Chapter 15
Programming

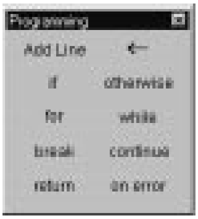
Pro

With Mathcad Professional, you can write your own programs using specialized programming operators. A Mathcad program has many attributes associated with programming languages including conditional branching, looping constructs, local scoping of variables, error handling, the ability to use other programs as subroutines, and the ability to call itself recursively. Mathcad programs make it easy to do tasks that may be impossible or inconvenient to do in any other way.

This chapter contains the following sections:

Defining a program

A Mathcad program is a special kind of Mathcad expression you can create in Mathcad Professional—it's an expression made up of a sequence of statements created using *programming operators*, available on the Programming toolbar. Click on the Math toolbar, or choose **Toolbars⇒Programming** from the **View** menu, to open the Programming toolbar.



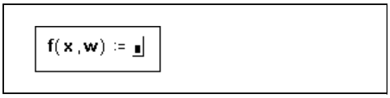
You can think of a program as a compound expression that involves potentially many programming operators. Like any expression, a program returns a value—a scalar, vector, array, nested array, or string—when followed by the equal sign or the live symbolic equal sign. Just as you can define a variable or function in terms of an expression, you can also define either in terms of a program.

The following example shows how to make a simple program to define the function:

$$f(x, w) \;=\; \log\left(\frac{x}{w}\right)$$

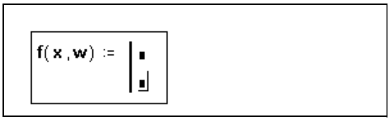
Although the example chosen is simple enough not to require programming, it illustrates how to separate the statements making up the program and how to use the local assignment operator, "←."

- Type the left side of the function definition, followed by a "**:=**". Make sure the placeholder is selected.



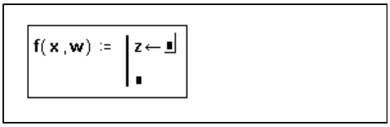
f(x, w) := ■

- Click Add Line on the Programming toolbar. Alternatively, press **]**. You'll see a vertical bar with two placeholders, which will hold the statements comprising your program.



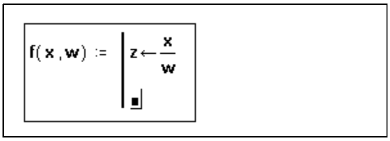
$$f(x, w) := \left|\begin{array}{l} ■ \\ ■ \end{array}\right.$$

- Click in the top placeholder. Type **z**, then click ← on the Programming toolbar. Alternatively, press **{** to insert a "←."



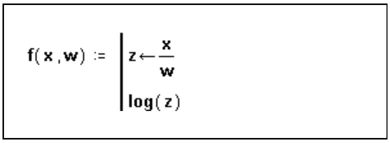
$$f(x, w) := \left|\begin{array}{l} z \leftarrow ■ \\ ■ \end{array}\right.$$

- Type **x/w** in the placeholder to the right of the "←." Then press [**Tab**] to move to the bottom placeholder.



$$f(x, w) := \left|\begin{array}{l} z \leftarrow \frac{x}{w} \\ ■ \end{array}\right.$$

- Enter the value to be returned by the program in the remaining placeholder. Type **log(z)**.

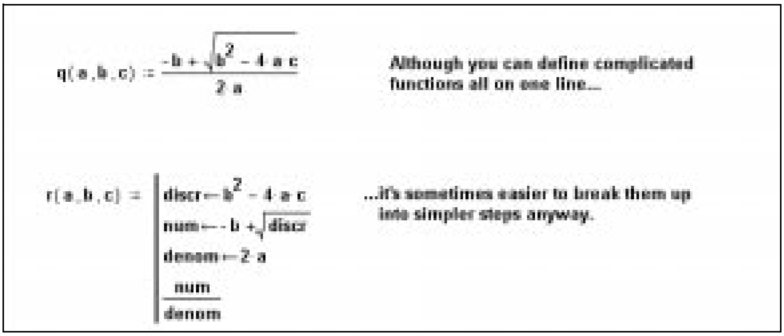


$$f(x, w) := \left|\begin{array}{l} z \leftarrow \frac{x}{w} \\ \log(z) \end{array}\right.$$

You can now use this function just as you would any other function in your worksheet.

Note You cannot use Mathcad’s usual assignment operator “:=” inside a program; you must use the local assignment operator instead. Variables defined inside a program with the local assignment operator “←,” such as *z* in the example above, are local to the program and are undefined elsewhere in the worksheet. However, you can refer to Mathcad variables and functions defined previously in the worksheet within a program.

Figure 15-1 shows a more complex example involving the quadratic formula. Although you can define the quadratic formula with a single statement as shown in the top half of the figure, you may find it easier to define it with a series of simple statements as shown in the bottom half.



$$q(a,b,c) := \frac{-b+\sqrt{b^2-4\cdot a\cdot c}}{2\cdot a}$$

Although you can define complicated functions all on one line...

$$r(a,b,c) := \left|\begin{array}{l} discr \leftarrow b^2-4\cdot a\cdot c \\ num \leftarrow -b+\sqrt{discr} \\ denom \leftarrow 2\cdot a \\ \dfrac{num}{denom} \end{array}\right.$$

...it's sometimes easier to break them up into simpler steps anyway.

