**Chapter 7 Short Summary**

\* *Multi-statements* and *terminators* (:, ;)

-> Group statements by putting them all in the same execution group and terminate each statement with either a *semicolon* or a *colon*.

A colon suppresses output display whereas a semicolon does not. The last semicolon or colon is optional. A colon at the end of a control structure prevents display of the values of all expressions evaluated inside the control statement.

e.g.

**> **



 (1.1.1)

Only the last value displayed can be accessed via the output label (1.1.1).

-> *To start a new line* without executing, press Shift+Enter.

e.g.

*> a;*

*b:*

*c*





(Execute the group by pressing Enter as usual.)

\* *Control Structures*

Control structures automate repeated calculations. A statement sequence inside a control structure is called the body of the control structure.

All input associated with a loop must be in the same execution group. This includes any initialization before the loop and any display of results after the loop.

\**Emergency Stops*

-> Click , to stop Maple.

(Remark: This does not always work so do not rely on it. Avoid infinite loops by including appropriate stop mechanisms in repetition statements. Always save your work before executing any repetition statement.)

-> You may be able to close Maple completely by closing the Maple window, which sometimes only interrupts Maple without actually closing it, but this is certainly not reliable.

-> Use Task Manager to close Maple only if nothing else works. Terminate any process with "maple" or "mserver" in its name.

\* *print()*

*print()* displays the value of its argument sequence, as a sequence, on the current output device.

It is generally necessary to put backward or double quotes around any text to be displayed by *print()*.

Backward quotes are not displayed (they make the text into a symbol). Double quotes are displayed (they make the text into a string).

(Remark: Do not use forward quotes unless you want to prevent evaluation.)

(Remark: *print()* is useful for displaying specific information from inside control structures.)

\* *Comments*

Any text following a # symbol is a comment and is ignored by Maple.

\* *Parallel Assignment*

Make several assignments in parallel by using an expression sequence with the same number of elements on both sides of the assignment operator.

e.g. **> **



e.g. Useful in structured data types:

**> **



**> **



e.g. Useful to exchange values of 2 variables:

**> **



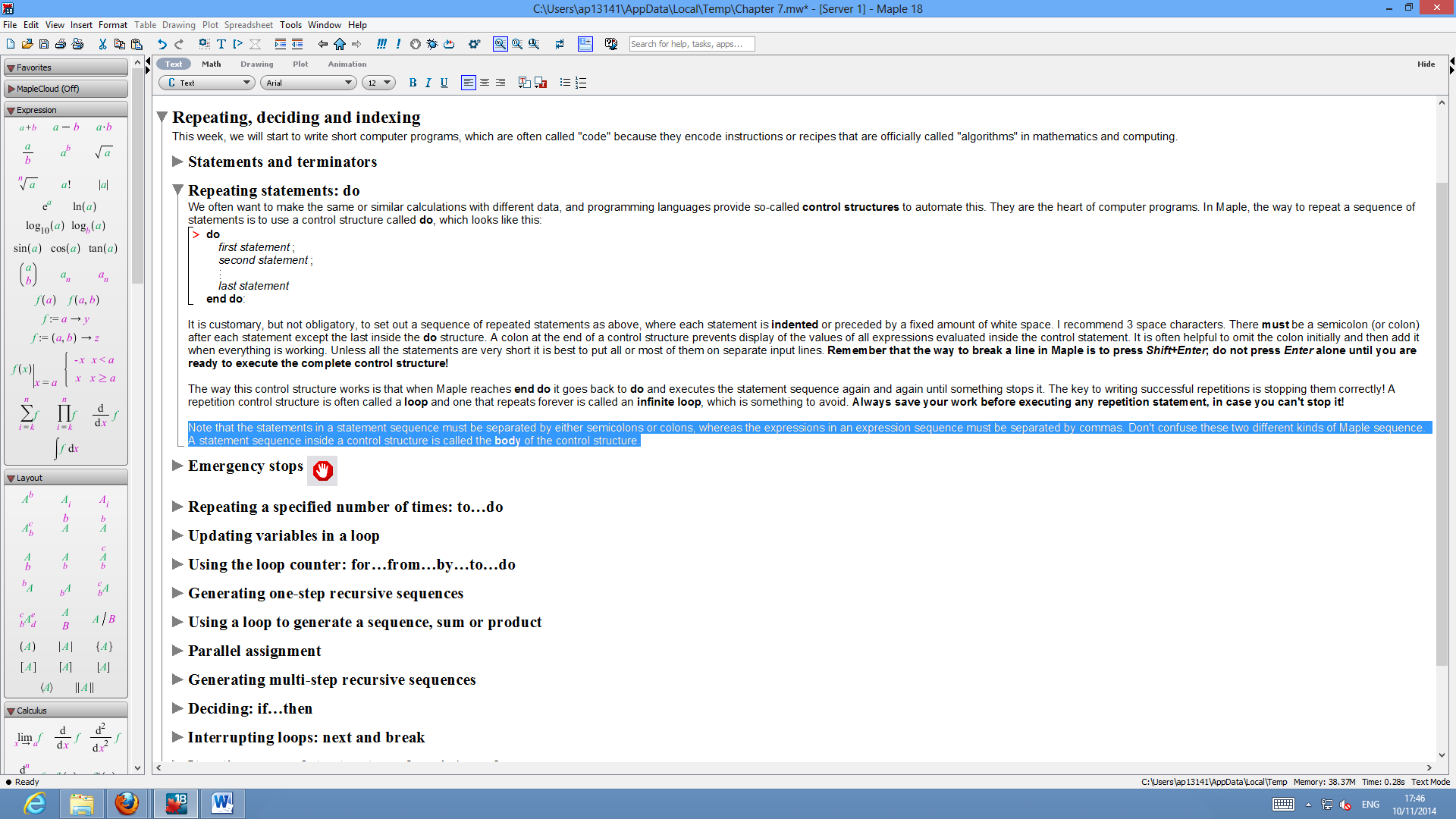
**> **



(Remark: Parallel assignment is not available in conventional programming languages.)

\****do*** *…****end do***

**do**…**end do**is a loop. (When Maple reaches **end do** it goes back to **do** and executes the statement sequence again and again until something stops it.)



There must be a semicolon (or colon) after each statement except the last inside the  structure.

\* ***for***… ***from***… ***by***… ***to***… ***do***… ***end do***

-> ***to*** *...do…end do*

To repeat a sequence of statements *n* times, put “**to** *n*” in front of the “**do”**. This increases the counter from 1 to n, as the loop repeats.

e.g.

**> **











(Remark: A common way to stop a loop is to specify a counter as above. You can provide a high but finite upper limit, to avoid any possibility of an infinite loop. This is called defensive programming.)

-> ***for***… *do*… *end do*

You can use the value of the loop counter, if you give the counter a name by including a **for** clause, like this:

**> **











-> ***from***… ***by***… *do*… *end do*

You can also make the loop counter start from any value  by including  and count in steps of  by including .

e.g.,

**> **











(Remark: You can count down by using a negative step. Remember the  value must be less than the  value if counting down.)

\* ***for***… ***in****… do… end do*/ ***for***…***∈***… *do… end do*

-> The  control structure iterates over each element of a data structure (such as a set or list) in turn.

e.g.

Here we find which of the sets S, in the set P, contain the element 5:

**> **







(The symbolcan be used instead of the keyword )

\* Aside: *Generating sequences, sums* and *products,* with a *loop*

-> To construct and display a sequence using a loop:

Choose a variable, such as , to store the sequence and initialize it to the empty sequence, which can be input as .

Then, in a loop, compute the value of the next element in the sequence, and append it to the sequence stored in  by using the comma operator.

e.g. The following constructs a sequence of five elements and then displays the result.

**> **



e.g. The following computes and stores a sequence of the first ten factorials in the variable *S*.

**> **



(Remark: A set or list is generated by enclosing a sequence in appropriate brackets, i.e.  or .)

-> Similarly, you can construct sums and products using loops:

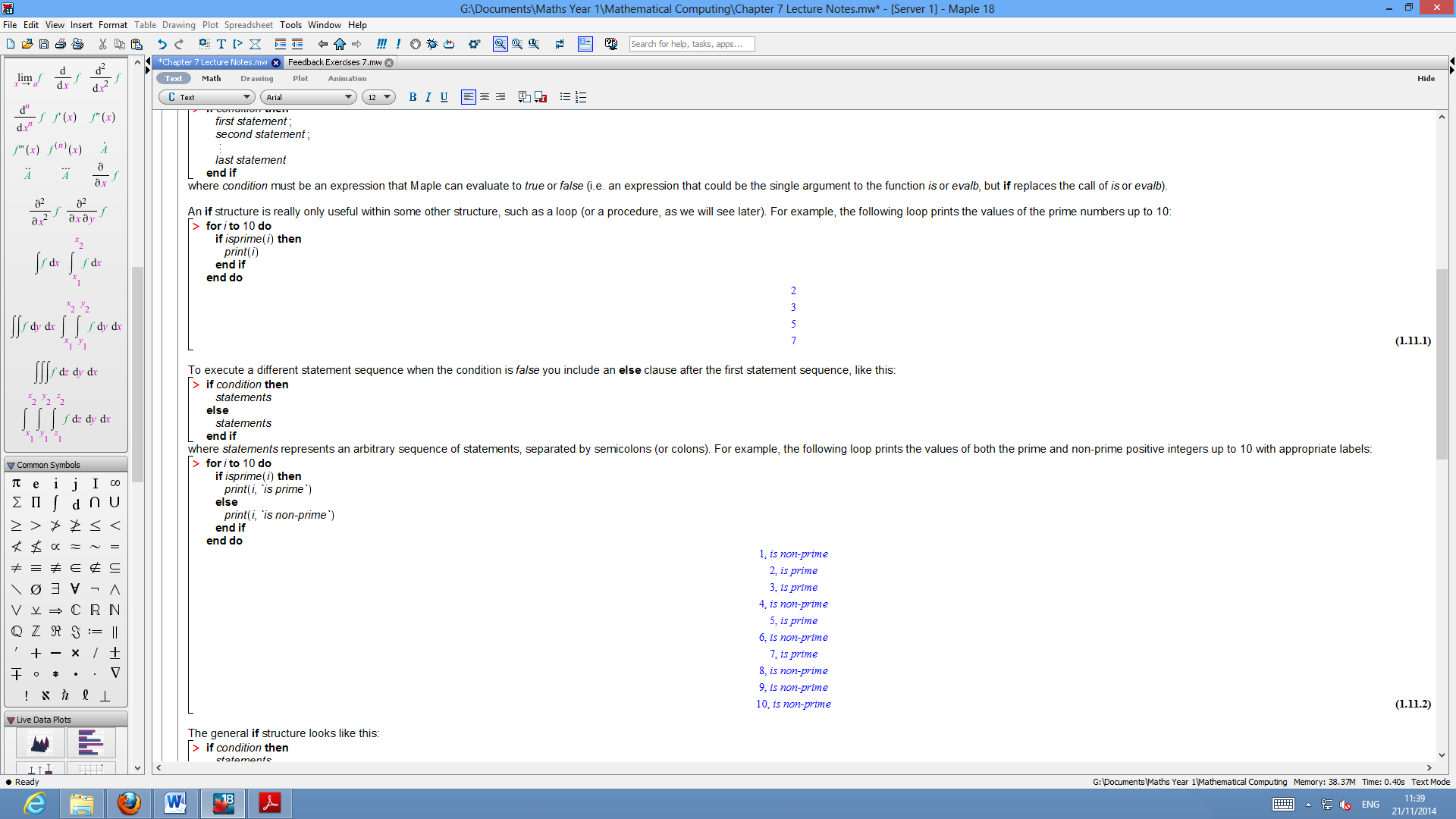
**> **



**> **



\* ***if***… ***then***



where  must be an expression that Maple can evaluate to  or . The *else* is used to do something else if the first statement does not meet the condition (is false).

The general  structure looks like this:

**> **

Only one statement sequence is executed. Maple evaluates the conditions in the order they are written and if one evaluates to  then the  following it (up to the next  or ) are executed; otherwise, any  following  (up to the next ) are executed. Then execution continues as normal after the .

(There can be any number of "" clauses and the final "" clause is optional.)

\* ***next*** and ***break***

-> **next** skips back to the start of the loop that contains it.

e.g.

The Mersenne numbers  that are not prime, where  is any of the first 10 primes:

**> **







(Remark: **next** can be an elegant alternative to putting a lot of statements inside an  statement).

-> **break** terminates the loop that contains it.

e.g.

A Fermat number is a number of the form where  is a non-negative integer, and Fermat incorrectly conjectured that such numbers are prime for all  The following loop displays Fermat numbers until it finds the smallest Fermat number that is not prime:

**> **













\* ***while***

It is possible to allow a loop to run only ***while*** some condition remains true (or equivalently stop a loop when some condition becomes false).

(A  clause can be used with any  loop, with or without any other control clauses.)

e.g. This computes and displays the Fermat numbers until it finds the smallest non-prime Fermat number.

**> **















(Remarks: It is often necessary, as above, to provide some initialization solely to ensure that the  condition is  for the first iteration; otherwise, the loop would never execute at all. The ‘**to** 100’ bit above, is a safety measure to prevent an infinite loop.)

(Remark: Using **while** is sometimes better than using **break**.)

\* Accessing data structures using *indexing*

You can access the elements of any list-like data structure (including arrays, vectors, matrices, etc.) using indexing, by subscripting. (See Chapter 3 for ways to subscript).

The index of the first element of a data structure in Maple is 1 (rather than 0).

The last element can be accessed using the index  and more generally negative indices count backwards from the last element. The last element can also be accessed using nops(), (since the index is the same as the number of elements in the data-structure).

e.g.

**> **



e.g.

**> **



**> **



e.g. A succinct way to reverse the list L:

**> **



If an unassigned name is indexed then the indexed name remains symbolic and is itself a name, e.g.

**> **



(Remark: Indexing in Maple is sophisticated, see help for more details.)

\* Aside: Multi-step *Recursive Sequences*

An -step recursive sequence is a sequence of terms  such that , where the values of  are specified (as initial conditions). To compute the values of the elements of an -step recursive sequence we need to use at least  variables:  variables (e.g. ) to store the values of the last  elements and one variable (e.g. ) to store the next element, (plus a variable to store the whole sequence if required, plus any variables required to compute the next element). The general loop structure looks like:

**> **

e.g. The Fibonacci sequence is a 2-step recursive sequence.