Moving into programming

Chapter Overview

The interactive document interface that we've worked with so far is good for quick development of calculations involving a few steps. Because feedback occurs after each step entered, this style of working is often the fastest way of getting such short calculations done. We've also seen how the effort to develop a script can be made to yield greater payback by finding and exploiting situations where reuse with minor editing and re-execution.

With longer scripts or extensive re-execution, editing and execution (other than of the entire worksheet) becomes tedious even if we are saving time by having the computer do the calculations. In this chapter, we introduce a new way of entering and executing scripts: code edit regions. While such regions make it more convenient to work and execute blocks of Maple instructions, the programmer must enter things using the textual version of Maple expressions and operations. Individual lines must be separated by semi-colons (;) so that the Maple language processor can more easily tell where one instruction ends and the next begins.

With code edit regions, the programmer must also work harder to see how the state of the computation is changing during the computation, rather than only looking at the final output. This is because if the final result is wrong, the programmer must find at what point in the script mistakes (called program bugs) occur. Work with code edit regions often requires adding print statements into the script to better see changes in variables and intermediate results.

As our first extended experience with code edit regions, we look at the CarSimulator package used for the first two labs of CS 122. To use it, two new programming elements are needed: procedures (blocks of code that are delimited by proc()... end proc) and conditional loops (blocks of code that are delimited by while condition do ... end do). Procedures (multi-line functions) and loops are used in most programming languages and in many of the programs that we will write from this point onwards.

Code edit regions: entering a block of code, executing it all at once

We have seen that even some of the easiest technical calculations break down into a series of operations, chained together. Realizing that we will typically need the computer to perform a "series of operations" is the motivation for making the transition to programming, where execution of many operations in a block is typical.

The Maple document interface that we have been using so far easily supports calculations where the user prompts to computer to do a series of steps by positioning the cursor at the first operation and then hitting return (or enter) repeatedly. We now introduce a second way to enter instructions and have them executed as a block. This alternative is often easier to work with when you have a few dozen instructions and will want to execute them all in a chain.
One can open a text field in the Maple document where one can enter a series of instructions. To do this, position your cursor where you want the text field to appear in the document, and with the mouse perform Insert->Code Edit Region

**Insert->Code Edit Region menu**

Once the field has been created, you can type the textual version of the Maple instructions you want executed. Each instruction (commonly referred to as a *statement*) must be separated by either a semi-colon, or a colon. If the statement ends in a semi-colon, then its value will be printed during execution of the region. If the statement ends in a colon, then printing of its value will be suppressed just as it is in operation of documents.

Once all the instructions have been entered, you can run them all in a series by typing control-e (command-e on Macintosh), or by entering right-click->Execute Code (on Macintosh, control-click->Execute Code). Instead of typing control-e, you can as an alternative click on the "!!" icon on the Maple toolbar. (The "!!!" icon executes the entire worksheet, including all code edit regions.)
The results for each statement will appear below the region in blue, except for the statements whose printing is suppressed by a colon. The output in blue is referred to as the execution trace produced when the code region's actions are carried out.

---

**Execute Group icon ("!" icon) on Maple Toolbar**

Any portion of a line that begins with a "#" is a program comment. The rest of the line is regarded by Maple as something that people will read, not an instruction to be performed by the computer. Typically what appears after the # is commentary written in English (or whatever language is convenient for communication with the intended audience) that helps explain/remind human readers what a segment of code is about. As you read programs with comments, you will see that some programmers are "chattier" than others. This is because they feel that their intended reading audience needs more explanation.

In professionally written code, you will often see comments at the beginning of the region that give an overview of what the code region does, the name of the author(s), and the date of creation/modification. Professional organizations often mandate a particular style and content for such "header" comments, so that it will be easy to find information about any code written by several programmers working on a single project.

---

**An example from Chapter 7 as a code edit region**

<table>
<thead>
<tr>
<th>Code region</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Segments of lines that begin with # are regarded as program comments (for the program reader's eyes), not operations for Maple to carry out.</td>
</tr>
<tr>
<td></td>
<td>Separate actions are separated by semi-colons.</td>
</tr>
</tbody>
</table>
# We reset Maple so previous assignments are ditched.
restart;

# We assign two variables with pressure and temperature data.
pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3];

# Load in the package with(CurveFitting);

# Develop a linear formula for temperature as a function of pressure.
pressureFormula := LeastSquares(tData, pData, t);

# We can also fit a line to pressure as a function of temperature.
tempFormula := LeastSquares(pData, tData, p);

[134.2, 142.5, 155.0, 159.8, 171.1, 184.2]
[0, 20.1, 39.8, 60.0, 79.9, 100.3]

[ArrayInterpolation, BSpline, BSplineCurve, Interactive, LeastSquares, PolynomialInterpolation, RationalInterpolation, Spline, ThieleInterpolation]
133.500049011592 + 0.485837074076807 \( t \)
-271.054958033645 + 2.03467442775863 \( p \)

The code region will develop a scroll bar if the amount of text entered exceeds the size of the window. The size of the field can be adjusted by right click->Component Properties, and then modifying the integers listed for the width and height.
Enlarging a code Region window by changing component properties via the clickable menu

```plaintext
# Figure 11.1.1
#initialize variables
i := 1;
val := .3; #evaluation point
term := (val^i)/i; # a term to compute
print("term is", term); #message
s := term; #s has a copy of the term
tol := 10e-7; #A small value.
```
The "Collapse Code Edit Region" menu item of the clickable menu will reduce the entire window to an icon. If the first line of the region is a program comment, it will be listed to the right of the icon. Clicking on the code icon will execute the code within.

### A collapsed code edit region

<table>
<thead>
<tr>
<th>Code region</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Figure 11.1.1" /></td>
<td>We collapsed the region by positioning the cursor in the code region window, and then entering right click-&gt; Collapse Code Edit Region. Note that the comment on the first line becomes displayed next to the code edit icon.</td>
</tr>
</tbody>
</table>

```
# Figure 11.1.1
#initialize variables
i := 1;
val := .3; #evaluation point
term := (val^i)/i; # a term to compute
print("term is", term); #message
s := term; #s has a copy of the term
tol := 10e-7; #A small value.
```

---

**Troubleshooting common errors in entry into code edit regions**

New kinds of errors and warnings can appear in code regions, due primarily to the new requirement
that statements
be separated by semi-colons and colons. Trying to enter too many lines at once before testing what
you have so far can make older errors appear more mysterious.

<table>
<thead>
<tr>
<th>Code regions with entry errors</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Example</strong></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5 + cos( Pi/2) ;</code></td>
<td>The error message indicates that a semi-colon was not expected where Maple scanned it. This is typically semi-colons can appear only at the end of a complete Maple statement. This message typically indicates that something went wrong in what was entered before the semi-colon. Edit the code region to make the statement correct, and re-execute the region. The problem here is a missing parentheses ). To fix this, you'd put a closing ) after the &quot;3.5&quot;, and then re-execute the region with a control-e (or command-e).</td>
</tr>
<tr>
<td><code>Error, </code>;<code> unexpected</code></td>
<td></td>
</tr>
<tr>
<td><code>a := sin(3.5) + cos( Pi/2) ;</code></td>
<td>This is only a warning, and executing this region does perform the computation correctly.</td>
</tr>
</tbody>
</table>

-0.3507832277
a := sin(3.5) + cos( Pi/2)

> Warning, inserted missing semicolon at end of statement
-0.3507832277

a := sin(3.5) + cos( Pi/2)
b := cos(3.5) + sin( Pi/2)

> Error, missing operator or `;``

If we add another line without a semi-colon between, we get just an error. Both lines are messed up because Maple needs an explicit separator between the statements. Just putting the next instruction on another line is not enough.
\( a := \sin(3.5) + \cos(\frac{\pi}{2}) \); \\
\( b := \cos(3.5) + \sin(\frac{\pi}{2}); \)

\(-0.3507832277 \\
0.0635433127

\( a := \sin(3.5 + \cos(\frac{\pi}{2}) \); \\
\( b := \cos(3.5) + \sin(\frac{\pi}{2}); \)

Error, `;` unexpected

If you try to enter several lines at once before entering, you have the additional problem of figuring out which line the mistake occurred on. In this case, you have to decide which of the two semi-colons was unexpected.