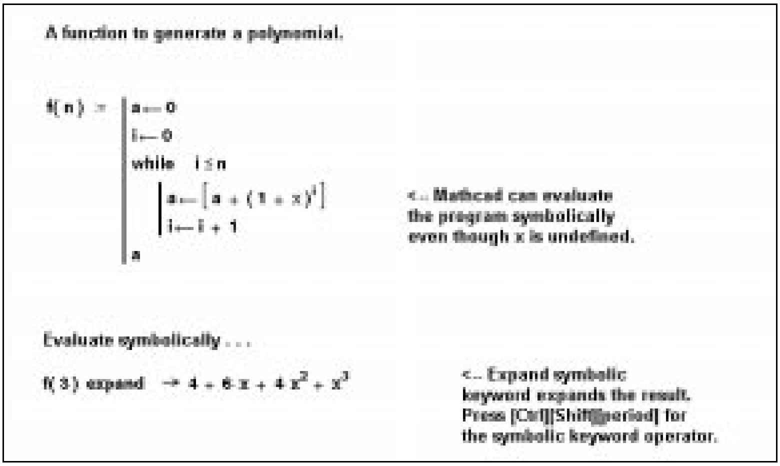
Figure 15-1: A more complex function defined in terms of both an expression and a program.

Tip A program can have any number of statements. To add a statement, click Add Line on the Programming toolbar. Mathcad inserts a placeholder below whatever statement you’ve selected. To delete the placeholder, click on it and press [**`Bksp`**].

As with any expression, a Mathcad program must have a value. This value is simply the value of the last statement executed by the program. It could be a string expression or a single number, or it could be an array of numbers. It could even be an array of arrays (see “Nested arrays” on page 233).

You can also write a Mathcad program to return a *symbolic* expression. When you evaluate a program using the live symbolic equal sign, “→,” described in Chapter 14, “Symbolic Calculation,” Mathcad passes the expression to its symbolic processor and, when possible, returns a simplified symbolic expression. You can use Mathcad’s ability to evaluate programs symbolically to generate complicated symbolic expressions, polynomials, and matrices. Figure 15-2 shows a function that, when evaluated symbolically, generates symbolic polynomials.

Note Programs that include the **return** and **on error** statements, described on page 308 and page 310, cannot be evaluated symbolically since the symbolic processor does not recognize these operators.



A function to generate a polynomial.

```
f(n) :=  | a ← 0
         | i ← 0
         | while  i ≤ n
         |     | a ← [a + (1 + x)^i]
         |     | i ← i + 1
         | a
```

<-- Mathcad can evaluate the program symbolically even though x is undefined.

Evaluate symbolically . . .

$f(3)$ expand $\rightarrow 4 + 6 \cdot x + 4 \cdot x^2 + x^3$

<-- Expand symbolic keyword expands the result. Press [Ctrl][Shift][period] for the symbolic keyword operator.

Figure 15-2: Using a Mathcad program to generate a symbolic expression.

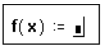
On-line Help For programming examples, see the “Programming” section in the Resource Center QuickSheets. The Resource Center also includes a special section, “The Treasury Guide to Programming,” which provides detailed examples and applications of Mathcad programs.

Conditional statements

In general, Mathcad evaluates each statement in your program from the top down. There may be times, however, when you want Mathcad to evaluate a statement only when a particular condition is met. You can do this by including an **if** statement.

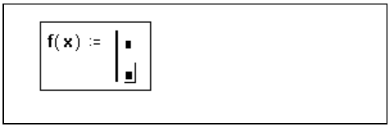
For example, suppose you want to define a function that forms a semicircle around the origin but is otherwise constant. To do this:

- Type the left side of the function definition, followed by a “**:=**”. Make sure the placeholder is selected.



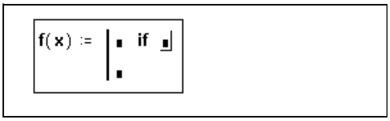
f(x) := ▪

- Click Add Line on the Programming toolbar. Alternatively, press **]**. You'll see a vertical bar with two placeholders. These placeholders will hold the statements making up your program.



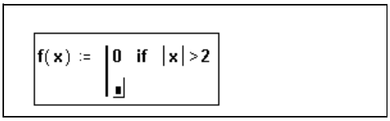
f(x) := ▪
▪

- Click if on the Programming toolbar in the top placeholder. Alternatively, press **}**. Do not type "if."



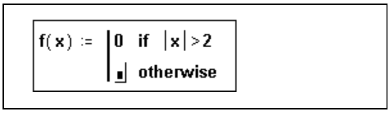
f(x) := ▪ if ▪
▪

- Enter a Boolean expression in the right placeholder using one of the relational operators on the Evaluation toolbar. In the left placeholder, type the value you want the expression to take whenever the expression in the right placeholder is true. If necessary, add more placeholders by clicking Add Line.



