Chapter 21 -- More on programming

Section 21.1 Striking out on your own in software development

Learning how to program can be like learning how to cook. It can be intimidating to novices to see expert chefs at work. Experts seem to know many techniques by heart and can put them together creatively to come up with new dishes. They can also determine how to reproduce an existing dish just from looking at it and tasting it.

The path taken by new cooks is to try to keep the complexity of the work manageable but providing challenging experiences that cover new ground. Being able to write a correctly operating program isn't a matter of just entering it perfectly and seeing that it works, it always involves making multiple mistakes and correcting them until the result works. If you introduce a complex situation too soon, you will not be able to figure out the mistake easily because there will be too many sources. In other words, "keep it simple all the time" when you're developing a program.

Keeping it simple can be boiled down to a few adages:

a) Always start out from a known state.

b) Grow solutions from something very simple that works, into progressively more complicated forms. Only add a few new things at a time.

c) Test for correctness at each stage. Provide yourself with lots of information about what the program is doing while it's running so that you can tell the first point where things go wrong.

Section 21.1.1 restart and scripting

In an extended Maple session, one typically performs many assignments to the same variables, as you re-run a script many times in developing it. Maple's worksheet interface allows you to jump around within it, executing lines of code in a different order than they are listed in the worksheet. This can be very convenient -- if you need to fix something, you just go back to the "bad line", edit it, and then re-execute that line. You can then jump forward and execute something much further down the worksheet to reap the benefits of the new assignment.

However, this feature is a two-edged sword. Jumping around a worksheet can cause you to forget exactly what you've changed after awhile. This can cause mystifying "errors" -- caused by a variable you are using to have a value that you've forgotten you've given it.

The "cure" for mystifying assignment values is to do a restart and start the scripting from the beginning.

As we have mentioned in section 8.2.3, restart is a command that clears all assignments from the Maple worksheet you are using. This is guaranteed to eliminate any accidental misassignments that might be affecting the run of your script. Typically, the way a developer wanting to start from a fresh state would work would be to develop a script that began with restart; and then did all the assignments and other operations needed in an entirely self-
contained way. In that way, re-executing that code region would guarantee that the results computed would be derived from a fresh state.

In rare situations, Maple's engine may get locked up by some actions you undertake. It may be necessary to kill the Maple application and window and start up the script again in a fresh version of Maple. Having all the actions you are taking in script saved in a file will allow you to carry on even if this happens.

### Example 21.2.1 restart at the start of a test run.

```maple
restart;
libname := ".", libname;
with(CarSimulator);
initialize();
setBackground([5,2], TARGET):
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

(More programming follows)
```

This script begins with a "restart". It then runs the script to do a Car Simulation program. By rerunning the code window that starts with the restart, the programmer knows that things begin with a "clean slate". This script can be saved in a worksheet file and re-executed later on. Since it always begins in a known state, it should always produce the same results. The downside of using restart in this way is that every step taken to initialize Maple so that the execution can proceed must be re-done since any previous initialization was wiped out by the restart. That's why it's useful to have all the initialization steps included in the script so that re-initialization is not laborious.

#### Section 21.1.2 Incremental development

In section 21.1.1, we explained the reasoning for tackling an extended software development project as a script saved in a file-- it's easier to reset things, and to overcome even bad
situations where you need to exit Maple to escape the damage. We can extend this idea to also cover the time that you are working on the development.

Enter the minimal amount of programming that will get you to "first base" as a restartable script. Execute the code region. Decide whether the results calculated so far are correct. If not, then go back and fix things. Rerun the script and try again.

Once you have success at first base, determine the second step you should take. Add this onto the script and get everything up to second base to work.

Continue until you reach the final goal. There may be four phases to the work if there are a lot of actions, but the idea is to proceed in small steps that are manageable for you to handle.

\section*{Section 21.1.3 Checkpoints for incremental development}

There are a number of ways to have the computer help you check that things are working at each stage. Experienced programmers, rather than hoping for the very unlikely outcome that everything works correctly, build into their program from the start extra operations that try to sound the alert if detectable things are going wrong. In other words, the notion is to assume that things will go wrong at some point and have extensive diagnostic information being issued while the program is being executed. This can be done through \textit{print} messages.

Such targeted messaging can augment searching through execution traces (as explained in Sections 11.3.3, 11.3.4, 13.4) since as the developer you can specify exactly what the messages do and what checking is done.

