Chapter 2 - Simple Examples

This section contains some simple examples of programs written in the language to give an idea of what it looks like.

Factorial

This example calculates the factorial of an argument.

```
factorial[0] -> 1;
factorial[n] -> n * factorial[n-1];

top[] -> outnum[factorial[6]];
```

A ReWrite program consists of a list of rewrite rules separated by semi-colons; each rule defines a transformation and is part of a function. The basic structure is

```
rule [patterns ]  -> rhs ;
```

A match needs to be made with the patterns in order for a particular rule to be used. The rule used will be the first one that matches.

For example, with the function `factorial` defined above, if it is called:

- with more than two arguments or no argument it will fail;
- with the integer `0` the first rule will be activated and `1` will be returned;
- with any value `n` other than 0 the second rule will be activated, and

```
n * factorial[n-1]
```

will be calculated, calling factorial again. (If n<0, this will be a problem)

A program must somewhere contain a rule `top[] -> ...`
When a program is executed, it needs to have a well defined starting point, so the function `top[]` is called, and whatever rule that matches is activated.

The above function will fail to terminate if the argument is not an integer or is negative. This problem could be prevented using the following modified function:

```
factorial[0] -> 1;
factorial[n:int]::n>0 -> n * factorial[n-1];
```

This introduces another idea used in ReWrite; conditions (sometimes called guards in functional programming languages). There are two sorts used here:

- `parameter :type`

This is used to indicate that a parameter to a function must be a certain type or the rule won't match.

- `rule [patterns ]::condition  -> rhs ;`

In this case the `patterns` must match and the `condition` must be satisfied, otherwise the rule won't match.

Equals

```
equal[x,x] -> true;
equal[x,y] -> false;
```

This function will return `true` if called with any two arguments that are the same, as the match with the first rule will only succeed if it is given two equal arguments, since both of the arguments have the same name. (the `x` appearing twice makes it a non-linear rule). `false` will be returned if the two arguments are different.

Non-linear rules are a very powerful part of the pattern matching.

Insert into ordered list

This function inserts a value (a number) into its correct place of a list of ascending order.

```
insert[x,] -> x;
insert[x,a,.rest]::x<a -> x,a,.rest;
insert[x,a,.rest] -> a,.insert[x,rest];
```

A list (represented ...) is a way of gathering a collection of values into a single value. A splice (represented by a preceding `.`) is the reverse of a list; it takes a list and turns it into its components.

In the example above,
`insert[x,a,.rest]`
will be matched if the first argument is any value and the second argument is a list containing at least one element. The first element of the list will be called `a`; the rest will be gathered into another list called `rest`. From this example it can be seen that list and splice act in the 'opposite' way when used as part of a match on the left hand side.

For example if
`insert[3,1,4,5]` was called then
`x` would have the value `3`,
`a` would have the value `1`,
rest would have the value `4,5` for the duration of that rule.

Continuing that example, x is not less than a, so the second rule is not matched, but the third one is so the next step will be:

```
1,.insert[3,4,5]
```

(*)
When `insert[3,4,5]` is called,
`x` will be `3`,
`a` will be `4`,
rest will be `5`,
so the second rule is matched and the right hand side of this will be:
`3,4,.5`
which becomes
`3,4,5` (remember `splice` undoes a `list`)
which going back into (*) gives:
`1,3,4,5`
The expected result.

Another (higher level) way of expressing the function is this:

```
insert[x,.lt,a,b,.gt]::a<=x & x<b -> .lt,a,x,b,.gt;
insert[x,a,.gt]::x<a -> x,a,.gt;
insert[x,l:lis] -> .lis,x;
```

This example shows the `splice` operator used in a way that requires backtracking - there is more than one match that may work, so all the possible matches are checked in a systematic way until a match is found that works, or the match fails.

The first rule is the usual case, where the two elements to insert `x` between are found, so the match breaks up the list into a part consisting of elements less than or equal to `x` (`.lt,a`) and a part consisting of elements greater than `x` (`b,.gt`),
The second rule deals with the case where `x` is smaller than all of the elements on the list,
The third rule deals with either the case where the list is empty or `x` is larger than all of the elements in the list (as if the first two rules are not matched, these are the only cases left).

This `list`/`splice` notation is very powerful and can mean easy definition of what would otherwise be low level functions.
For example (for those of you that know lisp):

```
car[x,.rest] -> x;

cdr[x,.rest] -> rest;

cons[x,rest] -> x,.rest;

join[x:lis,y:lis] -> .x,.y;
```

Bubblesort

This is a very simple sort algorithm on a list - it simply takes any two elements that are in the

wrong order and swaps them, repeating until done. Although simply expressed, it is not very efficient - the worst case is O(N^3).

```
sort[.a,b,c,.d]::b>c -> sort[.a,c,b,.d];
sort[x:lis] -> x;
```