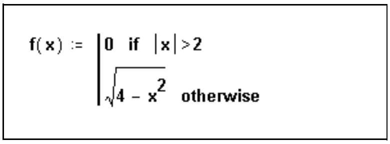
f(x) := 0 if $|x|>2$
▪

- Select the remaining placeholder and click otherwise on the Programming toolbar.



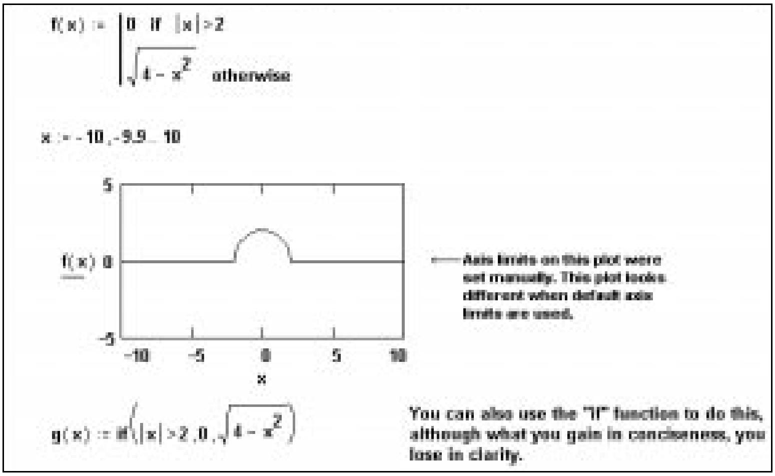
f(x) := 0 if $|x|>2$
▪ otherwise

- Type the value you want the program to return if the condition in the first statement is not met.



f(x) := 0 if $|x|>2$
$\sqrt{4-x^2}$ otherwise

Figure 15-3 shows a plot of this function.



f(x) := 0 if $|x|>2$
$\sqrt{4-x^2}$ otherwise

x := -10, -9.9 .. 10

Axis limits on this plot were set manually. This plot looks different when default axis limits are used.

g(x) := if$\left(|x|>2, 0, \sqrt{4-x^2}\right)$

You can also use the "if" function to do this, although what you gain in conciseness, you lose in clarity.

Figure 15-3: Using the **if** *statement to define a piecewise continuous function.*

Note The **if** statement in a Mathcad program is not the same as the *if* function (see “Piecewise continuous functions” on page 177). Although it is not hard to define a simple program using the *if* function, as shown in Figure 15-3, the *if* function can become unwieldy as the number of branches exceeds two.

Looping

One of the greatest strengths of programmability is the ability to execute a sequence of statements repeatedly in a loop. Mathcad provides two loop structures. The choice of which loop to use depends on how you plan to tell the loop to stop executing.

- If you know exactly how many times a loop is to execute, use a **for** loop.
- If you want the loop to stop upon the occurrence of a condition, but you don’t know how many loops will be required, use a **while** loop.

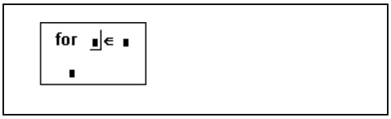
Tip See “Controlling program execution” on page 308 for methods to interrupt calculation within the body of a loop.

“for” loops

A **for** loop is a loop that terminates after a predetermined number of iterations. Iteration is controlled by an *iteration variable* defined at the top of the loop. The definition of the iteration variable is entirely local to the program.

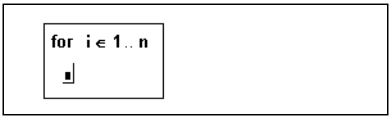
To create a **for** loop:

- Click for on the Programming toolbar. Do not type the word “for.”



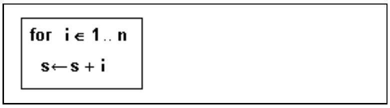
```
for ∎|∈ ∎
  ∎
```

- Type the name of the iteration variable in the placeholder to the left of the “∈ .”
- Enter the range of values the iteration variable should take in the placeholder to the right of the “∈ .” You usually specify this range the same way you would for a range variable (see page 125).



```
for i ∈ 1..n
  ∎|
```

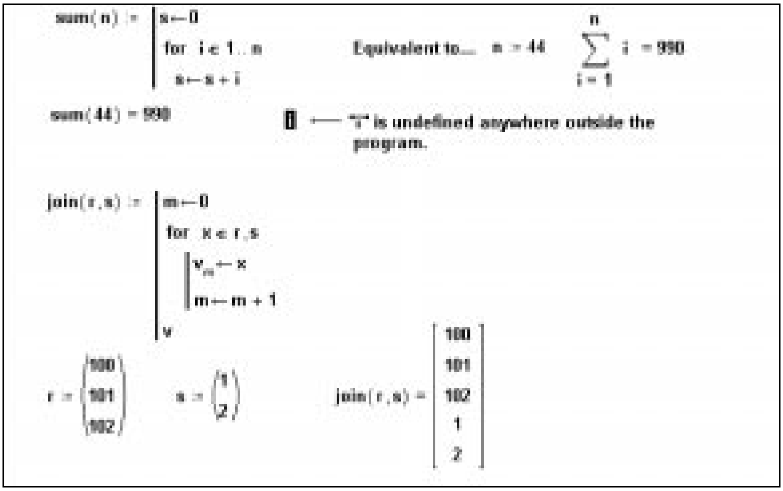
- Type the expression you want to evaluate in the remaining placeholder. This expression generally involves the iteration variable. If necessary, add placeholders by clicking Add Line on the Programming toolbar.



```
for  i ∈ 1..n
  s←s + i
```

The upper half of Figure 15-4 shows this **for** loop being used to add a sequence of integers.

