использовать!) ее физическую структуру.

Физическая структура программы образуется совокупностью (компилируемых) модулей, всегда отдельных (отделенных) друг от друга, и "подаваемых" компилятору в определенном порядке.

Логическую структуру готовой (завершенной) программы образуют сегменты. (Иногда их называют "программные модули").

Здесь уже совершенно не важен порядок компиляции.

Более того, при переходе от физической к логической структуре меняется само понятие компоненты программы. Так, с точки зрения физической структуры спецификация и тело (библиотечного) пакета - это разные модули, а в логической структуре - это единый сегмент - пакет, в котором при необходимости выделяется, скажем, видимая часть, не совпадающая в общем случае со спецификацией. (Из-за чего?).

Другой пример. Можно считать, что на уровне логической структуры фактически пропадает разница между первичными и вторичными модулями, а также становится ненужным само понятие библиотеки.

[Мы говорим о логической структуре с точки зрения разработчика комплекса программных услуг; с точки зрения пользователя логическая структура определяется только набором первичных модулей (спецификаций)].

4.22.5. Средства связывания модулей

Каковы же средства создания логической структуры программы над ее физической структурой? Часть из них мы рассмотрели - это способы одностороннего и двустороннего связывания : правила соответствия между спецификацией и телом библиотечного модуля, заглушки и (полные!) имена родительских модулей в заголовках вторичных модулей, указатель контекста, особая роль пакета STANDARD.

Другую группу средств связывания образуют правила видимости идентификаторов и правила идентификации имен. Про них - в следующем разделе.

Про раздельную компиляцию и библиотеку - все.

4.23. Именование и видимость (на примере Ады)

В предыдущем разделе в качестве средств связывания модулей были упомянуты идентификация и видимость имен. Это два важных аспекта общей проблемы именования в ЯП, которую мы и рассмотрим на примере Ады, выделяя по возможности общие принципы и концепции.

4.23.1. Имя как специфический знак

Начнем с основных терминов : имя и идентификация имени.

Проблема именования уже затрагивалась в курсе; кроме того, у читателя предполагается некоторый опыт работы с ЯП (следовательно, и общее представление об именах в ЯП). Подчеркнем, что практически в любом языке имя и идентификатор не одно и то же. Так, не является идентификатором имя компоненты массива "A(I)", имя компоненты записи "B.C". Если же поставить цель дать общее, достаточно точное определение имени, то возникнут неожиданные трудности (попробуйте это сделать сами).

Подход с позиций "бесконтекстного" КС-синтаксиса (БНФ-формул) устроить не может - строение и смысл имени зависит от контекста.

Действительно, во многих ЯП имя элемента массива локально неотличимо от вызова функции. В Аде подобных неоднозначностей довольно много. (Например, компонента записи неотличима от полного имени вложенной в пакет процедуры и т.п.).

Поэтому попробуем неформально определить, что такое имя, исходя из роли имен в программах. Можно сказать, что имя (именование) - это средство (способ) называть (обозначать) некоторые сущности (элементы) в программе. Если посмотреть на язык как на знаковую систему, то мы увидим, что (достаточно правильная) программа на языке представляет из себя иерархию знаков. В этой иерархии выделяются классы знаков (очень близкое (но не тождественное!) понятие - синтаксические классы, метасимволы, метапеременные в формальной грамматике). Для некоторых классов знаков денотат определяется в значительной степени по структуре знака (блок, цикл, объявление процедуры).

Имя как специфический знак характеризуется тем, что его денотат не может быть "вычислен" по "внешнему виду" знака без привлечения контекста (по знаку "A", невозможно сказать не только то, что конкретно он обозначает, но и даже

"приблизительно" определить класс обозначаемой сущности (процедура, переменная, тип , пакет и т.п.)).

Итак, понятие имени в некоторой степени уточнено. Идентификацией имени называют процесс/правила/алгоритм/результат определения той сущности, которую имя обозначает. С точки зрения знаковой системы идентификация - это процесс/результат определения денотата соответствующего знака.

4.23.2. Имя и идентификатор

В таких языках, как Алгол-60 и Фортран, эти понятия очень близки. В Аде, как видно из примеров, имя существенно сложнее. Однако, как можно заметить из примеров программ на Аде, любое имя "в значительной степени" "состоит" из идентификаторов. Идентификаторы могут входить в него "на разных уровнях" (сравните A.B.C.D и A(B(C(D)))). Идентификатор - это атомарное (неделимое) имя. Только идентификаторы можно непосредственно объявлять в программе.

4.23.3. Проблема видимости

Идентификатор в Ада-программе может появиться либо в "положении определения", либо в "положении использования". Чтобы различать разные в этом смысле вхождения одного и того же идентификатора, в Адовском лексиконе введено понятие простого имени (для обозначения идентификатора в положении использования). Для идентификаторов (простых имен) существует проблема установления соответствия между определяющими и использующими вхождениями. Следуя Адовской терминологии, будем называть ее проблемой видимости идентификаторов. Проблема идентификации имен включает проблему видимости идентификаторов, но не сводится к ней. Заметим, что следует различать статическую и динамическую идентификацию. Так, если объявлено :

```
A: array (1..10) of INTEGER ;
I: INTEGER ;
```

то со статической точки зрения A(I) - элемент массива A, а динамическая идентификация при I=3 даст A(3) (т.е. 3-ий элемент), а при I=11 - CONSTRAINT ERROR.

Мы уже говорили, что именование - это способ построения логической структуры программы над ее физической структурой. Действительно, если в компилируемом модуле идентифицировать все имена, то можно рассматривать его как составную часть теперь уже логической структуры программы, поскольку он окажется связанным не только с некоторыми библиотечными модулями (способы этой связи уже обсуждались), но и со всеми необходимыми ему понятиями и элементами программы (программного комплекса) как единого целого.

Таким образом, выполняя идентификацию, приходится работать в значительной степени в роли компилятора, который восстанавливает логическую структуру программы по компилируемому модулю и программной библиотеке. С другой стороны, программист, создавая компилируемый модуль, обязан постоянно помнить об идентификации, чтобы достичь своей конечной цели - описать логическую структуру программы.

Предварительные замечания окончены.

4.23.4. Общая структура проблемы именования

Выделим относительно независимые аспекты проблемы именования.

1). Разновидности объявлений (как и что можно объявлять).
2). Строение имен.
3). Строение "пространства имен".
4). Правила видимости идентифмкаторов.
5). Схема идентификации имен.

Всюду ниже в этом пункте будем игнорировать физическую структуру программы (ее разбиение на модули). Учитывать будем лишь ее логическую структуру (разбиение на сегменты). Таким образом, исключаем из рассмотрения имена модулей и их связывание.

При изучении основных конструктов ЯП обсуждение строилось по следующей схеме : выделялась технологическая потребность - обсуждалось средство ее реализации. Применим этот же подход к проблеме идентификации.

4.23.5. Основная потребность

Основная "внешняя" технологическая потребность очевидна - точно называть необходимые компоненты программы. Однако поскольку эти компоненты разнородны и обслуживают весьма разнообразные потребности, то существует сложная и многогранная

"внутренняя" технологическая потребность : именование должно быть хорошо согласовано со всеми средствами ЯП.

Более того, именование - способ связи различных языковых средств (способ объединения их в единую программу).

[Создавая абстракцию, мы даем ей имя, а затем ссылаемся на нее при помощи этого имени].

Наконец, именование должно отвечать общим целям языка (надежность, читаемость, эффективность и т.п.).

Таким образом, основные конструкты языка (а также заложенные в них концепции) и способ именования взаимозависимы.

[Сложные и многообразные конструкты ведут к сложному именованию, и наоборот, относительно простые способы именования требуют относительной простоты конструктов. Искусство автора ЯП проявляется в умении найти разумный компромисс между собственной сложностью ЯП и сложностью его использования для сложных задач (Фортран или Бейсик относительно просты, но сложные задачи на них программировать сложнее, чем на Аде)].

При создании Ады приоритет имела задача включения в него богатого набора средств (конструктов), позволяющих адекватно реализовывать большой набор технологических потребностей, причем для многих технологических потребностей заготавливалось специальное средство. В результате схема именования получилась довольно сложной. Это признают сами авторы языка. Значительная часть критических замечаний в адрес Ады также касается идентификации имен. В этой связи см. ниже принцип сундука.

4.23.6. Конструкты и требования, связанные с именованием

Выделим общие требования к языку и те конструкты Ады, которые оказали существенное влияние на способ именования.

4.23.6.1. Глубокая структуризация данных и операций

а) Составные типы данных.

Чтобы понять, какие в связи с этим возникают требования к именам, достаточно представить себе многоуровневую структуру из массивов и записей. Каждую ее компоненту надо уметь называть, вот и появились имена вида

A(I,J,K).B(L,M).C.D(E,F).K.all.E

(постройте пример соответствующей структуры).

б) Пакет как логически связанная совокупность ресурсов.

Мало объявить и реализовать ресурс, надо дать возможность им пользоваться. Как минимум, надо дать возможность его называть. Чтобы "развязать" именование ресурсов в непересекающихся пакетах (зачем это нужно?), приходится вводить составные (полные) имена. Другими словами, сначала называется имя пакета, а затем (через точку) имя ресурса в пакете.

в) Задача как автономное описание класса асинхронных процессов. Проблема именования входов задач решается аналогично пакетам : вызывая вход, надо назвать сначала задачу, где он объявлен, а уж затем сам вход.

4.23.6.2. Раздельная компиляция

Раздельно компилируемые модули должны иметь доступ к именам друг друга. Иначе создавать программу невозможно.

4.23.6.3. Относительная независимость именования внутри сегментов

Программисты, работающие над одной программой, не должны быть жестко связаны дисциплиной выбора имен (на нижних уровнях иерархии программы). Следствие - "блочная" структура областей локализации имен.

4.23.6.4. Необходимость переименования и сокращения длинных имен

4.23.6.5. Критичность проблемы полиморфизма

Следствие : потребовалось заменить "алгольный" запрет объявлять одинаковые идентификаторы в одном блоке более гибким ограничением, позволяющим объявлять одноименные операции.

4.23.6.6. Принцип обязательности объявлений для всех имен (кроме предопределенных)

4.23.6.7. Необходимость производных типов

Следствие : неявные объявления операций этих типов..

Пример.

```
package P is
       type T is (A,B);
       procedure Q(X : in T, Y : out INTEGER );
     end P;
```

```
...
type NEW_T is new T;
...
```

Тип NEW_T должен обладать свойствами, аналогичными всем свойствам типа T. В частности, иметь два перечисляемых литерала A и B (теперь уже типа NEW_T) и операцию-процедуру P с параметрами

(X : in NEW_T, Y : out INTEGER).

Чтобы не заставлять программистов переписывать соответствующие объявления (чем это плохо?) и вместе с тем соблюсти принцип обязательности объявлений, авторы Ады были вынуждены ввести так называемые неявные объявления. В частности, указанные выше литералы и процедура считаются объявленными неявно.

Вопрос. Зачем нужен принцип обязательности объявлений?

Подсказка. Для прогнозирования-контроля и, следовательно, повышения надежности.

4.23.6.8. Требование определенного "комфорта" при написании программ иногда оборачивается неприятными сюрпризами. Два примера:

а) Много локальных неоднозначностей. Скажем "A(I)" может обозначать

- элемент массива
- вызов функции
- вырезку массива (n-1-мерный подмассив n-мерного массива A);

б) Сложности и неожиданные ошибки из-за неявных инициализирующих выражений у (входных) параметров функции и процедур. Вот пример одной из них, взятый из журнала Ada LETTERS :

```
procedure test is
        type Enum is (Red, Green);
        type Vec is array (Enum) of Enum;
        X : Enum;
        Y : Vec;
        function F (A : Enum := Red) return Vec is
        begin
           return Y;
        end ;
     begin
        X := F (Red);
-- Что в последней строчке? Вызов функции с параметром RED
     -- или элемент массива, вычисленного вызовом функции без
     -- параметров (ведь инициализированные параметры
     -- можно опускать).
     -- [Надо бы F (  )(Red), как в Фортране-77 ].
   Y := F (Red);  -- здесь тоже неясно
           -- следует учесть, что правилами перекрытия пользоваться
           -- некорректно - функция одна и перекрытия нет
end ;
```

Замечание. Конечно, так программировать нельзя независимо от свойств ЯП. Программа не ребус. Ее нужно читать, а не разгадывать!

Конец замечания.

Еще хуже :

```
procedure F is
   type ARR;
   type ACC is access ARR;
   type ARR is array (1..10) of ACC;
   X : ACC;
   function f (X : INTEGER := 0) return ACC is
   begin
      return new ARR;
   end ;
begin
   X := f(1); -- допустимы две различные интерпретации
end ;
```

Итак, требования, которые в наибольшей степени повлияли на схему идентификации в Аде, названы. Рассмотрим эту схему.

4.23.7. Схема идентификации

4.23.7.1. Виды объявлений в Аде

а) явные объявления - с ними "все ясно".

б) части явных объявлений, синтаксически не выделяемых в отдельные конструкты :

- компоненты записей (в том числе и дискриминанты типа)
- входы задач
- параметры процедур и родовые параметры
- перечисляемые литералы
- параметр цикла

в) неявные объявления :

- имя блока
- имя цикла
- метка оператора
- перечисляемые литералы
- унаследованные подпрограммы производных типов
- предопределенные операции типов различных категорий.

Перечисляемые литералы считаются неявно объявленными функциями без параметров.

4.23.7.1.1. Зачем нужны неявные объяления

Неявные объявления оказываются в Аде "вынужденно-необходимыми".

Как уже не раз отмечалось, одним из важнейших требований к Аде было требование надежности, составной частью которого является требование обнаруживать и диагностировать как можно больше нарушений во время компиляции (до начала выполнения программы), т.е. требование статического прогнозирования и статического контроля. Это включает и контроль использования имен (идентификаторов). Чтобы контролировать использование, необходимо прогнозирование, т.е. тот или иной способ объявления.

Явно объявлять метки (как в Паскале) все-таки обременительно. С другой стороны метки могут конфликтовать с другими именами; чтобы контролировать такие коллизии с учетом областей локализации, удобно считать метки объявленными "рядом" с остальными (явно объявленными) именами рассматриваемой области локализации.

Основные неприятности возникают из-за неявных объявлений унаследованных подпрограмм при объявлении производных типов.

По замыслу производного типа в момент своего объявления он должен сохранить все те свойства (кроме имени), которые заданы в определяющем пакете родительского типа. Следовательно, вместе с новым типом должны считаться объявленными и подпрограммы, унаследованные от родительского типа. Но объявленными где? Ведь явных объявлений этих подпрограмм для нового типа нет, а от точного места объявлений зависит и смысл программы, и результаты контроля (приведите примеры такой зависимости).

В Аде эта проблема решается так : все унаследованные подпрограммы считаются неявно объявленными сразу вслед за объявлением производного типа. Эти неявные объявления "уравнены в правах" с явными объявлениями.

Далее, если родительский тип был перечисляемым, то производный должен иметь те же перечисляемые литералы; поэтому их тоже считают объявленными неявно (в качестве функций без параметров).

Вопрос. Причем эдесь функции, да еще без параметров?

Подсказка. Эти литералы могут перекрываться, поэтому их удобно всегда считать функциями для единообразия правил перекрытия.

Вопрос. А почему можно считать литералы функциями?

4.23.7.1.2. Другие особенности механизма объявлений

а) Механизм наследования в взаимодействии с указателем сокращений и перекрытием оказывается довольно сложным. Подробнее об этом будет сказано в связи с проблемами сокращений. Основная идея : наследуются только подпрограммы из определяющего пакета родительского типа, причем наследование возможно только вне пакета.

б) Формальная область действия (неявного) объявления метки может оказаться существенно шире, чем та часть текста, из которой на эту метку можно ссылаться. Например, запрещено передавать управление внутрь цикла, хотя формально внутренняя метка цикла может считаться (неявно) объявленной вне цикла, после явных объявлений внешнего "блока".

в) Наравне с идентификаторами объявляются строки - знаки опера

ций и символьные константы.

4.23.7.2. Устройство полных (составных) имен

Несколько упрощая суть дела, можно сказать, что основная проблема Ада-имен - потенциально неограниченная сложность составных имен (т.е. имен с разделителем ".").

Общая структура имени такова :

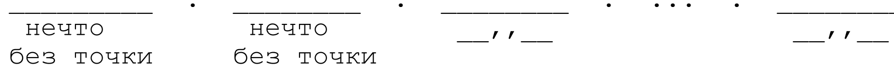


Рис. 4.8

Причем между точками стоит следующая структура :

идентификатор { нечто в скобках }

Идентификатор называет объект (определяет "суть дела"), а в скобках - либо индексы массива, либо параметры вызова функции. [Заметим, что в Аде возможен элемент массива вида a(i)(j)(k)(l)(m)(n)].

Рассмотрим три примера :

```
procedure P is                               -----------------
   type T is (A,B,C);                                         !
   type T1 is array (1..10) of T                              !
   type T2 is record                                          !
              A2 : T1;                                        !
              B2 : T1;                                        !
   end record                                              Массив-
   type T3 is array (1..10) of T2;                         записей
   type T4 is record                                       сложной
              A4 : T3;                                     структуры
              B4 : T3;                                        !
   end record                                                 !
   type T5 is array (1..10) of T4;                            !
   X : T5;                                                    !
begin                                                         !
   X(2).A4(3).A2(7):=C;                                       !
end;                                          ----------------
      или

procedure Q is                               ----------------
   package P1 IS                                              !
      package P2 is                                           !
         package P3 is                                     Способ
            package P4 is                                 "достать"
               type T is (A,B);                           из пакета
            end P4;                                        нужный
         end P3;                                       вычислительный
      end P2;                                              ресурс
   end P1                                                     !
   X : P1.P2.P3.P4.T;                                         !
   ...                                                        !
                                                              !
   ...                                       -----------------
```

или

```
procedure P is
   I : INTEGER ;
   ...
   procedure P1 is
      I : INTEGER ;
      ...
      procedure P2 is
         I : INTEGER ;
```

```
         ...
         procedure P3 is
            I : INTEGER ;
            ...
            begin
               ...
               ...
               P.I:= P1.I;
               P3.I:= P.P1.P2.P3.I     -- эквивалентно  I := I  (*)

            end P3;
         ...
      ...
   ...
end P;
```

Последний пример одновременно демонстрирует и мощь средств именования в Аде. В традиционных ЯП с блочной структурой (Алгол, Паскаль) имя, объявленное во внутренней области локализации, закрывает все глобальные омонимы. Другими словами, одноименные объекты в такой области локализации абсолютно недоступны.

В общем случае это не всегда удобно, иногда полезно иметь доступ к закрытому объекту (приведите примеры, когда это может понадобиться).

В Аде в таком случае достаточно указать полное имя закрытого объекта. Но для этого необходимо иметь возможность называть именами области локализации. Поэтому в Аде появились именованые блоки и циклы. [В цикле с параметром объявляется параметр цикла; имя цикла применяется в операторе выхода из цикла (exit)].

На этих примерах просматривается общий закон : значение имени "с точками" получается последовательным уточнением именуемого объекта при движении по составному имени слева направо.

4.23.7.2.1. Применение составных имен

Составное имя может использоваться в следующих случаях:

1. "Именуемая компонента"
   - 1.1. Компонента объекта комбинированного типа
   - 1.2. Вход задачи
   - 1.3. Объект, обозначаемый ссылочным значением
2. "Полное имя"
   - 4. Сущность, объявленная в видимой части пакета
   - 5. Сущность, объявленная в охватывающей области локализации

4.23.7.2.2. Источники сложности

(а) все эти пять случаев с точки зрения контекстно-свободного синтаксиса не различаются

(б) в одном и том же имени может комбинироваться несколько случаев.

Пример

```
procedure P is
   package Q is
      type T is
         record
            A:INTEGER ;
            B:BOOLEAN;
         end record ;
      X:T;
   end Q

begin
   ...
   Q.X.A:=1;
end P;
```

4.23.7.3. Области локализации и "пространство имен" Ада-программы

Ада относится к ЯП с "блочной структурой", точнее, со структурой вложенных областей локализации имен. Областью локализации называется фрагмент программы, в котором могут быть введены имена, непосредственно доступные в этом фрагменте и

непосредственно недоступные извне этого фрагмента. Таким образом, понятие области локализации позволяет структурировать "пространство имен" программы.

В Аде имеются следующие разновидности областей локализации :

- программный модуль (спецификация + тело)
- объявление входа вместе с соответствующими операторами приема входа (вводятся имена формальных параметров)
- объявление комбинированного типа (вводятся имена полей) вместе с соответствующим возможным неполным объявлением или объявлением приватного типа (вводятся дискриминанты), а также спецификацией представления
- переименование (воэможно, вводятся новые имена формальных параметров для новых имен подпрограмм)
- блок и цикл.

Замечание.

Область локализации физически не обязана быть связным фрагментом. В частности, возможны области локализации, состоящие из нескольких компилируемых модулей.

Конец замечания.

4.23.7.3.1. Особенности областей локализации Ады :

1). Неявные объявления: фактически в области локализации может быть объявлено больше имен, чем там текстуально присутствует.

2). В отличие от Алгола, в некоторых случаях имя доступно ("через точку") извне своей области локализации. Перечислим эти случаи:

(а). Можно получить доступ внутрь пакета при помощи полного имени:

```
procedure P is
   package Q is
      I : FLOAT;
   end Q;
   ...
begin
   Q.I := 1;        -- имя I доступно "через точку"
```

(б). Аналогично, извне непосредственной области локализации возможен доступ к именам компонент записи, именам входов задач, именам формальных параметров.

3). Борьба с коллизией имен.

В Алголе запрещено объявлять одинаковые идентификаторы в одном блоке. В Аде пойти по такому пути нельзя (иначе не решить проблему полиморфизма за счет перекрытия имен операций). Поэтому в Аде принято более тонкое решение. В одной области локализации запрещены не любые одинаковые идентификаторы, а только так называемые омографы.

Содержательно омографы - это объявления одинаковых имен, коллизия между которыми неразрещима с помощью правил перекрытияо. Перед тем, как дать более точное определение омографа, уточним правила перекрытия.

4.23.7.3.2. Правила перекрытия

В Аде разрешено перекрывать имена операций и процедур (включая предопределенные операции, такие, как +, -, *, and), перечисляемые литералы (поэтому-то они и считаются неявно объявленными функциями без параметров) и входы задач (вспомните, вызов входа по форме ничем не отличается от вызова процедуры). Для краткости будем в этом разделе называть имена всех указанных категорий перекрываемыми именами.

Профилем перекрываемого имени (и соответствующего объявления) называется кортеж, составленный в естественном порядке из формальных параметров и результата соответствующей подпрограммы (указываются имена и типы параметров и результата, если он имеется).

Два объявления одинаковых идентификаторов называются омографами либо когда их профили совпадают, либо когда одно из них не относится к перекрываемым.

Основное ограничение : омографы в одной области локализации в Аде запрещены. С другой стороны, во вложенной области локализации имя может быть закрыто только омографом, а не просто объявлением такого же имени.

Пример:

```
procedure P is
        function F (X : FLOAT) return INTEGER ;
        I : INTEGER ;
        ...
```

```
    procedure P1 is
       function F (X : INTEGER ) return INTEGER ;
       ...
       begin
          I := F(1.0); -- эквивалентно I:= P.F(1.0)
          ...
          I := F(1);   -- эквивалентно I := P1.F(1)
       end P1;
    ...
 end P;
```

4.23.7.4. Правила видимости

Рассмотрим вопрос : какие объявления непосредственно видимы из данной точки программы. Другими словами, будем решать задачу идентификации простого имени: определить область (непосредственной) видимости - множество (непосредственно) видимых имен (доступных без уточнений с помощью разделителя ".").

Область видимости в некоторой точке компилируемого модуля состоит из следующих частей.

Объявления, текстуально входящие в рассматриваемый модуль и видимые по традиционным правилам блочной структуры областей локализации (с учетом Ада-обобщения этого понятия - вспомните циклы, объявления комбинированных типов и т.п.). Эти объявления считаются самыми внутренними!

Объявления из предопределенного пакета STANDARD.

Контекст модуля, задаваемый двусторонними связями с программной библиотекой. Другими словами, рассматриваются объявления родительских модулей, если рассматриваемый модуль - вторичный.

Имена библиотечных модулей, упомянутых в указателе контекста рассматриваемого модуля.

Наконец, непосредственно видимыми становятся объявления из видимой части тех пакетов, имена которых перечислены в видимых указателях сокращений.

Пример:

```
package P is            -- первичный библиотечный пакет
   I : INTEGER ;
end P;

   with P; use P;
   procedure Q is
      package R is
         B : BOOLEAN ;
      end R;
      use R;
      begin
         B := TRUE;
         I := 1;
   end Q;
```

Итак, область видимости построена. Осталось выбрать "подходящее" объявление. Если имя перекрыто, то получаем несколько вариантов его объявления и, чтобы отобрать нужное, необходимо исследовать контекст использующего вхождения имени. Тут много дополнительных правил, на которых останавливаться не будем. Если в конечном итоге контекст позволяет отобрать ровно одно объявление, то перекрытие считается разрешенным и имя оказывается полностью идентифицированным; в противном случае имеет место нарушение языковых законов

4.23.7.5. Идентификация составного имени

Общая схема идентификации составного имени с точками выглядит следующим образом. Сначала идентифицируем первое слева имя без точки, затем (в зависимости от того, что получилось), корректируем область видимости и переходим через точку слева направо, и т.д

Пример:

```
with PACK; use PACK;
procedure P is
   package Q is
      type T is record
         A : T1;
```

```
      B : T2;  -- T1 и T2 объявлены в PACK
   end record ;
end Q;
use Q;
X : T;
Y : T1;
```

```
        procedure PP is
           X : FLOAT;
           ...
           begin
              P.X.A.:= Y;  -- все правильно (почему?)
        end PP;
     end P;
```

[Заметим, что перекрытие может "тянуться" вдоль имени. Например, перекрытая функция, выдавая два результата сложной структуры, заставляет анализировать каждый из них. Но при этом анализе для движения по структурам может снова потребоваться вычисление перекрытых функций и т.д. Конечно, так программировать нельзя (ненадежно!), но язык позволяет.]

