Chapter 7 - Trees, Sets, Misc

This sections firstly deals with the functions designed to work on two particular ways of looking at lists; trees and sets. These are more specialised functions, and may be safely ignored for most purposes.

After this, the section deals with a miscellany of functions, most of which are useful only to specialised applications. Some of the more unusual functions may disappear in future versions of ReWrite.

Tree operations

Given that lists may contain elements that are also lists etc., a list can be regarded as a tree.

For example, the list `1,2,3,4,5,6,7,8` can be represented like this:

eWrite has functions for working with this. First some definitions.

An indexed-occurrence of the tree is the list of numbers of choices of branches going down the tree. For instance, to get to the node (list) above the 5, we take the third branch, then the third again, so the indexed occurrence is `3,3`. This gets us to here:

et us call the tree `M` and the indexed occurrence `u`. There are two main things that we can do with this:

• We can take the subtree at this node (denoted `M¬u`).

his gives us the tree:

The list corresponding to this would be `5`.

• We can replace the subtree at this node by something else, say `N` (denoted `M[u∅N]`). For example, if we let `N = 1,2,` `M[u∅N]` gives the tree:

he list corresponding to this would be `1,2,3,4,1,2,6,7,8`.

Now for the functions:

treeslicesplice (tss)

```
tss[ints ,tree ,val ]-> ints ,tree ,val ;
tss[ints ,tree ,val ,fill ]-> ints ,tree ,val ;
```
(directly coded)

The main function (on which all the tree functions are based is the `treeslicesplice` or `tss` (either name can be used). This function has two forms.

• In the first form, it takes three arguments,
  - an indexed-occurrence `u` ,

- a tree `M` ,
- a value `N` to be spliced into the tree.

It returns

- the same indexed occurrence `u` ,
- `M [u ∅N ]`,
- `M ¬u` .

so `treeslicesplice[u ,M , N ] -> u ,M [u ∅N ],M ¬u` .

For example:

```
treeslicesplice[3,3, 1,2,3,4,5,6,7,8, 1,2] ->
                        3,3, 1,2,3,4,1,2,6,7,8, 5
```

Note: if the indexed occurrence has a non-integer in it, or the node in the tree doesn't exist, then `treeslicesplice` fails to match.

• An additional form has a fourth argument, which is used to fill out the tree if the node doesn't exist, then the operation is done normally. If the node exists, any fourth argument is ignored.

For example:

```
treeslicesplice[3,3, 1,2,3,4,5,6,7,8, 1,2, 9]
               -> 3,3, 1,2,3,4,1,2,6,7,8, 5

treeslicesplice[3,3, 5, 1,2, 7] ->
                        3,3, 7,7,7,7,1,2, 7
```

treetake, treeput

There are two derived functions for taking a subtree and for replacing a subtree by another object.

• `treetake[u ,M , N ] -> M ¬u` .
• `treetake[u ,M , N ,fill ] -> M ¬u` .

```
treetake[ints ,tree ,val ]-> val ;
treetake[ints ,tree ,val ,fill ]-> val ;
```

```
treetake[part,tree,.x] -> third[tss[part,tree,0,.x]];
third[a,b,c,.x] -> c;
```

Note that `treetake` will return 0 if a nonexistent node is accessed.

• `treeput[u ,M , N ] -> M [u ∅N ]`.

```
treeput[ints ,tree ,val ]-> tree ;
treeput[ints ,tree ,val ,fill ]-> tree ;
```

```
  treeput[.x] -> second[tss[.x]];
  second[a,b,.x] -> b;
```

The tree functions can be used as a 'poor man's sparse array', particularly with the optional 'fill' argument. The compiler uses this idea extensively until such time as a better array structure is provided in a future version.

Set Operations

ReWrite can use lists to represent sets just by ignoring the ordering on the list. However this is very inefficient except for very small sets, and has the problem of duplication of an element in a list. so there a number of functions built in to handle sets composed of non-negative integers only in a different more efficient way. The set is stored as a list of integers, with each integer representing 32 elements of the set. The storage is as follows:

or example, the set 2,3,5,7,11,13,17,19,23,29,31,37 would be represented:

```
%00110101000101000101000100000101,
 %00000100000000000000000000000000.
```

The bit operations (and the tree operations with a length 1 indexed-occurrence are used to implement the following functions:

osetaddmem, osetremmem, osetismem

`osetaddmem`, `osetremmem` and `osetismem` are used to add, remove and check membership of a set.

```
osetaddmem[int ,set ] -> set ;
```

```
  osetaddmem[x:int,set] ->
```

```
    osetaddmemx[bitand[x,$1F],tss[shift[-5,x]+1,set,0,0]];

  osetaddmemx[x:int,pos,set,part] ->
    treeput[pos,set,bitor[part,shift[-x,$80000000]]];

osetremmem[int ,set ] -> set ;

  osetremmem[x:int,set] ->
    osetremmemx[bitand[x,$1F],tss[shift[-5,x]+1,set,0,0]];

  osetremmemx[x:int,pos,set,part] ->
    treeput[pos,set,bitand[part,bitnot[shift[-x,$80000000]]]];

osetismem[int ,set ] -> bool ;

  osetismem[x:int,set] ->
    bitand[shift[-bitand[x,$1F],$80000000],
         treetake[shift[-5,x]+1,set,0]]!=0;
```

osetcount, osetfindfirst

`osetcount` returns the number of elements in a set.

```
osetcount[set ] -> int ;

  osetcount[] -> 0;
  osetcount[a,.b] -> bitcount[a]+osetcount[b];
```

