Chapter 5 - Lists, Eq and Output

This section gives the ReWrite functions that operate on lists, functions that deal with equality, and functions that send output to the screen. `list`, `splice` and `outnum` are particularly important.

List Operations

list, splice

`list` and `splice` are the two most important functions in ReWrite.

`list` makes a list out of whatever it is passed.

`splice` is the inverse of `list`: it takes a list as an argument, and returns the entries as results. `splice` is a good example of a function than returns a variable number of arguments.

`list[vals ]` or `vals` -> `list ;`
(directly coded)

`splice[list ]` or `.list` -> `vals ;`
(directly coded)

Example:
 `splice[1,2,3,4]` -> `1,2,3,4`

 `list[1,2,3,4]` -> `1,2,3,4`

Note:
 `splice[list[x ]]=x` , but
 `list[splice[x ]]` may not be identical to `x` , as `splice[x ]` requires `x` to be a list.

join

`join` combines several lists into one.

`join[lists ]` -> `list ;`

```
  join[] -> ;
  join[a:lis] -> a;
  join[a:lis,.rest] -> join[.a,.b,.rest];
```

Note that the first line of this definition is all that would be necessary to join two lists, the other lines merely extend this.

Example:
`join[1,2,3,,4,5,6]` -> `1,2,3,4,5,6`

nth, nthcdr

`nth` is a function that returns the nth element of a list.

`nth[int ,list ]->val ;`

```
nth[1,a,.b] -> a;
nth[x:int,a,.b] -> nth[x-1,b];
```

`nthcdr` is a function that returns the list with the first n elements removed.

`nthcdr[int ,list ]->list ;`

```
nthcdr[0,a] -> a;
nthcdr[x:int,a,.b] -> nthcdr[x-1,b];
nthcdr[x:int,] -> ;
```

Note that lists are indexed from 1 being the first element, so for example:
`nth[3,41,42,43,44]` -> `43`
`nthcdr[3,41,42,43,44]` -> `43,44`

length

`length` returns the number of elements in a list passed to it.

`length[list ]->int ;`

```
length[] -> 0;
length[a,.b] -> length[b]+1;
```

Example:
`length[41,42,4,3,9,44]` -> `4`

ismember

Given a value and a list, ismember returns `true` if the value is contained in the list, `false` otherwise.

`ismember[val ,list ]->bool ;`

```
ismember[x,._,x,._] -> true;
ismember[x,y:lis] -> false;
```

Example:

`ismember[4,1,2,3,5]` -> `false`
`ismember[3,1,2,3,5]` -> `true`

nil, prepend, append, car, cdr

These list constructor functions are somewhat superfluous, as the same results can usually be achieved more efficiently with list and splice. They are here only for completeness.

`nil` returns an empty list.
`nil[] ->;`

```
  nil[] -> ;
```

`prepend` prepends a value to the start of a list.
`prepend[val ,list ]-> list ;`

```
  prepend[x,l:lis] -> x,.l;
```

`append` appends a value to the end of a list.
`append[val ,list ]-> list ;`

```
  append[x,l:lis] -> .l,x;
```

`car` returns the first element of a list.
`car[list ]-> val ;`

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

`cdr` returns a list with the first element removed.
`cdr[list ]-> list ;`

```
  cdr[a,.rest] -> rest;
```

Examples:

`prepend[1,2,3,4]` -> `1,2,3,4`
This is the same as `1,.2,3,4` -> `1,2,3,4`

`append[1,2,3,4]` -> `2,3,4,1`

Equality Operations

These two functions compare two arguments two see whether they are equal or not. The two arguments can be anything. These should not need to be used much, since usually any checking they do can be done much easier as part of a match.

```
eq[val ,val ] or val = val -> bool ;
```

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

```
ne[val ,val ] or val != val -> bool ;
```

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

Screen Output

These are the functions that output to the screen. None of them are directly coded, but the full coding is not very illuminating for some or them, so has been omitted.

outnum, outbase, out, outln

`outnum` outputs its arguments to screen in the same form that they could be input, assuming that all the numbers are to be displayed base 10. `outbase` does the same as outnum, except you specify the base. `out` and `outln` are similar to outnum, but suppress the quotes on characters and tokens. Note that a separate function for displaying lists is not required, as all four will display the list structure.

```
outnum[vals ] ->;
outbase[int ,vals ] ->;
out[vals ] ->;
outln[vals ] ->;
```

```
  outnum[.x] -> outbase[10,.x];
```

For example if there was a rule:

```
  outall[.x] -> outnum[.x],outbase[2,.x],out[.x],outln[.x];

  outall[$41,66,,`token,"string",false,null]
```

would return nothing and print to the screen the following:

```
65,66,,`token,"string",false,null
1000001,1000010,,`token,"string",false,null
6566tokenstringfalsenull6566tokenstringfalsenull
```

Note that the result of `outnum` could be copied and pasted into a ReWrite program, the other three could not.

There are some other 'out' functions - `outs` and `outstring`. These should not be used, as they are only there for compatibility with some old code in the compiler.

prettyprint, prettyprint16

These are routines that are similar to `outnum` and `outbase` with a base of 16, the difference being that the attempt to break up long structures into manageable sized portions on each line so that the structure can be seen more easily. Note that they can also be suppressed by holding down the mouse button (useful if the list was much longer than expected, but you don't want to abort).

```
prettyprint[vals ] ->;
prettyprint16[vals ] ->;
```

SYSresidue

This is a function that tells ReWrite how to deal with any 'extra stuff' left over at the end of an evaluation. This can be overwritten by the user. It is important that SYSresidue returns nothing. The default definition is the following:

```
  SYSresidue[] -> ;
  SYSresidue[.x] -> outs['Returned:'],prettyprint[.x];
```

For example:

```
  top[] -> outnum[29],71;
```

returns:

```
29
Returned:
71
```

outand (a note on debugging)

`outand` is an example of a very simple diagnostic routine. If you put it around part or all of any expressions on the right hand side of a rule it will display the result, but otherwise be transparent. Also, since the other output functions return nothing, they can also be transparently inserted into the rhs of expressions.

```
outand[vals ] -> vals ;
```

```
  outand[.x] -> prettyprint[.x],.x;
```

For example we could change this:

```
  thecall[x,y,.z] -> buggytrickyfunction[x,y,z];
```

into this:

```
  thecall[x,y,.z] -> outstring['#tricky'],
        outand[buggytrickyfunction[outand[x,y,z]]];
```

This would output:

```
#tricky
```
the arguments passed to `buggytrickyfunction`
the result returned from `buggytrickyfunction`

all transparent to the code. (OK, it's not any sort of replacement for a full symbolic debugger with stack dump, trace and single step, but it's easier to write and will have to do for the moment).