Fortunately, you can sometimes get some help if you are observant -- the cursor will often flash on the line where the problem is first noticed (the first line in this case).

There's both a missing closing parentheses and a missing semi-colon here. The symptom is that there is a warning message (which isn't necessarily a sign of trouble, but is here), plus the fact that you are expecting a value to be printed from the computation but just see a red ">" with no numeric output.

Fixing both problems and re-executing the region will produce the correct result.
a := sin(3.5 + cos( Pi/2)

> Warning, premature end of input, use <Shift> + <Enter> to avoid this message.

b := cos(3.5) + sin( Pi/2);

Error, missing operator or `;`;

Sometimes when you enter a line, you make more than one error. Even though the same mistake has been made as in the previous example, there is a different error message than before. Executing this region causes the message indicated, plus the cursor will be observed to flash at the start of the second line.

Rather than being in an "I was expecting more" state which produced the first warning message, Maple is in a "I was expecting more, but wait what I've seen isn't what I was expecting". Since there's nothing before the cursor started flashing before any of the second line was parsed, it means that the error must have happened before that on the previous line.

There are still two things to fix. Fixing them will produce two lines of output.

print and annotating an execution trace

While executing a code region is convenient, a disadvantage is that the output of a result is separated from the line of code that produced it. Furthermore, many lines of output are not very useful other
than as confirmation that an intermediate action didn't fail.
As is true with interactive operation of Maple, ending a line with a colon (:) instead of a (;) will suppress the output. *print(....)* will cause a line of output which will list the value of each item between the parentheses. The programmer can tailor the use of each to produce more intelligible information.

<table>
<thead>
<tr>
<th>An example from Chapter 7 as a code edit region</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Code region</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
# We reset Maple so previous assignments are lost.
restart;

# We assign two variables with pressure and temperature data.
pData := [134.2, 142.5, 155.0, 159.8, 171.1, 184.2];
print("Pressure data is:", pData);
tData := [0, 20.1, 39.8, 60.0, 79.9, 100.3];
print("Temperature data is:", tData);

# Load in the package
with(CurveFitting):

# Develop a linear formula for temperature
# as a function of pressure.
pressureFormula := LeastSquares(tData, pData, t):
print("Fit of temperature to pressure is:", p=pressureFormula);

# We can also fit a line to pressure as a function
# of temperature.
tempFormula := LeastSquares(pData, tData, p):
print("Fit of pressure to temperature is:", t=pressureFormula);

"Pressure data is:", [134.2, 142.5, 155.0, 159.8, 171.1, 184.2 ]
"Temperature data is:", [0, 20.1, 39.8, 60.0, 79.9, 100.3 ]
"Fit of temperature to pressure is:", p = 133.500049011592
The CarSimulator package: a first view

Let's see an example of working with a package that requires us to write Maple programs in code regions. We are given the problem and a place to work out its solution as a Maple worksheet with multiple code edit regions.

Executing the first region does Simulator Set Up -- initializing some of the variables and data structures needed for the simulator to work. It also loads in the CarSimulator package (in the file CarSimulator.mla, which must be in the same directory as the worksheet we opened up).

Problem: run the simulator in a 20 x 20 empty arena, so that the car moves forward one square, turns left, and then moves forward two squares.

```
#Code region #1
#Reset all assignments in Maple so that they are forgotten
restart;

#Look for packages in the current directory ("." as well as
#built-in Maple library.
libname := ".", libname;

#Load CarSimulator package
with(CarSimulator);
```

".", "/opt/maple15/lib"
[ARROW, BARRIER, BLUESIGN, BLUESIGNHIT, CAR, CARBUMPED, CARFINAL, 
  CARNORMAL, CARRIAGECOLOR, CARSTATES, CARSTOPPED, CARTRANSPARENCY, 
  GREENSIGN, GREENSIGNHIT, HitState, MARKING, OCCUPANCYSYMBOLS, 
  OtherState, PURPLESIGN, PURPLESIGNHIT, REDSIGN, REDSIGNHIT, SIGNS, 
  STARTLOCATION, TARGET, TOLERANCE, YELLOWSIGN, YELLOWSIGNHIT, 
  carMovie, carStop, changeSignState, destroy, drawArrow, drawBackground, drawCAR, 
  drawRectangle, drawTrail, establishBoundaries, foundGoal, getCURRENTLOCATION, 
  getNextCoords, hasOverlap, initialize, isIn, isOccupied, isTouching, isTouchingSign, 
  maxMoves, move, moveReverse, onTopOfSign, record, roundTolerance, run, scanDistance, 
  seeIfObstacleRectangle, setBackground, turn, turnAbsolute, unsetBackground]

Table 1: First code edit region for CarSimulator work

After loading the package, Maple on-line help for it is also available. Clicking on the CarSimulator folder in the table of contents for on-line help indicates that there are several levels of help. For example, under Setup we some of the operations mentioned in the first code region. Under Car Operations we see move and turn. As with the rest of on-line help, clicking on any of the entries will bring up a help page with a further description. We realize that using a typical system's package, with dozens of features, will require doing more reading of on-line help to get the details of features we are likely to need.
Executing the second region sets up and shows the arena. The arena has a rectangular coordinate system. The car typically starts at coordinate (1,1) facing rightwards. The orientation of the car is given in radians, and the initial orientation is 0 radians. The arena is shown by calling the package operation `carMovie` with argument `stateTable` (a feature of the simulator that we could read about if we looked further in the Data folder of the package).
#Code region #2
# We don't put a restart here because we want to
# remember what we did in the previous code region.
# To start afresh, rerun all of the regions.

# Initialize car simulator
initialize();

establishBoundaries(20,20): #draw a 20 x 20 sandbox for the car to roam around in.
Coordinates range from [0,0] to [20,20].
drawBackground(): #Create a plot of the wall and the target for later use, but don't display it yet
carMovie(stateTable); #Show initial configuration
Table 3: Second code edit region with Car Simulator

We want to have the car move forward one square and turn left, and then move forward two more squares, like this:
To do this, we write a control program named run1a. The program consists of a script using operations from the CarSimulator package. The arguments we give to turn we derive from the information found in the on-line help page: a left turn requires an argument of π/2 radians. We see some new syntax in the control program: the control script is bracketed by proc() and the phrase end proc. This defines the script as a procedure. A procedure is a way of encapsulating a script so that it has a clearly defined beginning and end, along with named parameters. A procedure is expected to name parameters and optionally have a result, like the functions defined by the arrow notation described in chapter 7. run1a needs no parameters, which is why it has ( ) rather than a parenthesized parameter sequence ( x) or (tData, pData). For the next chapter all of our procedures will have no parameters, but we will eventually visit situations where we need to write procedures that need one or more parameters.
#Code Region #3

```maple
run1a := proc()
    move(1);  #move forward one square
    turn(Pi/2);  #turn Pi/2 radians (left 90 degrees)
    move(1);  #move forward one square
    move(1);  #move again.
end proc:
run(run1a);
```

"Car stoppped after 4 moves"

### Table 5: Second code edit region with Car Simulator

After defining the control procedure for the car, we submit the procedure (by its name, `run1a`) to the `run` operation of the CarSimulator package. The execution trace indicates that the simulation has taken 4 moves. However, there is no indication of what happened in the trace.