Note Although the expression to the right of the “∈ ” is usually a range, it can also be a vector or a list of scalars, ranges, and vectors separated by commas. The lower half of Figure 15-4 shows an example in which the iteration variable is defined as the elements of two vectors.



```
sum(n) := | s←0
          | for  i ∈ 1..n
          |   s←s + i
```

Equivalent to... $n := 44$ $\quad \sum_{i=1}^{n} i = 990$

$\text{sum}(44) = 990$

i ← "i" is undefined anywhere outside the program.

```
join(r,s) := | m←0
             | for  x ∈ r,s
             |   | v_m←x
             |   | m←m + 1
             | v
```

$r := \begin{pmatrix} 100 \\ 101 \\ 102 \end{pmatrix} \quad s := \begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \text{join}(r,s) = \begin{bmatrix} 100 \\ 101 \\ 102 \\ 1 \\ 2 \end{bmatrix}$

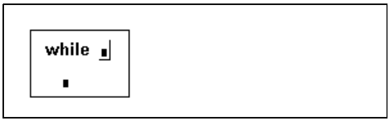
Figure 15-4: Using a **for** *loop with two different kinds of iteration variables.*

“while” loops

A **while** loop is driven by the truth of some condition. Because of this, you don’t need to know in advance how many times the loop will execute. It is important, however, to have a statement somewhere, either within the loop or elsewhere in the program, that eventually makes the condition false. Otherwise, the loop executes indefinitely.

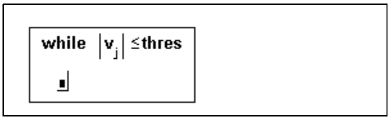
To create a **while** loop:

- Click while on the Programming toolbar. Do not type the word “while.”



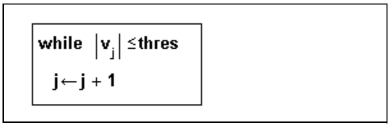
```
while ▪
  ▪
```

- Click in the top placeholder and type a condition. This is typically a Boolean expression like the one shown.



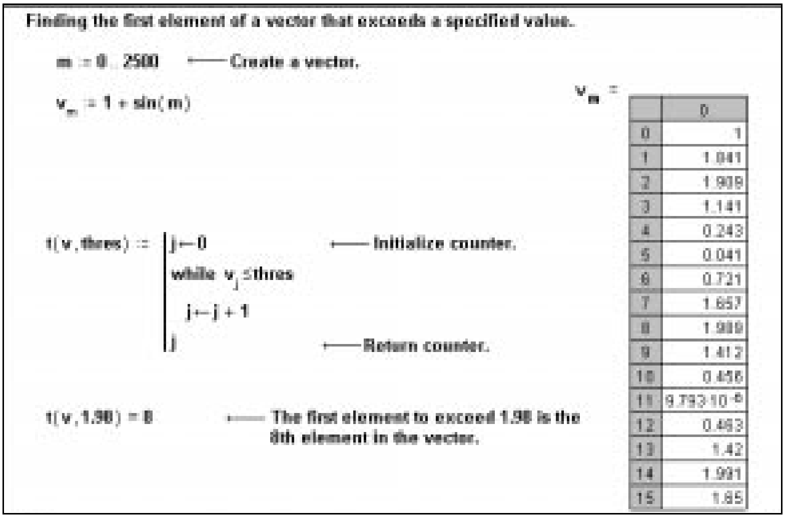
while $|v_j| \leq thres$
  ▪

- Type the expression you want evaluated in the remaining placeholder. If necessary, add placeholders by clicking Add Line on the Programming toolbar.



```
while |v_j| ≤thres
  j←j + 1
```

Figure 15-5 shows a larger program incorporating the above loop. Upon encountering a **while** loop, Mathcad checks the condition. If the condition is true, Mathcad executes the body of the loop and checks the condition again. If the condition is false, Mathcad exits the loop.



Finding the first element of a vector that exceeds a specified value.

```
m := 0 .. 2500        ←—— Create a vector.
v_m := 1 + sin(m)

t(v,thres) := | j←0                    ←—— Initialize counter.
               | while v_j ≤thres
               |   j←j + 1
               | j                      ←—— Return counter.

t(v,1.98) = 8         ←—— The first element to exceed 1.98 is the
                           8th element in the vector.
```

$v_m =$

	0
0	1
1	1.841
2	1.909
3	1.141
4	0.243
5	0.041
6	0.721
7	1.657
8	1.989
9	1.412
10	0.456
11	$9.793\cdot10^{-6}$
12	0.463
13	1.42
14	1.991
15	1.65

Figure 15-5: Using a **while** *loop to find the first occurrence of a particular number in a matrix.*

Controlling program execution

The Programming toolbar in Mathcad Professional includes three statements for controlling program execution:

- Use the **break** statement within a **for** or **while** loop to interrupt the loop when a condition occurs and move execution to the next statement outside the loop.
- Use the **continue** statement within a **for** or **while** loop to interrupt the current iteration and force program execution to continue with the next iteration of the loop.
- Use the **return** statement to stop a program and return a particular value from within the program rather than from the last statement evaluated.

