**Chapter 4 Short Summary**

\* *Finite set*

Define finite sets in Maple using braces. e.g. {1,2,3}

(Remark: You can’t use braces for any other purpose. Maple removes duplicates, and may change the order of elements.)

\* *Names* in Maple.

A name can be a letter followed by letters, digits or underscore characters (\_), or nothing. Uppercase and lowercase letters are distinct.

(Remark: Names can be more general and consist of a string of zero or more arbitrary characters enclosed in backward quotes, but this is best not used without good reason.)

\* *Assignment operator*

-> The assignment operator is used to assign a name to some object. Type := or find it in the common symbols palette.

e.g. A:={1,2,3}

(Remark: A new assignment replaces a previous assignment)

(Remark: Assignments are only remembered while a Maple file is open; they must be re-executed (if to be used again) when it is re-opened.)

(Remark: Certain symbols are protected and cannot be used as assignments, e.g. sin:= or π:=. You can circumvent this using advanced Maple techniques).

-> *Remove* an *assignment*

(An assignment may need to be removed if you want to use the name symbolically again for something else.)

Two ways to remove an assignment:

Option 1) **> **

(*unassign()* accepts an arbitrary number of variables).

Option 2) **> **



(Option 2 uses forward quotes.)

\* *restart* Maple server

This has the same effect as closing and opening a Maple file, but faster, (and unassigns all assigned variables).

Two ways to do this:

Option 1) Click on icon in toolbar:



Option 2) Execute the command:

**> **

\* *Predicates* and *is()*

-> A *predicate* is a function that returns a logical value, true or false.

e.g. *isprime(),* ****=*.*

\* Checking for *Set membership*

Two ways:

Option 1) e.g.

**> **



Option 2) *member()* is for specifically for testing membership (of sets, lists, etc.).

e.g. **> **



(Remark: ∈ and ∉ are inert operators, so do not do anything on their own.)

\* *Empty set*

Define the empty set using {} or using ∅ from the common symbols palette.

\* *Subsets* and *Proper Subsets*

-> There are three ways of testing *subsets*:

**> **



**> **



**> **



(The **subset** operator is a predicate so you do not need to use the *is()* function.)

(The subset symbol is found in the common symbols palette.)

-> Maple has no built-in notion of proper subset. A proper subset symbol ⊂, exists in the Relational Round palette, but it has no defined function.

To test for *proper* *subset*, e.g.

**> **



Aside: The way to make ⊂ work as an active proper subset operator is like this:

**> **

e.g. **> **



\* Finding the *Power set*

Use the following function to find a power set in maple:

**> **



\* *Unions* and *Intersections*

-> Unions: e.g.

**> **



**> **



The union can also be used as a function: (good for several sets)

**> **



-> Intersections: e.g.

**> **



**> **



As a function:

**> **



\* *Set Difference* and *Symmetric Difference*

-> For set difference, use the \ symbol in the common symbols palette or keyboard.

e.g.

**> **



-> Symmetric difference: *symmdiff()*

e.g.

**> **



The △ symbol (which can be found in the Miscellaneous palette) has no defined meaning in Maple.

We could make it represent symmetric difference, like this:

**> **

e.g. **> **



\* *nops()* and *set cardinality*

-> Use *nops()* to count the number of operands in an expression.

-> When applied to a finite set, *nops()* returns the cardinality of the set.

e.g.

**> **



(Remark: |A| is the notation for absolute value or modulus in Maple.)

\* *Binomial coefficient*

-> Find the binomial coefficient template in Expression palette

or use the function *binomial()*, i.e. **** = ****.

-> Binomial coefficients can be expressed in terms of factorials like this:

 = .

\* *Lists and sequences*

A list uses square brackets:

**> **



A sequence uses no parenthesis:

**> **



(Remarks: Ordering of elements and duplicate elements are preserved in lists and sequences. Lists and sequences are primarily computational.)

\* *seq()* and *$*, (generating sequences/lists/sets)

-> seq() examples:

**> **



**> **



**> **



**> **



**> **



**> **



*seq()* accepts an optional final argument representing the step size. e.g.

**> **



or more succinctly,

**> **



To generate a sequence in decreasing order, specify a negative step, e.g

**> **



Note: *seq()* cannot remain symbolic. (Has the same semantics as add() and mul()). E.g.

**> **

Error, unable to execute seq

Just like  and ,  also accepts data structures other than ranges, and instead of setting the control variable to each integer in a range, it sets it to each top-level operand of a data structure. E.g.

**> **



**> **



The $ operator described below, does not accept this syntax.

Remark: the sequence  will be empty unless ,

**> **



-> $ examples:

**> **



**> **



$ can remain symbolic:

**> **



The $ operator is most useful when used in one of its two shortcut forms:  generates a sequence consisting of  copies of , e.g.

**> **



$a ..b expands the range, e.g.

**> **



(The analogues using  are  = , and  = .)

(Remark: $ has the same semantics as *sum()* and *product()*)

(Remark:  can do everything that  can do, but using  where appropriate is more succinct.)

(\* Aside: Generating *nested* *structures*)

Examples:

1) Generating 

**> **



**> **



Now replace  by  to give

**> **



2) Generating 

**> **



**> **



**> **



Note that a sequence of sequences is just a sequence; sub-sequences lose their independence, e.g.

**> **