To see the simulation at work, we again perform the operation `carMovie(stateTable)`. Because the simulation had multiple moves, this now causes an animation window to appear which when played displays the movement of the car frame-by-frame. The animation is created by the Maple programming in the package -- we will learn how to make movies in that fashion in a few chapters.
Table 6: Fourth code edit region with Car Simulator animation result

The worksheet with the four code regions thus sets up the arena for the problem, describes the control procedure that is the solution to the stated problem, and shows the results of running the simulator with the procedure, which demonstrates that it solves the problem.
A second task with the CarSimulator

We set up a more elaborate arena (set up code not shown):

```
".", "/opt/maple15/lib"
15
12
"Starting configuration is:"
```

Setting up an arena with barriers

The goal is to move forward until the barrier is reached, then turn around and move forward until
the opposite barrier is reached.

The control program, illustrated in the following code region, uses two new programming elements, **while** and **isTouching**. A description of the isTouching function can be found in the on-line help for the CarSimulator package. isTouching is a function that returns true or false -- truth or Boolean values, rather than a number, formula, or plot structure as we have seen with other functions. **while** is part of the programming language notation Maple uses to describe a loop. Programming loops typically contain a block of actions that are to be performed a number of times, as well as a **continuation condition** that allows the computer to decide whether to continue to perform the block. The decision to repeat the loop is done typically only at the start of the would-be repetition. Once the decision is made to do it, the whole block is performed before the continuation condition is considered again.

```maple
barrierBounce := proc()
    while not(isTouching(0, 'bt')) do
        move(1);
        end do;
    turn(Pi); #turn around
    while not(isTouching(Pi, 'bt')) do
        move (1);
        end do;
    end:
run(barrierBounce);
print("Simulation result is:");
carMovie(stateTable);
```

In the loop in the control program, the block consists of the single action "move(1)". Thus the loop tells the simulated car to check if the front of the car is touching a barrier. If not, it moves forward one square. Repeating the loop means to check the continuation condition again and then perform the block's actions. Thus this loop performs check-move-check-move.... stopping when the next check determined that the front of the car is touching a barrier.

Once the loop is over, the program will perform the next action listed, which is the **turn**. After that, there is another loop which does the check-move-check.... repetition until the other barrier occurs.

"Car stopped after 10 moves"

"Simulation result is:"
Moving the car between two barriers: a control program defined and a simulator run of it

Motion of the car in the simulator

| At the start (partial image zoom in) | Taking several more moves to the wall, and turning. (zooming in some more) | Taking several moves towards the other wall | At the finish -- touching the other wall |

Programming to solve a class of problems

At this point, those of you not be familiar with while may be wondering why we go through the trouble of writing a loop with a conditional expression when we can just build something that has exactly the right number of moves and turns:
A solution to the first "bounce off the wall" problem

However, consider the following similar arena problem (details of set up code hidden in code edit region):

```
#Something that solves one kind of arena
lessRobustSoln := proc()
  move(1);
  turn(Pi);
  move(1);
  move(1):
  move(1):
  move(1):
  move(1):
  move(1):
  move(1):
  move(1):
  move(1);
  move(1);
  move(1);
  move(1);
  move(1);
  move(1);
  move(1);
  move(1);
  move(1);
end proc:
```

We're not resetting everything, but we will initialize with a new arena

```
15
12
"Starting configuration is:"
```
A similar arena problem

Our "less robust" solution won't work here because the number of moves is different. If we tried to write a similar program that would solve this new arena, it would get a bit tedious repeating the "move(1)" instruction the necessary number of times. However, the original solution `barrierBounce` will work fine:

```csharp
run(barrierBounce);
print("Simulation result is: ");
carMovie(stateTable);
```

"Car stopped after 21 moves"
"Simulation result is:"
Building a program that fits exactly one situation means that you will have to reprogram it if the situation changes. If you're going to go through the effort of building something, it will pay to design for a reasonable amount of flexibility. For that reason, appropriate testing of a program will put it through its paces in a number of different situations, to check that the desired breadth is really there.

**Reviewing old and introducing new programming concepts**

The first two labs of CS 122 uses a package called the CarSimulator. It allows you to write simple programs that control a simulated car moving around an obstacle course. The simulator is useful for rapidly prototyping ideas for controlling an actual robot car.

Using the CarSimulator on a problem requires the use of elements of Maple programming that we have already seen:
### Programming elements used in Car Simulation

We will also get a first look at some new elements:

<table>
<thead>
<tr>
<th>Element</th>
<th>First introduced in Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>4</td>
</tr>
<tr>
<td>Lists</td>
<td>5</td>
</tr>
<tr>
<td>Scripts</td>
<td>5</td>
</tr>
<tr>
<td>Library packages</td>
<td>7</td>
</tr>
<tr>
<td>On-line help</td>
<td>6</td>
</tr>
<tr>
<td>Textual entry of actions</td>
<td>6</td>
</tr>
<tr>
<td>Functions</td>
<td>7,8</td>
</tr>
<tr>
<td>Animations</td>
<td>9</td>
</tr>
<tr>
<td>Code edit regions</td>
<td>10</td>
</tr>
</tbody>
</table>

### New programming elements used in Car Simulation Lab

<table>
<thead>
<tr>
<th>Element</th>
<th>Discussed further in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedures</td>
<td>11</td>
</tr>
<tr>
<td><strong>While</strong> loops</td>
<td>12</td>
</tr>
<tr>
<td>Methodical testing and troubleshooting code blocks</td>
<td>13</td>
</tr>
<tr>
<td>Tables</td>
<td>14</td>
</tr>
<tr>
<td><strong>For</strong> loops</td>
<td>15</td>
</tr>
<tr>
<td><strong>If -then-else</strong> statements</td>
<td>16</td>
</tr>
</tbody>
</table>

### Summary of Chapter 10

**Use of print**
We set up a list of values. L has all floating-point values because the "10.0" in the denominator causes all computations to be done using limited-precision arithmetic. The display of the value of L is suppressed because we ended the assignment with a colon.

> print("This list presents some values of the function ", exp(x)); print(". The last element of the list is:", L[-1]);

> "This list presents some values of the function \(e^x\). The last element of the list is:", 0.7408182207 **(1.9.1)**

> print("There are ", nops(L), " elements in this list. The largest value is", max(L))

> "There are ", 3, " elements in this list. The largest value is", 0.9048374180 **(1.9.2)**

---

**Code edit regions**

Insert -> Code Edit Region to create a region.

control-e (command-e in Macintosh) or right-click -> Execute Code in a code edit region to execute all the code in it.

right-click->Component Properties to open a dialog box with the dimensions of the code region (in pixels). Change these dimensions by entering different numbers to enlarge or shrink the box. Sorry, dragging will not resize the code region.

right-click->Collapse Region to turn the code region into an icon.

Clicking on the icon will then execute the code inside.

The first line of code appears as the label for the icon.

right-click->Expand Region to turn a code region icon to display the region again.
All Maple statements must be separated or terminated with either a semi-colon (;) or a colon (:). Usually each statement executed generates a line of output which appears in blue after the region. This sequence of output is called the execution trace of the code. If a colon is used, then the output that normally results from evaluating the statement is suppressed from the execution trace.

Comments

Any portion of a line of code starting from a number sign (#) is treated as commentary and not as an operation to be performed.