The “break” statement

It is often useful to break out of a loop upon the occurrence of some condition. For example, in Figure 15-6 a **break** statement is used to stop a loop when a negative number is encountered in an input vector.

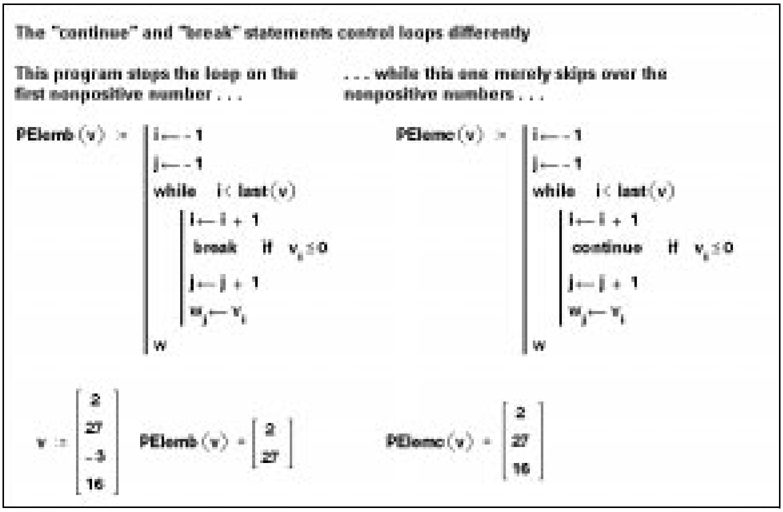
To insert a **break** statement, click on a placeholder inside a loop and click break on the Programming toolbar. Do not type the word “break.” You typically insert **break** into the left-hand placeholder of an **if** statement. The **break** is evaluated only when the right-hand side of the **if** is true.

Tip To create the program in Figure 15-6, for example, you would click break first, then click if.

The “continue” statement

To ignore an iteration of a loop, use **continue**. For example, in Figure 15-6 a **continue** statement is used to ignore nonpositive numbers in an input vector.

To insert the **continue** statement, click on a placeholder inside a loop and click continue on the Programming toolbar. Do not type the word “continue.” As with **break**, you typically insert **continue** into the left-hand placeholder of an **if** statement. The **continue** statement is evaluated only when the right-hand side of the **if** is true.



The "continue" and "break" statements control loops differently

This program stops the loop on the first nonpositive number . . .

```
PElemb(v) :=  i ← -1
              j ← -1
              while  i < last(v)
                  i ← i + 1
                  break  if  $v_i \le 0$
                  j ← j + 1
                  $w_j \leftarrow v_i$
              w
```

. . . while this one merely skips over the nonpositive numbers . . .

```
PElemc(v) :=  i ← -1
              j ← -1
              while  i < last(v)
                  i ← i + 1
                  continue  if  $v_i \le 0$
                  j ← j + 1
                  $w_j \leftarrow v_i$
              w
```

$v := \begin{bmatrix} 2 \\ 27 \\ -3 \\ 16 \end{bmatrix}$ $\quad \mathrm{PElemb}(v) = \begin{bmatrix} 2 \\ 27 \end{bmatrix}$ $\quad \mathrm{PElemc}(v) = \begin{bmatrix} 2 \\ 27 \\ 16 \end{bmatrix}$

Figure 15-6: The ***break*** *statement halts the loop, but execution resumes on the next iteration when* ***continue*** *is used.*

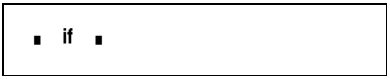
The “return” statement

A Mathcad program returns the value of the last expression evaluated in the program. In simple programs, the last expression evaluated is in the last line of the program. As you create more complicated programs, you may need more flexibility. The **return** statement allows you to interrupt the program and return particular values other than the default value.

A **return** statement can be used anywhere in a program, even within a deeply nested loop, to force program termination and the return of a scalar, vector, array, or string. As with **break** and **continue**, you typically use **return** on the left-hand side of an **if** statement, and the **return** statement is evaluated only when the right-hand side of the **if** is true.

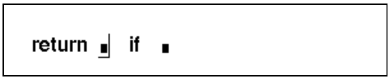
The following program fragment shows how a **return** statement is used to return a string upon the occurrence of a particular condition:

- Click if on the Programming toolbar.



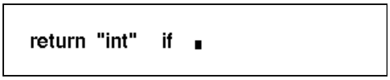
■ if ■

- Now click return on the Programming toolbar. Do not type "return."