4.23.8. Выводы. Недостатки именования в Аде

Основные принципы именования в языке Ада рассмотрены. Очевидно, что идентификация достаточно сложно устроенного имени оказывается нетривиальной.

Но не это самое страшное. Оказывается, что именование в Аде не удовлетворяет одному из основных требований к языку - надежности.

Основных источников ненадежности два.

Во-первых, пространство имен компилируемого модуля может формироваться неявно (вспомните об операциях производных типов).

Во-вторых, смысл указателя сокращений определен столь неестественно, что оказался нарушенным принцип целостности абстракций.

Вспомните, указатель сокращений действует так, что в точке, где он расположен, становятся непосредственно доступны имена из видимой части перечисленных в нем пакетов. Но не все такие имена(!), а только такие, которые не закрывают имена, непосредственно видимые в этой точке без указателя сокращений.

Здесь самое главное в том, что "не все". Обоснование несущественно, так как нарушен важнейший принцип - принцип целостности.

Действительно, если пакет - логически связанная совокупность ресурсов, или, иными словами, модель мира, то указатель сокращений, по основной идее, "вскрывая" пакет, позволяет удобно работать с этой моделью.

Однако в том то и дело, что пакет может быть вскрыт не полностью - и целостность модели оказывается нарушенной.

Пример:

```
     package P is               -- модель: типы, процедуры,
        type T1 is range 1..10;  -- переменные ...
        ...
        type T10 is  ...
        procedure P1( ...
        ...
        ...
        procedure P10( ...

        I1 : T1;
        ...
        I10 : T10;
     end P;
     with P; use P;   -- работа в модели
     procedure K is
        -- нет объявлений имен T1-T10, P1-P10, I1-I10
        begin
           I1 := 1:    -- I1 - компонента модели
           ...
           declare     -- блок, область локализации
              type T1 is range -10..10;
              I1 : INTEGER;
              use P;
           begin
              I1 := 1; -- не из модели   !! здесь целостность
              I2 := 1; -- снова в модели !! модели нарушена
              ...
```

Более того, соединение неявных объявлений с перекрытием позволяет "замаскировать" нарушения таким образом, что содержательно неправильное использование имени окажется правильным с точки зрения правил языка.

Пример:

```
    package Q1 is
       ...
       package Q2 is
          ...
          type T1 is (A,B,C);
          ...
          procedure  P(x : T1 := A);
          ...
          function F(x : T1) return INTEGER ;
          type T2 is range -100..100;
          ...
          function G(x : T2) return INTEGER ;
       end Q2;
     ...
    end Q1;

    with Q1;
    procedure P is
       ...
       A : INTEGER ;
       function F(X : INTEGER) return INTEGER ;
       ...
       type NT1 is new Q1.Q2.T1;
       type NT2 is new Q1.Q2.T2;
       function G(X : INTEGER ) return INTEGER ;
          procedure P1 is
             type NT10 is new Q1.Q2.T1;
             type NT20 is new Q1.Q2.T;
             I : INTEGER ;
             begin              -- P1
                I:= F(A);       -- Q1.Q2.F
                I:= G(1);       -- двусмысленность
          end P1;

    begin                       -- P
       I:= F(A)                 -- P.F
       I:= G(1)                 -- P.G
                                -- двусмысленность
    end P;
```

Упражнение. Постарайтесь найти доводы в пользу адовской семантики указателя сокращений. Ведь зачем-то она определена именно так!

Подсказка. Авторы отдавали приоритет явным объявлениям перед неявными (разумно, если не противоречит более важным принципам).

Аппарат именования должен быть согласован со средствами абстракции-конкретизации. В Аде это удалось сделать далеко не лучшим образом. Именование оказалось сложным и в определенных сочетаниях опасным, нарушающим важнейшие общеязыковые принципы.

4.24. Обмен с внешней средой

До сих пор мы занимались внутренним миром исполнителя, игнорировали одну из важнейших технологических потребностей - потребность управлять обменом данными между исполнителем и внешней средой (управлять, как часто говорят, вводом-выводом данных). Настало время поговорить о связи исполнителя с внешним миром.

Конечно, можно считать, что такая связь обеспечивается соответствующими базисными операциями исполнителя; они имеются в каждом исполнителе (почему?).

Управление обменом состоит в указании подходящей последовательности таких базисных операций. Казалось бы, никаких особых проблем.

На самом деле управление вводом-выводом обладает целым рядом особенностей, создающих специфические проблемы на всех уровнях проектирования - от создания аппаратуры до создания конкретных прикладных программ.

Источник всех таких проблем - в потенциально неограниченной сложности, изменчивости и непредсказуемости внешней среды, исключительном разнообразии требований к обмену с ее стороны. Коротко говоря, внутренний мир исполнителя несравненно беднее и определеннее, чем его внешний мир.

Конечно, проектировать связь исполнителя с внешним миром в каждом конкретном случае удается только за счет построения моделей внешнего мира, снужной степенью подробности отражающих его особенности. Базовый ЯП должен содержать средства для построения таких моделей. Другими словами, в нем должны быть подходящие средства абстракции и конкретизации. Вместе с тем возможны и встроенные в язык готовые модели, которые авторы ЯП посчитали особо значимыми.

И то, и другое есть в Аде. Перечислим специфические особенности внешнего мира с точки зрения обмена, дадим общую характеристику соответствующего аппарата абстракции-конкретизации, а затем продемонстрируем его реализацию в Аде.

4.24.1. Специфика обмена

4.24.1.1. Внешние объекты и их относительная независимость от исполнителя

Первая особенность в том, что объекты внешней среды и связи между ними, в отличие от внутренних объектов исполнителя, не находятся под его контролем полностью. Такие объекты могут возникать вне программы, взаимодействовать и изменяться, а также исчезать совершенно независимо от действий исполнителя. С точки зрения ЯП это означает принципиальную невозможность полностью зафиксировать в языке смысл взаимодействия исполнителя с любым мыслимым внешним объектом. Для области встроенных систем, характеризуемой исключительным разнообразием внешних объектов, это означает необходимость иметь в базовом ЯП средства описания взаимодействия с объектами заранее неизвестной природы.

Например, в Аде это уже знакомые нам средства определения новых типов вместе с детальным управлением конкретным представлением объектов (вплоть до программирования в терминах другого языка).

Характерное проявление относительной независимости внешних объектов - невозможность определить в ЯП единый метод их именования. Поэтому в базовом ЯП должен быть специальный способ представления внешних имен, причем подробности этого представления по необходимости должны зависеть от конкретной внешней среды. Например, в Аде внешнее имя предствляется строкой символов, заключенной в кавычки. Детали зависят от конкретной среды (строго говоря, "определяются реализацией").

4.24.1.2. Разнообразие периферийных (внешних) устройств

Вторая особенность тесно связана с первой и состоит в исключительном разнообразии и изменчивости набора устройств, выступающих во внешней среде партнерами исполнителя по обмену (взаимодействию). В сущности, такое разнообразие - прямое следствие все более широкого внедрения компьютеров в самые различные сферы деятельности.

Таким образом, с точки зрения набора устройств внешний мир исполнителя изменяется весьма интенсивно. Причем меняется и в зависимости от прикладной области, и в зависимости от решаемой задачи, и от конкретных условий ее решения, и6 наконец, динамически изменяется в процессе самого решения.

Например, зависимость от предметной области проявляется в том, что для числовых расчетов достаточно, скажем, ввода с перфокарт и вывода на печать; для игр нужно вводить с клавиатуры (или применять специальные манипуляторы) и выводить на экран (желательно цветной); для управления объектами в реальном времени нужно получать информацию от датчиков и выдавать управляющие сигналы органам управления объектами и т.п.

Зависимость от решаемой задачи проявляется, например, в том, что, скажем, при подготовке книги к изданию с помощью компьютера первоначальный набор текста удобнее вести без применения специальных манипуляторов (типа "мышь", например), а собственно редактирование - с их помощью. Соответственно вывод требуется на экран или принтеры различного качества (и соответственно, скорости), либо непосредственно на наборную машину.

Наконец, в процессе решения одной задачи внешняя среда может измениться - приборы, датчики, органы управления, устройства ввода-вывода могут выходить из строя, конфигупация оборудования может изменяться по различным причинам.

Итак, будем считать обоснованным тезис об изменчивости внешней среды и (или) связи исполнителя со средой.

Именно такого рода изменчивость создает особые сложности при программировании обмена и ставит проблему экономии (оптимизации) усилий создателя программы.

Общий прием борьбы со сложностью такого рода нам хорошо знаком. Конечно, нужен аппарат абстракции-конкретизации.

Современные средства программирования обмена обычно организованы так, что программист имеет возможность выбрать подходящий уровень абстракции при моделировании внешней среды своей программы. При этом удается в значительной степени игнорировать специфические особенности весьма разнообразных потенциальных внешних устройств. Так что содержательная часть программы пишется один раз, а настройка (конкретизация) на специфическое внешнее устройство выполняется в процессе связывания с этим устройством.

Однако процесс связывания в области обмена с внешней средой имеет важную особенность. В общем случае в этом процессе невозможно обойтись без полного жизненного цикла программы, вплоть до проектирования специальных программ заново. Таким образом, аппарат связывания, ориентированный на обмен с внешним миром, должен содержать, как уже было сказано, развитые средства программирования.

4.24.1.3. Человеческий фактор

Третья особенность в том, что в качестве источника и потребителя обмениваемых данных может выступать человек со своими специфическими свойствами и требованиями. Пример иакого свойства - способность человека ошибаться при вводе данных. Пример требования - необходимость представлять данные в удобочитаемом или общепринятом при определенной деятельности виде.

С учетом того, что данные должны располагаться на ограниченном пространстве экрана, стандартном листе бумаги или чертеже и при этом создавать впечатляющий зрительный и (или) звуковой образ, возникает потребность в изощренных средствах форматирования, а также управления графикой, цветом, звуком и иными средствами воздействия на человека.

С этой точки зрения в Аде определены только простейшие возможности (форматирования). Все остальное должно программироваться явно с применением средств развития.

4.24.1.4. Динамизм и относительная ненадежность

Четвертая особенность - динамизм внешних объектов. Из-за относительной независимости поведения внешних объектов достаточно полный статический (при трансляции программы) контроль их поведения невозможен.

Например, невозможно гарантировать, скажем, правильность работы человека с клавиатурой или сохранность файла на внешнем носителе. В случае внутренних объектов статический контроль возможен именно за счет того, что все поведение такого объекта находится под полным контролем программы. Скажем, у целой переменной не может быть вещественного значения, потому что нет способа в программе выполнить подобное присваивание. Но никакими ограничениями ЯП нельзя запретить человеку ошибаться или испортить внешний носитель.

Поэтому при управлении обменом с внешней средой совершенно необходимым оказывается динамический контроль с помощью аппарата исключений.

Итак, динамизм сочетается с ненадежностью внешних объектов.

4.24.1.5. Параллелизм

Пятая особенность - существенная асинхронность поведения внешних объектов по отношению к исполнителю. Это, конечно, также прямое следствие их относительной независимости и разнообразия. Исторически именно различия в скорости центрального процессора и внешних устройств привели к изобретению аппарата прерываний и других средств программирования асинхронных процессов. Стремление к рациональному, эффективному, и естественному взаимодействию с внешней средой, где могут относительно самостоятельно существовать активные объекты (устройства) со своим "жизненным ритмом", с необходимостью приводит к применению в области обмена с внешней средой аппарата управления асинхронными процессами.

Итак, завершая знакомство со специфическими особенностями обмена, подчеркнем, что в этой области требуется совместно использовать практически весь спектр изученных нами концепций и языковых конструктов - и моделирование, и связывание, и средства развития, и типы, и управление представлением, и исключения, и асинхронные процессы. А также родовые сегменты и перекрытия, как мы скоро увидим. С этой точки зрения рассматриваемая тема достойно завершает общее знакомство с базовым языком индустриального программирования, давая возможность продемонстрировать его характерные свойства в совокупности. Для Ады это естественно. Ведь программирование обмена - это в общем случае и есть программирование исполнителя как системы, встроенной в окружающую среду (т.е. объемлющую систему), работающую в реальном масштабе времени.

4.24.2. Назначение и структура аппарата обмена

В соответствии с принципом обозначения повторяющегося специфика обмена оправдывает появление в развитых ЯП специализированного аппарата обмена.

Этот аппарат предназначен для удовлетворения указанных выше потребностей (именование внешних объектов, связывание внешних объектов с внутренними, контроль и управление исключениями, форматирование, подкдючение устройств заранее неизвестной природы).

Аппарат обмена в традиционных ЯП обычно строится так, чтобы максимально освободить программиста от особенностей конкретных устройств ввода-вывода. Как правило, язык содержит достаточно абстрактную ("машинно-независимую") модель обмена, которая и поступает в распоряжение программиста. Все проблемы связывания (конкретизации обмена с учетом реальной внешней среды) решаются вне программы внеязыковыми средствами. Обычно это делается средствами операционной системы. При этом подключение совершенно новых устройств требует программирования соответствующих связывающих программ-драйверов, как правило, на уровне машинного языка. И выполняется оно не прикладными, а системными программистами.

В абстрактной модели обмена обычно пользуются понятием "логического внешнего устройства". Другими словами, это абстрактное устройство, отражающее существенные особенности реальных "физических" устройств некоторого класса (играющее роль модели таких устройств).

Важнейшая с точки зрения обмена особенность устройства ввода-вывода состоит в том, что к нему можно применять операции передачи и (или) получения данных определенного типа. Естественно, именно эта особенность отражена во всех логических устройствах обмена. Более тонкие особенности модели определяют, можно ли получить ранее переданные данные, как связан порядок передачи с порядком получения, какого рода контроль сопутствует обмену и т.п.

4.24.2.1. Файловая модель

Одна из наиболее распространенных моделей обмена - файловая модель. В ней внешние устройства представлены так называемыми файлами. Файлы естественно считать именованными объектами некоторого предопределенного типа (ограниченного приватного в смысле Ады) с подходящим набором операций. От других моделей файловая отличается независимостью файлов.

Если специально не оговорено обратное, то данные из различных файлов никак не связаны между собой. Другими словами, из файла невозможно получить данное, переданное в другой файл.

Абстрактные файлы называют также потоками, каналами, наборами, фондами, теками (иногда связывая с этими названиями определенную специфику).

Наиболее распространенные файловые модели - последовательная и индексно-последовательная, соответствующая реальным устройствам так называемого последовательного и прямого доступа.

Примерами устройств последовательного доступа служат магнитофоны (магнитные ленты), прямого доступа - дисководы (магнитные диски). Характерная особенность последовательного файла - возможность получать данные только в том порядке, в котором они были ранее переданы. Особенность индексно-последовательного файла - возможность произвольно менять этот порядок, управляя позицией, в которой выполняется обмен. Эта позиция однозначно определяется так называемым индексом (аналогом адреса внутренней памяти).

В качестве примера рассмотрим абстрактную модель обмена в Аде. Соответствующий аппарат обмена называют аппаратом обмена высокого уровня. Название связано с "высоким уровнем" абстракции соответствующей модели. Он проявляется в

практически полном отсутствии в этой модели специфических особенностей реальных устройств обмена (нет никаких зон МЛ, дорожек или цилиндров дисков и т.п.).

В Аде имеется и аппарат обмена "низкого уровня". Его зависимость от реальных устройств проявляется, в частности, в том, что программист обязан полностью определять организацию связи между последовательно передаваемыми или получаемыми данными (в соответствии с назначением и индивидуальными особенностями реального устройства). Априори не предполагается никакого аналога сохранения этих данных в именованных файлах.

4.24.3. Файловая модель обмена в Аде

Файловая модель представлена в Аде четырьмя предопределенными родовыми пакетами: последовательный_обмен, прямой_обмен, текстовый_обмен и исключения_обмена. Приведем спецификации этих пакетов с соответствующими пояснениями.

4.24.3.1. Последовательный обмен

```
with исключения_обмена;
generic
  type файловый is limited private ;
package последовательный_обмен is
  type файловый is limited private ;
  type режим_обмена is (ввод, вывод);
-- управление файлами
procedure создать (файл  :in out файловый; -- внутренний файл
                   режим : in режим_обмена;
                   имя   : in строчный := " "; -- внешний
                   доступ: in строчный := " ");
                   -- физическая организация, правила доступа
procedure открыть (файл :in out файловый;
                   режим : in режим_обмена;
                   имя   : in строчный;
                   доступ: in строчный := " ");

procedure закрыть (файл  :in out файловый);
procedure удалить (файл  :in out файловый);
procedure сначала (файл  :in out файловый;
                   режим : in режим_обмена);
procedure сначала (файл  :in out файловый);
function режим  (файл : in файловый)  return режим_обмена;
function имя    (файл : in файловый)  return строчный;
function доступ (файл : in файловый)  return строчный;
function открыт (файл : in файловый)  return BOOLEAN;
-- операции собственно обмена
procedure читать (файл :in файловый;
                  элемент : out тип_элемента);
procedure писать (файл :in файловый;
                  элемент : out тип_элемента);
function конец_файла (файл : in файловый)  return BOOLEAN;
-- исключения
статус_неправильный : exception renames
                      исключения_обмена.статус_неправильный;
-- файл не открыт или попытка открыть неоткрытый файл
режим_неправильный  : exception renames
                      исключения_обмена.режим_неправильный ;
-- ввод из выводного или наоборот
имя_неправильное    : exception renames
                      исключения_обмена.имя_неправильное ;
-- очевидно
использование_неправильное : exception renames
                      исключения_обмена.использование_неправильное ;
-- попытка создать входной с доступом выходного и т.п.
устройство_неисправно : exception renames
                      исключения_обмена.устройство_неисправно ;
-- отказ соответствующего внешнего устройства, не позволяющий
```

```
-- завершить операцию обмена
закончен_файл       : exception renames
                      исключения_обмена.закончен_файл ;
-- попытка прочитать маркер конца файла
данные_неправильные : exception renames
                      исключения_обмена.данные_неправильные ;
-- данные нельзя интерпретировать в соответствии с указанным
-- типом элемента .
private
-- определяется реализацией языка
end последовательный_обмен ;
```

4.24.3.2. Комментарий к пакету последовательный_обмен

Итак, в контексте этого пакета внешний мир исполнителя представлен совокупностью так называемых внешних файлов, идентифицируемых по уникальным именам-строкам. Дополнительные свойства внешних файлов (возможность только вводить, только выводить, данные о разметке и т.п.) указываются в специальной строке доступ. Таким образом, внешние файлы - это абстрактные внешние устройства.

Во внутреннем мире исполнителя внешние файлы представлены внутренними объектами ограниченного приватного типа "файловый" с набором операций и исключений, зафиксированным в определяющем пакете последовательный_обмен. После соответствующей конкретизации (настройки) этого пакета на нужный тип вводимых и (или) выводимых данных (тип_элемента) в контексте этого пакета можно:

а) объявлять внутренние файлы.

Например

А,В : файловый;

б) содавать внешние файлы и связывать их с объявленными внутренними.

Например

создать (А, вывод, "пример", "последовательный");

При этом правила указания имени и доступа зависят от конкретной внешней среды ("определяются реализацией").

в) открывать ранее созданные внешние файлы, связывая их с внутренними.

Например

открыть (А, ввод, "пример", "последовательный");

Ясно, что открывать для ввода имеет смысл только такие внешние файлы, в которые ранее что-то уже выводилось. Либо файлы, которым в реальной внешней среде соответствуют источники данных (клавиатура, устройство ввода с перфокарт и т.п.).

г) закрывать файлы, разрывая связь внутреннего файла с внешним.

Например

закрыть (А);

Операция открытия может потребовать установки подходящего диска или кассеты с нужным внешним файлом на подходящее устройство обмена. Операция закрытия позволяет освободить это устройство для других целей.

д) удалять файлы из внешней среды, делая их впредь недоступными.

Например

удалить (А);

Этой операцией следует пользоваться очень осторожно.

е) установить файл в начальную позицию.

Позиции линейно упорядочены начиная с 1. Операция чтения или записи увеличивает позицию на 1 (после своего выполнения).

Понятие позиции касается внутренних, а не внешних файлов. Допустимо связывать с одним внешним файлом несколько внутренних. Они могут быть разных режимов и могут находиться в разных позициях. Однако это возможно не во всякой среде.

ж) узнать режим обмена, внешнее имя, характеристику доступа, а также узнать, открыт ли файл.

з) наконец, можно прочитать или записать объект данных нужного типа.

Например, если объявлен тип "таблица", то после конкретизации

```
package обмен_таблиц is new последовательный_обмен (таблица);
use обмен_таблиц;
Т : таблица;
```

можно объявить

```
A : файловый;
открыть (А, вывод, "таблицы", "последовательный");
loop
-- формирование таблицы
  писать (А, Т);
end loop ;
закрыть (А);
```

Затем в аналогичном контексте можно прочитать сформированный ранее файл таблиц :

```
открыть (А, ввод, "таблицы", "последовательный");
if not конец_файла (А) then читать (А, Т);
закрыть (А);
```

Тем самым показано и применение функции "конец_файла". Смысл исключений указан в комментариях определяющего пакета.

Вопрос. Зачем нужен отдельный пакет исключений обмена, а также переименования в родовом пакете последовательный_обмен? Почему нельзя просто объявить исключения в этом родовом пакете?

Ответ. Такой прием описания позволяет организовать единообразную реакцию на исключения обмена. Если бы исключения не были переобъявлены, то они не были бы видимы в контексте, где применяется пакет последовательный_обмен (они были бы видимы только в самом этом пакете). Поэтому нельзя было бы в этом контексте задавать реакцию на эти исключения. А если объявить исключения в родовом пакете, то для каждой конкретизации этого пакета они были бы своими (другими) и при совместном применении различных конкретизаций (для обмена данных различных типов) попытка задать реакцию на исключения приводила бы к неудобствам или конфликту наименований.

Доказана неформальная теорема : исключения обмена рационально объявлять в предопределенном пакете и переименовывать в родовых специализированных пакетах.

Вопрос. Почему реализация обмена родовая? Почему нельзя в одном пакете определять обмен данных различных типов?

Ответ. В соответствии с концепцией уникальности типа процедуры обмена должны иметь точную спецификацию параметров. Тем самым фиксируется тип обмениваемых данных. Отсюда следует и фундаментальное свойство адовских файлов - однородность - каждый файл характеризеутся единым типом элементов (все элементы файла - одного типа!).

Доказана еще одна неформальная теорема : концепция уникальности типа влечет однородность файлов.

Замечание. Представление внутренних объектов во внешних файлах в общем случае в языке не определено (не фиксировано). Более того, оно не обязано быть зафиксированным в каких-либо документах, доступных программисту. Другими словами, это представление "зависит от реализации", но не "определяется реализацией" (последнее означает, что свойство обязано быть описано в документации для программистов).

Поэтому вводить можно только то, что ранее было выведено с помощью пакета для того же типа данных. Другими словами, последовательный (и прямой) обмен "неполноценны" в том отношении, что создавать и потреблять внешние данные при таком обмене невозможно без компьютера.

Это еще одна неформальная теорема. Она не распространяется на текстовый обмен, при котором можно вводить любые данные, представленные в соответствии с синтаксисом языка Ада. Именно синтаксис и служит документом, фиксирующим в этом случае правила представления внутренних объектов во внешних файлах. Как будет видно, при текстовом обмене допустимы и некоторые дополнительные возможности структуризации текстов (форматирования), выходящие за рамки синтаксиса Ады (разбиение на строчки, страницы и т.п.).

Конец замечания.