Example

```
# Figure 11.1.1
#initialize variables
i := 1;
val := .3; #evaluation point
term := (val^i)/i!; # a term to compute
print("term is", term); #message
s := term; #s has a copy of the term
tol := 10e-7; #A small value.
```

<table>
<thead>
<tr>
<th># Figure 11.1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize variables</td>
</tr>
<tr>
<td>i := 1;</td>
</tr>
<tr>
<td>val := .3; #evaluation point</td>
</tr>
<tr>
<td>term := (val^i)/i!; # a term to compute</td>
</tr>
<tr>
<td>print(&quot;term is&quot;, term); #message</td>
</tr>
<tr>
<td>s := term; #s has a copy of the term</td>
</tr>
<tr>
<td>tol := 10e-7; #A small value.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segments of lines that begin with # are regarded as program comments (for the program reader's eyes), not operations for Maple to carry out.</td>
</tr>
<tr>
<td>The results in blue are displayed after we position the cursor in the code edit region and type control-e (command-e on Macintosh), or enter Execute Code Region via the clickable menu.</td>
</tr>
<tr>
<td>Result of first assignment (to i)</td>
</tr>
<tr>
<td>Result of assignment to val.</td>
</tr>
<tr>
<td>Result of assignment to term.</td>
</tr>
<tr>
<td>Result of calling the print function.</td>
</tr>
<tr>
<td>Result of assignment to s.</td>
</tr>
<tr>
<td>Result of assignment to tol.</td>
</tr>
</tbody>
</table>

| 1 |
| 0.3 |
| 0.3 |
| "term is", 0.3 |
| 0.3 |
| 0.0000010 |
```maple
#Code region #1
#Reset all assignments in Maple so that
#they are forgotten
restart;

#Look for packages in the current
directory ("." ) as well as
#built-in Maple library.
libname := ".", libname;

#Load CarSimulator package
with(CarSimulator);
```

To successfully use the CarSimulator, the CarSimulator.mla and CarSimulator.hdb library files need to be located in a directory specified by the variable `libname`. Assigning `libname` to include ".", and then opening and working on a worksheet located in the same directory as the library files will work.

You can tell that when the loading works if the `with` (ending with a semi-colon rather than a colon) produces a list of all the items included in the simulator package.

Only do a `restart` if you want all previous assignments to be erased.
CarSimulator initialization

#Code region #2
#We don't put a restart here because we want to remember what we did in the previous code region.
#To start afresh, rerun all of the regions.

#Initialize car simulator
initialize();

establishBoundaries(20,20): #draw a 20 x 20 sandbox for the car to roam around in. Coordinates range from [0,0] to [20,20].
drawBackground(): #Create a plot of the wall and the target for later use, but don't display it yet
carMovie(stateTable); #Show initial configuration

Some actions are necessary to initialize the simulator before it is used. Typically they are CarSimulator set up actions such as initialize and establishBoundaries.
Control procedure for car control

A procedure begins with `proc()` and ends with `proc end`. The procedure contains a number of actions such as `move` and `turn`, although it can contain any other Maple operation such as assignments, `print`, or calculations.

A control loop within a procedure is of the form

```
while condition do
    action;
    action;
    ...
end do;
```

If the `condition` is true, then the actions are performed. Then the `condition` is evaluated again. If it is still
barrierBounce := proc()  
    while not(isTouching(0, 'bt')) do  
        move(1);  
    end do;  

    turn(Pi); # turn around  
    while not(isTouching(Pi, 'bt')) do  
        move(1);  
    end do;  
end:

true, the actions will be repeated. This continues until the condition is found to be false when it is evaluated. The condition is evaluated only once per repetition of the actions, at the start.

CarSimulator simulation and visualization of results

Use the run operation of the CarSimulator, giving it the name of a control procedure that has been previously created. After the simulation is over, see the results by applying the carMovie operation of the package to the stateTable generated by the simulation.
run(barrierBounce);
print("Simulation result is:");
carMovie(stateTable);

"Car stopped after 39 moves"
"Simulation result is:"
More on \textit{proc} and \textit{while}

Chapter Overview

We continue our discussion of programming with two new elements, procedures and loops under the control of \textit{while}.

Procedures are an alternative way (compared to the -> notation) of creating a function. As with all functions, they take arguments and returns results. Unlike arrow notation functions, their definition can span more than one line of code, including control constructs such as \textit{while}. Procedures can have local variables, whose use is encapsulated. This allows a procedure programmer to use a variable for internal use during a procedure execution and not have to worry about interfering with the use of that variable outside of the procedure.

Review of function terminology

Although they don't use the same syntax, procedures are a kind of function. The technical terminology of procedures shares a lot with the terminology we have already used for functions. Let's review what we know function terminology in Maple.

<table>
<thead>
<tr>
<th>Review of function terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Functions process input values and produce an output value. The function \textit{definition} describes how the values of those parameters are used to compute a function \textit{result}. When we defined functions using the -&gt; (arrow) notation, the result was described as an expression involving arithmetic, structure creators such as [], and built-in (e.g. \textit{sin}, \textit{sqrt}, or \textit{seq}) functions or functions previously defined by the user. The names given to the symbols used for the input values are called the \textit{formal parameters} to the function. Functions can have zero or more parameters. The \textit{function result} is typically the expression computed by the function.</td>
</tr>
<tr>
<td>$f := (a, b) \rightarrow \sqrt{a \cdot b}$</td>
</tr>
<tr>
<td>There are two formal parameters to the function defined above. \textit{a}, and \textit{b} are their name.</td>
</tr>
</tbody>
</table>

| 2. Functions are associated by a name by making the function definition the right hand |
| The function $(a, b) \rightarrow \sqrt{a \cdot b}$ is named $f$ through the assignment. |
3. Defining a function does not perform the computation, it just stores the directions for how to do the computation within the computer. After a function is defined, you can get the computer to perform the computation by invoking the function. Invocation involves giving the name of the function, and values for the function's arguments, enclosed in parentheses. The values supplied to the function when it is invoked are called the *actual parameters* or *arguments* to the function.

\[ f(3.1, \pi) \quad \# \text{The actual parameters (arguments)} \]
\[ \quad \# \text{are 3.1 and } \pi. \]

4. We can combine the actions of several functions together by composing (also known as daisy-chaining) them. Rather than the "one after another" sequencing of actions that we might get from using functions on several lines of a script, function composition uses "inside out" sequencing.

\[
\max(f(3.1, 6), f(4.7, 3))
\]
This is the same as the sequence

\[
\begin{align*}
\text{result1} & := f(3.1, 6); \\
\text{result2} & := f(4.7, 3); \\
\text{result3} & := \max(\text{result1}, \text{result2})
\end{align*}
\]

Programming languages feature functions (and procedures) prominently because of their advantages in the construction of programs. Writing a computation as a function *encapsulates* it: there is a well-defined starting point given by the values supplied for the parameters, there is a specific computation described within the function definition, and there is a final result to the computation when the function delivers its output. Only when the function is later invoked are those actions performed. The function may be invoked multiple times (as would occur when the users invokes it when doing several problems, or when *map* is used with a function) after it is defined. The expectation of reuse is one of the justifications for the extra work needed to define a function, since re-invocation is typically easier than even copying and pasting the actions.

#### Creating a short function with proc... end

One line functions created by -> have the limitation that we can't create functions that have code which includes assignment, loops, or other actions that require more than one line of code. In the previous chapter, we saw the use of proc ()... end proc

---

<table>
<thead>
<tr>
<th>Maple procedure definition -- the simplest form</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>function name := proc (formal parameters )</code></td>
</tr>
<tr>
<td><code>statement;</code></td>
</tr>
<tr>
<td><code>statement;</code></td>
</tr>
<tr>
<td><code>... (one or more)</code></td>
</tr>
</tbody>
</table>
The *procedural result* is by default the last thing computed in the sequence of actions computed by the function when it is invoked.

