**Chapter 10 Short Summary**

\* *Procedures*

A procedure is a sub-program that packages a program into smaller components (so that it can be re-used and independently developed and tested).

-> Format:

(We have assigned the name *myproc* to the procedure below).

**> **



A semicolon or colon after the last statement in the procedure body is optional, but all other statements must be terminated. A colon after **end proc** suppresses display of the procedure definition.

By default, when a procedure is executed it returns the value of the last expression evaluated in its body. (A procedure is called by using and evaluating it as a function.)

e.g.

**> **



(Remark: Procedures need not accept input or return output.)

e.g.

**>** 



 

**> **



-> A procedure that consists of only a single expression can be written using the  shorthand notation.

 e.g.

**> **



is shorthand for

**> **



(Tip: It is a good idea to begin each procedure definition with a comment that states what the procedure returns, what its parameters represent and any assumptions about their values.)

\* *Local and Global Variables*

Variables that are used only inside a procedure should be declared , like this:

**> **

This stops them getting mixed up with variables with the same name that are used outside the procedure. (Variables that are not declared local are called *global*). Local variables are completely distinct from global variables, even if they have the same names.

(Remark: Function parameters are not variables, so they are neither local nor global. Do not declare parameters to be local.)

(Remark: Maple will try to guess which undeclared variables should be declared local and do it for you, with a warning, but never rely on this.)

(Remark: A terminator before  causes a syntax error, so never put a terminator just after **proc()**, e.g.

**> **

Error, unable to parse

)

(Remark: The control variable used in  (i.e. the variable *n* in *seq*(*n*!, *n*=0 ..5) for example) does not cause a warning if it is not declared local, although it is nevertheless good practice to do so. The same applies to  and .)

\* ***return***

The  statement can only be used in a procedure. It does two things: it terminates execution of the procedure and it returns the value of the expression following the keyword , as the value of the procedure call.

e.g.

The following procedure returns  if no set in a given set of sets, is empty (or null).

**> **

**> **



**> **



\**Recursive* Procedures

A recursive function calls itself or is defined in terms of itself. (It must include a base case that does not involve a recursive function call, and the general case must lead toward the base case.)

e.g. The factorial function , defined by

base case,

general recursive case.

**> **



Note that the ‘**if** $n\leq -1$ **then** *undefined*’ bit in the above procedure avoids an infinite loop if the argument entered is not a natural number. E.g.

**> **



**> **



(Remark: Be careful of infinite loops, and check the base case when writing recursive procedures.)

\* *Tracing* and *Debugging Procedures*

-> Tracing primarily displays the input and returns values at every call of the traced procedure.

e.g. To trace procedures ,  and , evaluate the following function:

**> **

To turn off tracing, evaluate the following function:

**> **

Re-executing procedure definitions also turns off tracing (since the trace function adds some information  not normally included explicitly in procedure definitions).

e.g.

**> **



**> **



**> **



**> **

{--> enter myfactorial, args = 3

{--> enter myfactorial, args = 2

{--> enter myfactorial, args = 1

{--> enter myfactorial, args = 0



<-- exit myfactorial (now in myfactorial) = 1}



<-- exit myfactorial (now in myfactorial) = 1}



<-- exit myfactorial (now in myfactorial) = 2}



<-- exit myfactorial (now at top level) = 6}



(This tracing output shows how recursive function calls are stacked until the base case is reached, and then unstacked to compute the required result.)

-> Maple also provides an interactive debugger (similar to that found in other programming environments) which pops up its own window. Search for help on the Interactive Debugger for details.

\* *Aside*: See lecture notes for a case study of *computing permutations* of a list using *recursive* procedures.