Chapter 6 - Numerical, Boolean

This sections deals with the functions designed to work on atoms (integers), including integer, boolean and bitwise functions.

Numerical Operations

One thing worth noting about the n-ary numerical functions is that they can be called with zero or one argument.

If an n-ary function is called with one argument, it just returns that one argument.
If an n-ary function is called with zero arguments, it returns some sort of identity for that function. For example:

```
add[6] -> 6;
mul[] -> 1;
```

Basic Arithmetic

ReWrite supports the following operations, all of which work on integers. The usual precedence rules apply.

```
add[ints ]  or  int + int  -> int ;
mul[ints ]  or  int * int  -> int ;
neg[int ]  or  -int  -> int ;
sub[int ,int ]  or  int - int  -> int ;
div[int ,int ]  or  int / int  -> int ;
mod[int ,int ]  or  int % int  -> int ;
```

(all directly coded)

Example:

```
2 + 127 % 109 -> 9
```

In addition, the following comparison operators are given for integers and characters (don't forget that eq, ne are also defined, but for general values rather than just integers).

Comparison

```
ge[int ,int ]  or  int >= int  -> bool ;
gt[int ,int ]  or  int > int  -> bool ;
le[int ,int ]  or  int <= int  -> bool ;
lt[int ,int ]  or  int < int  -> bool ;
```

(all directly coded)

```
ge[char ,char ]  or  char >= char  -> bool ;
```

```
gt[char ,char ]  or  char > char  -> bool ;
le[char ,char ]  or  char <= char  -> bool ;
lt[char ,char ]  or  char < char  -> bool ;
```
(not directly coded)

Example:
```
  2 + 127 % 109 > 8 -> true
```

There are several other numerical functions supported:

Absolute value

```
abs[int ] -> int ;
```

```
  abs[x:int]::x>=0 -> x;
  abs[x:int] -> -x;
```

max, min

```
max[ints ] -> int ;
```

```
  max[] -> minint;
  max[x:int] -> x;
  max[x:int,y:int,.rest]::x>=y -> max[x,.rest];
  max[x:int,y:int,.rest] -> max[y,.rest];
```

```
min[ints ] -> int ;
```

```
  min[] -> maxint;
  min[x:int] -> x;
  min[x:int,y:int,.rest]::x<=y -> min[x,.rest];
  min[x:int,y:int,.rest] -> min[y,.rest];
```

Note that these require the arguments not to be in a list.
For example:
```
  max[1,2,6,3] -> 6;
  max[1,2,6,3] -> fails
```
If you want to find the largest number in the list `lis`, use `max[.lis]`.

maxint, minint

These functions just provide easy access to these constants. Note that I consider minint to be -2,147,483,647, not -2,147,483,648.

```
maxint[] -> $7FFFFFFF;
```

```
  maxint[] -> $7FFFFFFF;
```

```
minint[] -> -$7FFFFFFF;
```

```
  minint[] -> -$7FFFFFFF;
```

random, Uniqnum, tickcount

`random` returns a pseudo-random integer in the range -32767 to 32767. See Inside Macintosh, Vol I, page 194.

```
random[] -> int ;
```
(directly coded)

Logical Operations

These are the usual logical operations.

```
and[bools ]  or  bool & bool  -> bool ;
```

```
  and[] -> true;
  and[false,.rest] -> false;
  and[x:bool,.rest] -> and[.rest];
```

```
or[bools ]  or  bool | bool  -> bool ;
```

```
  or[] -> false;
  or[false,.rest] -> or[.rest];
  or[x:bool,.rest] -> x;
```

```
not[bool ]  or  !bool  -> bool ;
```

```
  not[false] -> true;
  not[true] -> false;
```

Bit Operations

bitand, bitor, bitnot

These are just the typical bit operations, treating an integer as 32 bits

```
bitand[ints ] -> int ;
bitor[ints ] -> int ;
bitnot[int ] -> int ;
```
(directly coded)

For example:
```
  bitand[%1011,%01011101] -> 9;
```
(%1001)

hi, lo

Given `x` an integer consists of 32 bits:
`hi[x ]` returns the most significant 16 bits (shifted right by 16 bits),
`lo[x ]` returns the least significant 16 bits.
This function is mostly used by the assembler.

```
hi[int ] -> int ;
lo[int ] -> int ;
```
(directly coded)

shift

`shift[x ,y ]` shifts the integer y left by x bits - effectively it returns y*2^x

```
shift[int ,int ] -> int ;
```
(directly coded)

Examples:
```
  shift[3,100] -> 800;
  shift[-3,800] -> 12;
```

bitff1, bitff0, bitcount

`bitff1` and `bitff0` are somewhat anomalous functions. They are only documented here because they are used later for the definition of sets.

If an integer (considered in binary) is indexed the following way:

hen `bitff1` scans through from the most significant to the least and returns the index of the first 1 that it finds. `bitff0` similarly looks for a 0.

```
bitff1[int ] -> int ;
```
(directly coded)

```
bitff0[int ] -> int ;
```

```
  bitff0[x:int] -> bitff1[bitnot[x]];
```

Examples:
```
  bitff1[100] -> 25;
```

```
bitff0[100] -> 0;
bitff1[0] -> 32;
```

To create a simple integer log base 2 function, we could do the following:

```
log2[x:int] -> 31-bitff1[x];
```

bitcount returns the number of 1's in the binary representation of a number.

```
bitcount[int ] -> int ;
```
(directly coded)

For example:

```
bitcount[%10101011] -> 5;
```