### Example: The simple function defined with -> and then again with proc

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f := (a, b) \rightarrow \frac{\sqrt{a \cdot b}}{2} )</td>
<td>This is a function named ( f ) defined using the arrow notation.</td>
</tr>
<tr>
<td>( f(3.0, \text{Pi}) )</td>
<td></td>
</tr>
<tr>
<td>( f2 := \text{proc}(a, b); \frac{\sqrt{a \cdot b}}{2}; \text{end} )</td>
<td>( f2 ) has the same behavior as ( f ). Note that there are no local variables here -- all the symbols that appear in the function come from the parameters.</td>
</tr>
<tr>
<td>( f2(3.0, \text{Pi}) )</td>
<td>Note that you can end a procedure with either &quot;end&quot; or &quot;end proc&quot;.</td>
</tr>
</tbody>
</table>

### Creating a function with encapsulated (local) variables

**General form of a Maple procedure definition with local variables**

```
function name := proc( formal parameters )
local variables used within procedure;
statements;
end proc;
```

**Notes**

1. There is no semi-colon between the proc ( ) containing the formal parameters and the word local. If there are no local variables for the procedure, then you need to put in the semi-colon: proc ( formal parameters ) ; statements; ....
2. The opposite of a "local variable" is a "global variable". We don't use them much in the Maple procedures we will be working with. You can read more about them if you look up "global" in Maple on-line help.
3. You can't assign a value to a formal parameter in the statements of the procedure. parameters are used only as a way of providing input information to the function.
Usually local variables arise when the function needs to compute and store intermediate results that are used for the final answer but are not part of them.

An important feature of local variables is that use of their use inside the function is encapsulated. When a variable is declared to be a local variable of a function, the value that the symbol outside of the function invocation may have is temporarily set aside. The local symbol within the function starts off with no value, but may be assigned values within the programming of the function. Once the function invocation is over, the old value that the symbol had is restored to it.

### Demonstration that local variables do not pay attention to assignments made outside the scope of the procedure definition.

```plaintext
# This function demonstrates the use of local variables.

testFunction := proc() local i;
        print("Line 1: i's value is",i);
        i := 1;
        print("Line 2: i's value is",i);
end proc;
```

We define a `testFunction`, whose purpose is to assign a value to `i`, and to print out it's value before and after assignment.

The execution trace we get from executing this code region after the previous one is explained by:

a) the first zero is from the assignment to `i`.
b) The "Line 0" output is the assigned value of `i`.
c) Once `testFunction` begins execution, the declaration of `i` as a local variable means that previous assigned values are temporarily set
#This is a "non-local" use of i;
i := 0;
print("Line 0: i's value is",i);
testFunction();
print("Line 3: i's value is",i);

aside. This is why the Line 1 output is just i.
d) The Line 2 output reflects the assignment within testFunction.
e) The Line 3 output comes after the invocation of testFunction is completed. By that time, the previously assigned value of i is restored.

Table 1: Demonstrate of assignment of local variables

The behavior of local variables is seems more complicated, but the reality is that it frees the programmer from having to worry about the whether their use of a variable in a function will break other code outside of the function if they don't restore the previously assigned value. One could avoid local variables if one never used functions or procedures, but this isn't practical for serious projects -- the complexity of coding will defeat the attempt to complete the project successfully without the ability to break the task into the smaller pieces that coding with procedures allows.

Conditional repetition through while... do... end do

The key understanding of loops written with while is: "keep on repeating this while a condition is true".

The general form of the while statement

```
while condition do
    loop body;
end do;
```

The condition is something that is either true or false. An error results if it isn't. The condition is
evaluated only once per repetition, at the start before any of the code in the loop body is executed. If the condition is true, then the entire loop body is executed. If not, the repetition is finished and execution moves onto the next instruction after the end do.

The condition is evaluated at the start of the loop iteration. It does not matter if the condition changes in the middle of the loop body; this won't terminate execution of the loop until the execution reaches the end of the loop body and the condition looked at again before the start of the next repetition of the loop.

The continuation condition is can be a function invocation that returns true or false, such as the isTouching or onTopOfSign functions of the CarSimulator package. It can also be an inequality such as \( x < .1 \) or \( \text{abs}(y-z) > 0.1 \). It always must be an expression that can be evaluated to be true or false. If the inequality is numeric such as \( x < .1 \), variables in the condition such as \( x \) must have been assigned a numeric value earlier in the Maple session.

It is also possible to combine several conditions together using the operations and, or, and not(....).

true and false are referred to as Boolean values (after the British 19th century, George Boole -- http://www.gap-system.org/~history/Biographies/Boole.html for more information). Functions that return Boolean values are referred to as Boolean functions, while and, or and not are referred to as Boolean operations.

**Example: car control procedures with conditional expression built from Boolean operations and multiple loops**

The problem here is to go to the first four way intersection, and turn left. After the turn, go forwards to the target.

Here is a sample scenario. We want to write a control program that will work this scenario as well as with similar ones. The similar scenario will have a four way intersection and a target to the left, but it may have a number of other (three way) intersections before that. The distance to the four way intersection may vary between scenarios.

Generate test problem (code not shown)

"", "/opt/maple15/lib"
The car will automatically stop when it is pointing at and directly next to a blue target, so the control program should just run the car forward while it hasn't seen the 4-way intersection, then turn and run forward until it is touching the target. We can express "not at a four way intersection" by using Boolean operations to construct the condition that the car is touching something in one of the four directions -- (istouching(-Pi,'bt')) or (isTouching(Pi/2,'bt')) or (isTouching(0,'bt')) or (isTouching(-Pi/2,'bt'));
goToFourWay := proc()
    while isTouching(-Pi,'bt') or(isTouching(Pi/2,'bt')) or (isTouching(-
    (isTouching(0))) do
        move(1);
        end do;
        turn(Pi/2);
        while not(isTouching(Pi/2,'bt')) do
            move(1);
            end do;
    end proc:
run(goToFourWay);
carMovie(stateTable);

"Car stoppped after 27 moves with error.", "Car stopped after 27 moves"
Example: car control procedures with a local variable

Our scenario here is to go around four corners in a counterclockwise direction, then stop.

Generate test problem

```
./, "/opt/maple15/lib"
8
9
```
We present two solutions to this. The first solution repeats the code that says "move forward while touching the wall to the left, then turn left" four times. Since the isTouching operation uses absolute rather than relative directions, we must change the angle with each turn from $\frac{\pi}{2}$ to $\pi$ to $\frac{3\pi}{2}$ to $2\pi$.

We do this by introducing a local variable "angle" that we increment by $\pi/2$ each time. The reason why we do the incrementing angle := angle + $\pi/2$; rather than just angle := $\pi/2$; angle := $\pi$; angle := $3\pi/2$, etc. will become clear in the sequel.
```
turnFourTimes := proc()
    local bt, angle;
    # Move forward while touching a wall, then
    # turn left
    # Initial scan angle is $\pi/2$
    angle := $\pi/2$;
    while isTouching(angle, 'bt') do
        move(1);
        end do;
    turn(Pi/2);
    move(1);
    angle := angle + Pi/2; # angle is now $\pi$
    # Do that again.
    while isTouching(angle, 'bt') do
        move(1);
        end do;
    turn(Pi/2);
    move(1);
    angle := angle + Pi/2; # angle now $3\pi/2$
    # Do that again.
    while isTouching(angle, 'bt') do
        move(1);
        end do;
    turn(Pi/2);
    move(1);
    angle := angle + Pi/2; # $2\pi$ is the same as 0.
    # Do that one more time
    while isTouching(angle, 'bt') do
        move(1);
        end do;
    turn(Pi/2);
    move(1);
    end proc:

run(turnFourTimes);
carMovie(stateTable);
```
Example: the four corner problem with a shorter program using a nested loop

In the previous example, our solution involved describing the actions needed to traverse one side of our four sided problem, then repeating that code four times. We write a shorter program by putting all of this inside a loop, whose purpose is to perform everything inside it four times.
turnFourTimesA := proc()
  local bt, angle, count;
  # Move forward while touching a wall, then
  # turn left
  # Initial scan angle is Pi/2
  angle := Pi/2;
  # Initial count is zero.
  count := 0;
  while(count < 4) do
    while isTouching(angle, 'bt') do
      move(1);
      end do;
      turn(Pi/2);
      move(1);
      angle := angle + Pi/2;
      count := count + 1;
    end do;
  end do;
end proc:
run(turnFourTimesA);
carMovie(stateTable);

"Car stopped after 32 moves"
The programming for this solution is dramatically more succinct. If you can handle in stride the greater conceptual complexity of putting one loop inside another loop, you will probably get your solution working faster, since there are fewer characters to type, therefore decreased chances of making an entry error.

