**Chapter 9 Short Summary**

\* *Defining Finite Relations*

(& defines a Neutral operator).

Option 1) (General approach)

e.g. A relation R on A:

**> **

**> **

**> **



Use the relation like this:

**> **



Option 2) (explicitly state each pair)

e.g. A relation R on {1, 2}:

**> **

\* Checking if R is *reflexive, symmetric, transitive*:

To check these for &R on domain A use:

For reflexive:

**> **

Symmetric:

**> **

Transitive:

**> **

\* For *a partition of a set A*, the following must hold:

a) every set in  is non-empty:

**> **

b) the sets are pairwise disjoint:

**> **

c) the union of the sets in is :

**> **

\* *Partitions and Equivalence Relations*

Recall theorem: For an equivalence relation R on A, Equivalence classes of R  Partition of A.

Part 1) *(A Partition defines an Equivalence Relation)*

After having defined a Partition , (e.g. **> **), we need to define the relation as follows:

**> **

(We can then check that this is really an equivalence relation.) (Note that the neutral operator  can be used as a function if it is enclosed in backward quotes.)

Part 2) *(An Equivalence Relation defines a Partition)*

(For an **equivalence** relation R on A, we need to define a function  such that, for each , ; which gives the equivalence classes that make up the partition). Do this as follows:

**> **



Then the Partition is given by:

**> **

(We can then check that this really is a partition.)

\* *Solving Equations* using *solve()*

-> Use solve() to solve equations of the form $f(x)=0$.

**> **



or equivalently,

**> **



-> If the equation contains more than one variable then you must specify the variable to solve for as the second argument, e.g.

**> **



-> For expressions with more complicated solutions, *solve()* may not display explicit solutions by default. In this case, use *explicit* as a second argument. e.g. **>** 

(output not displayed in these notes)

-> For non-polynomial equations solve() generally finds only one solution. To find all solutions, give *solve()* a final argument of *allsolutions*. e.g. **** = ****

Maple has expressed this solution in terms of a parameter named  The tilde indicates that there are assumptions on this variable and we can find out what they are by applying the function  to the variable (without the tilde):

**> **

Originally \_Z1, renamed \_Z1~:

 is assumed to be: integer

-> To solve a system of simultaneous equations, put them into a set (or list), and when specifying variables to solve for (in the second argument), put them into a set (or list) too.

e.g.

**> **



-> You can also put single equations in a set as the argument of *solve()*, e.g.

**> **



This makes it easy to check solutions using , (which accepts a set or list as its second argument), e.g.

**> **



-> To solve single inequalities, put it in a set or list: e.g.

**> **



**> **



-> Solve simultaneous inequalities (or a mix of equations and inequalities) similarly:

e.g.

**> **

(output hidden due to being long)

**> **

(output hidden due to being long)

\* *Approximating solutions* using *fsolve()*

The Maple function *fsolve()* computes numerical approximations to the roots of equations to the current precision. (Useful when no exact solutions or the exact solutions not needed).

fsolve() is used in a similar way to solve(). E.g.

 **> **



-> But by default, fsolve() computes only real solutions. To find complex solutions give second argument *complex*, e.g.

**> **



-> (One way to use a different precision is to call fsolve() within *evalf()*, like this):

**> **



-> For non-polynomial equations fsolve() returns only one root, the one typically closest to 0, e.g.

**> **



Force fsolve() to find a different root by specifying an *isolating interval* (i.e. an interval that contains one and only one root), e.g.

**> **



(Tip: Try graphing to get an idea for which intervals to choose)

-> fsolve() also solves systems of simultaneous equations. You can find all the roots by specifying ranges for both variables, as above.

e.g.

Root 1:

**> **



Root 2:

**> **



\* *numer()* and *denom()*

Use numer() and denom() to extract the numerator and denominator of a fraction.

e.g.

**> **= ,  = .

\* *iquo* and *irem()*

Integer quotient and remainder are implemented as functions *iquo()* and *irem()* such that $q=iquo(a,b)$ and $r=irem(a,b)$ satisfy the equation $a=bq+r$ and the conditions that $|r|<|b|$ and the sign of *r* is the same as the sign of *a*. (Thus *q* and *r* are unique).

e.g.

**> **



**> **



**> **



-> (Remark: In Maple, divisibility functions that apply only to integers have names that begin with ‘i’ to distinguish them from those that apply to polynomials.)

-> Aside: Testing for divisibility

There appears to be no way to use the vertical bar notation actively for divisibility, but we could implement an infix integer divisibility predicate and test it like this:

**> **



**> **



\* *igcd()* and *ilcm()*

(i.e. Integer greatest common divisor (gcd) and Integer lowest common multiple (lcm)).

Use as follows:

e.g.

**> **



**> **



Note that igcd() and ilcm() accept any number of arguments. However gcd() and lcm() only accept two arguments.

Note that **** = .

(Recall Euclid’s Algorithm for computing gcd.)

(Recall for computing lcm.)