`osetfindfirst` returns the lowest number contained in the set. Note that if the set is empty, some negative number is returned (no guarantee as to the exact number).

```
osetfindfirst[set ] -> int ;

  osetfindfirst[] -> minint[];
  osetfindfirst[0,.a] -> 32+osetfindfirst[a];
  osetfindfirst[x,.a] -> bitff1[x];
```

osetunion, osetintersect, osetdiff

`osetunion`, `osetintersect` and `osetdiff` find the union, intersection and difference of two sets.

```
osetunion[set ,set ] -> set ;

  osetunion[,a] -> a;
  osetunion[a,] -> a;
  osetunion[a,.r1,b,.r2] -> bitor[a,b],.osetunion[r1,r2];

osetintersect[set ,set ] -> set ;

  osetintersect[,a] -> ;
  osetintersect[a,] -> ;
```

```
  osetintersect[a,.r1,b,.r2] ->
    bitand[a,b],.osetintersect[r1,r2];

osetdiff[set ,set ] -> set ;

  osetdiff[,a] -> ;
  osetdiff[a,] -> a;
  osetdiff[a,.r1,b,.r2] ->
    bitand[a,bitnot[b]],.osetdiff[r1,r2];
```

osetfill

`osetfill` of a number n generates the set `0,1,2,3, ... ,n`

```
osetfill[int ] -> set ;

  osetfill[x:int]::bitand[x+1,$1F]=0 ->
     second[tss[shift[-5,x+1],0,-1,-1]];
  osetfill[x:int] ->
     second[tss[shift[-5,x+33],0,shift[32-bitand[x+1,$1F],-1],-1]];
```

Other functions in brief:

first, second, third

These functions return their first, second or third argument respectively.

```
first[vals ] -> val ;

  first[a,.x] -> a;

second[vals ] -> val ;

  second[a,b,.x] -> b;

third[vals ] -> val ;

  third[a,b,c,.x] -> c;
```

tickcount, Uniqnum

`tickcount` returns the number of 1/60ths of a second since the Macintosh was turned on (this is good for timing code). See Inside Macintosh, Vol I, page 260.

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

`Uniqnum` returns a number guaranteed to be different to any other call of Uniqnum since the

memory has been reset. At the moment it just starts at 1,000,000,000 and increments this by one each time. This is probably not very useful in most applications.

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

button, buttonpress

`button` returns whether the mouse button is down or not. `buttonpress` returns whether there is a mouse down event on the toolbox event queue.

```
button[] -> bool ;

buttonpress[] -> bool ;
```

****

An important note about the functions from here to the end of the chapter: Whenever a '`char`' is referred to, they really mean '`int`' as they were all written when there were just lists and longints as types, so characters were just integers that were interpreted as ascii values. Now that the character type has been added, these will be replaced in future versions by (incompatible) versions using the proper types. They have been left in this form because the current version of the compiler uses them.

****

getkey, getline

`getkey` gets a 'character' from the keyboard. `getline` gets a sequence of characters terminated by a return . (The return character is not included)

```
getkey[] -> char ;

getline[] -> chars ;
```

```
  getline[] -> outs[$d],getlinex[,getkeyx[]];
  getlinex[sofar,$d] -> .sofar;
  getlinex[sofar,x:int] -> outs[x],getlinex[.sofar,x,getkeyx[]];
```

iswhitespace, isalpha, isdigit

`iswhitespace`, `isalpha`, `isdigit` determine whether a character is a whitespace (non-printing character), an alphanumeric character (_ is considered an alphanumeric character), or a decimal digit respectively. These are mainly useful for parsing.

```
iswhitespace[int(char) ] -> int(bool) ;
```

```
  iswhitespace[x:int] -> x<=$20 | x>=$7f;
```

```
isalpha[int(char) ] -> int(bool) ;

  isalpha[x:int] ->
       or[(x>='A' & x<='Z'),(x>='a' & x<='z'),(x = '_')];

isdigit[int(char) ] -> int(bool) ;

  isdigit[x:int] -> x>='0' & x<='9';
```

atoi, itoa

`atoi` converts a string of characters that forms an `integer constant` (see definitions section) into that constant. If `atoi` cannot interpret its input as a number, it returns .

```
atoi[ints(chars) ] -> int ;
```
(not given here, but see the next chapter)

For example:
```
  atoi['16'] -> 16;
  atoi['-$16'] -> -22;
```

`itoa` converts an integer into a string of characters in an arbitrary base (<=36).

```
itoa[int ,int ] -> ints(chars)  ;

  itoa[b:int,0] -> ;
  itoa[b:int,$80000000] -> '$80000000'; (* special case *)
  itoa[b:int,x:int]::x<0 -> '-',itoa[b,-x];
  itoa[b:int,x:int] -> itoa[b,x/b],todigit[x%b];

  todigit[x:int]::x<10 -> x+'0';
  todigit[x:int] -> x+'7';
```

For example:
```
  itoa[3,10] -> '101';
```

This returns the decimal number 10 in base 3.

outstring

`outstring` is an output routine that takes a structure consisting of lists and longints, and displays it, turning all the integers to characters. Do not use any types other than integers or lists with this function or it will crash. This function will no appear in later versions.

```
outstring[vals* ] -> ;
```

For example:

```
  outstring[65,,$42] -> ;
```

and displays:

```
AB
```