Example: a procedure that does a calculation instead of controlling a car.

Much of technical programming has to do with the calculation of mathematical formulae, visualizations, or numbers rather than control of devices. Procedures can be used for such tasks. If you have a calculational script, in many cases it is easy to turn it into a procedure:

1. Rather than initializing script parameters at the start, remove those assignments and put the parameters into the sequence of parameters of the procedure.
2. The last thing calculated should be the result returned. In the example below, we return a list of numbers, which allows us to communicate two results that the procedure has computed. One of the
advantages of using a data structure such as a list is that it can be used as a "data container" for several results.

```maple
#Given two sides of a right triangle, compute #the hypotenuse and the angle formed by it.
triangleInfo := proc (side1, side2)
    local hypotenuse, angle, dAngle;
    #Use whatever values are provided when the #procedure is invoked
    angle := arctan(side1, side2);
    dAngle := convert(angle, units, radians, degrees);
    hypotenuse := sqrt(side1^2 + side2^2);
    #Return a list of the hypotenuse and the angle #(in degrees).
    [hypotenuse, dAngle];
end proc:
```

Once we've defined the function, we can use it for several triangles:

```
triangleInfo(3, 4)

[5, \frac{180 \arctan \left( \frac{3}{4} \right)}{\pi}]

\text{(1.9.1)}
```

```
triangleInfo(3.0, 4.0)

[5.000000000, 36.86989764]

\text{(1.9.2)}
```

If we want one value or the other, we can use part extraction as explained in chapter 5.

```
triangleInfo(6, 6)

[6 \sqrt{2}, 45]

\text{(1.9.3)}
```

```
\rightarrow 45

\text{(1.9.4)}
```

We note that while the procedure will calculate either with exact or floating point numbers, the floating point results are only approximations and may differ slightly from the correct answer.

```
triangleInfo(6.0, 6.0)

[8.485281374, 44.99999998]

\text{(1.9.5)}
```

```
\rightarrow 8.485281374

\text{(1.9.6)}
```
## Troubleshooting small function definitions

Function definitions with proc...end proc will exhibit typical "imbalance" error messages such as errors if you forget to balance the typical things that occur in computer syntax: ( and ), if and end if, [ and ], proc and end proc. If you do it in a code edit region, then in addition to printing out an error message, the cursor will pause over the first place where the computer saw the imbalance.

### Troubleshooting problems with small function definitions

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
</table>
| `findPower := proc (x,tol)`
  `local i;`
  `i := 0;`
  `term := x^i;`
  `while (term>tol) do`
  `  term := x^i;` | This function loops to find last power of x that is greater than tol. |

Warning, premature end of input, use <Shift> + <Enter> to avoid this message.

We note that there's no "end proc" to end the procedure definition. We edit the code region and execute it again.

After making that correction, we get a message that indicates that the proc wasn't expected. The cursor is positioned in the midst of the "end proc" so we know that it's the last proc where things ran into problems.

We know that procedures must end with end proc, so removing it doesn't seem to be an
findPower := proc (x,tol)
local i;
i := 0;
term := x^i;
while (term>tol) do
    term := x^i;
i := i+1;
end proc;
end proc;

Warning, `term` is implicitly declared local to procedure findPower
proc(x, tol)
    local i, term;
i := 0;
term := x^i;
while tol < term do
    term := x^i; i := i + 1
end do
end proc

(1.10.1)
This doesn't have any errors or warnings, but is it right? We need to test the function.

| findPower := proc (x,tol) 
| local i,term; 
| i := 0; 
| term := x^i; 
| while (term>tol) do 
| 
| term := x^i; 
| i := i+1; 
| end do; 
| end proc; |

| proc(x, tol) 
| local i, term; 
| i := 0; 
| term := x^i; 
| while tol < term do 
| 
| term := x^i; i := i + 1 
| end do 
| end proc |

| findPower(.9,.1) |
| 23 |
| .9^23 |
| 0.08862938120 |

The function returns 23. But .9^23 isn't greater than .1, so that can't be the last power greater than .1.

When we analyze what is going on, the value of i computed in the loop is one too far. So we think we should really make the last value of the function one less.
```maple
findPower := proc (x,tol)
local i,term;
i := 0;
term := x^i;
while (term>tol) do
    term := x^i;
i := i+1;
end do;
i -1;
end proc;
```

```maple
proc(x, tol)
    local i, term;
i := 0;
term := x^i;
while tol < term do
    term := x^i; i := i + 1
end do;
i - 1
end proc
```

Evidently our analysis is still off. Looking harder, we see that we should change the order of operations inside the loop, or we will go one too far.

<table>
<thead>
<tr>
<th>Value</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.09847709022</td>
<td>(1.10.5)</td>
</tr>
<tr>
<td>0.1094189891</td>
<td>(1.10.6)</td>
</tr>
<tr>
<td>0.1094189891</td>
<td>(1.10.7)</td>
</tr>
</tbody>
</table>
```maple
findPower := proc (x,tol)
local i,term;
i := 0;
term := x^i;
while (term>tol) do
  i := i+1;
term := x^i;
end do;
i -1;
end proc;
```

This works. But it took some analysis even beyond the point where we stopped getting error messages and warnings.

Troubleshooting problems with use of variables in procedures

This is a function that's supposed to take \( n \) square roots of a number:

\[
\sqrt{\sqrt{\sqrt{\ldots\sqrt{a}}}}
\]

There are several problems with it. The first
g := proc(a,n);
local i, count;
count := 0;
while count<n do
  a := sqrt(a); #take multiple square roots
  count := count +1;
end do;
return a;
end;

Error, unexpected `local` declaration in procedure body

With this corrected, we get Maple to accept the definition, but it still doesn't work. The problem is that formal parameters can't be assigned to in Maple procedures. Formal parameters do not work the same as local variables.

The way to get the desired result is to set up a local variable and to initialize it with the value of `a` before the loop starts.

The function works both for exact numbers and floating point numbers.
Troubleshooting local variable declarations

<table>
<thead>
<tr>
<th>Warning messages from undeclared local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>If you forget to declare a variable within a function definition as a local variable, then Maple will do it for you but print out a warning. While this default action is almost always what is wanted, the warning is get you to think about whether you really</td>
</tr>
</tbody>
</table>
eApprox2 := proc(x, tol)
    local term, i;
    s := 1;
    term := 1;
    i := 1;
    while abs(term) >= tol do
        term := term*x/i;
        s := s + term;
        i := i + 1;
    end do;
    return s;
end proc:

Warning, `s` is implicitly declared local to procedure `eApprox2`

We see that s's value within this script is not affected to what happened to s within eApprox2. s was automatically declared local to the eApprox2 procedure when it was defined, even if we didn't include it in the code.

By putting a colon at the end of the "end do", we suppress the execution trace of what's happening inside the loop. The print actions do cause output though.
We can redefine eApprox2. This doesn't change the meaning but it does get rid of the warning messages.