An additional feature of Maple that can produce messaging during execution that is designed specifically to help find problems is by using the \texttt{ASSERT} function. A kind of streamlined \texttt{if} statement, it will check a condition and stop execution of the program with a message if the condition does not hold. It can be good practice to place a number of \texttt{assert} statements into a program to see if everything that you think is happening can be verified. See Maple on-line help for an example of \texttt{ASSERT}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Example 21.1.3.1 Example of incremental development using ASSERT and print statements.} & \\
\hline
\texttt{nextprime}(10^9 + 1) & Maple can tell us the next prime after a given value. Here was ask what the next prime is after a billion. \\
& \texttt{1000000007} \\
& \textbf{(1.1.3.1)} \\
\hline
\texttt{isprime}(1000000005) & We've seen "isprime" before. It returns true or false depending on whether the number is prime. It works fairly quickly for small numbers. \\
& \texttt{false} \\
& \textbf{(1.1.3.2)} \\
\hline
\end{tabular}
\end{table}
isPrimeIn := proc(r)
    local low, high;
    low := op(2, r);
    high := op(1, r);
    ASSERT(low < high,
        "low should be less than high", low, high);
    return low, high;
end:

isPrimeIn(5..10)
Error, (in isPrimeIn)
assertion failed, low should be less than high, 10, 5

isPrimeIn := proc(r)
    local low, high;
    low := op(1, r);
    high := op(2, r);
    ASSERT(low < high,
        "low should be less than high", low, high);
    return low, high;
end:

isPrimeIn(5..10)
5, 10

primes are in that range. If so, it will return true otherwise it will return false.

Examples we are thinking of are:

isPrimeIn(5..10) is true
isPrimeIn(11..13) is true
isPrimeIn(5..6) is true
isPrimeIn(25..27) is false

It's been awhile since we've used ranges so we are not sure that we know how to extract the end points of the range. We develop incrementally and just try to write a procedure to do that

According to the online documentation on ASSERT, this is the way to enable assertions.

We think that this should make the low the lesser end point and high the greater one. But we put in an assertion that this should be so and then test the program.

We get an error message indicating that we goofed. We can see that low is 10 and high is 5. We must have reversed things so we should do low := op(1, r), and high := op(2, r);

We enter the revised procedure and try the test again.

This test passes. We try to move onto the next step of development, to return true or false rather than just the end points.
Example 21.1.3.2 Example of incremental development using ASSERT and print statements, part 2

isPrimeIn := proc(r)
  local low, high;
  low := op(1, r);
  high := op(2, r);
  ASSERT(low < high,
    "low should be less than high", low, high);
  np := nextprime(low);
end:

In the next phase we try to compute the next prime. But we have to go back
and insert a missing local declaration.

isPrimeIn := proc(r)
  local low, high, np;
  low := op(1, r);
  high := op(2, r);
  ASSERT(low < high,
    "low should be less than high", low, high);
  np := nextprime(low);
  print("np is:", np);
  return low, np, high;
end:

We put in a print message to tell us what np is.

isPrimeIn(5 ..10)
  "np is:", 7
5, 7, 10  (1.1.3.5)

This seems to work.

isPrimeIn := proc(r)
  local low, high, np;
  low := op(1, r);
  high := op(2, r);
  ASSERT(low < high,
    "low should be less than high", low, high);
  np := nextprime(low);
  ASSERT(np > low,
    "low should be less than next prime", low, np, high);
  if np < high then return true else return false end if;
end:

Now we add the conditional statement that will return true or false.

isPrimeIn(5 ..10)
  true  (1.1.3.6)

This seems to work. But passing one test does not mean that things work all the time.

This is wrong.
isPrimeIn(11..13) false

Example 21.1.3.2  Example of incremental development using ASSERT and print statements, part 3

isPrimeIn := proc(r)
local low, high, np;
low := op(1, r);
high := op(2, r);
ASSERT(low < high,
"low should be less than high", low, high);
np := nextprime(low);
ASSERT(np > low,
"low should be less than next prime",
low, np, high);
print("np is:", np);
if np <= high then return true else
return false end if;
end:

isPrimeIn(11..13)

"np is:", 13 false

We put in an extra assertion to check that np is in range.

We run the test again but neither assertion is triggered. So evidently low<high and np<low. Looking at the output, we see that the next prime is = high. But we see that the if statement we have put in only returns true if np is less than high.

We change the if statement is that it returns true if np is less than or equal to high.

Now the test works.

This one works, too.

We change the if statement is that it returns true if np is less than or equal to high.