4.24.3.3. Прямой обмен

```
with исключения_обмена ;
generic
  type тип_элемента is private ;
package прямой_обмен is
```

```
  type файловый is limited private ;
  type режим_обмена is (ввод, обновление, вывод) ;
  type позиция is range 0..определяется_реализацией ;
  subtype индекс is позиция range 1..позиция'последний ;
-- управление файлами
procedure создать (файл  :in out файловый;
                   режим : in режим_обмена := обновление ;
                   имя   : in строчный := " ";
                   доступ: in строчный := " ");
procedure открыть (файл  :in out файловый;
                   режим : in режим_обмена ;
                   имя   : in строчный ;
                   доступ: in строчный := " ");
procedure закрыть (файл  :in out файловый);
procedure удалить (файл  :in out файловый);
procedure сначала (файл  :in out файловый;
                   режим ;in режим_обмена);
procedure сначала (файл  :in out файловый);
-- Обратите внимание, процедуры объявлены парами.
-- Вопрос. Зачем? Ведь можно и так опускать параметры,
-- да и подразумеваемое значение файла можно задать.
-- Ответ.  Но тогда нельзя было бы задавать новый подразумеваемый
-- файл (почему?). Именно из-за такого удобства и нужны столь
-- громоздкие объявления.
function режим  (файл  :in файловый) return режим_обмена ;
function имя    (файл  :in файловый) return строчный ;
function доступ (файл  :in файловый) return строчный ;
function открыт (файл  :in файловый) return BOOLEAN ;
-- операции обмена
procedure читать  (файл  :in файловый;
                   элемент : out тип_элемента ;
                   из    : in индекс);
procedure читать  (файл  :in файловый;
                   элемент : out тип_элемента);
    -- читать из текущей позиции
procedure писать  (файл  :in файловый;
                   элемент : in тип_элемента;
                   в       : in индекс);
procedure писать  (файл  :in файловый;
                   элемент : in тип_элемента);
    -- писать в текущую позицию
procedure установить-индекс (файл  :in файловый;
                             в     :in индекс);
function дай-индекс (файл  :in файловый) return индекс ;
function размер (файл  :in файловый) return позиция ;
function конец_файла (файл  :in файловый) return BOOLEAN ;
-- исключения
статус_неправильный : exception renames
                      исключения_обмена.статус_неправильный;
-- файл не открыт или попытка открыть неоткрытый файл
режим_неправильный  : exception renames
                      исключения_обмена.режим_неправильный ;
-- ввод из выводного или наоборот
имя_неправильное    : exception renames
                      исключения_обмена.имя_неправильное ;
-- очевидно
использование_неправильное : exception renames
                      исключения_обмена.использование_неправильное ;
-- попытка создать входной с доступом выходного и т.п.
устройство_неисправно : exception renames
                      исключения_обмена.устройство_неисправно ;
-- отказ соответствующего внешнего устройства, не позволяющий
-- завершить операцию обмена
закончен_файл       : exception renames
```

```
                         исключения_обмена.закончен_файл ;
      -- попытка прочитать маркер конца файла
      данные_неправильные : exception renames
                         исключения_обмена.данные_неправильные ;
      -- данные нельзя интерпретировать в соответствии с указанным
      -- типом элемента .
      private
      -- определяется реализацией языка
      end прямой_обмен ;
```

Повидимому, почти все понятно. В отличие от последовательных файлов появилась возможность явно ураравлять позицией (индексом) обмена в операциях "читать" и "писать", а также устанавливать и узнавать текущую позицию с помощью операций установить_индекс и дай_индекс. Функция "размер" позволяет узнать максимальное значение индекса, по которому производилась запись в данный файл (т.е. узнать число элементов во внешнем файле).

4.24.3.4. Текстовый обмен

Последовательный и прямой обмен учитывают относительную независимость внешних объектов, их динамизм и (частично) разнообразие внешних устройств. Однако совершенно не учитывают человеческий фактор. В сущности, последовательный и прямой обмен предназначены для взаимосвязи с устройствами внешней памяти (лентами, дисками ит.п.) и не предназначены для взаимодействия с человеком или устройствами, которые служат не для хранения данных (датчики, органы управления и т.п.).

Аппаратом, явно учитывающим человеческий фактор, в Аде служит предопределенный пакет текстовый_обмен, о котором и пойдет речь в этом разделе. Его характерная особенность - управление форматированием внешних объектов (текстовых файлов). Средства аналогичного назначения имеются и в других ЯП. Часто их совокупность называется форматным обменом (в отличие от "бесформатных" последовательного и прямого). Следующий раздел - о программировании специальных устройств с учетом их параллелизма и непредсказуемого разнообразия.

```
      with исключения_обмена;
      package текстовый_обмен is  -- это не родовой пакет!
        type файловый is limited private ;
        type режим_обмена is (ввод, вывод);
        type позиция is range 0..определяется_реализацией ;
        subtype граница is позиция range 1..позиция'последний  ;
        subtype размер_поля is INTEGER range 0..определяется_реализацией ;
        subtype основание is INTEGER range 2..16 ;
           -- основание системы счисления
        неограничено : constant позиция := 0 ; -- означает отсутствие
                       --  ограничения на длину строчки или страницы
      -- управление файлами
      procedure создать (файл  :in out файловый;
                         режим : in режим_обмена := вывод ;
                         имя   : in строчный := " ";
                         доступ: in строчный := " ");
      procedure открыть (файл  :in out файловый;
                         режим : in режим_обмена ;
                         имя   : in строчный ;
                         доступ: in строчный := " ");
      procedure закрыть (файл  :in out файловый);
      procedure удалить (файл  :in out файловый);
      procedure сначала (файл  :in out файловый;
                         режим ;in режим_обмена);
      procedure сначала (файл  :in out файловый);
      function режим  (файл  :in файловый) return режим_обмена ;
      function имя    (файл  :in файловый) return строчный ;
      function доступ (файл  :in файловый) return строчный ;
      function открыт (файл  :in файловый) return BOOLEAN ;
      -- управление подразумеваемыми (по умолчанию) входными и выходными
      -- файлами
      procedure установить_вводной  (файл  :in файловый) ;
      procedure установить_выводной (файл  :in файловый) ;
      function стандартный_вводной return файловый ;
```

```
function стандартный_выводной return файловый ;
  -- выдают файловые объекты, подразумеваемые по умолчанию
  -- в самом начале работы программы
function текущий_вводной return файловый ;
function текущий_выводной return файловый ;
  -- выдают файловые объекты, подразумеваемые по умолчанию
  -- в текущий момент работы программы
  -- Вопрос. Зачем нужны эти функции?

-- указание длины строчек и страниц
procedure установить_длину_строчки  (файл : in файловый ;
                                     длина : in позиция) ;
procedure установить_длину_строчки  (длина : in позиция) ;
procedure установить_длину_страницы (файл : in файловый ;
                                     длина : in позиция) ;
procedure установить_длину_страницы (длина : in позиция) ;
function длина_строчки (файл :in файловый) return позиция ;
function длина_строчки return позиция ;
function длина_страницы (файл :in файловый) return позиция ;
function длина_страницы return позиция ;
   -- вторые варианты - для подразумеваемых файлов
-- управление колонкой (табуляцией), строчкой и страницей
procedure новая_строчка (файл : in файловый ;
                         интервал : in граница := 1) ;
procedure новая_строчка (интервал : in граница := 1) ;
-- для выводных файлов, с соответствующими исключениями
procedure пропустить_строчку (файл : in файловый ;
                         интервал : in граница := 1) ;
procedure пропустить_строчку (интервал : in граница := 1) ;
-- для вводных файлов, с соответствующими исключениями
function конец_строчки (файл  :in файловый) return BOOLEAN ;
function конец_строчки return BOOLEAN ;
procedure новая_страница (файл  :in файловый) ;
procedure новая_страница  ;
-- для выводных файлов
procedure пропустить_страницу (файл  :in файловый) ;
procedure пропустить_страницу ;
-- для вводных файлов
function конец_страницы (файл  :in файловый) return BOOLEAN ;
function конец_страницы return BOOLEAN ;
function конец_файла (файл  :in файловый) return BOOLEAN ;
function конец_файла return BOOLEAN ;
procedure установить_колонку  (файл : in файловый ;
                               номер : in граница) ;
procedure установить_колонку  (номер : in граница) ;
procedure установить_строчку  (файл : in файловый ;
                                номер : in граница ) ;
procedure установить_строчку (номер : in граница) ;
function колонка (файл :in файловый) return граница ;
function колонка return граница ;
function строчка (файл :in файловый) return граница ;
function строчка return граница ;
function страница (файл :in файловый) return граница ;
function страница return граница ;

-- символьный обмен
procedure получить (файл : in файловый ;
                    элемент : out символьный) ;
procedure получить (элемент : out символьный) ;
procedure послать  (файл : in файловый ;
                    элемент : in  символьный) ;
procedure послать  (элемент : in  символьный) ;
-- Процедуры "получить" и "послать" близки по назначению к процедурам
-- "читать" и "писать". Они названы по-другому из-за принципиального
```

```
-- отличия - это настоящие процедуры внешнего обмена в том смысле,
-- что создавать данные для "получить" и использовать данные от
-- "послать" можно без компьютера. Формат внешнего представления
-- фиксирован синтаксисом Ады и дополнительными соглашениями о
-- строчках и страницах.
--
-- строчный обмен
procedure получить (файл  :in файловый;
                   элемент : out строчный);
procedure получить (элемент : out строчный);
procedure послать (файл  :in файловый;
                   элемент : in строчный);
procedure послать (элемент : in строчный);
procedure получить_строчку (файл  :in файловый;
                   элемент : out строчный;
                   последний : out натуральный);
procedure получить_строчку (элемент : out строчный;
                   последний : out натуральный);
procedure послать_строчку (файл  :in файловый;
                           элемент : in строчный);
procedure послать_строчку (элемент : in строчный);
-- ввод и вывод целыми строчками. При вводе с помощью
-- "получить_строчку" воспринимается строчка текста и заполняется
-- внутренний объект типа "строчный". Номер последней заполненной в
-- этом объекте позиции указывается выходным параметром "последний".
-- При выводе с помощью процедуры "послать_строчку" передается
-- внутренний объект типа "строчный" так, что его содержимое
-- образует новую строчку внешнего текста.
-- Имеются общие правила, определяющие "естественное" поведение
-- исполнителя на границах внешних строчек и внутренних объектов.
-- Например, ввод выполняется до конца текущей строчки во внешнем
-- файле или до конца внутреннего объекта.
--
-- Родовой пакет для обмна значений целых типов
generic
  type число is range <>;
package целочисленный_обмен is
  подразумеваемое_поле : размер_поля := число'поле;
  подразумеваемое_основание : основание := 10;
  procedure получить (файл : in файловый; элемент : out число;
                      поле : in размер_поля := 0);
  procedure получить (элемент : out число;
                      поле : in размер_поля := 0);
  procedure послать  (файл : in файловый; элемент : in число;
                  поле : in размер_поля := подразумеваемое_поле;
                  осн  : in основание := подразумеваемое_основание);
  procedure послать  (элемент : in число;
                  поле : in размер_поля := подразумеваемое_поле;
                  осн  : in основание := подразумеваемое_основание);
  procedure получить (из : in строчный; элемент : out число;
                      последний :out индекс);
  procedure послать  (в   : out строчный; элемент : in число;
                   осн : in основание := подразумеваемое_основание);
-- последние две процедуры - для "обмена" с внутренними строками.
-- Это удобно, в частности, при подготовке обмена с файлами.
-- Кроме того, единообразие при работе с файлами и строками помогает
-- создавать модули, пригодные для сопряжения как через внешние
-- файлы, так и через внутренние объекты строчного типа. Такими
-- модулями могут быть, например, блоки трансляторов.
end целочисленный_обмен;
--
-- Вопрос. Почему пакет родовой?
-- Ответ. В Аде много различных целых типов. Вместе с однородностью файлов
-- это обстоятельство объясняет целесообразность и необходимость
```

```
-- родового пакета для обмена целых. Заодно возникает возможность
-- удобно настраиваться на нужный размер поля во внешнем
-- представлении. Его можно не указывать - срабатывает
-- подразумеваемая атрибутная функция. Какая?
--
-- Родовой пакет для обмена вещественных
--
generic
  type число is digits <>;
package плавающий_обмен is
  подразумеваемое_до_поле : размер_поля := 2;
  подразумеваемое_после_поле : размер_поля := число'digits - 1;
  подразумеваемый_порядок : размер_поля := 3;
  -- число имеет вид "до_поле.после_полеЕпорядок"
  procedure получить (файл : in файловый; элемент : out число;
                      поле : in размер_поля := 0);
  procedure получить (элемент : out число;
                      поле : in размер_поля := 0);
  procedure послать  (файл : in файловый; элемент : in число;
               до    : in размер_поля := подразумеваемое_до_поле;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок;
  procedure послать  (элемент : in число;
               до    : in размер_поля := подразумеваемое_до_поле;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок;
  procedure получить (из : in строчный; элемент : out число;
                      последний :out индекс);
  procedure послать  (в   : out строчный; элемент : in число;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок);
-- Параметра "до" здесь не нужно, так как его можно вычислить по
-- длине строчного параметра "в". Подразумевается, что число
-- (с возможными ведущими нулями) заполняет весь строчный объект.
end плавающий_обмен;
-- Повидимому, все ясно и можно переходить к следующему пакету.
generic
  type число is delta <>;
package фиксированный_обмен is
  подразумеваемое_до_поле : размер_поля := число'до;
  подразумеваемое_после_поле : размер_поля := число'после;
  подразумеваемый_порядок : размер_поля := 0;
  -- число имеет вид "до_поле.после_полеЕпорядок"
  procedure получить (файл : in файловый; элемент : out число;
                      поле : in размер_поля := 0);
  procedure получить (элемент : out число;
                      поле : in размер_поля := 0);
  procedure послать  (файл : in файловый; элемент : in число;
               до    : in размер_поля := подразумеваемое_до_поле;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок;
  procedure послать  (элемент : in число;
               до    : in размер_поля := подразумеваемое_до_поле;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок;
  procedure получить (из : in строчный; элемент : out число;
                      последний :out индекс);
  procedure послать  (в   : out строчный; элемент : in число;
               после : in размер_поля := подразумеваемое_после_поле;
             порядок : in размер_поля := подразумеваемый_порядок);
end фиксированный_обмен;
--
-- Родовой пакет для обмена перечисляемых типов.
generic
```

```
  type перечисляемый is (<>);
package перечисляемый_обмен is
  подразумеваемое_поле : размер_поля := 0;
  подразумеваемый_нижний : BOOLEAN  := FALSE;
  -- подразумеваемый_регистр - верхний, т.е. буквы прописные.
  procedure получить (файл : in файловый;
                      элемент: out перечисляемый);
  procedure получить (элемент : out перечисляемый);
  procedure послать  (файл : in файловый; элемент : in перечисляемый;
              поле : in размер_поля := подразумеваемое_поле;
            нижний : in BOOLEAN     := подразумеваемый_нижний);
  procedure послать  (элемент: out перечисляемый;
              поле : in размер_поля := подразумеваемое_поле;
             нижний : in BOOLEAN     := подразумеваемый_нижний);
-- то же для строк
  procedure получить (из : in строчный; элемент : out перечисляемый;
                        последний :out индекс);
  procedure послать  (в   : out строчный; элемент : in перечисляемый;
             нижний : in BOOLEAN     := подразумеваемый_нижний);
end перечисляемый_обмен;
--
-- исключения (как в последовательном обмене плюс одно дополнительное
-- "нет_места"
--
статус_неправильный : exception renames
                        исключения_обмена.статус_неправильный;
-- файл не открыт или попытка открыть неоткрытый файл
режим_неправильный  : exception renames
                        исключения_обмена.режим_неправильный ;
-- ввод из выводного или наоборот
имя_неправильное    : exception renames
                        исключения_обмена.имя_неправильное ;
-- очевидно
использование_неправильное : exception renames
                        исключения_обмена.использование_неправильное ;
-- попытка создать входной с доступом выходного и т.п.
устройство_неисправно : exception renames
                        исключения_обмена.устройство_неисправно ;
-- отказ соответствующего внешнего устройства, не позволяющий
-- завершить операцию обмена
закончен_файл       : exception renames
                        исключения_обмена.закончен_файл ;
-- попытка прочитать маркер конца файла
данные_неправильные : exception renames
                        исключения_обмена.данные_неправильные ;
-- данные нельзя интерпретировать в соответствии с указанным
-- типом элемента .
нет_места : exception renames исключения_обмена.нет_места;
-- при вводе в строку ее длина недостаточна для вводимого элемента
-- Обратите внимание, объяснен смысл всех исключений из пакета исключения_обмена.
private
-- определяется реализацией
end текстовый_обмен ;
```

Итак, формализованное описание абстрактной модели обмена, принятой в Аде, полностью завершено. В качестве метаязыка мы воспользовались самим языком Ада.

Замечание.

Спецификации пакетов можно рассматривать в качестве синтаксической части такого формализованного описания. Действительно, они полностью определяют строение вызовов операций и объявлений данных, допустимых в некотором специализированном языке. Семантика этих операций и объявлений окончательно фиксируется отсутствующими частями (в том числе телами) пакетов. Для пользователя она становится известной из комментариев и специально для него предназначенной документации.

Таким образом, мы одновременно продемонстрировали метод и результат описания посредством Ады реального специализированного (проблемно-ориентированного) языка (в другой терминологии - пакета прикладных программ) для управления обменом.

Итак, показано, что Ада действительно пригодна в качестве базового языка.

Обратите внимание : таким методом можно описывать и проблемно-ориентированные языки (прикладные пакеты), которые не предполагается полностью (или даже частично) реализовывать на Аде. Кстати, именно таков наш язык управления обменом (для реализации предопределенных пакетов придется воспользоваться всеми возможностями инструментальной и целевой машин).

Использование Ады в качестве языка спецификаций в настоящее время довольно широко распространено.

Конец замечания.

4.24.3.5. Пример обмена. Программа диалога

Прототип этого примера заимствован из стандарта Ады и изменен в соответствии с концепциями нашего курса. В особенности нас интересуют возможности пошаговой детализации и управления видимостью имен.

4.24.3.5.1. Постановка задачи

Следует организовать диалог с пользователем системы, хранящей сведения о товарах (скажем, автомобилях), имеющихся в продаже.

Обратите внимание, это совершенно новый вид задачи. Мы уже программировали алгоритм вычисления некоторой функции - задача ставилась в форме спецификации требуемой функции. Программировали совокупность модулей, предоставляющую комплекс программных услуг - задача ставилась в форме спецификации перечня услуг. Теперь нужно организовать диалог. Это не функция и не комплекс услуг - это взаимодействие.

Удобной формой спецификации взаимодействия служит сценарий. Другими словами, это описание ролей партнеров по взаимодействию (описание их поведения с учетом возможного поведения партнера). Отличие от обычного театрального сценария в том, что в общем случае последовательность действий партнеров не фиксируется.

Вопрос. В чем отличие сценария от комплекса услуг?

Таким образом, при решении "диалоговых" задач начинать проектирование следует с разработки сценария как исходной "функциональной" спецификации задачи, а затем продолжать решение обычной детализацией.

Сценарий нашего диалога прост:

Система: начинает диалог, предлагая пользователю выбрать желательный цвет (автомобиля).

Пользователь: отвечает, печатая название цвета (тем самым запрашивая автомобиль указанного цвета).

Система: в ответ на запрос сообщает число автомобилей нужного цвета, имеющихся в продаже, либо указывает на ошибку в запросе и предлагает повторить попытку.

4.24.3.5.2. Пример диалога

Ответы пользователя выделены.

```
Выберите цвет: Черный.
Недопустимый цвет, попытаемся еще раз.
Выберите цвет: Голубой.
  Голубой цвет : 173
Выберите цвет: Желтый.
  Желтый цвет  :  10
```

4.24.3.5.3. Пошаговая детализация

Постараемся планирповать действия системы, считая, что в нашем распоряжении имеются все нужные ресурсы. А затем, как, если потребуется, займемся дальнейшей детализацией.

Первое приближение. Глядя на пример диалога, нетрудно выписать первое приближение:

```
loop
  ввод-цвета ;
  вывод_ответа ;
end loop ;
```

Как детализировать дальше? Во-первых, из примера видно, что вводимый цвет используется в ответе. Поэтому нужна переменная для связи ввода с выводом. Она должна иметь тип. Естественно назвать его "цвет" и задать перечислением, характеризующим все цвета автомобилей, имеющихся в продаже. Далее очевидно, что нужна таблица, содержащая сведения о количестве автомобилей каждого цвета. Имеем

Второе приближение.

```
procedure диалог is
  type цвет is (белый, красный, оранжевый, желтый,
               зеленый, голубой, коричневый);
  таблица : array (цвет) of INTEGER  := (20, 17, 43, 10, 28, 173, 87);
  выбранный_цвет : цвет;
begin
  loop
    ввод-цвета (выбранный_цвет);
    вывод_ответа (выбранный_цвет, таблица);
  end loop ;
end диалог ;
```

Третье приближение. Будем разворачивать ввод цвета и вывод ответа непосредственно в последовательность операторов, считая, что многократно пользоваться ими (в качестве процедур) не придется. Учтем, что при вводе цвета пользователь может ошибиться.

Предопределенные процедуры текстового обмена проверяют его корректность (в частности, при вводе проверяется, представляет ли вводимая последовательность символов значение передаваемого типа в соответствии с синтаксисом Ады). При любой некорректности возникает соответствующее предопределенное исключение. Следоваетльно, в соответствии со сценарием и примером диалога "ввод_цвета" должен содержать реакцию на исключение "данные_неправильные".

Детализируем лишь тело внешнего цикла в процедуре "диалог".

```
loop
declare -- блок нужен для размещения реакции на исключение
  -- ввод цвета
  послать ("Выберите цвет" :");
  получить (выбранный_цвет);
  -- конец ввода цвета
  -- вывод ответа
установить_колонку (5); -- отступ - 5 позиций
послать (выбранный_цвет);
послать ("цвет :");
установить_колонку (40); -- чтобы выделялось количество автомобилей
послать (таблица(выбранный_цвет), 4); -- размер поля
                      -- в 4 позиции достаточен для чисел из таблицы
новая_строчка;
  -- конец вывода ответа
exception -- реакция на ошибки пользователя
  when данные_неправильные =>
         послать ("Недопустимый цвет. Еще раз.");
         новая_строчка (2);
end; -- конец блока (и реакции на ошибку)
end loop ;
```

Четвертое приближение. Мы воспользовались процедурами родовых пакетов из предопределенного пакета "текстовый_обмен". Чтобы можно было писать такое "тело диалога", нужно позаботиться о формировании подходящего контекста. Именно, требуется применить указатель контекста и конкретизацию родовых процедур с учетом типов обмениваемых значений. В нашем случае это "цвет", "INTEGER" и "строчный".

Вопрос. Где используется "строчный"?

Ответ. Конечно, при выводе строк, указанных в кавычках.

Итак,

```
with текстовый_обмен; use текстовый_обмен;
procedure диалог is
  type цвет is (белый, красный, оранжевый, желтый,
               зеленый, голубой, коричневый);
  таблица : array (цвет) of INTEGER := (20,17,43,10,28,173,87);
```

```
  выбранный_цвет : цвет;
  package для_цвета is new перечисляемый_обмен (цвет);
  package для_чисел is new целочисленный_обмен (INTEGER);
  use для_цвета, для_чисел;
begin
  loop
    declare -- блок нужен для размещения реакции на исключение
      -- ввод цвета:
      послать ("Выберите цвет" :");
      получить (выбранный_цвет);
      -- конец ввода цвета
      -- вывод ответа:
    установить_колонку (5); -- отступ - 5 позиций
    послать (выбранный_цвет);
    послать ("цвет :");
    установить_колонку (40); -- чтобы выделялось
                             -- количество автомобилей
    послать (таблица(выбранный_цвет), 4); -- размер поля
                -- в 4 позиции достаточен для чисел из таблицы
    новая_строчка;
      -- конец вывода ответа
    exception -- реакция на ошибки пользователя
      when данные_неправильные =>
        послать ("Недопустимый цвет. Еще раз.");
        новая_строчка (2);
    end; -- конец блока (и реакции на ошибку)
  end loop;
end диалог;
```

Задача решена. Обратите внимание: использованы подразумеваемые внешние файлы. Обычно это клавиатура для ввода и экран для вывода. Точнее управлять назначением устройств ввода-вывода в рамках абстрактной модели невозможно. Требуются знания (а, возможно, и операции), описываемые в конкретной реализации языка.

4.24.3.6. Отступление о видимости и родовых пакетах

Мы уже отмечали ряд неприятных свойств аппарата управления видимостью в Аде. Покажем еще один пример, где он проявляет себя не лучшим образом.

Представим, что пользоваться обменом для нескольких конкретных типов приходится часто. Возникает идея создать подходящий пакет, полностью обеспечивающий нужный контекст. Слово "полностью" подчеркивает естественное требование, чтобы для работы в нужном контексте пользователю было достаточно написать указатель контекста с одним только именем нашего пакета.

Пусть для определенности нужны именно те процедуры, которыми мы воспользовались при программировании диалога. Казалось бы, достаточно написать пакет

```
with текстовый_обмен; use текстовый_обмен;
package обмен_чисел_цветов_строк is
  type цвет is (белый, ... , коричневый);
  package для_цвета is new перечисляемый_обмен (цвет);
  package для_чисел is new целочисленный_обмен (INTEGER);
  use для_цвета; для_чисел;
end обмен_чисел_цветов_строк;
```

Так что процедуру, аналогичную нашей процедуре "диалог", можно начинать так

```
with обмен_чисел_цветов_строк; use обмен_чисел_цветов_строк;
procedure новый_диалог is ... ;
```

Однако правила видимости в Аде не позволяют так работать. Ведь в процедуре "новый_диалог" будет непосредственно видимо лишь то, что объявлено в пакете из указателя сокращений. Так что нужные процедуры придется указывать полным именем. Например

```
для_чисел.послать (...);
для_цветов.получить (...); и т.п.
```

Таким образом, пользователь вынужден знать не только имя пакета, но и его содержимое, да к тому же вынужден применять длинные имена. Это явно не входило в наши планы. Другими словами, мы не достигли нужного уровня абстракции предоставляемой услуги.

Чтобы его достичь, нужно в пакете для_чисел_цветов_строк применить переименование всех нужных процедур из предопределенных пакетов. Например

```
procedure послать (элемент : out цвет;
  поле : in размер_поля := для_цвета.подразумеваемое_поле;
  нижний : in BOOLEAN := подразумеваемый_нижний)
    renames для_цвета.послать;
```

И так для всех (!) нужных процедур. Именно поэтому переименованы исключения в предопределенных обменных пакетах.

Назовем отмеченную неприятность проблемой транзита импортированных имен (ведь суть проблемы - в передаче заимствованных имен коротких имен через промежуточный модуль). Эта проблема характерна и для других ЯП с развитым управлением контекстом (например, для Модулы-2, которой мы еще займемся). В Аде она решена (в отличие от той же Модулы-2), хотя, как было показано, с весомыми накладными расходами.