Note that the computed result is based on x (the script version, not the local variable used inside eApprox2)'s last value from the previous computation in the worksheet. Since a the last computed value of x from the while loop is 1.1, this is the value used in the computation here.
print("Computed:", EApprox2(x, 10^(-5)), "Built-in:", exp(x));
"Computed:", 3.004165230, "Built-in:", 3.004166024
x; 1.1

Troubleshooting while loops: loops that don't stop

We've already reviewed some of the basic problems with loops that come from forgetting to put in a end do or do, or putting too many in. These typically give some kind of Error, message and were discussed here.

More pernicious are loops which do not give error messages but either stop with the wrong answer, or do not stop. Such programming is called an infinite loop and they are usually something to avoid because the program will never come to a conclusion and let you regain interactive control of the computer.

You can tell that this is happening because you are either flooded with a lot of output, or the status window in the lower left hand corner of the worksheet says Evaluating.... but never seems to return to Ready.

A runaway "infinite" loop

x := 3.0;
while (x<10) do
    print("sin(x)", sin(x));
end do;

3.0
"sin(x)", 0.1411200081
"sin(x)", 0.1411200081
"sin(x)", 0.1411200081
The problem with this loop is that it will never stop. The continuation condition is \( x < 10 \), which is true initially. Because the loop body never changes the value of \( x \), it will continue to be true for every successive repetition. Presumably the programmer made a mistake and forgot to put in lines of code into the loop body that changed the value of \( x \) and (with enough repetition) made it eventually grow to 10 or larger.

In theory, you can stop long-running computations such as infinite loops with the "red stop hand" found on the Maple tool bar. Clicking on the red stop hand will eventually stop the output, but not until after a long, long wait while all the output that resulted before you told the computer to stop looping, is printed. It typically takes a long time for it to print all the output from the repetitions, and you won't be able shut off the printing with the red hand.

So in practice, if you have a runaway loop you often will have to force-quit your Maple session and recover the most recent version of your worksheet through File -> Recent Backups -> Recover Backup, as described in chapter 2. If you absolutely must execute a runaway loop again, first change it to print very little so that it will stop before you get old waiting for it.

<table>
<thead>
<tr>
<th>Interrupting an evaluation with the &quot;red stop hand&quot;</th>
<th>After entering an equation to solve, we have realized that we really don't want to see millions of roots listed, assuming that Maple can find them all. So we click on the red hand (circled in green).</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Interrupting an evaluation with the &quot;red stop hand&quot;" /></td>
<td>After a significant pause, the message &quot;computation interrupted&quot; appears and we can enter another computation for Maple to perform. There is no result from the interrupted computation, so no result is displayed. The hand becomes</td>
</tr>
</tbody>
</table>

After entering an equation to solve, we have realized that we really don't want to see millions of roots listed, assuming that Maple can find them all. So we click on the red hand (circled in green).
gray instead of red when execution is over.

This technique does not work well if the long running computation prints a lot, as might occur in a runaway loop with a verbose execution trace.

The CarSimulator package won't have the problem of having the car take infinite numbers of moves by accident, because it has a built-in limit to the number of steps that the car can take. However, you may discover that you have a "runaway car" who was only prevented from an infinite loop by the built-in limits. It's still possible to write a car control program that doesn't end if you include a loop that doesn't move the car (such as one that only prints, or assigns a variable a value).

### Summary of Chapter 11

#### General form of a Maple procedure definition with local variables

```maple
function name := proc ( formal parameters )
local variables used within procedure;

statements;

end proc;
```

**Notes**

1. There is no semi-colon between the proc ( ) containing the formal parameters and the word local. If there are no local variables for the procedure, then you need to put in the semi-colon: `proc ( formal parameters ); statements;` .

2. The opposite of a "local variable" is a "global variable". We don't use them much in the Maple procedures we will be working with. You can read more about them if you look up "global" in Maple on-line help.

3. You can't assign a value to a formal parameter in the statements of the procedure. Parameters are used only as a way of providing input information to the function.
### The general form of the while statement

```markdown
while condition do
    loop body;
end do;
```

The *condition* is something that is either *true* or *false*. An error results if it isn't. The *condition* is evaluated only once per repetition, at the start before any of the code in the *loop body* is executed. If the *condition* is true, then the entire *loop body* is executed. If not, the repetition is finished and execution moves onto the next instruction after the *end do*.

The *condition* is evaluated at the start of the loop iteration. It does not matter if the *condition* changes in the middle of the *loop body*; this won't terminate execution of the loop until the execution reaches the end of the *loop body* and the *condition* looked at again before the start of the next repetition of the loop.

The continuation *condition* is can be a function invocation that returns *true* or *false*, such as the `isTouching` or `onTopOfSign` functions of the CarSimulator package. It can also be an inequality such as \( x < 0.1 \) or \( \text{abs}(y-z) > \text{tol} \). It always must be an expression that can be evaluated to be *true* or *false*. If the inequality is numeric such as \( x < 0.1 \), variables in the *condition* such as \( x \) must have been assigned a numeric value earlier in the Maple session.

It is also possible to combine several conditions together using the operations *and*, *or*, and *not* etc.

*true* and *false* are referred to as Boolean values (after the British 19th century, George Boole -- [http://www.gap-system.org/~history/Biographies/Boole.html](http://www.gap-system.org/~history/Biographies/Boole.html) for more information). Functions that return Boolean values are referred to as *Boolean functions*, while *and*, *or* and *not* are referred to as *Boolean operations*.

---

### Stopping a never-ending (infinite) loop

Stop the "red hand" icon on the Maple toolbar. If that doesn't work, then shut down the Maple session by killing the process (in Windows) or doing a Force Quit (in Macintosh). Unfortunately the latter will cause you to lose all the work you've done since last saving the worksheet, so it should be done only as a last resort.
Index

Symbols
! (exclamation point
  factorial), 11
!!! (execute worksheet), 62
" (double-quote), 71
' (apostrophe), 78
(, 10
), 10
- (minus or dash), 10
:=, 42
=(equation), 20
^ (caret), 10
_ (underscore), 41
` (backquote), 78
π
  symbolic constant, 25, 29, 30
, 101

A
abs (absolute value), 96
actual parameter, 71
acute accent marks (’), 78
addition (+), 9
altitude, 156
animate (plots package), 141, 160
animation controls, 142
animations
  exporting to GIF format, 162
apostrophe, 120
apostrophes, 78
arccos (inverse of cosine), 25
arcsin (inverse of sine), 25, 96
arctan (inverse of tangent), 25
arguments
  of function, 71, 95
arithmetic, 8
  approximate, 26
  exact, 12
  operations, 9
arrow, 101
arrow key
  right, 11
assignment, 21
  mental model of, 43
  of a plot structure to a variable, 133
undoing, 44
assistant
  Unit Converter, 123
asterisk, 9
autosaving, 19
average
  of numbers in a list, 85
Axiom, 4

B
backquotes, 78
backspace key, 15
backups
  retrieving, 19
ball, 155
base of natural logarithm (e), 29
blue, 136
brackets, 58

C
C (programming language), 4, 6, 50
C++ (programming language), 5
Canadian, 84
Canadian roots, 137
Canvas (Maple word processor feature), 38
caret (power), 10
circle, 137, 140, 157
  function, 137
circle-X, 7
clickable interface, 74
Colorado, 59
colornames (plot), 150
colors
  plotting, 83
coma, 58
command- =, 37
command-C (copy operation), 15, 61
command-R, 37
  switch to math mode, 36
command-T
  switch to text entry mode, 36
command-V (paste operation), 15, 61
command-X (cut operation), 15
completion
  command, 98
composition
  of functions, 113, 116, 117
computing
  algebraic, 1
  graphical, 1
  high performance, 2
  logical, 1
  numerical, 1
constant
  symbolic, 52
control- =, 37
control-C (copy operation), 15, 61
control-click, 20
control-R, 37
    switch to math mode, 36
control-T, 37
    switch to text entry mode, 36
control-V (paste operation), 15, 61
control-X (cut operation), 15
convert, 156
    a list into a sequence, 85
    a list into a string, 85
    between units, 109, 123
copy (editing operation), 15
copy and paste, 61
    Maple, 19
crash
curly braces, 58
currency, 119
current state of a Maple session, 43
cut (editing operation), 15