This one works, too.

We put in an extra assertion to check that np is in range.
true

isPrimeIn(5 ..10)

"np is:", 7

true

isPrimeIn(5 ..6)

"np is:", 7

false

This result is wrong. There definitely is a prime (namely 5) between 5 and 6 but the procedure returns false. We see that "np" as computed is 7 and 7 is not less than 6. So the procedure is doing what we told it but the answer is wrong. We must have told it to do the wrong thing. Time to think over the situation some more.

Example 21.1.3.2 Example of incremental development using ASSERT and print statements, part 4

isPrimeIn := proc(r)
local low, high, np;
low := op(1, r);
high := op(2, r);
ASSERT(low < high,
"low should be less than high", low, high);
np := nextprime(low);
ASSERT(np > low,
"low should be less than next prime", low, np, high);
print("np is:", np);
if np ≤ high or isprime(low) then
    return true
else
    return false
end if;
end:

isPrimeIn(5 ..6)

"np is:", 7

true

isPrimeIn(5 ..10)

"np is:", 7

true

isPrimeIn(11 ..13)

"np is:", 13

true

isPrimeIn(25 ..27)

"np is:", 29

false

isPrimeIn := proc(r)

Thinking reveals that we want to return true if the next prime after low is in range, or if low itself is a prime. We alter the procedure to say this and try our tests again.

Okay.

Still okay. We didn't break anything by putting in a fix.

Also still okay.

This works too.

Now we ready the procedure for "production" use. We comment out the print statement.
local low, high, np;
low := op(1, r);
high := op(2, r);
ASSERT(low < high,
"low should be less than high", low, high);
np := nextprime(low);
ASSERT(np > low,
"low should be less than next prime", low, np, high);
# print("np is:", np);
if np ≤ high or isprime(low)
then
  return true
else
  return false
end if;
end:

We turn off the assertions. They are still in the procedure but deactivated. This is a bit faster than commenting out every print statement.

# kernelopts(assertlevel = 0);

   2 (1.1.3.16)
isPrimeIn(5..6)  true (1.1.3.17)
isPrimeIn(5..10)  true (1.1.3.18)
isPrimeIn(11..13)  true (1.1.3.19)
isPrimeIn(25..27)  false (1.1.3.20)
isPrimeIn(100..105)  true (1.1.3.21)
isPrimeIn(200..205)  false (1.1.3.22)

More tests on bigger intervals.

We can now use the procedure to do a little exploring. This is a loop that tries to find the first interval where there is no prime in five consecutive locations.

Evidently its 24, 25, ....28.

Evidently the first time where there are 47 consecutive non primes is 19610..19656.

for low from 1 while isPrimeIn(low .. low + 4) do end do;

low 24 (1.1.3.23)

for low from 1 while isPrimeIn(low .. low + 46) do end do;
Section 21.2 Quoted arguments to functions -- getting information out of a function through another channel

Most Maple library procedures return information by the function result. Sometimes this is a single number or formula, sometimes the value returned is an aggregate such as a list, set or table. There are certain functions that return Boolean (true or false) results such as the isprime or isPrimeln examples of Section 21.1.3. The results are intended to be used in conditional statements (ifs or whiles). Some of these functions begin with the letters is to indicate that their results are of the true/false variety.

There are some circumstances where it would be useful to return other information besides the true/false result. The problem with returning a list of values (one of which might be true/false, the others being the other information desired) is that then more work would have to be done in a script to disassemble and use the information. It can be awkward to program the disassembly in the midst of an if or while condition. It disrupts the natural use of the function in if s and whiles.

Maple solves this situation of "awkward expressiveness" by providing a second channel by the use of quoted arguments. The arguments specify names of variables that get assigned values in addition to the calculation and return of the function's result. This allows use of the functions in if or while even with the additional information being returned through the assignment to quoted arguments.

Example 21.2.1 Functions with additional return channels through the use of quoted arguments

listOfPlayers := convert(ImportData( ), list)
["Chad Durbin", "Matt Stairs",
 "Raul Ibanez", "Jayson Werth",
 "Pedro Feliz", "Chase Utley",
 "Ryan Howard",
 "Shane Victorino", "Lou Marson",
 "Brett Myers", "Carlos Ruiz",]
"Chris Coste", "Jimmy Rollins", 
"Greg Dobbs", "Eric Bruntlett", 
"Joe Blanton", "Miguel Cairo", 
"Chan Ho Park", "Jamie Moyer", 
"Cole Hamels", "J.A. Happ", 
"Jack Taschner"]

We read in a list of corresponding batting averages. These values are floating point numbers rather than strings.