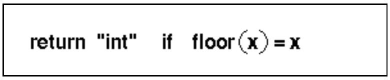
return ■ if ■

- Create a string by typing the double-quote key (**"**) on the placeholder to the right of **return**. Then type the string to be returned by the program. Mathcad displays the string between a pair of quotes.



return "int" if ■

- Type a condition in the placeholder to the right of **if**. This is typically a Boolean expression like the one shown. (Type [**Ctrl**]**=** for the bold equal sign.)



return "int" if floor(x) = x

In this example, the program returns the string "int" when the expression $floor(x) = x$ is true.

Tip You can add more lines to the expression to the right of **return** by clicking Add Line on the Programming toolbar.

Error handling

Errors may occur during program execution that cause Mathcad to stop calculating the program. For example, because of a particular input, a program may attempt to divide by 0 in an expression and therefore encounter a singularity error. In these cases Mathcad treats the program as it does any math expression: it marks the offending expression with an error message and highlights the offending name or operator in a different color, as described in Chapter 8, "Calculating in Mathcad."

Mathcad Professional gives you two features to improve error handling in programs:

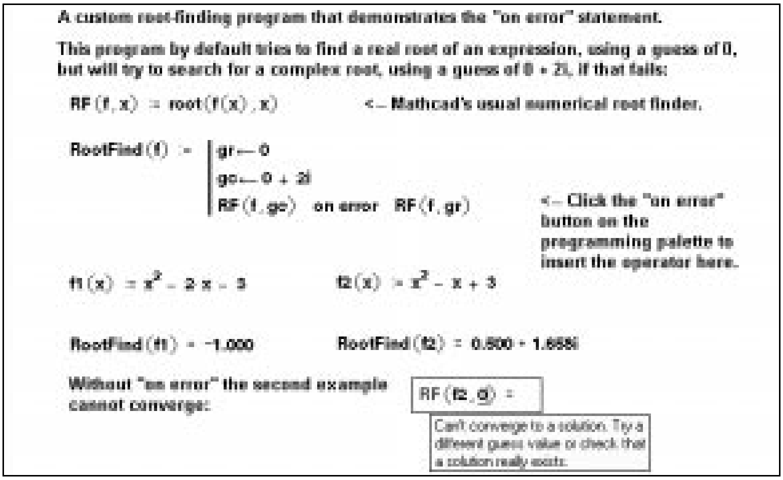
- The **on error** statement on the Programming toolbar allows you to trap a numerical error that would otherwise force Mathcad to stop calculating the program.
- The *error* string function gives you access to Mathcad's error tip mechanism and lets you customize error messages issued by your program.

"on error" statement

In some cases you may be able to anticipate program inputs that lead to a numerical error (such as a singularity, an overflow, or a failure to converge) that would force Mathcad to stop calculating the program. In more complicated cases, especially when your programs rely heavily on Mathcad's numerical operators or built-in function set, you may not be able to anticipate or enumerate all of the possible numerical errors that can occur in a program. The **on error** statement is designed as a general-purpose error trap to compute an alternative expression when a numerical error occurs that would otherwise force Mathcad to stop calculating the program.

To use the **on error** statement, click on error on the Programming toolbar. Do not type "on error." In the placeholder to the right of **on error**, create the program statement(s) you ordinarily expect to evaluate but in which you wish to trap any numerical errors. In the placeholder to the left create the program statement(s) you want to evaluate should the default expression on the right-hand side fail.

Figure 15-7 shows **on error** operating in a program to find a root of an expression.



A custom root-finding program that demonstrates the "on error" statement.

This program by default tries to find a real root of an expression, using a guess of 0, but will try to search for a complex root, using a guess of 0 + 2i, if that fails:

RF(f, x) := root(f(x), x)   <-- Mathcad's usual numerical root finder.

RootFind(f) := | gr ← 0
               | gc ← 0 + 2i
               | RF(f, gc) on error RF(f, gr)

<-- Click the "on error" button on the programming palette to insert the operator here.

f1(x) := $x^2 - 2x - 3$ f2(x) := $x^2 - x + 3$

RootFind(f1) = −1.000   RootFind(f2) = 0.500 + 1.658i

Without "on error" the second example cannot converge: RF(f2, g) =

Can't converge to a solution. Try a different guess value or check that a solution really exists.

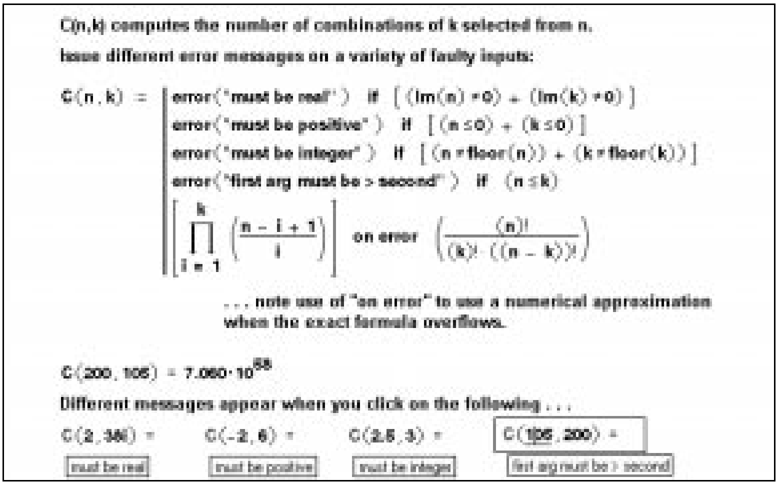
Figure 15-7: The **on error** *statement traps numerical errors in a program.*

Issuing error messages

Just as Mathcad automatically stops further evaluation and produces an appropriate "error tip" on an expression that generates an error (see the bottom of Figure 15-7 for an example), you can cause evaluation to stop and make custom error tips appear when your programs or other expressions are used improperly or cannot return answers.