Задача (повышенной трудности). Предложите и обоснуйте решение проблемы транзита.

4.24.3.7. Выводы об абстрактной модели обмена

Итак, абстрактная модель Ады характеризуется

1). Понятием файла, разграничением внешних и внутренних файлов и предопределенным типом "файловый".

2). Развитым аппаратом связывания с файлами подходящих наборов операций, представленным совокупностью пакетов с родовыми параметрами.

3). Однородностью файлов.

4). Полиморфизмом и вариаргументностью предоставляемых операций. (Вариаргументность - это возможность применять процедуры с переменным числом аргументов за счет подразумеваемых значений опущенных аргументов).

5). Форматированием, встроенным непосредственно в операции управления обменом. (Отсутствует, скажем, аналог фортранского именованного формата).

Упражнения. Придумайте варианты развития реализованного нами диалога и реализуйте его. В случаях, когда возникают сомнения в точной семантике операций, самостоятельно опишите (и обоснуйте) нужное Вам уточнение эффекта операции или воспользуйтесь указанной литературой по Аде.

4.24.4. Программирование специальных устройств

Чтобы управлять специальным внешним устройством, необходимо, естественно, иметь соответствующую аппаратуру - само устройство и аппаратный адаптер, связывающий подключаемое устройство с основным исполнителем. При этих условиях новое устройство становится источником (и потребителем) определенных воздействий на исполнитель (со стороны исполнителя). Так как обычно внешнее устройство работает асинхронно с основным исполнителем, целесообразно рассматривать его как задачу, с которой можно взаимодействовать посредством аппаратных прерываний, специальных регистров, выделенных адресов и т.п.

Назовем такую задачу аппаратной.

Изменить принцип функционирования аппаратной задачи (перепрограммировать ее) часто практически невозможно или нецелесообразно. Будем считать его фиксированным. С другой стороны, с точки зрения создаваемого комплекса программ, возможности обмена, предоставляемые аппаратной задачей, обычно выглядят как возможности излишне низкого уровня (абстракции). Они излишне детально отражают поведение устройства и весьма далеки от планируемых его содержательных функций в создаваемой системе.

Однако перепрограммировать аппаратную задачу мы считаем невозможным. Поэтому единственный способ построить более содержательное (и вместе с тем более абстрактное) устройство - создать содержательную программную модель устройства в виде асинхронного процесса, взаимодействующего с аппаратной задачей. Такая программная модель обычно называется драйвером устройства.

При создании драйвера естественно моделировать аппаратные регистры и адреса данными подходящих типов, использовать спецификацию предсталения для привязки этих данных к конкретным компонентам аппаратуры, а аппаратные прерывания использовать

для синхронизации драйвера с аппаратной задачей, связывая входы драйвера с адресами соответствующих аппаратных прерываний.

Итак, общий метод программирования специальных внешних устройств сводится к построению подходящего драйвера. При этом оказываются полезными развитые типы данных и спецификации их представления.

Важно понимать, что специфицировать представление приходится не только для привязки к адресам и структуре регистров. Скажем, перечисляемые типы удобны в модели-драйвере, но их обычно аппаратура не воспринимает. Поэтому часть работы, в обычных условиях "отдаваемой на откуп" компилятору, приходится делать вручную, выписывая явно конкретные коды для значений перечисляемого типа. Например

type защита is (обычная, ограниченная, строго_ограниченная, секретная, совершенно_секретная);

for защита use (обычная => 0, ограниченная => 1, строго_ограниченная => 2, секретная => 4, совершенно_секретная => 8);

В результате гарантируется, что драйвер передаст аппаратуре вполне определенные числовые коды, предусмотренные ее конструкцией.

4.24.4.1. Пример драйвера

Этот пример заимствован из книги Пайла "Ада - язык встроенных систем".

В ЭВМ PDP-11 символ, вводимый с клавиатуры, вырабатывает прерывание с адресом, скажем, 8#100# и выдачей соответствующего символа в буферный регистр. [Человек за клавиатурой играет роль аппаратной задачи].

Напишем драйвер, который сохраняет введенный символ в локальной переменной, доступной из любой обслуживаемой задачи по входу "взять_символ".

```
task драйвер_клавиатуры is
  entry взять_символ (симв : out символьный);
  entry есть_символ; -- аппаратное прерывание
  for есть_символ use at 8#100#;
end драйвер_клавиатуры;
-- драйвер позволяет обслуживаемой задаче быть независимой
-- от конкретных адресов и прерываний и в этом смысле
-- служит абстрактной моделью клавиатуры.

task body драйвер_клавиатуры is
  символ : символьный; -- рабочая переменная
  буф_регистр : символьный;
for буф_регистр use at 8#177462#;
  -- так элементы аппаратуры представляются данными
  -- адовских типов
begin
  loop
    accept есть_символ do символ := буф_регистр end есть_символ;
    accept взять_символ (симв : out символьный) do
      симв := символ;
    end взять_символ;
  end loop;
end драйвер_клавиатуры;
```

Достигнув в цикле оператора приема "есть_символ", драйвер ждет аппаратного прерывания. Предварительно он обязан разрешить его и заслать в вектор адресов прерываний ссылку на тело оператора приема этого входа. Делать все это обязана реализация оператора приема аппаратного входа в Аде. Отличить аппаратный вход можно по спецификации его представления.

После прерывания тело оператора приема выполняется (происходит аппаратно-программное рандеву). И драйвер готов обслужить "машинно-независимую" задачу по входу "взять_символ". Здесь рандеву самое обычное. Далее все повторяется в бесконечном цикле.

4.24.4.2. Выводы

Итак, при программировании специального устройства потребовалось

- построить модель аппаратной задачи (в нашем случае она представлена переменной буф_регистр и входом есть_символ).
- связать эту модель с конкретной аппаратурой (в нашем случае - две спецификации представления).

- построить на основе аппаратной модели содержательную модель устройства - задачу-драйвер.

Теперь можно пользоваться драйвером в качестве специального устройства обмена в прикладных программах.

Упражнение. Разберитесь а примерах управления специальными устройствами в книгах Пайла и Янга. Убедитесь, что в них явно или неявно присутствуют и аппаратная модель, и связывание, и драйвер.

4.25. Заключение: Модель А. Языковое пространство

Итак, мы завершили изучение основных абстракций ЯП с технологической позиции. Язык Ада как язык примеров оказался удобен тем, что он достаточно богат концепциями и конструктами, характерными для современного индустриального программирования. Изученные аспекты Ады образуют нашу четвертую модель - модель А.

Если модели Н,М и Б "минимальны" (каждая по-своему; у них небольшой базис и единственное средство развития), то модель А близка к "максимальной" модели современного (неймановского) ЯП - в ней содержатся практически все устоявшиеся языковые концепции и конструкты, имеется обширный базис и разнообразные средства развития.

Представим совокупность современных ЯП в виде многомерного "пространства", оси координат в котором соответствуют определенным языковым концепциям и конструктам, а координаты конкретного ЯП - степени воплощения в нем этих концепций и конструктов. С этой точки зрения каждой модели соответствует целое подпространство ЯП с различной степенью воплощения тех или иных свойств модели. Тогда моделям Н, М и Б соответствуют почти ортогональные "подпространства", а модели А - достаточно богатое подпространство неймановских языков.

Языковое пространство - обогащение исходной концептуальной схемы. Оно послужит для детальной характеристики и оценки конкретных ЯП.

5. АВТОРСКАЯ ПОЗИЦИЯ: ДВА АЛЬТЕРНАТИВНЫХ ПРИНЦИПА СОЗДАНИЯ ЯП

Построив языковое пространство, мы достигли переломного момента нашего курса. До сих пор накапливалось знание о технологических потребностях и удовлетворяющих эти потребности языковых конструктах. Теперь накоплено достаточно материала, чтобы взглянуть на него критически, с позиции автора современного ЯП.

И раньше мы не забывали об авторской позиции, выделяли концепции и конструкты, помогающие создавать ЯП, анализировать и оценивать его. Однако эти принципы и концепции, как правило, касались отдельных групп конструктов. Наша ближайшая крупная цель - указать на два принципа, касающихся языка в целом, и обсудить их влияние на современное языкотворчество.

Для краткости и выразительности дадим им названия принцип сундука и принцип чемоданчика.

5.1. Принцип сундука

На примере Ады мы видели, как выявляемые технологические потребности приводили к новым конструктам. Может показаться, что на этом пути будут получаться все более высококачественные ЯП. К сожалению, большинство современных индустриальных ЯП носят на себе родимые пятна такого примитивного критерия качества. Это характерно и для Кобола, и для ПЛ/1, и для Фортрана-77, и для Ады.

Основной принцип конструирования, которым руководствовались авторы этих ЯП, в упрощенной форме можно сформулировать так: для каждой значимой в проблемной области технологической потребности в языке должно быть готовое выразительное средство. Короче: каждой значимой потребности - готовый конструкт. Этот принцип заготовленности конструктов и назовем принципом сундука (именно в сундуках хранят много всякого на всякий случай).

Как показывает опыт, безудержное применение принципа сундука ведет к громоздким, сложным, дорогим в реализации, обучении и использовании языкам-монстрам с тяжеловесным базисом и несбалансированными средствами развития. Сундук и есть сундук!

5.2. Закон распространения сложности ЯП

Бывают взаимодействия, сложность которых по существу не зависит от собственной сложности взаимодействующих объектов. Например, процесс и результат столкновения человека с автомобилем в первом приближении никак не связан со

сложностью человека и автомобиля. Сложность вызова процедуры непосредственно не связана с ее внутренней сложностью и сложностью вызывающего контекста. В подобных ситуациях сложность инкапсулирована. Образно говоря. простота взаимодействия обеспечивается "небольшой площадью" взаимодействия потенциально весьма сложных объектов.

Язык программирования сам служит "поверхностью" взаимодействия авторов, реализаторов, преподавателей, пользователей ЯП и т.д.

Такая специфическая роль ЯП определяет справедливость для него следующего закона (распространения сложности) : собственная сложность ЯП распространяется на все аспекты его внедрения (описание, реализацию, использование, обучение и т.д.).

Никлаус Вирт отмечает частный случай этого закона [20] как самое главное, что следует усвоить о реализации ЯП.

[Н.Вирт - один из самых авторитетных специалистов по ЯП, лауреат премии Тьюринга за создание таких известных ЯП, как Алгол W, Паскаль, Модула, Модула-2.]

Итак, ориентация на принцип сундука повышает собственную сложность ЯП, что по закону распространения сложности приводит к росту сложности его внедрения (которая, в свою очередь, может оказаться катастрофической для его судьбы - достаточно сопоставить судьбу Паскаля с судьбой Алгола 68).

Вспомним, однако, что основной принятый нами критерий качества базового ЯП - его способность снижать сложность, помогать в борьбе с основной проблемой программирования. Налицо тупик, в который ведет принцип сундука. Вирту принадлежит принцип, указывающий выход из этого тупика.

5.3. Принцип чемоданчика

Н.Вирт неоднократно отмечал, что самое трудное при создании ЯП - решить, от чего следует отказаться. Объясняя принципы конструирования своего последнего языка Модула-2 (поразившего специалистов элегантностью), Вирт развил эту идею и сформулировал следующий принцип языкового минимума: в ЯП следует включать только такие концепции и конструкты, без которых совершенно невозможно обойтись.

Назовем этот принцип минимума принципом чемоданчика по контрасту с принципом сундука (в чемоданчик кладут только абсолютно необходимое).

Продемонстрируем этот важнейший метаязыковый принцип на примере конкретных решений, принятых при создании языка Модула-2 и сопоставим их с решениями, воплощенными в Аде. Но сначала придется в общих чертах познакомиться с Модулой-2.

5.4. Обзор языка Модула-2

Модула-2, созданный в конце семидесятых годов (около 1978), прямой наследник ЯП Паскаль и Модула, созданных Н.Виртом в конце шестидесятых (около 1969) и в середине семидесятых (около 1976) годов соответственно. Это ЯП общего назначения, ориентированный на относительно скромные ресурсы как инструментальной, так и целевой машины. Автор предназначал его для небольших, в том числе персональных, компьютеров ("для небольшой однопроцессорной ЭВМ").

И не ошибся. В настоящее время наблюдается своего рода бум интереса к Модуле-2 именно как к базовому языку индустриального программирования на микрокомпьютерах, "законному наследнику" Паскаля. В Модуле-2 автору удалось соединить простоту и естественность основных конструктов Паскаля с мощными и изящными средствами модуляризации. Коротко говоря, Модула-2 - это модульный Паскаль.

Специалисты обратили внимание на очередное достижение Вирта с первых публикаций. В настоящее время интерес к Модуле-2 становится всеобщим, так как появились высококачественные его реализации (в том числе примечательная авторская реализация на компьютерах DEC PDP-11/70). В нашей стране доступны реализации Модулы-2 на машинах серий СМ, ЕС и др. Замечательная особенность языка, обеспечившая его достоинства - строгое следование принципу чемоданчика.

Этот принцип наглядно проявляет себя уже в том, что авторское описание Модулы-2 требует около 80 000 печатных знаков против 1 500 000 печатных знаков стандарта Ады.

[Конечно, сказывается относительная нестрогость "авторского" описания по сравнению со стандартом (обычно стандарт в 2-2.5 раза объемнее авторского описания), но результаты сопоставления впечатляют.]

Точные данные об относительной сложности компиляторов Ады и Модулы-2 отсутствуют, но естественно предположить, что она тесно связана с объемом определения языка.

5.4.1. Характеристика Модулы-2 в координатах неймановского языкового пространства (технологическая позиция).

По сравнению с моделью А отсутствуют производные типы; концепция типа ориентирована скорее на структуру, чем на имена; существенно меньше предопределенных типов; резко ограничен аппарат управления асинхронными процессами; сильно упрощено управление видимостью и аппарат раздельной компиляции (в частности, отсутствуют вторичные модули); отсутствует аппарат управления точностью расчетов; упрощена предопределенная модель обмена; отсутствуют родовые модули; резко ограничено управление представлением; отсутствует управление исключениями.

С другой стороны, с точки зрения языкового пространства Модула-2 ничего не добавляет к модели А. Иначе говоря, верна неформальная теорема : для всякого конструкта Модулы-2 в Аде найдется конструкт с аналогичными возможностями. Другими словами, Модулу-2 можно назвать технологическим подмножеством Ады.

5.4.2. Характеристика Модулы-2 в терминах концептуальной схемы

Для краткости свойства Модулы-2 назовем м-свойствами, а свойства Ады (А-модели) - А-свойствами.

Базис. Скалярная м-сигнатура - точное подмножество А-сигнатуры. Отсутствуют фиксированные вещественные типы и вообще управление точностью. Но имеются целые, вещественные, символьные, логические (с более или менее устоявшимся набором операций).

Структурная м-сигнатура содержит регулярные, комбинированные и ссылочные типы, аналогично А-сигнатуре. Но в ней имеются также процедурные и множественные типы. Правда, весьма ограниченные.

Именно, значением процедурного типа не может быть стандартная процедура или процедура, вложенная в другую процедуру. А "множественный" тип (SET OF T), класс значений которого состоит из всех возможных множеств значений исходного типа T, можно образовывать только для исходных типов малой мощности (чтобы множество можно было представить одним машинным словом - двоичной шкалой; это пример влияния на язык реализаторской позиции).

Имеются обычные управляющие структуры (последовательности, развилки, циклы, процедуры) и ограниченные средства управления асинхронными процессами. При этом настоящий параллелизм подразумевается только для так называемых периферийных процессов (в нашей терминологии - аппаратных задач, соответствующих внешним устройствам).

Развитие. Можно создавать операционные абстракции (процедуры и функции) и абстракции данных (именованные типы с возможностью ограничивать набор применимых операций - аналог приватных А-типов). Основное средство развития - модуль. Это аналог А-пакета.

Защита. Обязательные объявления с соответствующим контролем поведения. Подразумевается статический контроль так называемой совместимости типов и динамический контроль за допустимостью значений переменных. Роль подтипа по существу играет тип, объявленный как отрезок другого типа. Отрезки одного исходного типа совместимы между собой и с исходным типом. Формально понятие подтипа отсутствует. Производные типы отсутствуют (и формально, и по существу).

[Понятие типа и совместимости типов в авторском описании определено нечетко. Неясно, какие типы равны. Можно подозревать, что равными считаются типы, названные одинаковыми именами. Так как нет производных типов, то операции явно с типом не связаны (нет проблемы неявного определения операций для производных типов).]

Аппарат исключений не предусмотрен.

Мощное средство прогнозирования и контроля фактически предоставляет аппарат управления видимостью - списки экспортируемых и импортируемых имен. С их помощью обеспечивается инкапсуляция в рамках модулей (аналогов А-пакетов).

Исполнитель. Характеризуется набором предопределенных и определяемых реализацией модулей, в совокупности обеспечивающих рациональное для небольших компьютеров сочетание общеязыковых (резидентных, постоянно присутствующих в компиляторе и (или) целевой машине) и специализированных средств. Среди последних - аппарат файлового обмена, управления параллельными процессами посредством так называемых сигналов, управление динамическим распределением памяти.

Архитектура. Характеризуется принципом чемоданчика. Именно поэтому мы особенно интересуемся Модулой-2.

5.5. Пример м-программы.

Рассмотрим решение уже известной нам задачи об управлении сетями. Цель - создание у читателя "зрительного образа" м-программ, а также подробное знакомство с теми свойствами языка, которые помогут продемонстрировать принцип чемоданчика.

Постараемся выполнять те же требования к реализации комплекса услуг по управлению сетями, что и в А-случае (надежность, целостность, модифицируемость).

Однако в самом начале - о ключевом понятии Модулы-2. Название этого понятия отражено в названии языка. Конечно, это понятие - модуль.

[В свое время мы характеризовали его как материализованное воплощение абстракции. Действительно, абстракция может существовать лишь в сознании человека, может находить свое отражение в разрозненных результатах его деятельности. А может быть целенаправленно воплощена во вполне определенном материальном объекте, аккумулирующем результаты интеллектуальной деятельности. В случае ЯП идеальный вариант такого материальногто объекта - отдельный текст воплощающего эту абстракцию модуля, который можно независимо создавать, хранить и использовать.]

Как и в Аде, спецификация в Модуле-2 отделена от реализации. Представлены они соответственно определяющими (DEFINITION) и реализующими (IMPLEMENTATIN) модулями, аналогами спецификации и тела пакета в Аде.

Продуманная интерпретация фундаментальной концепции модуля - основа элегантности и конкурентоспособности Модулы-2. Именно на этой интерпретации мы и сосредоточим свой анализ.

5.5.1. Управление сетями на Модуле-2

Ниже следует определяющий модуль ПараметрыСети (аналог спецификации соответствующего А-пакета).

```
1. DEFINITION MODULE ПараметрыСети;
2.   EXPORT QUALIFIED МаксУзлов, МаксСвязей;
3.     CONST МаксУзлов  = 100;
4.           МаксСвязей =   8;
5. END ПараметрыСети;
```

[Из описания языка неясно, нужен ли в этом случае реализующий модуль. Если нужен, то у него может быть пустое тело

```
IMPLEMENTATION MODULE ПараметрыСети;
END ПараметрыСети;                    ]
```

Как видите: очень похоже на Аду. Отличаются ключевые слова; в идентификаторах недопустимы разделители-подчеркивания (поэтому применяются большие буквы для отделения слов); допустимы серии объявлений типов, констант и переменных, выделяемых соответствующим ключевым словом; вместо is применяется знак "=". Короче говоря, Модула-2 в перечисленных отношениях ближе к своему старшему родственнику - Паскалю, чем Ада.

Главное отличие - во второй строке. Она представляет собой так называемый список экспорта. В нем явно перечисляются те и только те имена, определенные в модуле, которые считаются доступными (видимыми) в объемлющем контексте.

[Точнее говоря, ключевое слово QUALIFIED указывает на косвенный экспорт, когда доступны лишь полные имена (с указанием имени экспортирующего модуля). А именно, доступны ПараметрыСети.МаксУзлов и ПараметрыСети.МаксСвязей. При отсутствии этого ключевого слова имеется в виду прямой экспорт - непосредственно доступны "короткие" миена МаксУзлов и МаксСвязей.]

В использующих модулях можно управлять доступом с помощью так называемых списков импорта.

5.5.2. Определяющий модуль

```
1. DEFINITION MODULE УправлениеСетями;
2. FROM ПараметрыСети IMPORT МаксУзлов, МаксСвязей;
        (* это список импорта *)
3. EXPORT QUALIFIED Создать, Вставить, Удалить, Связать,
                    ИмяУзла, Связи, Присвоить, УзелЕсть,
                    ВсеСвязи, Сети;
        (* это список экспорта *)
4.   TYPE ИмяУзла = [1..МаксУэлов];
5.        ЧислоСвязей = [0..МаксСвязей];
6.        ИндексУзла = [1..МаксСвязей];
        (* производных типов нет. Все три типа совместимы. *)
7.        ПереченьСвязей = ARRAY ИндексУзла OF ИмяУзла;
        (* регулярный тип (тип массива). Все неформальные массивы - с
        постоянными границами. Точнее говоря, границы -
        константные выражения, вычислимые в период компиляции *)
8.        Связи = RECORD
9.                Число : ЧислоСвязей; (* инициализации нет *)
10.               Узлы  : ПереченьСвязей;
11.             END;
    (* комбинированный тип (тип записи). Допустимы и вариантные. *)
12.       Сети;
    (* указано только имя типа. Это так называемое непрозрачное
     объявление типа. Аналог объявления приватного типа. *)
13.    PROCEDURE Создать (VAR Сеть : Сети);
```

(* В Аде этого не было. Непрозрачные типы могут быть только ссылочными или отрезками предопределенных типов. Содержательные сети у нас - массивы. Поэтому тип "Сети" будет ссылочным (а никак не отрезком). Реализовать процедуру создания соответствующего массива можно только в модуле, экспортирующем тип "Сети", т.е. в реализующем модуле УправлениеСетями. Дело в том, что в отличие от Ады приватной части в определяющем модуле нет. Поэтому нет во внешнем контексте и информации об устройстве непрозрачных типов (ее нет даже для транслятора). Поэтому приходится определять специальную процедуру создания содержательных сетей. *)

(* Параметр "Сеть" специфицирован ключевым словом "VAR" - это так называемый параметр-переменная - аналог А-переменной, вызываемой в режиме in out. В результате исполнения процедуры будет создан указатель на массив-сеть и этот указатель присвоен формальному параметру-переменной. *)

```
14.    PROCEDURE Вставить (Узел : ИмяУзла; ВСеть : Сети);
```

(* оба параметра - параметры-значения (аналог режима in). Хотя второй параметр указывает на содержательно изменяющуюся сеть, сам указатель при этом остается неизменным. *)

```
15.     PROCEDURE Удалить (Узел : ИмяУзла; ИзСети : Сети);
16.     PROCEDURE Связать (АУзел, ВУзел : ИмяУзла; ВСети : Сети);
17.     PROCEDURE Присвоить (Сеть1, Сеть2 : Сети);
```

(* В Аде этого не было. Во внешнем контексте содержательное присваивание сетей описать невозможно из-за отсутствия информации об их строении даже у транслятора - приватной части нет! Поэтому и приходится определять специальную процедуру для присваивания содержательных сетей. *)

```
18.     PROCEDURE УзелЕсть (Узел : ИмяУзла; ВСети : Сети) :BOOLEAN ;
      (* Так объявляют в Модуле-2 логическую функцию. *)
19.     PROCEDURE ВсеСвязи (Узел : ИмяУзла; ВСети : Сети) : Связи;
20. END УправлениеСетями;
```

Строка 1 - заголовок определяющего модуля. Строка 2 - список импорта. Перечисленные в нем имена (экспортированные модулем ПараметрыСети) доступны (прямо, по коротким именам) в модуле УправлениеСетями. Важно, что никакие другие внешние имена (кроме так называемых стандартных) недоступны. Часть FROM списка импорта указывает имя модуля-экспортера и тем самым позволяет применять короткие имена. Если бы список начинался сразу словом IMPORT, то в нем должны были бы фигурировать косвенные имена (с точкой) для случая косвенного экспорта (и прямые имена для случая прямого экспорта).

Строка 3 - список косвенного экспорта (требующего во внешнем контексте в общем случае косвенных имен). Косвенный экспорт называют иногда "квалифицированным", а действие фрагмента FROM "снятием квалификации". Такие обороты выглядят чужеродными в русском тексте.

В строке 12 - так называемое непрозрачное объявление типа. По назначению оно соответствует объявлению приватного типа в Аде. Во внешнем контексте становится известно имя объявленного непрозрачного типа, но применять к его объектам можно лищь фиксированный набор операций, объявленных в этом же (определяющем модуле), так как о природе объектов непрозрачного типа во внешнем контексте ничего неизвестно.

[Конечно, имя непрозрачного типа и имена соответствующих операций должны быть в списке экспорта. Иначе тип незачем объявлять непрозрачным.]

5.5.3. Использующий модуль

```
MODULE ПостроениеСетей;
(* это так называемый главный модуль, ничего не экспортирующий *)
(* определяющий модуль для главного не пишется *)
  FROM УправлениеСетями IMPORT Создать, Вставить, Связать,
                                  Присвоить, Сети;
  VAR Сеть1, Сеть2 : Сети; (* объявление переменных типа Сети *)
BEGIN
  Создать (Сеть1); (* содержательную сеть - в отличие от объекта *)
                   (* типа Сети - можно создать только *)
  Создать (Сеть2); (* с помощью импортированной процедуры *)
  Вставить (33, 13, Сеть1);
         ...
  Присвоить (Сеть1, Сеть2);
(* объекту, указанному Сеть2, присваивается значение объекта, указан
   ного Сеть1. См. реализующий модуль, экспортирующий тип Сети. *)
END ПостроениеСетей;
```

В этом модуле отражено уже упоминавшееся важное ограничение, касающееся непрозрачных типов:

Объектами непрозрачных типов могут быть только ссылки (указатели) или скалярные объекты.