The built-in member function returns true or false, depending on whether an occurrence of the first argument appears in the second argument which is typically a list or set.

The "extra channel" form of member provides a third argument which has single quoted (apostrophe '). This does not refer to a string but as explained in Section 18.2 a symbol whose evaluation is blocked by the quotes. The function result is still true or false, but if it's true, then ithPlace will be assigned the first position in the list where the name occurs. In this case, we see that "Jamie Moyer" is in the list, and the first occurrence of that string is in position 19 of listOfnames.

We write a utility function to serve up information from the list. The procedure wants a name and the two lists of information. It uses an if to print out the batting average (if the name is in the first list), or a sensible message to indicate that the name lookup did not work. Note that it's natural to want member to continue to return just true or false so that the if statement will work. If member returned a list of true/false plus the position, then we'd have to do more work.
Mickey Mantle does not appear in the list of names.

While it is relatively straightforward to use these multi-channel functions, it is a bit harder to properly set them up in your own programming. For the purposes of this course, we will settle on your being able to use built-in functions that use this multi-channel feature, such as the `isTouching` function of the car simulation package of Lab 4.

### Section 21.Z Chapter Summary

**Example 21.1.1 Functions with additional return channels through the use of quoted arguments**

```lisp
listOfPlayers := convert(ImportData( ), list)
["Chad Durbin", "Matt Stairs",  "Raul Ibanez", "Jayson Werth",
 "Pedro Feliz", "Chase Utley",  "Ryan Howard",
 "Shane Victorino", "Lou Marson",  "Brett Myers", "Carlos Ruiz",
 "Joe Blanton", "Miguel Cairo",  "Chan Ho Park", "Jamie Moyer",
 "Cole Hamels", "J.A. Happ",  "Jack Taschner"]

listOfAverages := convert(ImportData( ), list)
[0.5, 0.389, 0.328, 0.299, 0.291, 0.286, 0.27, 0.259, 0.235, 0.231, 0.212, 0.204, 0.2, 0.138, 0.136, 0.125, 0.125, 0.125, 0.111, 0., 0.]

member("Chad Durbin", listOfPlayers)  \textbf{(1.3.3)}  \textbf{true}

member("Mickey Mantle", listOfPlayers) \textbf{(1.3.4)} \textbf{false}
```

We read in a list of names of baseball players given to us from a file (using the information in http://csnphilly.stats.com/mlb/teamreports.asp?yr=2009&tm=22&btnGo=Go&report=stats, May 14, 2009). Note that the players' names are strings, not symbols. We can tell that because each one is surrounded by double quotes ".

We read in a list of corresponding batting averages. These values are floating point numbers rather than strings.

The built-in `member` function returns `true` or `false`, depending on whether an occurrence of the first argument appears in the second argument which is typically a list or set.

The "extra channel" form of member provides a third argument which has single quoted (apostrophe '). This does not refer to a string but as explained in Chapter X a symbol whose evaluation is blocked by the quotes. The function result is still true or false, but if it's
true, then \texttt{ithPlace} will be assigned the first position in the list where the name occurs. In this case, we see that "Jamie Moyer" is in the list, and the first occurrence of that string in position 19 of \texttt{listOfNames}.

We write a utility function to serve up information from the list. The procedure wants a name and the two lists of information. It uses an \texttt{if} to print out the batting average (if the name is in the first list), or a sensible message to indicate that the name lookup did not work. Note that it's natural to want \texttt{member} to continue to return just true or false so that the if statement will work. If \texttt{member} returned a list of true/false plus the position, then we'd have to do more work.

```
member("Jamie Moyer", listOfPlayers, 'ithPlace')
    true
(1.3.5)

ithPlace
19
(1.3.6)
listOfAverages[ithPlace]
0.111
(1.3.7)

lister := proc(player, L1, L2)
    local ithPlace;
    if member(player, L1, 'ithPlace')
        then printf("%s's batting average on May 14, 2009 was %f", player, L2[ithPlace]);
        else printf("%s does not appear in the list of names.", player);
    end if;
end:

lister("Jimmy Rollins", listOfPlayers, listOfAverages)
Jimmy Rollins's batting average on May 14, 2009 was 0.200000

lister("Mickey Mantle", listOfPlayers, listOfAverages)
Mickey Mantle does not appear in the list of names.
```