Mathcad Professional’s *error* string function gives you this capability. This function, described in “String functions” on page 213, suspends further numerical evaluation of an expression and produces an error tip whose text is simply the string it takes as an argument. Typically you use the *error* string function in the placeholder on the left-hand side of an **if** or **on error** programming statement so that an error and appropriate error tip are generated when a particular condition is encountered.

Figure 15-8 shows how custom errors can be used even in a small program.



C(n,k) computes the number of combinations of k selected from n.

Issue different error messages on a variety of faulty inputs:

$$C(n,k) = \left| \begin{array}{l} \text{error("must be real")} \quad \text{if} \quad [(\mathrm{Im}(n)\ne 0)+(\mathrm{Im}(k)\ne 0)] \\ \text{error("must be positive")} \quad \text{if} \quad [(n\le 0)+(k\le 0)] \\ \text{error("must be integer")} \quad \text{if} \quad [(n\ne \mathrm{floor}(n))+(k\ne \mathrm{floor}(k))] \\ \text{error("first arg must be > second")} \quad \text{if} \quad (n\le k) \\ \left[\prod_{i=1}^{k}\left(\frac{n-i+1}{i}\right)\right] \quad \text{on error} \quad \left(\frac{(n)!}{(k)!\cdot((n-k))!}\right) \end{array}\right.$$

. . . note use of "on error" to use a numerical approximation when the exact formula overflows.

$C(200,105) = 7.060\cdot 10^{58}$

Different messages appear when you click on the following . . .

$C(2,3i) =$ must be real

$C(-2,6) =$ must be positive

$C(2.5,3) =$ must be integer

$C(105,200) =$ first arg must be > second

Figure 15-8: Generating custom errors via the error *string function.*

Programs within programs

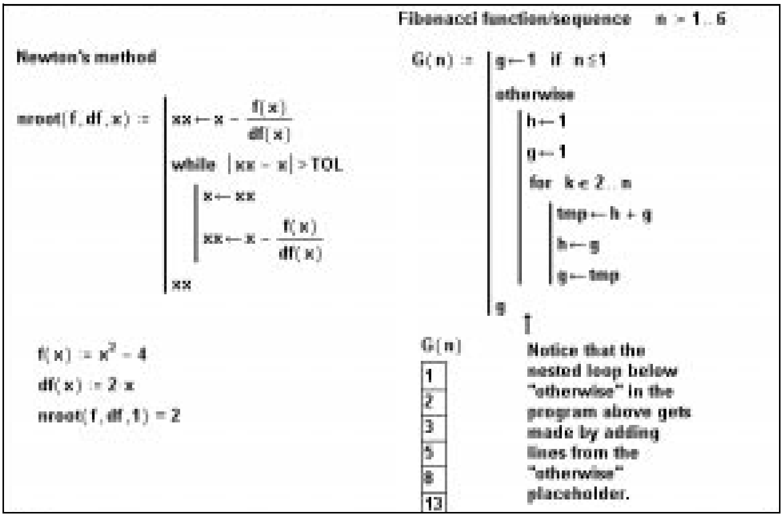
The examples in previous sections have been chosen more for illustrative purposes rather than their power. This section shows examples of more sophisticated programs.

Much of the flexibility inherent in programming arises from the ability to embed programming structures inside one another. In Mathcad, you can do this in the following ways:

- You can make one of the statements in a program be another program, or you can define a program elsewhere and call it from within another program as if it were a subroutine.
- You can define a function recursively.

Subroutines

Figure 15-9 shows two examples of programs containing a statement which is itself a program. In principle, there is no limit to how deeply nested a program can be.



Fibonacci function/sequence $n := 1..6$

```
$G(n) :=$
    $g \leftarrow 1$  if  $n \le 1$
    otherwise
        $h \leftarrow 1$
        $g \leftarrow 1$
        for  $k \in 2..n$
            $tmp \leftarrow h + g$
            $h \leftarrow g$
            $g \leftarrow tmp$
    $g$
```

Newton's method

```
$nroot(f, df, x) :=$
    $xx \leftarrow x - \frac{f(x)}{df(x)}$
    while  $|xx - x| > TOL$
        $x \leftarrow xx$
        $xx \leftarrow x - \frac{f(x)}{df(x)}$
    $xx$
```