Однако при использовании непрозрачных типов неизвестно, как они устроены. Поэтому с ними можно выполнять лишь операции, явно экспортированные соответствующим определяющим модулем. Именно поэтому по сравнению с Ада-реализацией добавились операции Создать и Присвоить.

Так что непрозрачные типы Модулы-2 по использованию близки к ограниченным приватным Ады.

5.5.4. Реализующий модуль

Ниже следует реализующий модуль (аналог тела пакета).

```
IMPLEMENTATION MODULE УправлениеСетями;
  TYPE ЗаписьОбУзле = RECORD
                        Включен : BOOLEAN;
                        Связан :  Связи;
                      END;
       Сети = POINTER TO ARRAY ИмяУзла OF ЗаписьОбУзле;
(* описание устройства содержательных сетей *)
(* действует правило последовательного определения *)
  PROCEDURE Создать (VAR Сеть : Сети);
  BEGIN
    Сеть := NEW Сети;
(* работает генератор динамического объекта. Создается объект анонимного регулярного типа (содержательная сеть) и указатель на этот объект. Созданный указатель присваивается ссылочной переменной-параметру "Сеть". Обратите внимание, в генераторе Модула-2 используется ссылочныхй тип, а не базовый, как в Аде. Поэтому базовый вполне может оставаться анонимным. *)
  END Создать;

  PROCEDURE УзелЕсть (Узел : ИмяУзла; ВСети : Сети) : BOOLEAN;
  BEGIN
    RETURN ВСети_[Узел].Включен;
(* "_" означает так называемое разыменование - переход от имени к его значению. Явное разыменование в Модуле-2 применяется только для объектов ссылочных типов. "ВСети_" означает массив, на который ссылается указатель "ВСети".
```

Квадратные скобки выделяют список индексов (алгольная традиция). Точка имеет тот же смысл, что и в Аде. *)

```
        END УзелЕсть;

        PROCEDURE ВсеСвязи (Узел : ИмяУзла; ВСети : Сети) : Связи;
        BEGIN
          RETURN ВСети_[Узел].Связан;
        END ВсеСвязи;

        PROCEDURE Вставить (Узел : ИмяУзла; ВСеть : Сети);
        BEGIN
          ВСеть_[Узел].Включен := TRUE;
          ВСеть_[Узел].Связан.Число := 0;
        END Вставить;

        PROCEDURE Присвоить (Сеть1, Сеть2 : Сети);
        BEGIN
          Сеть2_ := Сеть1_;
        END Присвоить;

        PROCEDURE Удалить (Узел : ИмяУзла; ИзСети : Сети);
          VAR i : 1..МаксСвязей; (* переменную цикла нужно объявлять. При этом
контроль диапазона менее жесткий, чем в аналогичной А-программе, так как граница
отрезка типа обязана быть константным выражением. *)
        BEGIN
          ИзСети_[Узел].Включен := FALSE;
          FOR i := 1 TO ИзСети_[Узел].Связан.Число DO
            Чистить (Узел, ИзСети_[Узел].Связан.Узлы[i], ИзСети);
          END (* цикла *);
        END Удалить;

        PROCEDURE Чистить (Связь, ВУзле : ИмяУзла; ВСети : Сети);
          VAR i : 1..МаксСвязей;
        BEGIN
          FOR i := 1 TO ВСети_[ВУзле].Связан.Число DO
            IF ВСети_[ВУзле.Связан.Узлы[i] = Связь THEN
              Переписать (ВУзле, i, ВСети);
            END (* условия *);
          END (* цикла *);
        END Чистить;
      (* мы сознательно программируем близко к соответствующей А-программе,
      хотя можно было бы действовать рациональней *)

        PROCEDURE Переписать (ВУзле : ИмяУзла;
                              После : ИндексУзла; ВСети : Сети);
          VAR j :1..МаксСвязей;
        BEGIN
          WITH ВСети_[ВУзле].Связан DO  (* присоединяющий оператор *)
            FOR J := После TO Число-1 DO
              Узлы[j] := Узлы [j+1];
            END (* цикла *);
          END (* присоединяющего оператора *)
        END Переписать;
```

(* Вместо переименования (которого нет) с успехом применен так называемый "присоединяющий оператор" вида

```
WITH ИмяЗаписи DO Опереторы END
```

Его смысл в том, что между DO и END селекторы полей записи, указанной через ИмяЗаписи, доступны по коротким именам. В нашем случае это селекторы "Число" и "Узлы". Присоединяющий оператор имеется и в Паскале. *)

```
      END УправлениеСетями;
```

(* Подчеркнем, что во внешнем контексте доступны только имена из списка экспорта определяющего модуля. Реализующий модуль может включать лишь список

импорта (когда в нем используются внешние имена, которые не потребовались в определяющем модуле. Как и в Аде, все имена, доступные в определяющем модуле, доступны и в его реализующем модуле. *)

5.5.5. Выводы

Итак, поставленная задача полностью решена. Обеспечена аналогичная А-случаю целостность сетей, модифицируемость комплекса и надежность программирования.

Вопрос. За счет чего?

Ответ. Непрозрачный тип "Сети", модульность (в частности, разделение спецификации и реализации) и явные объявления (в частности, отрезки типов).

Основной вывод из нашего эксперимента : обычные программы можно писать на Модуле-2 практически с тем же успехом и комфортом, что и на Аде.

Конечно, такие выводы не делают на основании одного эксперимента с неотлаженным и тем более не применявшимся на практике комплексом программ. Однако для наших целей важно, что при решении поставленной задачи мы ни разу не попали в ситуацию, когда вместо А-средств не нашлись бы подходящие м-возможности, не приводящие к необходимости кардинально перерабатывать программу. При этом задача не подбиралась специально с учетом особенностей Модулы-2, а была просто взята первая задача, на которой мы изучали возможности Ады.

Накоплено достаточно сведений о Модуле-2, чтобы содержательно обсудить принцип чемоданчика в сопоставлении с принципом сундука. Однако так как при этом не обойтись без сопоставления А- и м-решений, уясним, в каком смысле целесоообразно их сопоставлять, насколько это может быть правомерно и почему поучительно.

Ключевые понятия при ответах на эти вопросы - языковая ниша и авторская позиция. О втором уже шла речь, а первым займемся в следующем разделе.

5.6. Языковая ниша

Вопрос о том, насколько правомерно сравнивать Модулу-2 с Адой как потенциальных конкурентов, тесно связан с понятием "языковая ниша".

Языковая ниша - это комплекс внешних условий, при которых активная жизнь двух различных языков невозможна.

Языковая ниша (или просто ниша) характеризуется по меньшей мере классом пользователей ЯП, классом решаемых задач (проблемной областью), классом инструментальных и классом целевых компьютеров (точнее, программных сред, включающих компьютеры, операционные системы, используемые прикладные пакеты и т.п.). На нишу, соответствующую ЯП, влияют, кроме собственных свойств ЯП, также и особенности технической и социальной поддержки (наличие и качество реализаций, экономическая или иная заинтересованность организаций, фирм, ведомств, стран и т.д.).

Сравнивать в качестве потенциальных конкурентов имеет смысл лишь ЯП, претендующих на одну и ту же или пересекающиеся ниши. Иначе различия между ЯП всегда можно объяснить различием "условий их активной жизни".

С другой стороны, если у конкретных ЯП сформировалась определенная ниша, то довольно бесперспективно говорить о его вытеснении потенциальным конкурентом. Опыт показывает, что справедлив закон консерватизма ниш : ниша сопротивляется замене ЯП. Этот же закон можно понимать как закон сохранения (обитателей) ниш. Поэтому к ЯП (возможно даже более, чем к программам), применим известный афоризм В.Л. Темова, который в применении к ЯП можно перефразировать так: "Языки не внедряются, а выживают".

Так что мало смысла обсуждать, скажем, замену Фортрана или ПЛ/1 на Аду или Модулу-2 без изменения класса используемых компьютеров, контингента пользователей, решаемых задач и (или) других характеристик ниши.

Однако сравнивать в качестве потенциальных конкурентов Аду с Модулой-2 вполне имеет смысл, хотя в первом приближении это ЯП различного предназначения. Ада, как уже говорилось, ориентирована в первую очередь на программирование встроенных (встраиваемых) систем с применением кросс-компиляции. А Модула-2 - на задачи системного программирования на относительно бедных однопроцессорных компьютерах.

Но реальная языковая ниша для этих ЯП еще не сформировалась. Скорее всего, эти языки будут конкурентами в ситуациях, когда принцип чемоданчика окажется более жизнеспособным, чем принцип сундука, даже подкрепленный мощной поддержкой директивных органов.

Скажем, еще до появления доступных высококачественных реализаций Ады Модула-2 может "захватить" классы применений, где ресурсы целевого компьютера позволяют отказаться от кросс-компиляции, если сам компилятор достаточно компактен. Подробное сопоставление потенциальных ниш Ады и Модулы-2 выходит за рамки курса.

Особенно поучительно сравнивать проектные решения с авторской позиции. В этом случае проектируемые ЯП могут быть предназначены для совершенно разных языковых ниш. Желательно лишь, чтобы сопоставляющий по возможности четко представлял себе особенности этих ниш и их связь с принимаемыми проектными решениями.

При таком подходе принцип чемоданчика как авторский принцип может проявиться в том, чтобы отказаться от борьбы за определенную нишу, если для этого не созрели технические или социальные условия. Другими словами, принцип минимальности можно интерпретировать и так, что следует выбирать ниши, где предлагаемые языковые решения будут выглядеть как совершенно необходимые.

Итак, будем сравнивать А- и м-решения прежде всего с авторской позиции. Однако учтем, что возможно и пересечение соответствующих ниш.

5.7. Проявление принципа чемоданчика в проектных решениях Модулы-2 (авторская позиция)

5.7.1. Видимость

Занимаясь управлением видимостью в Аде, мы обнаружили, во-первых, технологические потребности, которые привели к созданию соответствующего А-аппарата (пакетов, блоков, указателей контекста и сокращений, правил перекрытия и наследования операций производных типов, переименования, неявных объявлений).

Во-вторых, были продемонстрированы проблемы, вызываемые нежелательным взаимодействием многочисленных компонент этого сложного аппарата. В частности, было показано, как неявные объявления в сочетании с указателем сокращений могут приводить к неожиданным последствиям, явно входящим в противоречие с основной целью языка - обеспечить надежное программирование.

Как должен поступить автор языка, руководствующийся принципом чемоданчика?

Он должен заново проанализировать потребности и поискать компромисс, в максимальной степени удовлетворяющий критические потребности при минимальной возможной сложности предлагаемого языкового аппарата.

Ключевая идея такого компромисса, найденная Виртом для Модулы-2 - полный отказ от любых косвенных (неявных) объявлений.

Компромисс состоит в том, что в первом приближении такая идея должна быть не слишком удобной для пишущего программу. А иногда и для читающего. Ведь появляются (потенциально довольно длинные) списки экспортируемых и импортируемых миен. Их нужно писать, читать, проверять (и хранить). К тому же по ним все равно невозможно узнать свойства именуемых сущностей и приходится как читателю, так и транслятору анализировать экспортирующие модули.

Однако потому это и компромисс, что зато становятся совершенно тривиальными правила видимости и контроля имен - ведь все они объявлены явно в каждом модуле.

Вместе с тем этот компромисс следует именно принципу чемоданчика. Для обеспечения раздельной компиляции совершенно необходимо управлять видимостью имен, смысл которых определен в других модулях. Применяем простейший способ - явное перечисление нужных имен с указанием модуля-экспортера. Получаем списки импорта. Для надежности, облегчения понимания и модификации применяем двойственный конструкт - списки экспорта. Получаем логически завершенный аппарат связывания модулей.

У принятого решения - несколько важных следствий.

Во-первых, экспорт-импорт становится точнее, чем в Аде, где из-за правила последовательного определения видимыми могут оказаться несущественные для внешнего контекста, по сути промежуточные имена (приведите примеры).

Во-вторых, концепция явных объявлений несовместима с концепцией производных типов (с естественными для них неявными определениями операций). Весьма вероятно, что именно отмеченная несовместимость - одна из причин отсутствия производных типов в Модуле-2. А ведь это целый языковый пласт.

В-третьих, отказ от производных типов не позволяет применять типы в качестве характеристики содержательной роли объекта, что меняет саму концепцию типа. Такого рода последствия можно обнаруживать и дальше.

С одной стороны, они подчеркивают взаимозависимость и взаимообусловленность компонент языка как знаковой системы. С другой стороны, указывают цену

компромисса. С третьей стороны, показывают, от каких заманчивых возможностей (скажем, контроль содержательных ролей) приходится отказываться ради простоты определения, реализации и использовании языка, помятуя о неумолимом законе распространения сложности.

В связи с управлением видимостью интересно понять, почему в Модуле-2 остался присоединяющий оператор (остался от Паскаля). Казалось бы, это полный аналог указателя контекста, и неявные (локальные) объявления полей записи - очевидное противоречие с концепцией явных объявлений.

Однако серьезного противоречия нет. Скорее наоборот, можно и здесь усмотреть следование принципу чемоданчика. Только понимать его нужно глубже, трактуя "совершенно необходимо" не только в чисто техническом, но и в "социальном" смысле.

Во-первых, область действия неявных объявлений полей строго ограничена - между DO и END одного оператора. Во-вторых, как было показано, становятся менее нужными переименования (с такими своими проблемами, как синонимия - доступ к одному объекту по различным именам; это очень ненадежно; почему?). В-третьих, присоединяющий оператор допускает весьма эффективную реализацию. Запись, с которой предполагается работать, можно разместить в сверхоперативной памяти, на рабочих регистрах и т.п. С этой точки зрения отказаться от присоединяющего оператора - решение сомнительное.

Однако чтобы понять, почему он совершенно необходим в Модуле-2, нужно привлечь соображения социального характера, явно учитывающие потенциальную нишу для этого языка. Присоединяющий оператор совершенно необходим в Модуле-2 потому, что он имеется в Паскале (к нему привыкли те самые пользователи, которые с большой вероятностью начнут пользоваться Модулой-2 как естественным преемником Паскаля (если угодно, его модульным диалектом)). Другими словами, у этих языков - потенциально пересекающиеся ниши.

Обратите внимание: закон консерватизма ниш в данном случае работает не против нового языка, а за него, потому что Модулу-2 следует рассматривать не как конкурента Паскаля, а как его естественное развитие, учитывающее консерватизм ниши.

На этом закончим раздел о видимости в Модуле-2.

5.7.2. Инкапсуляция

Особенно наглядно принцип чемоданчика проявляется в методе м-инкапсуляции. Обсуждая необходимость приватной части в Аде, мы привлекали реализаторскую позицию (соображения эффективности реализации - без приватной части компилятор не в состоянии распределять память под объекты приватных типов). И отмечали, что при этом нарушается согласованность с концепцией разделения спецификации и реализации, а также пошаговой детализации.

Другими словами, эффективность реализации в этом случае достигается за счет ряда нарушений общих принципов и усложнения языка (кстати, тоже нарушение общего принципа, а именно принципа чемоданчика).

Анализируем проблему по принципу чемоданчика. Что совершенно необходимо? - Инкапсуляция как средство достижения надежности и целостности. Ищем компромисс между потребностями и возможностями простых проектных решений. Находим его в отказе от особо эффективной реализации (по сути - от статического распределения памяти под инкапсулированные объекты - сравните работу с А- и м-сетями). Следствие - возможность отказаться от весьма неприятной приватной части и тем самым обеспечить соблюдение четких принципов проектирования программы (точное разделение спецификации и реализации между двумя категориями модулей).

Главная цель достигнута. Язык стал проще (и технологичнее). За счет того, что распределение памяти под составные инкапсулированные объекты должно выполняться не компилятором, а динамически - генератором "непрозрачных" указателей.

Ограничение непрозрачных типов ссылочными и отрезками предопределенных типов позволяет компилятору выделять память для них как для скаляров (по одной "единице" памяти). Остальное выполняется в динамике, процедурами модуля-экспортера. Изящное решение.

Упражнение. Докажите неформальную теорему : отсутствие приватной части в сочетании с раздельной компиляцией спецификаций и тел модулей влечет динамизм составных инкапсулированных типов. Как следствие - ограничение непрозрачных типов ссылочными или скалярными.

Об инкапсуляции все.

5.7.3. Обмен

Мы видели, как вся мощь А-модели использовалась для управления обменом. Но вся мощь потребовалась именно потому, что А-модель претендует на удовлетворение отнюдь не минимальных потребностей. Например, можно создавать файлы с совершенно произвольными (а не, скажем, лишь предопределенными) типами элементов - именно это потребовало родовых пакетов обмена. Аналогичные претензии удовлетворяются при создании драйверов специальных устройств обмена - именно это потребовало развитых спецификаций представления.

С другой стороны, для реализации драйверов привлекается универсальный аппарат управления асинхронными процессами. Когда он уже имеется в языке, такое решение может показаться даже изящным. Однако на уровне машинных команд организация взаимодействия с аппаратной задачей может существенно отличаться от организации взаимодействия "полностью программных" задач. Мы уже отмечали это, приводя пример драйвера клавиатуры.

Так что компилятор вынужден выделять драйверы и все-таки программировать их не так, как другие задачи. Выделять драйверы приходится по спецификациям представления. Итак, применение универсального аппарата асинхронных процессов для реализации драйверов заставляет сначала тщательно замаскировать то, что затем приходится столь же тщательно выискивать.

Создавать трудности, чтобы потом их преодолевать - не лучший принцип не только в программировании.

Наконец, хотя асинхронность внешних устройств - одна из причин появления в языке асинхронных процессов, совершенно не очевидно, что для создания драйверов требуется столь абстрактный (и дорогой в реализации) аппарат, как рандеву из А-модели.

Итак, с учетом ориентации Модулы-2 в основном на однопроцессорные компьютеры (но с асинхронными устройствами обмена) взглянем на управление обменом, руководствуясь принципом чемоданчика.

Что совершенно необходимо? Дать возможность писать драйверы, обеспечивающие взаимодействие основной программы с асинхронно работающим устройством обмена.

Снова ищем компромисс между потребностями в асинхронных процессах и возможностями простых проектных решений.

Находим его в отказе, во-первых, от того, чтобы, чтобы драйвер работал асинхронно с основной программой (оставаясь программной моделью аппаратной задачи), и, во-вторых, от абстрактного механизма взаимодействия относительно равноправных задач (подобного рандеву).

Действительно, минимальные потребности состоят в том, чтобы основная программа (точнее ее часть, драйвер устройства) имела лишь возможность:

Запустить устройство для выполнения конкретного обмена.

Продолжать работать, пока устройство исполняет задание.

Реагировать на завершение обмена (на факт выполнения устройством задания).

Именно такие минимальные (совершенно необходимые) возможности управления устройствами встроены в Модулу-2. Точнее говоря, то, что мы назвали аппаратной задачей, в Модуле-2 называется периферийным (асинхронным) процессом.

[Периферией обычно называют совокупность устройств обмена. В Модуле-2 имеются еще и так называемые квазипараллельные процессы или сопрограммы. Они отличаются тем, что выполняются одним исполнителем в результате явных его переключений с одной сопрограммы на другую. При этом обычно сопрограмма продолжает работать с того места, откуда в последний раз из нее поступила команда переключения на (в общем случае другую) сопрограмму.]

5.7.3.1. Периферийные процессы в Модуле-2. Пример

И в Модуле-2 обмен требует всей мощи языка. По крайней мере, существенно используется основной механизм абстракции - модули. Определяемые реализацией (так называемые "системно-зависимые") имена инкапсулированы в предопределенном модуле Система (SYSTEM). Так как транзит импортированных имен запрещен [судя по всему; в авторском описании это явно не указано, но косвенные указания имеются], то любые модули, где применяются системно-зависимые имена, должны явно импортировать модуль Система. (По этому признаку системно-зависимые модули легко распознавать).

Модуль Система экспортирует, в частности, типы "Адрес" (ADDRESS), "Слово" (WORD - машинное слово), "Процесс" (PROCESS - к этому типу относятся как сопрограммы, так и периферийные процессы), А также процедуры для работы с объектами этих типов: "НовыйПроцесс" (NEWPROCESS), "Переключить" (TRANSFER) и "ПереключитьСЗаказом" (IOTRANSFER).

Так что в системно-зависимых модулях (в том числе в драйверах) можно работать с машинными адресами, словами и процессами. Последние характеризуются двумя компонентами - телом (представленным некоторой процедурой) и рабочей областью, в свою очередь характеризуемой начальным адресом и размером (представленным натуральным числом).

Процедура

НовыйПроцесс (P, A, n, p1)

создает новый процесс (объект типа "Процесс" с телом P и рабочей областью с начальным адресом A и размером n) и присваивает его переменной p1 типа "Процесс". Новый процесс при этом не запускается (ведь процессор один), продолжает выполняться текущий процесс (основная программа также считается процессом).

Переключение на новый процесс осуществляется процедурой

Переключить (p1, p2);

При этом текущий процесс приостанавливается и присваивается переменной p1, а активным становится процесс-содержимое переменной p2. [Напомним, что это сопрограммы; p2 начинает работать с начала своего тела или с того места, где ранее приостановился.]

Процедура

ПереключитьСЗаказом (p1, p2, A);

делает то же, что и предыдущая, но еще и заказывает переключение снова на p1 после прерывания по адресу A.

Именно эта процедуоа и позволяет обеспечить потребности б) и в) из предыдущего пункта. Для этого достаточно указать в качестве p2 процесс, который должен работать асинхронно (параллельно) с устройством, а в качестве адреса A указать адрес вектора прерываний, приписанный управляемому устройству.

Тогда если непосредственно перед выполнением процедуры ПереключитьСЗаказом запустить обмен с устройством, то текущий процесс, приостановившись на этой процедуре, будет ждать прерывания, свидетельствующего о завершении обмена. При этом параллельно с устройством будет работать процесс p2. А после прерывания произойдет заказанное переключение снова на p1, т.е. на процесс, запустивший обмен с устройством (с заказом прерывания). Обычно обмен запускается засылкой единицы в соответствующий разряд регистра состояния устройства - так реализуется потребность "а" из предыдущего пункта.

Вот такими скупыми средствами реализовано в Модуле-2 управление периферийными процессами. С точки зрения языка не понадобилось вообще ничего нового, а с точки зрения пакета "Система" - всего одна процедура ПереключитьСЗаказом. Так действует принцип чемоданчика! Чтобы лучше понять взаимодействие описанных средств, приведем (с переводом идентификаторов на русский язык) модуль обмена с телетайпом из авторского описания Модулы-2.

5.7.3.2. Драйвер на Модуле-2

Чтобы все в этом примере было понятно, нужно сказать несколько слов о приоритетах процессов.

Приоритет - это целое число, характеризующее срочность процесса. Приоритет связывается с каждым модулем и с каждым устройством, посылающим прерывания. Исполнение программы может быть прервано тогда и только тогда, когда приоритет прерывающего устройства выше приоритета исполняемого (текущего) процесса. Приоритет процессора (т.е. приоритет текущего процесса) можно временно понизить процедурой УменьшитьПриоритет (LISTEN) из пакета Система. Нужно это для того, чтобы разрешить прерывания от устройств.

```
 1  MODULE Телетайп [4] ; (* приоритет этого модуля равен 4 *)
 2    FROM Система IMPORT Слово, Процесс, НовыйПроцесс,
         Переключить, ПереключитьСЗаказом, УменьшитьПриоритет ;
 3    EXPORT Печатать ;
 4    CONST N = 32 ; (* размер буфера литер *)
 5    VAR n : INTEGER ; (* текущее количество литер в буфере *)
 6    Класть, Брать : [1..N] ; (* индексы в буфере, отмечающие,
                                куда класть и откуда брать литеры *)
 7    Буф : ARRAY [1..N] OF CHAR ; (* буфер, массив литер *)
 8    Дай, Возьми : Процесс ;
 9    РабОбл : ARRAY [0..20] OF Слово ;
                              (* рабочая область драйвера *)
10    РегСост [177564B] : BITSET ; (* регистр состояния телетайпа *)
11    РегБуф  [177566B] : CHAR ; (* буферный регистр телетайпа *)
```

```
12    PROCEDURE Печатать (Лит : CHAR) ;
13    BEGIN
14      INC (n) ; (* предопределенная процедура; n := n + 1 *)
15      WHILE n > N DO УменьшитьПриоритет END ;
16      Буф [Класть] := Лит ;
17      Класть := (Класть MOD N) + 1 ; (* MOD - операция
     взятия по модулю; Индекс "Класть" циклически пробегает буфер *)
18      IF n = 0 THEN Переключить (Дай, Возьми) END ;
19    END Печатать ;

20    PROCEDURE Драйвер ;
21    BEGIN
22      LOOP
23        DEC (n) ; (* предопределенная процедура; n := n - 1 ; *)
24        IF n < 0 THEN Переключить (Возьми, Дай) END ;
25        РегБуф := Буф [Брать] ; Брать := (Брать MOD N) + 1 ;
26        РегСост := {6} ; (* шестой разряд инициирует обмен *)
27        ПереключитьСЗаказом (Возьми, Дай, 64B) ;
28        РегСост := { } ; (* обмен завершен *)
29      END ;
30    END Драйвер ;
31  BEGIN  n := 0 ; Класть := 1 ; Брать := 1 ; (* Инициализация *)
32    НовыйПроцесс (Драйвер, ADR(РабОбл), SIZE(РабОбл), Возьми) ;
       (* Предопределенные функции доставляют соответственно адрес и
          размер объекта *)
33    Переключить (Дай, Возьми) ;
34  END Телетайп ;
```

5.7.3.3. Подробности о функционировании модуля Телетайп

Представим себе применение этого модуля по такой схеме:

```
35  MODULE Печать ;
36    FROM Телетайп IMPORT Печатать ;
37    CONST M = 100 ;
38    VAR Текст : ARRAY [1..N] OF CHAR ;
          ...
39    FOR J := 1 TO M DO
40      Печатать (Текст [J]) ;
41    END ;
42  END Печать ;
```

Будем отслеживать взаимодействие компонент программы, указывая обрабатываемые (выполняемые) номера строк.