D
daisy chaining, 113
    functions, 117
dashed rectangle, 8
days, 120
degrees, 109
delimiters in Maple: ( ) [ ] { }, 13
digits, 26
disk (plottools function), 150
display, 135, 160
    function, 134, 137
divide and conquer, 63
division (/), 9
documentation
    need for, 1
dot
    centered, 11
double-quotes, 78

E
e
    from keyboard not a symbolic constant, 29
    natural logarithm and symbolic constant, 25, 29, 30, 38
Edit->Execute->Selection, 62
Edit->Execute->Worksheet., 62
ellipse, 140
ENGR101, 63
ter, 36, 45, 62
equation, 20, 57
error
    attempting to assign to Pi which is protected, 52
    attempting to assign to 'solve' which is protected , 52
    controlling variable of for loop must be a name, 52
    illegal use of an object as a name, 51
    invalid fraction, 14
    invalid matrix/vector product, 13
    invalid operator parameter name, 105
    invalid power, 72
    invalid sequence, 14
    invalid sum/difference, 12
    logic, 18
    missing operation, 12
    numeric operation: division by zero, 13
    semantic, 17
    unable to match delimiters, 13, 72
    unexpected option, 77
    vocabulary, 17
error message, 12
escape key, 38
escape key (ESC), 98
Euro, 116
eval, 108
evalf, 109
evaluating
    preventing, 120
    state, 19
evaluation, 42
    at a point, 62
    of a formula, 139
    of a function, 72
    of expressions, 9
exp (exponential function), 97
exponentiation, 16
Exporting animations, 162
expressions
    algebraic, 20
    from palette, 25

F
factorial (!), 11
fence post error, 28
find something similar, 63
Format -> Create Document Block, 38
Formatting
    output
        as currency, 119
Fortran, 4
freeze frame
    animation, 142
fruit flies, 100
fsolve, 107
ft, 119
function
    idea of, 95
function definition, 104
function definition alternative, 106
function evaluation, 72
functions
define, 100
designing from word problems, 115
thousands of built-in, 98
fuzzy notions, 63

G
gallons, 109, 120
   Imperial, 84
gigaflop, 2
 graphical user interface (GUI), 4
 Graphics Exchange Format (GIF), 162
green, 136

H
help
   on-line, 81, 92, 99
   quick, 7
   through command completion, 98
   using on-line, 100
Help (Maple menu), 38
horsepower, 120
hour, 109, 120
hypotenuse, 127

I
I (square root of -1), 52, 59
Ideal Gas Law, 113
Image (Maple word processor feature), 38
inch, 109, 119
index, 57, 137
inequalities, 57
infin (shortcut for infinity in math mode), 38
Insert (Maple menu item), 38
Insert->2-D Math, 36
Insert->Text, 36
interative, 5
inverse trig functions, 25

J
Java, 4, 5, 50

K
Kessel Run, 121
kilometer, 119, 121
kilopascals, 114
kilowatt, 120

L
label, 9
labels for axes

M
plotting, 83
LeastSquares, 106
left arrow key, 15
left hand side, 57
left hand side (right-click menu), 27, 31
lhs, 108
line
   function, 136
linear algebra, 13
list, 58, 75
   add together items from, 85
   count number of items in, 85
   one item from, 85
   operations on, 84
   part of, 85
liters, 109
log10 (base 10 logarithm), 96
logarithm
   base e, 25
logic error, 18, 28
Los Alamos National Laboratory, 2

N
Macsyma, 4
Maple, 4
   as a word processor, 35
document mode, 7
math entry mode, 35
Mathematica, 4, 6
mathematical model, 1
mathematical models, 138
Matlab, 4, 50
   Symbolic Toolkit, 6
max (maximum of arguments), 96, 107
maxdigits, 16
meter, 109, 119
micrometer, 120
mile, 119
millimeter, 119
min (minimum of arguments), 96, 107
minus, 10
minute, 120
mistake
   typographical
      how to fix, 15
multiple systems, 5
multiplication (*), 9
MuPad, 4, 50

N
name
   assignment to, 41
National Forest Service, 61
negation (-), 10

O
Oak Ridge National Laboratory, 2
Octave, 4
omega (Greek letter), 38
Omega (upper case Greek letter), 38
on-line documentation, 83
options
  plot, 108
orange, 136
oscilloscope, 63

P
palette, 25
  Expression, 74, 95, 100, 102, 107
  square root, 25
palettes, 38
parameter, 62
parameterized plots, 139
parentheses, 10, 71
parsec, 121
paste (editing operation), 15
petaflop, 2
Pi, 52
Pi (symbolic constant), 52
pick names, 62
pitfalls
  common
    in Maple, 95
playing
  animation, 142
plot, 108
  a formula, 21
  builder, 80
colors, 78
  for insight into a mathematical relationship, 139
labels, 78
options, 83
structure, 133
symbol for point plot, 157
symbol size for point plotting, 157
plot options, 108
PlotBuilder, 23
plots
  package, 135
plottools, 136
plus, 9
pop-up, 105
power (\(^\)), 10
pressure, 125
preventing
  evaluation, 120
Python, 6
Q
Quick Reference (help menu item), 38
quotation, 120
R
radians, 109
ready (status of Maple), 19
rectangle
  dashed (entry), 8
red, 136
refactor
  code, 118
replay
  animation, 142
restart, 51, 52, 63
reuse
  of software, 1
rhs, 108
right arrow key, 11
right hand side, 57
right hand side (of equation), 27
right hand side (right-click menu), 27, 31
right-click, 20
S
SAGE, 4
scaling=constrained (option to plot), 140
scripting, 61
search field
  for help, 81
select
  entry, 57
  item from a set, 27
select entry (right-click menu), 27, 31
semantic error, 17
sequence, 14, 58, 127
set, 58
  empty, 69
sheep, 59
shortcuts
  keyboard, 38
Show Al Palettes, 38
simplify -> assuming real (clickable operation), 98
simulation, 1
single-quotes, 78
slash, 9
small steps, 63
Solo
  Han, 121
solve, 57, 72, 92, 107
  a system of equations, 85
by right-clicking or control-clicking, 20
numerical, 26
won't work with an assigned variable, 52
solving an equation, 139
speed
animation, 142
sqrt (square root), 96
square root
from palette, 25
Star Wars, 121
state
current of Maple session, 43
state display
of Maple, 19
state of a Maple session, 45
stop playing
animation, 142
sublist, 85
subscript, 37
subtraction (-), 10
supercomputer, 2
symbol
for point plotting, 157
symbolsize, 157

T
Table (Maple word processor feature), 38
temperature, 125
template
textual, 98
teraflop, 2
test, 63
testing, 115
a program, 105, 106
functions, 115
text entry mode, 36
text mode to math mode
can't switch back, 38
textual entry, 71
Tools->Spellcheck, 39
triangle, 127
troubleshooting, 63
assignment, 53
function definitions, 105
textual input, 92
unit conversion, 121
word processing, 38

U
unassign, 43, 44, 51
underscore, 37
underscore (_), 41
Unit Converter assistant, 123

units, 156
Units Calculator, 84

V
value
of an unassigned name, 42
variable
programming, 41
vertical range, 74
View (Maple menu), 38
visualization, 1

W
what you see is what you get, 35
WYSIWYG, 35

Y
yard, 120