$f(x) := x^2 - 4$

$df(x) := 2 \cdot x$

$nroot(f, df, 1) = 2$

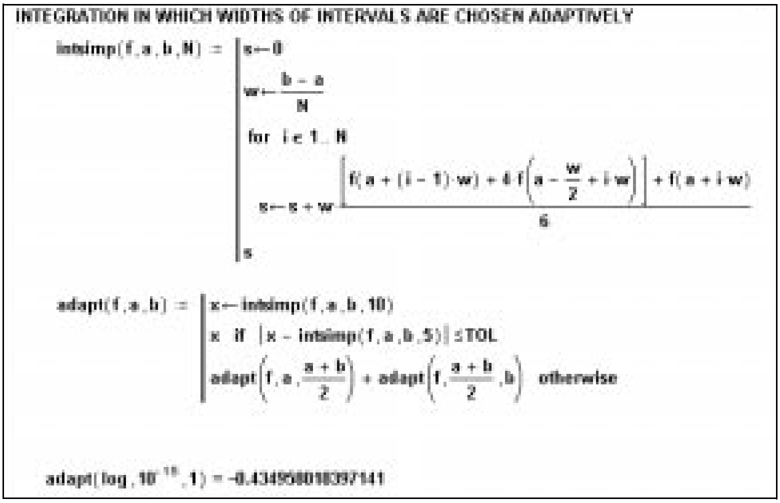
$G(n)$
1
2
3
5
8
13

Notice that the nested loop below "otherwise" in the program above gets made by adding lines from the "otherwise" placeholder.

Figure 15-9: Programs in which statements are themselves programs.

One way many programmers avoid overly complicated programs is to bury the complexity in *subroutines*. Figure 15-10 shows an example of this technique.

Tip Breaking up long programs with subroutines is good programming practice. Long programs and those containing deeply nested statements can become difficult for other users to understand at a glance. They are also more cumbersome to edit and debug.



INTEGRATION IN WHICH WIDTHS OF INTERVALS ARE CHOSEN ADAPTIVELY

```
$intsimp(f, a, b, N) :=$
    $s \leftarrow 0$
    $w \leftarrow \frac{b - a}{N}$
    for  $i \in 1..N$
        $s \leftarrow s + w \cdot \frac{\left[f(a + (i - 1) \cdot w) + 4 \cdot f\left(a - \frac{w}{2} + i \cdot w\right)\right] + f(a + i \cdot w)}{6}$
    $s$

$adapt(f, a, b) :=$
    $x \leftarrow intsimp(f, a, b, 10)$
    $x$  if  $|x - intsimp(f, a, b, 5)| \le TOL$
    $adapt\left(f, a, \frac{a + b}{2}\right) + adapt\left(f, \frac{a + b}{2}, b\right)$  otherwise
```

$adapt(\log, 10^{-15}, 1) = -0.434958018397141$

Figure 15-10: Using a subroutine to manage complexity.

The function *adapt* carries out an adaptive quadrature or integration routine by using *intsimp* to approximate the area in each subinterval. By defining *intsimp* elsewhere and using it within *adapt*, the program used to define *adapt* becomes considerably simpler.

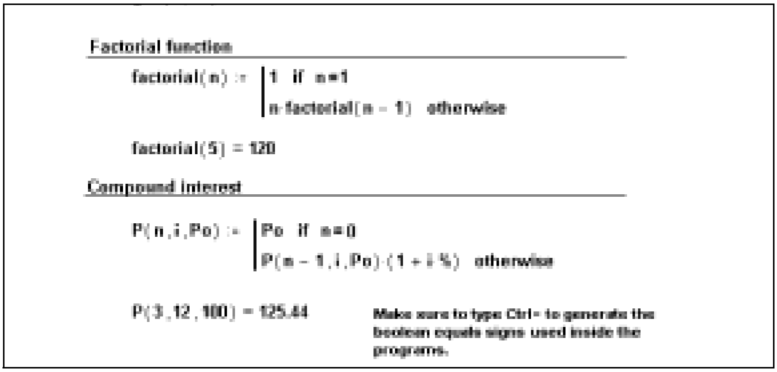
Recursion

Recursion is a powerful programming technique that involves defining a function in terms of itself, as shown in Figure 15-11. See also the definition of *adapt* in Figure 15-10. Recursive function definitions should always have at least two parts:

- A definition of the function in terms of a previous value of the function.
- An initial condition to prevent the recursion from going forever.

The idea is similar to that underlying mathematical induction: if you can determine $f(n+1)$ from $f(n)$, and you know $f(0)$, then you know all there is to know about *f*.

Tip Recursive function definitions, despite their elegance and conciseness, are not always computationally efficient. You may find that an equivalent definition using one of the iterative loops described earlier will evaluate more quickly.



Factorial function

$$\text{factorial}(n) := \left| \begin{array}{l} 1 \;\; \text{if} \;\; n=1 \\ n \cdot \text{factorial}(n-1) \;\; \text{otherwise} \end{array} \right.$$

$\text{factorial}(5) = 120$

Compound interest

$$P(n, i, Po) := \left| \begin{array}{l} Po \;\; \text{if} \;\; n=0 \\ P(n-1, i, Po) \cdot (1 + i\,\%) \;\; \text{otherwise} \end{array} \right.$$

$P(3, 12, 100) = 125.44$

Make sure to type Ctrl= to generate the boolean equals signs used inside the programs.

Figure 15-11: Defining functions recursively.