5.7.3.3.1. Инициализация

В самом начале модуля Печать происходит связывание с модулем Телетайп и выполнение его "инициализирующих" строк 31-33. Создается процесс с телом Драйвер и присваивается переменной Возьми. С этого момента Возьми используется для идентификации сопрограммы, непосредственно работающей с внешним устройством.

[Ее принципиальное отличие от процедуры Драйвер состоит в том, что переключение на Возьми означает продолжение работы сопрограммы, а не вызов процедуры Драйвер (с ее начала)].

Затем (строка 33) эта сопрограмма запускается и одновременно текущий процесс (т.е. основная программа) присваивается переменной Дай и приостанавливается (перед выполнением строки 37).

С этого момента основная программа выступает как процесс Дай, а драйвер - как Возьми. [Названия оправданы тем, что основная программа подает литеры в буфер Буф, а драйвер забирает их оттуда].

Итак, запомним, что строка 32 нужна для создания процесса Дай, а строка 33 - для создания процесса Возьми. Взаимодействие начинается.

5.7.3.3.2. Начало. Буфер пуст

После строки 33 управление достигает цикла 22 с условием n = 0, свидетельствующим о пустом буфере. Поэтому после строки 23 в строке 24 следует переключение на основную программу Дай. [Вернется оно в драйвер на строку 25!]. Так будет всегда, когда драйвер в своем основном цикле освобождает буфер и при n = -1 переключается на основную программу Дай.

Эта программа продолжается со строки 37, рано или поздно доходит до строки 40 и вызывает Печатать с очередной литерой текста. Через строку 14 при условии n = 0 проходим на 16 и кладем литеру в буфер. Строка 18 отправляет на драйвер (строка 25) при n = 0 (несколько неестественном условии; ведь в буфере имеется одна литера).

5.7.3.3.3. Основное взаимодействие. Буфер не пуст и не полон

Извлекая очередную литеру из буфера (в строке 25), драйвер запускает обмен с внешним устройством в строке 26 (присваивая его регистру состояния 1 в шестом разряде и активизируя тем самым "аппаратную задачу").

Принципиально важная для нас строка 27 приостанавливает драйвер, переключает управление на основную программу (в первый раз - на строку 19, т.е. сразу же на 39) и заказывает прерывание по концу обмена очередной литеры. Это прерывание (от телетайпа) в соответствии с семантикой процедуры ПереключитьСЗаказом приводит к переключению от Дай снова на Возьми в момент окончания обмена.

Пока идет обмен (работает аппаратная задача асинхронно с исполнением процессов Дай и Возьми), процесс Дай в цикле 39-41 может наполнять буфер. После прерывания драйвер в цикле 22-29 очищает буфер по одной литере. Это и есть основное взаимодействие процессов Дай и Возьми. При этом скорости заполнения и очистки буфера жестко не связаны.

[Вопрос. За счет чего буфер может очищаться быстрее, чем наполняться, и наоборот?].

5.7.3.3.4. Особые ситуации. Буфер полон и пуст

Основное взаимодействие прекращается, если буфер оказывается полным (в строке 15 n > N) или пустым (в строке 24 n < 0).

Когда полон, нужно дать приоритет процессу Возьми, очищающему буфер, приостановив заполняющий процесс Дай. Это реализует цикл уменьшения приоритета (строка 16). Ведь по логике модуля Телетайп заполнение буфера более чем на одну позицию возможно только одновременно с работой аппаратной задачи (собственно обменом или ожиданием ей разрешения на прерывание (убедитесь в этом!).

Поэтому переполнение буфера означает, что нужно обеспечить беспрепятственное выполнение очищающего цикла драйвера. Для этого процесс Дай и задерживается на цикле 15, в конечном итоге уступая (единственный!) процессор драйверу (при достаточном понижении приоритета). И буфер начинает очищаться.

Когда хе буфер пуст, то строка 24 переключает управление на Дай с n = -1. Это соответствует уже разобранной ситуации "Начало. Буфер пуст".

5.7.3.4. Еще одно решение

Не видно причин, почему не написать модуль Телетайп концептуально проще, изъяв строку 33 и (как следствие) попадание в зону отрицательных n (не соответствующих назначению этой переменной - считать количество литер в буфере). Найдите это решение.

Решение таково (пишем только "Печатать" и "Драйвер" при условии, что строки 33 нет).

```
PROCEDURE Печатать (Лит : CHAR);
BEGIN
  WHILE n = N DO УменьшитьПриоритет END;
  Буф[Класть] := Лит; INC (n); Класть : = (Класть MOD N) + 1;
  IF n = 1 THEN  Переключить (Дай, Возьми) END;
END Печатать;
PROCEDURE Драйвер;
BEGIN
  lOOP
    РегБуф := Буф [Брать]; DEC(n); Брать := (Брать MOD N) + 1;
    РегСост := {6}; ПереключитьСЗаказом (Возьми, Дай, 64B);
    РегСост := {};
```

```
      IF n = 0 THEN Переключить (Возьми, Дай) END;
    END (* цикла *);
  END Драйвер;
```

Упражнение. Докажите эквивалентность первому решению.

5.8. Принцип чайника

Обсуждая методы борьбы со сложностью программирования, полезно обратить внимание на принцип, интуитивно хорошо знакомый опытным программистам и выражающий своего рода защитную реакцию на сложность и ненадежность их операционной среды. Суть этого принципа хорошо иллюстрирует старый анекдот : "Как вскипятить чайник? - Берем чайник, наливаем воду, ставим на огонь, доводим до кипения. Как вскипятить чайник с водой? - Выливаем воду из чайника и сводим задачу к предыдущей!".

Почему математики так любят "сводить задачу к известной"? Потому, что для них главное - ясность ("прозрачность", "надежность") доказательства, а прямое решение новой задачи рискует оказаться ошибочным.

Но ведь и для программистов главное - надежность и понятность программы. Поэтому опытный программист без особой нужды не станет пользоваться элементами операционной среды, которые он лично не проверил.

Это относится, в частности, к использованию отдельных команд, языковых конструктов, программ, пакетов, а также языков программирования. Важными оказываются не столько их свойства сами по себе, сколько то, что программист эти свойства знает и этому своему знанию доверяет.

Если окажется возможным "свести задачу к предыдущей", она будет, как правило, решена традиционными, обкатанными методами. Намекая на упомянутый анекдот, назовем соответствующий технологический принцип "принципом чайника".

Очевидное проявление принципа чайника - долгожительство классических ЯП, в особенности Фортрана. Менее очевидное (указанное впервые Дональдом Кнутом и затем многократно подтвержденное другими исследователями) - пристрастие программистов к самым тривиальным оборотам (фразам) при использовании ЯП. Если шаг цикла, то 1; если прибавить, то 1; если присвоить, то простейшее выражение; если проверить, то простейшее отношение и т.п.

Принцип чайника помогает обосновать принцип чемоданчика (на этот раз уже с точки зрения психологии пользователя) - необязательными, чересчур изощренными конструктами будут редко пользоваться, они окажутся экономически неоправданными.

6. РЕАЛИЗАТОРСКАЯ ПОЗИЦИЯ: СТРУКТУРА РЕАЛИЗАЦИИ

В самом начале курса мы выделили пять общих позиций, с которых намеревались рассмотреть ЯП. До сих пор основное внимание уделялось технологической позиции, так как она позволяет понять, зачем нужен язык в целом и отдельные его понятия, концепции и конструкты. Остальные позиции лишь затрагивались по мере необходимости (в частности семантическая - при определении языка как знаковой системы, математическая - при фиксации определяющих соотношений наших моделей, реализаторская - в явном виде только при объяснении структуры спецификаций пакета в Аде). Обычно реализаторской позиции придают большее значение. Например, хорошая книга Т.Пратта "Языки программирования: разработка и реализация" [14] вся написана с реализаторской позиции. Настало время и нам поговорить о реализации ЯП (отдавая дань реализаторской позиции).

Замечание (методического характера).

Безусловно, возможности и требования реализации оказывают существенное влияние на свойства ЯП. Долгое время это влияние считалось (а в значительной степени и было) определяющим. С ростом возможностей аппаратуры и методов трансляции оно ослабевает, уступая технологической позиции. Основной методический тезис книги состоит в том, что подавляющее большинство свойств современных ЯП можно достаточно убедительно объяснить, не прибегая к реализаторской позиции. [А если пришлось прибегнуть - значит ценность соответствующего свойства, концепции или конструкта сомнительна.]

С другой стороны, о реализации ЯП написано много полезных книг (с точки зрения общих потребностей выпускников специализации "Б" вполне достаточно книжки Пратта). Поэтому постараемся уделить внимание лишь тем аспектам реализаторской позиции, которые в доступной литературе освещены недостаточно.

Конец замечания.

Начнем с напоминания роли "реализатора" во взаимодействии с представителями остальных пяти сторон нашего языкового пятиугольника. Реализатор призван обеспечить эксплуатацию языка на всех технологических этапах, опираясь на замысел автора.

[Такое понимание роли реализатора (и реализации) языка не стало, к сожалению, общепринятым. Иногда еще приходится бороться с устаревшей точкой зрения, что задача реализатора - обеспечить язык исполнителем (языковым процессором, транслятором) и только. Именно такая узкая "реализаторская позиция" (имеющая глубокие корни) - одна из причин положения, при котором мы вынуждены пользоваться ненадежными трансляторами, колдовать над невразумительными "диагностическими сообщениями", страдать от произвольных изменений языка, отсутствия сервиса, помогающего создавать и сопровождать программы, низкого уровня учебников, отсутствия методических материалов и т.п.].

Конечно, нам не удастся рассмотреть задачу реализатора во всей ее полноте достаточно подробно. Поэтому поступим так же, как в случае технологической позиции. Как вы помните, мы кратко рассмотрели жизненный цикл изделия в целом, а затем выделили только проектирование как представительный этап цикла. Аналогичным образом дадим общее представление о задаче реализации языка в целом, а затем выделим лишь один аспект реализации и займемся только им.

Итак, будем считать, что реализация в целом должна обеспечить эксплуатацию языка на всех этапах ЖЦКП (жизненного цикла комплексного программного продукта). Рассмотрим три этапа (стадии) жизненного цикла - проектирование, эксплуатацию и сопровождение продукта. Их достаточно, чтобы выделить важнейшие компоненты реализации.

6.1. Компоненты реализации

Будем исходить из того, что авторское определение ЯП имеется (для базового языка индустриального программирования в настоящее время это обычно отраслевой, национальный или международный стандарт; в других случаях определение языка может иметь менее высокий официальный статус).

К авторскому определению предъявляются исключительно высокие требования. Их обсуждение выходит за рамки нашего курса. Но одно из них стоит сформулировать. Авторское определение в идеале должно исчерпывающим образом фиксировать синтаксис и семантику языка. Другими словами, оно должно быть способно служить единственным источником сведений о допустимых языковых конструктах и их смысле. Поэтому можно ожидать (и опыт уверенно подтверждает), что авторское определение не пригодно в качестве методического материала (а тем более, учебника) по созданию программ на этом языке. Точно, понятно и полно описать ЯП - столь непростая задача, что не стоит ее усложнять погоней за двумя зайцами.

Реализация с точки зрения этапа проектирования. Чтобы обеспечить эксплуатацию языка на этапе проектирования программы, требуется скорее методический материал, чем авторское определение. Нужда в нем особенно очевидна в случае базового языка индустриального программирования, ориентированного на массовое применение. Недаром в случае с Адой первые учебники появились практически одновременно с официальным определением языка (среди них - уже упоминавшийся учебник Вегнера). Итак, первая важнейшая компонента реализации, необходимая в особенности при проектировании программы - это методическое руководство (учебник) по программированию на рассматриваемом ЯП. Конечно, учебником не исчерпываются потребности этапа проектирования, которые призвана удовлетворять квалифицированная реализация.

[В последние годы появились, в частности, программные средства, поддерживающие пошаговую детализацию, проверку правильности, создание тестов, управление проектом и другие элементы проектирования.]

Реализация с точки зрения этапа эксплуатации. Сразу ясно, что здесь не обойтись без исполнителя, соответствующего ЯП. Причем не абстрактного, мыслимого, а вполне конкретного, материального, обладающего достаточными физическими ресурсами и приемлемыми характеристиками эффективности (ресурсоемкости). Как известно, в настоящее время исполнители для ЯП представляют собой комплекс аппаратуры и программных продуктов, называемых трансляторами. Будем считать, что создание аппаратуры выходит за рамки задач, связанных с реализацией конкретного языка (хотя имеется тенденция к изменению этого положения). Тогда в качестве второй важнейшей компоненты реализации, выделим транслятор - без него невозможно

обеспечить этап эксплуатации программы. Снова ясно, что потребности этого этапа не исчерпываются транслятором.

[В частности, нужна операционная система, обеспечивающая нормальное функционирование аппаратуры, нужна так называемый резидент, обеспечивающий нормальное выполнение целевой программы и т.п.]

Реализация с точки зрения этапа сопровождения. Анализируя этап сопровождения, обратим внимание на основную технологическую потребность этого этапа - корректировать программу с минимальным риском внести ошибки. Вы знакомы со средствами редактирования текстов (редакторами), позволяющими вносить изменения в исходные программы. Риск ошибиться уменьшается, если редактор "знает" ЯП и позволяет вносить исправления "в терминах ЯП". Например, такому "языковому" редактору можно дать задание "в процедуре Р заменить формальный параметр А на В".

Сравните указание обычному редактору "заменить А на В" и соответствующий риск заменить "не то" А. Итак, третьей важнейшей компонентой квалифицированной реализации служит языковый редактор.

Совсем хорошо было бы вносить исправления не "в терминах языка", а "в терминах решаемой задачи" (тогда редактор должен был бы "знать" и язык, и проблемную область, и задачу). Но это - пока дело будущего.

Итак, беглого взгляда на три этапа жизненного цикла программы хватило для выделения трех важшейших компонент реализации ЯП: учебника, транслятора и редактора.

Другие компоненты. Не вдаваясь в подобности, перечислим еще некоторые компоненты. Ограничимся только компонентами, непосредственно связанными с языком, считая, что реализация погружена в некоторую многоязыковую систему программирования (типа ОС ЕС или Мониторной системы "Дубна"), предоставляющую необходимые общесистемные услуги, если они не определены в языке (базу данных, связь с другими языками, фонды готовых программ, документов и т.п.).

Укажем этапы жизненного цикла, где применение называемых компонент особенно целесообразно (хотя очевидно, что они полезны и для других этапов, в том числе и выпавших из нашего рассмотрения).

По этапу проектирования - различного рода процессоры, помогающие готовить тексты исходных программ. Примерами могут служить уже упомянутые препроцессоры, поддерживающие метод пошаговой детализации программ, "знающие" определенный язык программирования. Они в состоянии воспринять запись шагов детализации и выдать текст законченной (или еще незаконченной) программы, попутно контролируя его правильность (в диалоговом режиме, если нужно). Полезны процессоры, позволяющие писать на структурных расширениях Фортрана, Кобола, ПЛ/1 и других "заслуженных" языков. Еще один класс компонент реализации - отладчики.

По этапу эксплуатации - средства контроля и измерений как программ, так и трансляторов. Это система тестов, проверяющих соответствие исполнителя определению языка, оптимизаторы и конкретизаторы, настраивающие программы на определенные условия эксплуатации.

По этапу сопровождения - уже упоминавшиеся измерительные средства; средства для отслеживания и контроля изменений (версий); контролеры программ, проверяющие соответствие стандартам (особенно важно для переноса программ в другую среду) или выявляющие особо ресурсоемкие места.

Кроме того, следует понимать, что развитая реализация может содержать учебники для различных категорий пользователей, трансляторы с различными предпочтительными режимами эксплуатации (особо быстрый, особо надежный, особо оптимизирующий) и, конечно, для различных машин, редакторы с различными уровнями "интеллекта" и т.п.

Итак, будем считать достаточно обоснованным следующий тезис: квалифицированная реализация ЯП - дело сложное, дорогое, длительное и многоплановое. От качества реализации в этом широком смысле слова зависят "потребительские свойства" ЯП. Реализация - один из наиболее очевидных аспектов, переводящих понятие "язык программирования" из категории научно-технической в социальную категорию.

[Дополнительную яркую социальную окраску этому понятию придает весьма своеобразное образование - пользователи определенного ЯП, иногда официально объединенные в ассоциации. Так что ЯП, тем более базовый язык индустриального программирования в наше время - явление социальное и именно такого подхода к себе и требует.]

Закончим теперь заниматься реализацией в целом. Сконцентрируем внимание на более традиционной ее части - трансляторах, точнее, компиляторах.

6.2. Реализаторская позиция: компиляторы

Компилятором называется программный продукт, предназначенный для перевода с исходного ЯП на целевой (объектный) язык (обычно - язык загрузки или иной язык, близкий к машинному).

Если для целевого ЯП исполнитель имеется, то за счет компилятора появляется возможность выполнять исходные программы в два этапа. На первом этапе - компиляция (трансляция) - исходная программа переводится на целевой язык с помощью компилятора; на втором - этапе исполнения - исполителем целевого языка выполняется переведенная (оттранслированная, целевая, объектная) программа.

Современные языки индустриального программирования, как можно видеть на примере Ады, ориентируются прежде всего на технологические потребности пользователей и поэтому довольно сильно отличаются от наиболее распространенных машинных языков. Вместе с тем, как мы видели, в них многое сделано для того, чтобы можно было позаботиться о надежности и эффективности целевых программ еще на этапе компиляции (вспомните квазистатический аппарат прогнозирования - контроля). По этим двум причинам компиляторы (а не, скажем, интерпретаторы) служат обязательными компонентами реализации практически всех языков индустриального программирования.

Мы не стремимся дать исчерпывающее определение компилятора. Дело в том, что это понятие скорее инженерное, чем математическое. Во всяком случае, хороший компилятор должен не только "переводить", но и сообщать об ошибках, и накапливать статистические сведения об обработанных программах, и оптимизировать свою работу с учетом особенностей потока программ. Возможны и иные требования (гибкое управление свойствами объектной программы, печатью листинга, диагностическими режимами и пр.).

Создать компилятор - дело очень непростое. Высококачественный компилятор с современного языка программирования требует нескольких лет работы и может содержать сотни тысяч команд. При этом не случайно не названо количество требуемых специалистов. Несколько лет нужно независимо от того, можно ли выделить на это 10 человек или 200. Близкая к оптимальноой - группа из 5 - 15 человек.

Увеличение группы только удлиннит сроки или приведет к полному краху (закон Брукса); если не удается найти для новых людей совершенно независимой работы (такой, скажем, как создание системы тестов, проверяющих качество компилятора)).

Технологии создания компиляторов посвящена огромная литература. Выделены важнейшие технологические этапы, основные компоненты компилятора, предложены многочисленные методы реализации отдельных компонент, имеются автоматизированные системы, предназначенные для создания компиляторов.

Их успешно применяют в относительно простых случаях, когда сами переводы не слишком сложны и к ресурсоемкости компиляторов не предъявляют жестких требований. В таких условиях два-три специалиста с помощью соответствующей инструментальной системы могут изготовить компилятор примерно за месяц интенсивной работы.

Однако ЯП развиваются, требования к качеству реализации ужесточаются, возможности аппаратуры растут. В результате разработка компиляторов для языков индустриального программирования продолжает требовать значительных творческих усилий (правда, теперь чаще приходится не столько "изобретать" методы компиляции, сколько квалифицированно "выбирать" из имеющегося арсенала).

Полноценные учебники по созданию компиляторов еще ждут своих авторов. Много интересного и полезного на эту тему можно найти в книгах [14-16].

6.3. Основная функция компилятора

Рассмотрим лишь одну, выделяемую традиционно, функцию компилятора - строить целевую программу. Выделяется она потому, что лучше других отражает специфику компилятора и соответствует его основному назначению. Однако и остальные функции компилятора в определенных условиях могут оказаться исключительно важными. Например, для студентов важнейшей может оказаться диагностическая функция, т.е. способность компилятора помогать отлаживать программу.

Итак, будем считать, что основная задача компилятора - перевести программу с исходного языка на целевой.

Обозначим через LL1 - исходный язык, а через LL2 - целевой язык для планируемого компилятора. Пусть L1 - множество текстов, допустимых в LL1 (т.е. определяемых синтаксисом LL1), а L2 - множество текстов, допустимых в LL2.

Переводом (проекцией) с языка LL1 на язык LL2 называется отношение p из L1 в L2 ,т.е. подмножество декартова произведения L1 * L2.

Легко догадаться, что всякий компилятор характеризуется единственной проекцией (обратное неверно!). Этой проекции принадлежат те и только те пары

(t1 ,t2)

где t1 из L1, t2 из L2, для которых t2 может быть получен в результате применения компилятора к t1.

Данное выше определение проекции в виде отношения подчеркивает факт, что компилятор может переводить не все тексты из L1 (например, для слишком длинных текстов может не хватить ресурсов), переводить различные тексты в один (например, если они обозначают одно и то же), переводить один и тот же текст по-разному (скажем, в зависимости от режима трансляции - с оптимизацией или без нее).

[Данное определение проекции выглядит совершенно симметричным относительно языков LL1 и LL2, хотя они содержательно играют различные роли. Чтобы подчеркнуть эти роли, иногда говорят, что проекция - частичное многозначное отображение из L1 в L2

p : L1 -> L2].

6.4. Три принципа создания компиляторов

Небольшой опыт по созданию компилятора у нас уже есть. В модифицированной модели Маркова мы практиковались в создании компилятора с языка обычных (инфиксных) выражений в польскую инверсную запись (в язык постфиксных выражений).

Наш компилятор представлял собой программу из четырех предложений:

```
{ e1 + e2 } R -> { e1 } { e2 } +   .
{ e1 * e2 } R -> { e1 } { e2 } *   .
{ ( e ) } -> { e }  .
{ e } -> e  .
```

Проекция p, соответствующая этому компилятору, должна удовлетворять определяющему соотношению

p(F1 op F2) = p(F1) p(F2) op

где F1, F2 - правильные инфиксные формулы, op - операция.

Уже на примере такого простого компилятора можно продемонстрировать три важные положения.

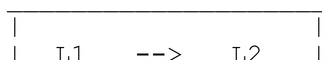
Во-первых, созданию компилятора должна предшествовать разработка связанной с ним проекции. Это не обязательно означает, что проекция полностью фиксируется до начала программирования компилятора. Но, во всяком случае, ее замысел предшествует разработке соответствующего алгоритма перевода и последовательно уточняется, определяя всю разработку компилятора. Например, четыре предложения нашего компилятора не могли быть написаны, если бы мы фактически не "держали в голове" приведенное определяющее соотношение для проекции.

Эти соображения можно оформить в виде проекционного принципа: создание компилятора можно разбить на два технологически независимых этапа - П-этап или этап разработки проекции и А-этап или этап алгоритмизации проекции. Полученное на П-этапе описание проекции (например, в виде системы определяющих соотношений) может служить техническим заданием (спецификацией) для работы на А-этапе. Опыт показывает, что в некоторых практически важных случаях А-этап удается полностью автоматизировать. Делаются попытки частично автоматизировать и П-этап (например, в созданной под руководством профессора В.Н.Редько системе построения трансляторов (СПТ) Дефипс). Это удается за счет предварительного формального описания как исходного, так и целевого языка сходным образом на одном и том же метаязыке. В отличие от знакомой вам БНФ такой метаязык должен позволять описывать не только синтаксис, но и семантику языков. В сущности, при этом приходится создавать проекцию описываемого языка на метаязык. Так что П-этап всегда носит творческий характер. К реальным языкам индустриального программирования автоматизация П-этапа пока неприменима из-за непрактичности метаязыков и соответствующих СПТ.

Во-вторых, когда проекция достаточно проработана и можно приступать к ее алгоритмизации, полезно выделять две фазы компиляции - фазу анализа и фазу синтеза. В нашем примере мы воплощали первую фазу левой частью рефал-предложения, вторую - правой частью.

В отличие от исходного и целевого языка рефал выступает в нашем компиляторе в роли инструментального языка, т.е. языка, на котором написан компилятор.

Тройственную связь исходного, целевого и инструментального языков удобно изображать Т-образной диаграммой



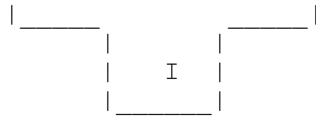


Рис. 6.1

С ее помощью легко описываются довольно сложные процессы компилиции (в частности, так называемая раскрутка, применяемая при переносе компиляторов).

В-третьих, анализ и синтез далеко не всегда удается столь четко сопоставить некоторому определенному конструкту инструментального языка, как это сделано в нашем простом примере.

Дело в том, что в рефале непосредственными составляющими при анализе выражения могут быть только выражения, термы и символы. При компиляции с более сложных исходных языков приходится переводить операторы, объявления, области действия и т.п. Анализ исходного текста и синтез соответствующего целевого не удается представить в этих случаях одним предложением. И для анализа, и для синтеза приходится писать специальные подпрограммы.

В первых компиляторах взаимодействие таких подпрограмм было довольно запутанным. Но уже в начале 60-х годов был предложен (Айронсом) принцип упорядочивания этого взаимодействия на основе иерархической структуры исходных текстов. Структура эта задается синтаксисом исходного языка, поэтому сам принцип получил название принципа синтаксического управления трансляцией (компиляцией, в частности).

В синтаксически-управляемых компиляторах синтаксическим категориям исходного языка ставятся в соответствие так называемые семантические действия. Они то и синтезируют целевой текст в процессе так называемой структурной индукции.

В этом процессе семантические действия, соответствующие определенным синтаксическим категориям, используют результы семантических действий, соответствующим непосредственным компонентам этих категорий. Структура, по которой ведется индукция, строится в процессе анализа (декомпозиции) исходного текста в соответствии с определением исходного языка.

Принцип синтаксического управления и структурную индукцию можно в первом приближении понять на примере нашего компилятора для перевода выражений.

При этом левые части рефал-предложений выполняют декомпозицию (выделяя сумму, произведение, скобочную первичную формулу), а правые части выполняют структурную индукцию, пользуясь уже готовыми переводами компонент соответствующих синтаксических категорий.

В нашем компиляторе анализ и синтез чередуются (т.е. компилятор, как говорят, однопроходный). Это не обязательно. Можно сначала полностью проанализировать исходный текст, получив в результате его структуру (обычно в виде так называемого синтаксического дерева), а затем (как говорят, на втором проходе) выполнить структурную индукцию. Иногда применяют и большее число проходов (обычно при ограничениях на память для размещения компилятора или при необходимости оптимизировать программу).

Итак, мы выделили один из принципов, позволяющий структурировать процесс создания транслятора - проекционный принцип; один из принципов, позволяющих структурировать сам компилятор - принцип явного выделения анализа и синтеза, и один из принципов, позволяющих структурировать синтез - принцип синтаксического управления (несколько упрощая, можно отождествить его с принципом структурной индукции).

[Отметим, что термин "структурная индукция" обозначает также один из способов доказательства свойств структурированных объектов.

Было бы очень полезно рассмотреть более представительный пример компилятора и на нем продемонстрировать выделенные принципы в действии. Однако пока придется ограничиться примером разработки только одной компоненты компилятора - синтаксического анализатора. Зато это та самая компонента, за счет которой становится возможным синтаксическое управление.

К тому же мы построим в конечном итоге параметрический анализатор. Его параметром будет описание синтаксиса конкретного языка. Тем самым будет обозначено целое направление : параметрические компиляторы. Чтобы получить из такого абстрактного компилятора конкретный, нужно задать в качестве параметра не только описание синтаксиса, но и полное определение исходного языка или даже описание проекции с исходного языка на целевой. При подходящем выборе метаязыка и класса

проекций это оказывается более простым делом, чем написать компилятор для нужной пары языков. Такие параметрические (в том числе проекционные) системы построения трансляторов (СПТ) уже применяются на практике (например, в области создания трансляторов для станков с числовым программным управлением). Однако эта исключительно интересная тема выходит за рамки курса].

6.5. Пример параметрического синтаксического анализатора

Покажем, как построить синтаксический анализатор, применимый для широкого класса языков. В качестве модельного языка (представителя рассматриваемого класса) выберем уже знакомый нам язык Дейк.

[* это уже было в доказательном программировании*] Начнем с описания синтаксиса языка Дейк с помощью несколько модифицированной БНФ. И синтаксис, и БНФ удобно объяснять в процессе пошаговой детализации строения исходной абстракции (понятия, синтаксической категории) "программа". На каждом шаге детализации при этом указываются допустимые конкретизации промежуточных абстракций (синтаксических категорий).

В зависимости от интерпретации применяемые при этом правила можно считать или соотношениями, связывающими классы текстов (формальные языки), или правилами вывода в порождающей системе (контекстно-свободной грамматике).

Ниже выписаны 16 шагов такой детализации (т.е. 16 правил модифицированной БНФ).

```
          Естественный синтаксис языка Дейк.
1. программа = `begin '{объявление `;'} {оператор `;'} `end '.
2. объявление = (`var' | `arr') имя {`,' имя}.
3. оператор = присваивание | развилка | цикл | `null'.
4. присваивание = переменная `:=' выражение.
5. переменная = имя [индекс].
6. индекс = `['выражение`]'.
7. выражение = переменная | число | функция.
8. имя `(' выражение {`,' выражение }`)'.
9. развилка = `if'{вариант}`fi'.
10.цикл = `do'{вариант}`od'.
11.вариант = условие `-->' {оператор `;'}.
12.условие = выражение ( < | < | = | = | > | > ) выражение.
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

13.имя = буква { буква | цифра }.               |
14.число = цифра { цифра }.                     |
15.буква = `a'| . . . |`z'.                     | - лексемы
16.цифра = `0'| . . . |`9'.                     |
```

Примененная модификации БНФ сответствует предложениям Вирта (принята в качестве английского национального стандарта). Отличия от оригинальной БНФ (использованной нами в разделе о моделях языков программирования) сводятся, во-первых, к тому, что выделяются специальными скобками не названия абстракций (синтаксических категорий, метасимволов), а конкретные символы так называемого терминального алфавита. В языке Дейк терминальный алфавит состоит из букв, цифр и символов begin, end, var, arr, do, od, if, fi, null (подобно Алголу). В качестве таких символов применяются кавычки. Во-вторых, применяются круглые, квадратные и фигурные скобки. Круглые, чтобы сгруппировать несколько вариантов конкретизации (несколько альтернатив). Квадратные - чтобы указать, что их содержимое можно опускать. Фигурные - чтобы указать, что их содержимое можно выписывать подряд нуль или более раз в процессе пошаговой детализации (или, как говорят, порождения) конкретной программы.

Некоторый текст признается допустимой программой на языке Дейк тогда и только тогда, когда его можно получить последовательной конкретизацией (породить) по указанным 16 правилам из исходной абстракции "программа". Такой текст называется выводимым из метасимвола "программа".

Например, текст

```
    begin var x, i, n;
      x:=a[1] ; i:=1 ;
      do i<n --> i:=plus(i,1) ;
```

```
                    if a[i]>x --> x:=a[i] ;
                       a[i]<x --> null ; fi ;
         od ;
       end
```

допустим в языке Дейк, а если вместо plus(i,1) написать "i+1", то получится недопустимый текст (так как "выражение" может быть только "переменной", "числом" или "функцией").

Аналогичным образом можно начинать пошаговую детализацию (порождение, вывод) с любого промежуточного уровня абстракции, получая тексты, которые называются порождениями соответствующего метасимвола (терминальными порождениями, если они состоят только из терминальных символов).

Например, текст

(1) if вариант вариант fi

выводим из метасимвола "развилка" и служит его порождением (но не терминальным).

Текст

(2) a[i]>x --> x := a[i] ;

и текст

(3) a[i]<x --> null ;

служат различными порождениями метасимвола "вариант" (причем терминальными его порождениями), а текст

```
if a[i] > x --> x := a[i] ; a[i] < x --> null ; fi
```

терминальным порождением метасимвола "развилка".

Всякому выводимому тексту соответствует синтаксическое дерево (дерево вывода, дерево порождения, дерево последовательной детализации). Например, тексту (1) соответствует дерево

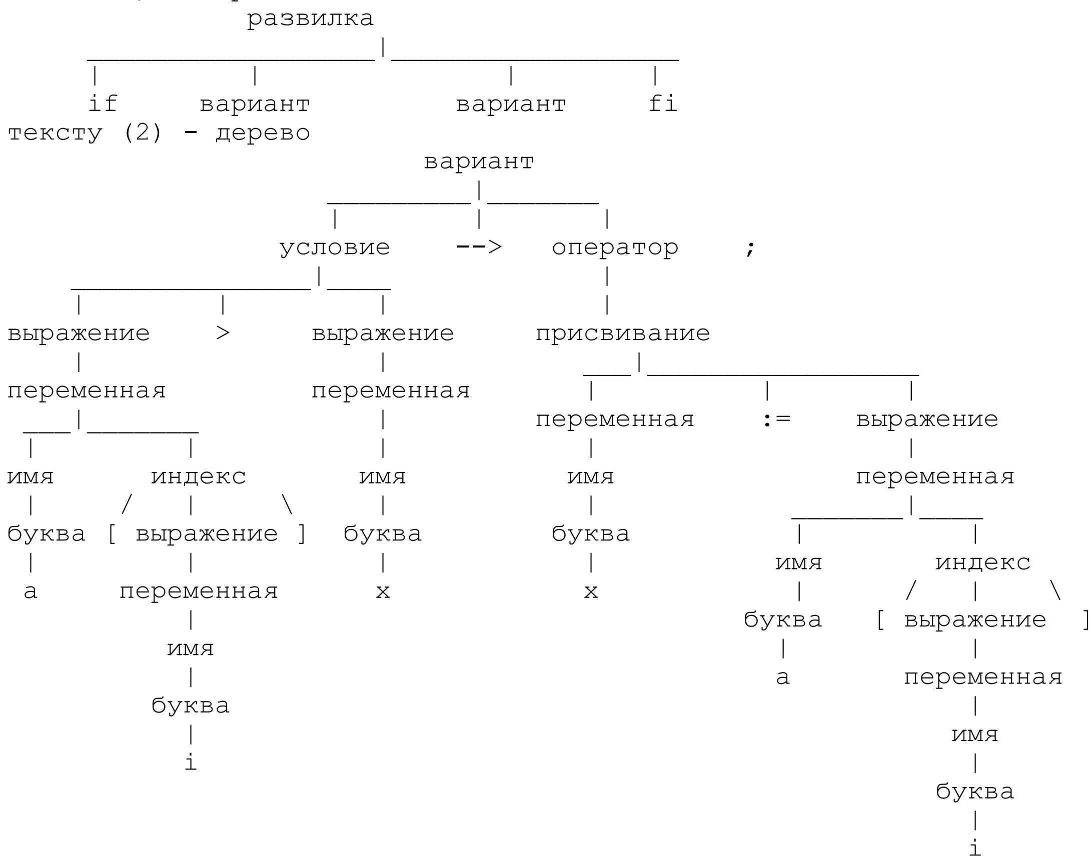


Рис. 6.2

Итак, будем считать понятным, каким образом синтаксис языка Дейк определяет класс допустимых программ и их древовидную структуру.

6.6. Задача анализа

Задача анализа формулируется как задача, обратная пошаговой детализации (конкретизации). Имея конкретный текст в терминальном алфавите, требуется восстановить шаги его порождения или убедиться, что оно не возможно. Другими словами, требуется попытаться последовательно абстрагироваться от конкретного вида порождений, выделяя подходящие понятия (синтаксические категории), пока не

получится исходное понятие ("программа") или не станет ясно, что при любых возможных вариантах последовательного выделения понятий исходное понятие (цель) не получить. Короче говоря, требуется по заданному конкретному тексту построить его синтаксическое дерево или указать, что это не возможно.

В реальных условиях задача сложнее - нужно еще выдавать разумную диагностику, объясняющую, почему построение дерева оказалось невозможным.

Упростим нашу задачу.

Во-первых, не будем заниматься диагностикой.

Во-вторых, отделим четыре последние правила (с 13 по 16) и будем считать, что понятия "имя" и "число" являются лексемами и выделяются лексическим анализатором. Он же сворачивает лексемы begin, end и т.п.

Тем самым "терминальными" символами в нашем синтаксисе можно считать метасимволы "имя" и "число". Подобный прием широко применяется на практике (он позволяет использовать различные методы анализа для различных классов метасимволов). Концепция терминальных и метасимволов становится при этом относительной, а сам синтаксис (иногда говорят, "грамматика"), в котором допустимы "терминальные" метасимволы, называется относительным (относительной).

Как мы чуть позже увидим, применяемые анализаторы накладывают ограничения на вид синтаксических правил. Еще одно важное преимущество относительных грамматик в том, что они позволяют по разному доопределять "терминальные" метасимволы без изменения относительного синтаксиса и, следовательно, без нарушения ограничений анализатора.

Третье упрощение состоит в том, что не будем требовать полного построения синтаксического дерева. Представлять такое дерево можно по-разному. Сопоставим каждой альтернативе обозначение вида

X(j),

где X - название синтаксической категории, а j - номер альтернативы. Анонимные альтернативы будем обозначать звездочкой '*'; будем опускать индекс, когда альтернатива - единственная.

Тогда нетрудно убедиться, что "польская" запись синтаксического дерева (когда обозначения альтернатив помещаются непосредственно после записи соответствующих поддеревьев), позволяет однозначно его восстановить. Например, последовательность :

индекс переменная выражение *(6) переменная выражение условие переменная индекс переменная выражение присваивание оператор вариант

- однозначно описывает дерево текста (2) до уровня лексем.

Обозначения альтернатив часто называют семантическими символами или просто "семантиками", так как им естественно соответствуют семантические действия, выполняемые при структурной индукции.

Итак, будем считать, что задача анализа - выдать по исходному тексту последовательность семантик.

6.7. Синтаксический анализатор для разделенных грамматик

Замечание.

Изобретено много методов синтаксического анализа, отличающихся разнообразными характеристиками. Одни применимы к любому синтаксису, написанному на БНФ (или, как говорят, к любой контекстно-свободной грамматике; слово "контекстно-свободной" подчеркивает факт, что порождение метасимвола в таких грамматиках, как мы видели, не зависит от контекста этого символа. Поэтому с помощью БНФ не удается описать так называемые контекстные условия. Например, нельзя описать требование, чтобы типы левой и правой частей присваивания совпадали. Проверка подобных ограничений - дело так называемого контекстного анализатора. Так что мы выделили уже три фазы анализа - лексический, синтаксический (контекстно-свободный) и контекстный. Вместе с фазой синтеза, выполняемой синтезатором (генератором) объектной программы, они образуют четыре классические фазы компиляции).

Есть методы анализа, применимые не ко всем грамматикам, зато особо быстрые. Другие требуют мало памяти, третьи легко настраиваются на нужную грамматику.

Рассмитрим один из самых простых методов, к тому же обеспечивающий высокую скорость анализа.

Конец замечания.

Итак, задача анализатора - последовательно выделять понятия, продвигаясь по исходному тексту.

Обычно ведут анализ так, чтобы постоянно иметь перед собой текущую цель (понятие, которое нужно выделить), и указатель на текущую просматриваемую позицию в анализируемом тексте, В исходном состоянии цель - это "программа", а указатель стоит в первой позиции текста.

Если у цели несколько альтернатив, то нужно понять, какая из них могла быть применена.

Основная идея метода, который мы рассмотрим, состоит в следующем.

Допускаются только такие синтаксисы (грамматики), в которых по цели и символу, стоящему в текущей позиции (текущему символу), альтернатива всегда выбирается однозначно. Тогда следует ее выбрать и согласованно продвигаться как вдоль альтернативы, так и вдоль текста.

Встречая в альтернативе терминальные символы, нужно проверять, что текущие символы текста им соответствуют, а встречая в альтернативе метасимвол, нужно выполнить для него анализ как для новой текущей цели (отправив старую в стек, чтобы вернуться к ней после завершения анализа новой цели).

Если очередной терминальный символ альтернативы не соответствует текущему символу текста или текст оборвется до того, как будет выделена исходная цель, то обнаружена ошибка (ведь каждый раз выбирается единственная возможная альтернатива). Вот и вся идея.

Как видим, метод довольно прост, но прост именно за счет возможности уверенно выбрать альтернативу по одному текущему символу (никогда не приходится вовращаться по тексту или альтернативам, чтобы проверять другие варианты, если альтернатива не подойдет).

В случае произвольного контекстно-свободного синтаксиса альтернативы текущей цели могут иметь общее начало. Например, в алгольском определении условного оператора:

```
условный_оператор = оператор_если ! оператор_если else оператор !
                условие оператор_цикла ! метка ': ' условный_оператор.
условие = if логическое_выражение then .
оператор_если = условие безусловный_оператор .
```

Здесь до конца "условия" нельзя понять, какую из трех первых альтернатив выбрать, а до конца оператора_если непонятно, какую выбрать из первых двух.

Ситуация, когда анализатор обнаруживает, что с выбранной альтернативой продолжать анализ нельзя и нужно вернуться по тексту назад, занявшись еще не проверенными альтернативами, называется тупиком. На перебор альтернатив с возвратами из тупиков тратится много времени. Поэтому на практике предпочитают быстрые беступиковые методы анализа. К ним относится и метод, который мы рассмотрели.

6.8. (Строго) разделенные грамматики

Бывают различные ограничения на синтаксис, обеспечивающие возможность беступикового анализа. Простейшее состоит в том, чтобы в каждом правиле все его альтернативы начинались различными терминальными символами.

Синтаксисы (грамматики), а также отдельные правила, удовлетворяющие этому требованию, называются (строго) разделенными.

Очевидно, что в строго разделенном синтаксисе заданному текущему символу всегда соответствует не более одной альтернативы.

Но очевидно также, что естественный синтаксис не только ЯП класса Алгола, но и языка Дейк не является разделенным.

Оказывается, что для языков, используемых на практике, обычно несложно построить разделенный синтаксис. Конечно, он должен описывать те же допустимые тексты, что и естественный синтаксис (быть ему эквивалентным). Более того, последовательность "старых" семантик должна быть сохранена для каждого допустимого текста (это свойство называется сильной эквивалентностью синтаксисов).

Свойство сильной эквивалентности гарантирует, что разделенный синтаксис пригоден для той же структурной индукции, что и естественный (так что при переходе к более быстрому (разделенному) методу анализа семантические действия менять не придется).

6.9. Разделенный синтаксис языка Дейк

Можно написать и строго разделенный синтаксис языка Дейк. Но он неприятно громоздкий. Тем более, что требование разделенности должно касаться не только явных, но и скрытых альтернатив (содержимого фигурных и квадратных скобок).

Чтобы сделать синтаксис компактнее, сделаем ограничение менее жестким.

Во-первых, позволим начинаться с метасимвола тем альтернативам, которые единственны в правиле (ясно, что это не мешает всегда сразу выбирать единственную альтернативу - основное условие беступиковости анализа сохранено).

Во-вторых, разрешим пользоваться специальным символом "коф". Будем считать его особым терминальным символом. Этот символ может появляться только в самом начале последней альтернативы правила, а его порождение считается пустым.

Правило называется разделенным (не строго), если в нем либо одна альтернатива, либо все начинаются различными терминальными символами ("коф" может начинать только последнюю альтернативу).

Кроме того, в разделенном синтаксисе не должно быть круглых и квадратных метаскобок, а порождение фигурных не может начинаться тем же терминальным символом, который может быть порожден непосредственно после этих скобок.

Нетрудно понять, что все это условия, обеспечивающие возможность по одному текущему символу выбрать единственный допустимый вариант продолжения анализа.

6.9.1. Преобразование естественного синтаксиса в разделенный

Следующие ниже разделенные правила демонстрируют основную идею преобразования естественного синтаксиса в разделенный. Для этого нужно ,во-первых, устранить метасимволы в начале альтернатив и, во-вторых, избавиться от квадратных скобок.

```
3'. оператор = имя конец_присваивания ! 'if 'конец_развилки !
                    'do ' конец_цикла ! 'null '.
     7'. выражение = число ! имя конец_перем_или_функции.
     7". конец_перем_или_функции = '('выражение {',' выражение} ')'.
                                   ! коф .
     5'. переменная = имя индекс_0.
     5". индекс_0 = '[ ' выражения '] '  ! коф.
```

Напомним, что коф - особый "метасимвол-терминал", а "имя" и "число" мы также договорились считать терминальными.

Упражнение.

Придумайте прием избавления от круглых и квадратных скобок.

Подсказка.

```
       Все группы (круглые скобки) заменены по правилу
       (A1!...!An)g ==> A1g!A2g!...!Ang
  Все вставки (квадратные скобки) - по правилу
       [A] ==> A_0
            A_0 = разделенное_A! коф
```

Упражнение (повышенной сложности). Сформулируйте "разделяющее ограничение" для синтаксиса с семантиками.

Подсказка. Семантика похожа на коф. (Почему?).

6.10. Разделенный анализатор, написанный на Аде

6.10.1. Общая структура

Теперь напишем анализатор, применимый к любому разделенному синтаксису. Будем для простоты считать, что фигурных скобок нет. Можно модифицировать этот анализатор, чтобы в синтаксисе были допустимы и фигурные скобки (итерации).

Допустим, что основная идея анализатора понятна. Осталось воплотить ее в виде программы на Аде.

Применим снова метод пошаговой детализации. Однако из-за недостатка времени шаги будут относительно крупными.

Сначала представим себе общую структуру фазы анализа, затем детализуем синтаксический анализатор в расчете на последующее уточнение всех использованных в нем связей с контекстом, затем до некоторой степени уточним контекст и на этом остановимся.

Общая структура фазы анализа (с точки зрения синтаксического анализатора) такова.

Во-первых, имеется программный контекст, представляющий очередные лексемы исходного текста (его естественно оформить на Аде пакетом "лексика"). Во-вторых, имеется контекст, потребляющий вырабатываемые синтаксическим анализатором очередные семантики (его естественно оформить пакетом "семантика"). Наконец, в-третьих, сам синтаксический анализатор работает в контексте, предоставляющем ему услуги по чтению синтаксических формул. Этот контекст естественно оформить пакетом "синтактика".

Подчеркнем, что лексический, синтаксический и семантический анализ полезно и естественно оформлять как асинхронные процессы. Как мы уже видели, связь между подобными процессами можно осуществить с помощью очередей. Нам понадобится очередь лексем, связывающая лексический анализатор с синтаксическим, и очередь семантик, связывающая синтаксический анализатор с семантическим.

6.10.2. Детализация синтаксического анализатора

Приступим теперь к детализации синтаксического анализатора, считая, что все используемое в нем определено в подходящем контексте (контексты уточним потом). Оформим синтаксический анализатор как вторичный компилируемый модуль, первичным для которого служит пакет "синтактика".

```
separate (синтактика) task body анализатор is
   S : лексема ; -- для текущих лексем
   своб_S : BOOLEAN := true ; -- признак того, что
                              -- очередная лексема в S
                              -- еще не поступила или уже обработана.
   удача : BOOLEAN := true ;  -- признак того, что удалось выделить
          -- понятие, соответствующее читаемой формуле.
          -- За счет разделенности неудача при чтении самой
          -- внутренней (последней) из читаемых формул означает
          -- неудачу при чтении всех внешних еще не законченных
          -- формул. Поэтому можно ограничиться лишь одним
          -- признаком.
   рез : сем_запись ; -- для очередных семантик, т.е. результатов
                      -- работы синтаксического анализатора.
procedure пройти_формулу (начало : указатель) is
  -- рекурсивная процедура, читающая формулы (правила БНФ).
  -- параметр указывает место в синтаксисе, где формула начинается.
   R : указатель := начало ;  -- это указатель на текущую цель.
  -- Текущих целей может быть много. За счет рекурсивного
  -- синтаксиса - сколь угодно много; образуется стек текущих целей.
begin   -- итак, нужно выделить понятие, соответствующее текущей
        -- цели, начиная с текущей лексемы;
   loop -- по элементам формулы ;
        -- существенные элементы : метасимвол, терминал, семантика;
        -- доступ к компонентам и признакам элементов
        -- (соответствующие функции) обеспечивает контекст.
      case дай_ключ (R) is
         when метасимвол ==> пройти_формулу (нач_опред(R)) ;
            -- рекурсивный вызов (для обработки формулы-определения
            -- метасимвола, на который указывает R).
         when семантика ==> рез.номер := знач_семантики (R);
                            стань_в_очередь(рез) ;
            -- соответствующая R семантика записывается в "рез"
            -- и передается в очередь семантик (при рандеву);
         when терминал ==>
            -- нужно обеспечить, чтобы в S была очередная лексема;
            if своб_S then выйди_из_очереди (S);
                           -- рандеву с очередью лексем.
                           своб_S := false ;
            end if ;
            -- выбор альтернативы.
            loop -- по альтернативам, если они есть ;
                 -- иначе единственная
                 -- альтернатива считается "последней"
               if знач_терминала (R) = абстр_лексема (S) then
```

```
                  рез.лексема := конкр_лексема (S);
                  своб_S := true ;  -- лексема обработана;
  -- лексема в общем случае состоит из абстрактной
  -- компоненты (например, "имя") и конкретной (например,
  -- "arctan"). Абстрактная используется в синтаксическом
  -- анализе, конкретная может пригодиться для контекстного анализа.
  -- Поэтому передается ему через "рез".
                  exit ; -- альтернатива выбрана (по одному текущему
                         -- символу!).
               else if посл_альтернатива (R) then
                  exit when коф (R) ; -- если дошли до кофа, то
                         -- нужно выбирать последнюю альтернативу;
                  удача := false ; -- текущая лексема
                         -- не соответствует текушей цели.
                  коллапс (R,S); -- вызов диагностической процедуры;
                  return ;        -- возврат при ошибке (неудача).
               else вниз (R) ; -- к следующей альтернативе формулы ;
                  -- по свойству разделенности там в начале только
                  -- терминалы ; поэтому останемся в том же цикле.
               end if ;
            end loop ;
       -- альтернатива выбрана!
      end case ; -- элемент формулы обработан;
      if конец_альтернативы (R) or not удача then return ;
         -- формулу прошли (удачно или неудачно);
         -- возвращаемся к предыдущей текущей цели, выходя из
         -- (в общем случае рекурсивного) вызова;
      else вправо (R) ; -- к следующему элементу формулы;
         -- семантики считаются явно вписанными в синтаксис в конце
         -- соответствующих альтернатив;
      end if ;
   end loop ; -- по элементам формулы
end пройти_формулу;
-- процедура пройти_формулу - это "сердце" анализатора!

begin
   пройти_формулу (цель) ; -- исходная цель задается контекстом;
   while not удача loop
      пройти_формулу (аварийная_цель) ;
   end loop ; -- синтаксический анализ после обнаружения ошибки и
         -- сообщения о ней выполняется по так называемому
         -- аварийному синтаксису ; его начало  - аварийная_цель -
         -- задается контекстом.
end анализатор ;
```

6.10.3. Контекст для синтаксического анализатора.

Определим точнее контекст задачи "анализатор".

```
package лексика is
   type лексема is private ;
   type а_лексема is (... начало, конец, имя, число, ...);
                              -- тип абстрактных лексем;
   type к_лексема is new строка ;  -- тип конкретных лексем;
    -- строка - предопределенный тип последовательностей символов;
   function абстр_лексема (X : лексема) return а_лексема ;
   function конкр_лексема (X : лексема) return к_лексема ;
   package очередь_лексем is new очередь (клиент ==> лексема) ;
    -- становятся доступны, в частности, выйди_из_очереди (X), где
    -- X - лексема.
   ... -- здесь уже можно объявить лексический анализатор
end лексика;

package семантика is
   type сем is new INTEGER range 1..200 ; -- примерно столько
                     -- семантик нужно в реальном трансляторе.
```

```
   type сем_запись is
      record
         номер : сем ;
         лексема : к_лексема ;
      end record ;
      -- можно было бы объявить этот тип приватным, но
      -- "откроем"  его для простоты
   package очередь_семантик is new очередь (клиент ==> сем_запись) ;
     -- становится доступным стань_в_очередь(X), где X - сем_запись.
      ...
end семантика ;

with лексика, семантика ; use лексика, семантика ;
use очередь_лексем, очередь_семантик ;
package синтактика is
   type указатель is private ; -- собственно синтаксис скрыт
    -- в теле пакета, поэтому скрыто и устройство указателей
    -- на элементы синтаксических правил.
   type ключ is  (терминал, метасимвол, семантика) ;
    -- это три вида элементов, существенных для анализа.
   function дай_ключ (P : указатель) return ключ ;
   procedure вниз (P : in out указатель) ; -- перебирает начала
                                         -- альтернатив.
   procedure вправо (P : in out указатель ; -- перебирает
                  -- последовательные элементы альтернативы.
   function конец_альтернативы (P : указатель) return BOOLEAN;
   function посл_альтернатива (P : указатель) return BOOLEAN;
  -- эта функция истинна для единственных и последних альтернатив.
   function нач_опред (P : указатель) return указатель;
  -- выдает указатель на начало определения
  -- того элемента-метасимвола, на который указывает P.
   function знач_терминала (P: указатель) return a_лексема;
  -- выдает значение терминала, указанное в синтаксическом элементе.
   function знач_семантики (P : указатель) return сем;
  -- выдает значение семантики, указанное в синтаксическом элементе.
   procedure коллапс (P : указатель, S : лексема);
  -- диагностическая процедура; выдает по диагностическому каналу
  -- вывода сообщения об ошибках.
   function коф (P : указатель) return BOOLEAN; -- распознает
                                  -- специальный терминал.
   function цель return указатель ; -- устанавливает исходную цель
    -- (для языка Дейк это "программа").
   function аварийная_цель return указатель;
  -- устанавливает аварийную цель (например,начало синтаксиса,
  -- определяющего всевозможные допустимые продолжения после ошибки;
  -- такой синтаксис можно получить из исходного систематической
  -- процедурой.
   task анализатор end ; -- задача "анализатор" - без входов.
   task body анализатор is separate; -- именно здесь "размещается"
                         -- тело анализатора после связывания.
end синтактика;
```

На этом прекратим дальнейшую детализацию контекста для синтаксического анализатора. Анализатор естественно считать параметрическим, так как он пригоден для работы с любым разделенным синтаксисом (воплощенным в теле пакета "синтактика").

[Несложно модифицировать анализатор, чтобы он обрабатывал формулы с фмгурными и квадратными скобками. Для круглых скобок он пригоден и в описанном виде. (Почему? Какое условие при этом следует наложить на пакет "синтактика"?)].

6.11. Заключение

Рассмотрен ряд принципов и компонент реализации ЯП. Пример синтаксического анализатора демонстрирует, в частности, меру возможной общности различных реализаций. Вместе с тем мы не зря прекратили детализацию. Она зависит от таких

факторов, как объем языка, требования к скорости анализа, размер машинного слова базовой машины и т.д., и т.п. Эти детали определяют специфику конкретных трансляторов даже в условиях, когда метод анализа фиксирован.

7. ЗАКЛЮЧИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

Многие важные аспекты предмета ЯП остались почти не затронутыми. К ним относятся прежде всего реализация ЯП. Об этом много написано. Очень хороши книги Пратта, а также Ахо и Ульмана.

Почти ничего не сказано о классификации ЯП. Наш основной тезис состоит в том, что пока обоснованной классификации ЯП нет. Но было бы полезно дать примеры классификации. Об этом также есть немного у Пратта.

Осталась не развитой математическая позиция. Полезно было бы уделить внимание новым тенденциям и идеям в области ЯП. На все это пока не хватило времени.

Стоит добавить, что реализация не рассматривалась еще и по той причине, что многие принципы, концепции и особенности ЯП удалось объяснить и без привлечения тонкостей реализации. Это и естественно, так как сам ЯП несравненно стабильнее (консервативнее), чем аппаратура и методика реализации (хотя интересно отметить, с другой стороны, что самая последняя авторская реализация Модулы-2 выполнена оправдавшими себя методами двадцатилетней давности - еще одно подтверждение принципа чайника). В сущности, лишь вопрос о принципиальной возможности или невозможности реализации в современных условиях может кардинально повлиять на грамотно проектируемый ЯП. Остальное влияние реализаторской позиции в целом иллюзорно.

Основной тезис состоит в том, что сущность ЯП можно излагать почти без привлечения реализаторской позиции. Когда же это не удается, значит излагаются недостаточно фундаментальные концепции.

Итак, рассмотрено пять моделей ЯП, на каждой продемонстрированы определенные концепции и принципы создания, оценки и применения ЯП.

Модель Неймана, отражая особенности машинных ЯП, послужила для демонстрации функций высших порядков, а также неразработанности аппарата абстракции-конкретизации и прогнозированмия-контроля.

Модифицированная модель Маркова проявила роль мощных средств анализа и синтеза в аппарате развития.

Модель Бэкуса познакомила с принципом концептуальной ясности и указала путь к его воплощению за счет мощных операционных абстракций высших порядков. На фиксированном наборе таких абстракций была продемонстрирована идея алгебры программ, а также технологии создания корректных и эффективных программ за счет эквивалентных преобразований программ в этой алгебре. Тем самым была достигнута определенная логическая завершенность в области базовых ЯП с технологической позиции - был указан путь построения операционных абстракций, ведущий к достижению технологического идеала.

Однако этот идеал, как было показано, не учитывал многих реальных потребностей современного программирования. Поэтому технологическая позиция была представлена в курсе еще двумя моделями.

Модель А позволила представить подпространство современных неймановских ЯП - самой распространенной их категории - и тем самым облегчить их самостоятельное изучение, сопоставление и оценку.

Модель М-2 (Модула-2) дала, во-первых, повод привести пример характеристики реального ЯП с помощью концептуальной схемы и языкового подпространства. Другими словами, это характеристика того, что есть в ЯП из обязательных атрибутов базового языка и чего нет в ЯП по отношению к максимальной модели такого ЯП в рассматриваемом подпространстве. Во-вторых, она дала возможность закончить курс сравнительным анализом двух ЯП, вызывающих особый интерес программистского сообщества в качестве кандидатов на наибольший успех в ближайшие годы. В-третьих, дала возможность показать в действии ключевой принцип рационального языкотворчества - принцип чемоданчика.

Такова "концептуальная схема" курса в целом с технологической и частично авторской позиции. Коротко ее можно выразить следующим тезисом :

От перспективных экспериментов в теории (модели М и Б) к перспективным экспериментам на практике (модели А и М-2).

Вторую часть тезиса можно и развернуть:

Сундук или чемоданчик или ... ?

[Может показаться, что в курсе слабо отражены современные ЯП массового программирования. На самом деле с учетом уровня университетского курса о них достаточно сказано при обсуждении модели А. Но дополнительный краткий обзор был бы полезным. Немало об этом сказано у Пратта. Стоит попытаться включить и сведения о наиболее интересных "пионерских" языках - содержащих новые идеи, но не ставших по различным причинам языками массового программирования - ПРОЛОГе и др.]

Упомянем и о других концептуальных схемах (слоях) курса.

Во-первых, мы смотрели на ЯП с нескольких различных позиций, стремясь к тому, чтобы взаимодействие этих позиций было продуктивным. Так, технологическая позиция постоянно давала материал для формулировки принципов и концепций, интересных прежде всего с позиции авторской. Таковы принцип согласованности абстракций; принцип РОРИУС (раздельного определения, реализации и использования услуг); концепция уникальности типа; понятие критичной потребности и неформальной теоремы о существовании ее решения; концепция регламентированного доступа (инкапсуляция); принцип реальности абстракций; принцип целостности объектов; концепция внутренней дисциплины при организации доступа к разделяемым ресурсам; концепция единой модели числовых расчетов; принцип защиты авторского права; концепция раздельной трансляции; динамический принцип выбора реакции на исключение; принцип динамической ловушки и др.

К сожалению, из-за временных ограничений не удалось уделить внимание многим другим полезным принципам, нашедшим свое воплощение в новейших ЯП.

Аналогичным образом семиотическая позиция взаимодействовала с авторской и технологической. А именно, занимаясь моделями Н, М, и Б, мы изучали различные виды семантик. С другой стороны, изучение дедуктивной семантики не только позволяет пояснить такой технологический элемент, как доказательство корректности программ, но и позволяет обосновать требования к управляющим структурам в ЯП (эти требования иногда неудачно называют принципами структурного программирования; такая трактовка отвлекает внимание от корректной структуризации всех аспектов программирования, в частности, структуризации данных).

Во-вторых, имея дело со всеми моделями, мы, с одной стороны, старались демонстрировать возможность выделять достаточно четко фиксированные модели, критерии, оценки и способы рассуждений, а также убедительных обоснований, вплоть до строгого математического доказательства содержательных свойств моделей (основные семантические соотношения в модели М, теоремы "теории трансляции" о свойствах универсального специализатора).

Но, с другой стороны, мы постоянно подчеркивали сложность ЯП как объекта конструирования и исследования. Мы старались показать, как выводы о свойствах проектных решений зависят от точки зрения, от самых общих подходов к проектированию ЯП. Особенно наглядно это проявилось при сопоставлении принципов сундука и чемоданчика. Ведь оказались под сомнением такие ранее "обоснованные" решения, как указатель контекста, приватные типы и концепция рандеву.

Этот аспект курса можно резюмировать так:

ЯП как сложное явление реального мира (лингвистическое, техническое, социальное, математическое) всегда уязвимо с точки зрения односторонней критики. ЯП всегда - плод компромиссов между технологическими потребностями и реализационными возможностями. Продуктивное творчество в области ЯП - скорее высокое искусство, чем предмет точной инженерной или тем более математической науки.

Отсюда следует, в частности, что современный ЯП следует воспринимать не как формальный объект, описанный произвольным способом, а как единство авторского описания, его денотата и соответствующей языковой ниши. Это положение способно оказать влияние практически на все фазы жизненного цикла ЯП.

7.1. О сущности предмета

Пояснение. Накоплено достаточно материала, чтобы можно было содержательно воспринять тезисы о сущности предмета "Языки программирования" в университетском курсе, которые были сформулированы автором в самом начале проектирования курса и в этом качестве обсуждались на кафедре алгоритмических языков факультета ВМиК МГУ весной 1982 года. Теперь можно сравнить замысел с реализацией и оценить необходимые модификации и того, и другого.

Собственно тезисы занумерованы, комментарии к ним выделены двумя предшествующими дефисами.

1). Язык программирования (ЯП) - это, во-первых, инструкция (пользователя) для планирования поведения абстрактного исполнителя и, во-вторых, средство общения между пользователями в связи с их деятельностью по программированию исполнителей.

-- подразумевается, что свойства абстрактного исполнителя (процессора, системы процессоров), которыми разрешено пользоваться при планировании его поведения, явно перечислены - тем самым пользователь вынужден абстрагироваться от остальных свойств реальных исполнителей.

-- сознательно говорится о "планировании поведения", а не, скажем, о "представлении алгоритмов", так как сущность современного программирования не укладывается в рамки представления алгоритмов хотя бы в части управления параллельными процессами или управления доступом к ресурсам.

2). Реально существующие ЯП - сложный. но необходимый и поучительный объект изучения для специалиста в области программирования. При кажущемся разнообразии они обладают достаточным сходством, чтобы было возможно и разумно рассматривать их с единых позиций, выявлять общие принципы, вырабатывать единые критерии оценки. Хотя эти позиции и принципы не стали еще компонентами сложившейся научной дисциплины, их изучение и усвоение уже сейчас способно помочь в овладении конкретным языком как инструментом повседневной деятельности, улучшении стиля программирования, выборе языка для использования и реализации, проектировании специальных языков.

3). В силу тезиса (1) и массовости пользователей, ЯП - явление прежде всего социальное, а затем инженерное и научное.

-- умозрительные критерии и оценки достоинств и недостатков языков по меньшей мере неубедительны - основной критерий: практика массового использования.

-- особенности пользователя - существенный фактор при конструировании, выборе и пропаганде ЯП, а также в их развитии, "выживании" или отмирании.

4). В силу (1) и (3). а также исключительной трудоемкости программирования, особое значение имеют языки, активно способсвтующие сохранению и накоплению законченных продуктов программистской деятельности, предназначенных для массового использования - программных изделий. В настоящее время это базовые языки индустриального программирования.

-- поэтому в курсе им уделено особое внимание.

5). Для конструирования и понимания особенностей , а также использования и оценки базовых языков ключевое понятие - полный жизненный цикл программного изделия и самого ЯП; существенно, в частности, распределение затрат по различным этапам жизненного цикла.

-- тем самым признаются неприемлемо узкими такие распространенные критерии, как легкость написания программ или эффективность объектного кода или простота изучения языка и т.п.

6). Требования, предъявляемые к ЯП (а следовательно и критерии оценки) существенно зависят от технологии (характера. стереотипа) его использования. Эти требования различны в зависимости от разновидности программирования (индустриальное, экспериментальное, игровое, учебное и т.п.), этапа жизненного цикла изделия (программирование, эволюция, перенос и т.п.), этапа жизненного цикла языка (конструирование, реализация. внедрение, стандартизация и т.п.). Тенденция развития в искусстве создания ЯП состоит во все более четкой формулировке требований к языку и применении соответствующих критериев отбора выразительных средств языка. В идеале справедлив принцип технологичности - ЯП в целом должен оптимально соотвествовать технологии (стереотипу) использования.

7). Основными требованиями к современному базовому языку индустриального программирования признаются (с точки зрения качества программ) надежность, познаваемость, переносимость и эффективность.

8). Важнейшей технологической характеристикой ЯП (с учетом принципа технологичности) считается его способность помогать в борьбе со сложностью программ, программирования, понимания программы и других объектов и процессов, вовлекаемых в сферу использования ЯП.

9). Применение аппарата абстракции-конкретизации - основное средство борьбы со сложностью в программировании. (может бытъ, "обобщение-настройка" или "выделение главного - учет тонкостей").

-- он применяется при пошаговой детализации.

-- он применяется при любой параметризации, при отделении спецификации от реализации, при введении абстрактных (виртуальных) ресурсов, в смешанных вычислениях, представлении знаний и т.п.

10). В развитии ЯП прослеживается тенденция к постепенному развитию и последующей стабилизации специфических выразительных средств для важнейших абстракций, соответствующих ключевым технологическим (операционным) потребностям в жизненном цикле программ. Высшим проявлением этого принципа "конструктивизации абстракции" служит развитие языковых модулей, соотвествующих важнейшим абстракциям.

-- в этом смысле модульность - самый развитый вариант реализации (конструктивизации, воплощение в конкретной языковой конструкции) абстракции.

-- этот принцип: каждой абстракции - своя конструкция может противоречить, скажем, принципу ортогональности (и обычно побеждает его).

11). Характерные примеры абстракции:

ОТ ЧЕГО	СРЕДСТВО АБСТРАКЦИИ	СРЕДСТВО КОНКРЕТИЗАЦИИ
размещения	имя	загрузчик
исполнения	процедура	вызов, специализатор
компьютера	ЯП (виртуальная машина)	транслятор (эмулятор)
контекста	модуль, блок(частично)	транслятор, загрузчик вызов
реализация	спецификация	программирование
представления	абстрактные ресурсы (АТД, в частности)	пакет, класс

-- ниже следуют менее разработанные абстракции:

исключений	нормальные сегменты	реакция на исключение
взаимодействия	последовательные сегменты	рандеву, семафоры
ЯП	псевдокод, лексикон	программист, конвертор
изменений	слой (по Фуксману)	новый слой
порождения	родовые объекты, макросы	макровызов, настройка, макрогенератор
именования	образец, условие	конкретизация образца

-- перспективные абстракции:

проблемной области	лексикон	творец прикладной теории
модели (конструкции)	теория, система соотношений	дополнительные соотношения
задача	моделъ, тип	имена аргументов и результатов
программы	задача на модели, (на конструкции)	значения аргументов

-- очевидно. что далеко не все эти абстракции обеспечены соответствующими языковыми конструктами (и, по-видимому, некоторые никогда не будут обеспечены; но приведенный спектр абстракций дает возможность анализировать конкретные языки на предмет развития в нем аппарата определенных абстракций и судить о языках не по "низкому" и "высокому" уровню, а существенно более содержательно с технической стороны. Подчеркнем, что принцип технологичности требует не наивысшего, а оптимального уровня абстракции в соответствии с требованиями к ЯП.

-- этот список абстракций показывает важность социального аспекта ЯП; скажем, абстрагироваться от компьютера - дело творцов ЯП, а вот средства конкретизации обеспечивают реализаторы трансляторов, авторы учебников и т.п. Другими словами, оценка разработанности аппарата абстракции-конкретизации в языке выходит за рамки его внутренних свойств, причем это может касаться важнейших для пользователя абстракций.

-- важна также возможная изменчивость оцекни языка, связанная с внешней по отношению к нему человеческой деятельностью - поддержкой, пропагандой, накоплением

реализаций и программ, появлением хороших учебников, развитием возможностей аппаратуры и методов реализации, технологии программирования.

12). Если аппарат абстракции-конкретизации - основное средство борьбы со сложностью (в принципе способный повысить и познаваемость, и надежность, и переносимость, и эффективность), то следующий по важности аппарат (также в принципе ориентированный на все четыре перечисленные характеристики, но прежде всего - на надежность) - это аппарат прогнозирования-контроля. С этой точки зрения важна также проверяемость требуемых ЯП свойств программ (особенно для переносимости).

13). С точки зрения специфических требований к базовым языкам индустриального программирования выделенной технической характеристикой языка следует считать пригодность к построению на его основе виртуальных машин требуемого класса (другими словами, пакетов прикладных программ). Иначе говоря, его пригодностъ к развитию.

14). Ключевым понятием, связанным со способностью языка к выживанию в конкурентной борьбе , служит понятие языковой ниши - это комплекс внешних условий, при которых активная жизнь двух различных языков невозможна.

-- понятие это "носится в воздухе", но недостаточно исследовано; поэтому давать в курсе исчерпывающие характеристики ниш для рассматриваемх языков не предполагается, однако следует сказать то, что известно.

15). Ключевыми компонентами комплексной характеристики (оценки) языка могут служить

модель исполнения - чисто внутренняя характеристика
анализ основных средств абстракции-конкретизации - промежуточная
стереотип использования - чисто внешняя (часть ниши)
ниша - чисто внешняя
выдающиеся черты

-- эти компоненты определяют единую схему комплексного анализа конкретных языков в курсе; такой анализ призван оставить у слушателей достаточно четкий индивидуальный "образ" языка. [Пока не получилось - мало конкретных данных.]

16). Компонентами модели исполнителя, наряду с классическими "данными", "действиями" и "управлением", служат "именование", "прогнозирование", "контроль", "исключения", "синхронизация" и "защита". [Где связывание?].

-- более развитая схема лучше проявляет индивидуальные свойства и тенденции развития ЯП.

17). Пониманию и запоминанию частных свойств ЯП, целостному их восприятию, способствует знание совокупности часто используемых принципов их построения, каждый из которых следует рассматривать как конструктивный прием, направленный на достижение разумного компромисса между стремлением к концептуальной целостности языка и необходимостью удовлетворять многочисленным и противоречивым требованиям.

-- естественно, не вся совокупность рассматриваемых принципов действует одновременно в каждом ЯП; некоторые принципы противоречат друг другу.

18). Характерные принципы: однопроходность, динамизм, статический контроль, ортогональность, модульность, уникалъностъ типа, согласованностъ абстракций.

19). До последнего времени абстракциями, определяющими выживание базового языка, были абстракции контекста и компьютера, в ближайшем будущем к ним добавится абстракция реализации.

-- с ней связан принцип, который можно было бы назвать принципом индустриальности - язык должен способствовать развитию индустрии программ м в смысле защиты авторских интересов создателей программных изделий и, в частности, гарантии качества предоставляемых услуг.

20). Проблемы реализации языков остаются весьма важным фактором, влияющим как на отбор выразительных средств, так и на их судьбу в целом, а так же на стиль программирования и психологию программистов.

-- тем не менее большинство решений в ЯП можно объяснить и не прибегая к непосредственным ссылкам на проблемы реализации в узком смысле. Часто важнее оказываются технологичность, концептуальная целостность, социальный характер языка или незрелость искусства создания языков и науки об их использовании, развитии и т.п.

21). Реализация ЯП содержит по меньшей мере следующие компоненты: программную реализацию - транслятор плюс инструментарий, обеспечивающий жизненный цикл программ на реализуемом языке (это могут быть и компоненты многоязыковой системы программирования); описание реализованной версии языка (или допустимых

уточнений реализации); средства проверки реализаций на соответствие описанию (в частности, стандарту, если он принят); руководство по программированию (учебник).

22). Создание трансляторов - одна из прикладных областей, для которых разработаны теории и технологии, свои пакеты прикладных программ - СПТ. В этой области возникли и используются многие передовые концепции современного программирования - функциональное и реляционное программирование, доказательное программирование, смешанные вычисления и др.

-- конкретный пример инструментального пакета - МАСОН с его словом и таблицей как базовыми понятиями. [Пока не получилось - нет времени.]

23). Несмотря на значительные достижения, создание трансляторов в целом остается делом трудным, долгим и дорогим, а готовые трансляторы часто не удовлетворяют весьма скромным требованиям по надежности, удобству в эксплуатации и полноте спецификации.

24). Формально ЯП можно и полезно рассматривать как отображение

текст --> мир

подчеркивая различие "миров" при рассмотрении ЯП с разных точек зрения.

-- мир функций - денотационная семантика, мир утверждений - дедуктивная, мир текстов - проекционная, мир вычислений - операционная и т.п.

25). Стандартизация - высший этап в развитии ЯП. Точность и проверяемость описания - ключевые технические свойства стандарта. Метод взаимно-дополнительных определений - естественный общий метод определения языка. Частичная формализация - естественное обобщение программирования. Частично формализованный псевдокод - естественное обобщение и предтеча ЯП.

Литература

1. С.С.Лавров. Основные понятия и конструкции языков программирования. - М.: Финансы и статистика, 1982.
2. Базисный рефал и его реализация на вычислительных машинах (методические рекомендации). - ЦНИПИАСС, Госстрой СССР, 1977.
3. J.Backus. Can Programming Be Liberated from von Neumann Style? A Functional Style and Its Algebra of Programs. - CACM, 1978, v. 21, n.8, 613-641.
4. Д.Грис. Наука программирования.-М.: Мир, 1984.
5. Н.Вирт. Модула-2. (Пер. с английского Л.А.Захарова.- В кн.: Языки программирования.- М.:Наука,1985. (Алгоритмы и алгоритмические языки).
6. Э.З.Любимский, В.В.Мартынюк, Н.П.Трифонов. Программирование. - М.:Наука, 1979 (?).
7. Дж.Хьюз, Дж.Мичтом. Структурный подход к программироованию. - М.: Мир, 1980.
8. С.Янг. Алгоритмические языки реального времени. Конструирование и разработка. - М.: Мир, 1985.
9. Ф. Брукс. Как проектируются и создаются программные комплексы.
10. В.Л.Темов. Метаалгоритмическая система общего назначения МАСОН.
11. В.М.Пентковский. Язык программирования Эль-76.
12. Бринч-Хансен. Методы проектирования операционных систем.
13. В.Л. Темов. Язык и система программирования Том. (В печати).
14. Т. Пратт. Языки программирования. Разработка и реализация. (появилось второе американское издание 1983 года).
15. Д.Грис "Конструирование компиляторов для цифровых вычислительных машин" - М.:Мир,1975.
16. Р.Хантер "Проектирование и конструирование компиляторов" - М."Финансы и статистика",1984.
17. Пайл Я. "Ада - язык встроенных систем" (М., Ф. и С., 1984), главы 9 и 10.
18. П.Вегнер. Программирование на языке Ада.
19. Стандарт Ады.
20. Н. Вирт. Алгоритмы + структуры данных = программы. - М.: Мир, 1985.