Res. Lett. Inf. Math. Sci., (2000) **1**, 37-64
Available online at http://www.massey.ac.nz/~wwiims/rlims/

C/C++ Implementation of Functions of the Class LT_0.

Elena Calude
I.I.M.S., Massey University Albany Campus, Auckland, N.Z.
E.Calude@massey.ac.nz

Peter Kay
I.I.M.S., Massey University Albany Campus, Auckland, N.Z.
P.Kay@massey.ac.nz

Weiwei Luo
NASA Huntsville, USA
Weiwei_Luo@hotmail.com

Abstract
This report describes an on-going implementation, in C/C++, of the functions and schemes of the formal system LT_0, presented in the paper Caporaso, Pani and Covino [1]. The final aim is to be able to effectively construct a "small manageable" Exponential Diophantine Equation which represents (in the sense of Chaitin [2]) an algorithmical random binary sequence.

Introduction

At the beginning of the century mathematicians (led by David Hilbert) attempted to give mathematics a solid, rigorous base. They started with a few basic axioms and attempted to derive all mathematical theorems.

> "Hilbert's idea is the culmination of two thousand years of mathematical tradition going back to Euclid's axiomatic treatment of geometry, going back to Leibniz's dream of a symbolic logic and Russell and Whitehead's monumental *Principia Mathematica.* Hilbert's dream was to once and for all clarify the methods of mathematical reasoning. Hilbert wanted to formulate a **formal axiomatic system** which would encompass all of mathematics.
>
> Hilbert emphasised a number of key properties that such a formal axiomatic system should have. It's like a computer programming language. It's a precise statement about the methods of reasoning, the postulates and the methods of inference that we accept as mathematicians. Furthermore, Hilbert stipulated that the formal axiomatic system encompassing all of mathematics that he wanted to construct should be "**consistent**" and it should be "**complete**".
>
> Consistent means that you shouldn't be able to prove an assertion and the contrary of the assertion. You shouldn't be able to prove *A* and not *A*…
>
> Complete means that if you make a meaningful assertion you should be able to settle it one way or the other. It means that either *A* or not *A* should be a theorem, should be provable from the axioms using the rules of inference in the formal axiomatic system.
>
> …
>
> Consistent and complete means only truth and all the truth. They seem like reasonable requirements. There's a funny consequence, though, having to do with something called the **decision problem**. In German it's the Entscheidungsproblem.
>
> Hilbert ascribed a great deal of importance to the decision problem.

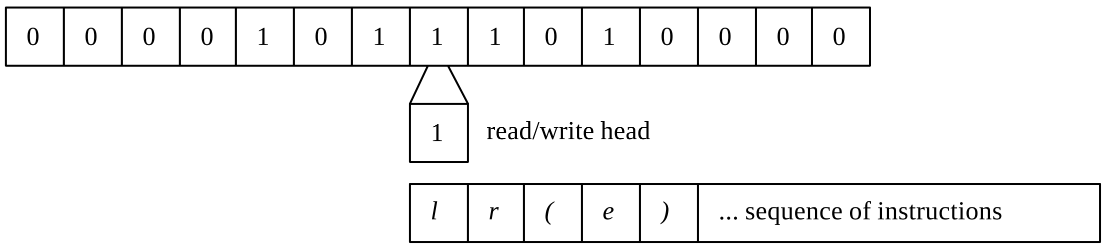
Solving the decision problem for a formal axiomatic system means finding an algorithm that enables you to decide whether any given meaningful assertion is a theorem or not. A solution of the decision problem is called a decision procedure.

…The formal axiomatic system that Hilbert wanted to construct would have included all of mathematics: elementary arithmetic, calculus, algebra, everything. If there's a decision procedure, then mathematicians are out of work. This **algorithm**, this **mechanical procedure**, can check whether something is a theorem or not, can check whether it's true or not. " [3]

In the 1930s, Kurt Gödel [4] showed that a consistent formal axiomatic system with integers and addition/multiplication could not be **complete**, i.e. there exist "true" but unprovable statements. Hilbert's dream was never going to work.

To this aim Alan Turing [5], also in the 1930s, presented an argument that once again destroyed Hilbert's dream. Turing invented a machine on paper, that is now called a "Turing Machine".

A Turing Machine consists of a paper tape which contains a series of numbers, either 0 or 1. One particular number, the **observed symbol** is under a 'read/write' head.



Each machine instruction can move the tape either one square to the left or right, write either a 0 or a 1 into the current square, or decide which instruction to execute, depending on the value of the observed symbol. Eventually (perhaps) the sequence of instructions will finish (or halt).

Turing showed the following:

"…That there is no algorithm, no mechanical procedure, which will decide… if a computer program ever halts. …what Hilbert wanted was a formal axiomatic system from which all mathematical truth should follow, only mathematical truth, and all mathematical truth. If Hilbert could do that, it would give us a mechanical procedure to decide if a computer program will ever halt.

Why? You just run through all possible proofs until you either find a proof that the program halts or you find a proof that it never halts. So if Hilbert's dream of a finite set of axioms from which all of mathematical truth should follow were possible, then by running through all possible proofs checking which ones are correct, you would be able to decide if any computer program halts. In principle you could. But you **can't** by Turing's very simple argument…" [3]

Formal systems cannot live up to Hilbert's dream. They have, however, other interesting properties. One is that it is possible to create a machine (a computer program) that, given a detailed (step by step) proof of a theorem, the machine is able to verify the correctness of the proof.

The paper [1] constructs a formal system $\mathbf{LT_0}$, rich enough to be incomplete, but which scales **linearly** with proof complexity.

The intention of this paper is to produce an annotation of the original paper [1], together with an implementation in C/C++. Sections 2 and 3 are covered in this first report. (The numbering of the original paper is referred to in parentheses.)

Functions, schemes and a paper tape example.

In this section we introduce the basic building blocks of the system. Data is represented by **lists**, to which we can apply **functions** and **schemes**. Functions are built from 3 'basic functions' that can extract parts of a list. Schemes are built from 'basic schemes', equality, negation and disjunction.

(Definition 1.1) All objects in our system can be written in terms of a 'list'.

0	is a ternary list, all other lists are in the form:
$[X, \quad Y, \quad Z]$	where X, Y and Z are themselves lists

Examples of lists are:

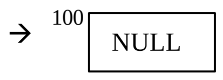
0
[0, 0, 0]
[[0, 0, 0], 0, 0]
[0, [0, [0, 0, 0], 0], 0]

In C we represent 0 by a pointer with the value NULL.

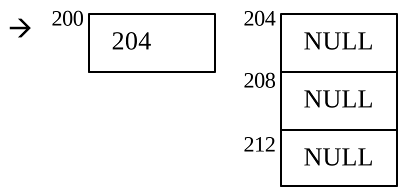
A list in the form $[X, \quad Y, \quad Z]$ is represented by a pointer that points to an array of 3 more pointers.

The examples above are represented by:

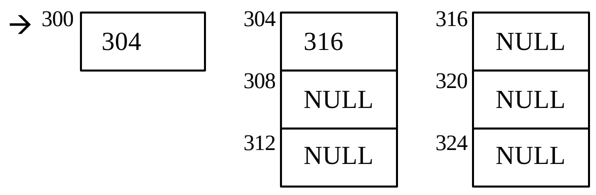
0



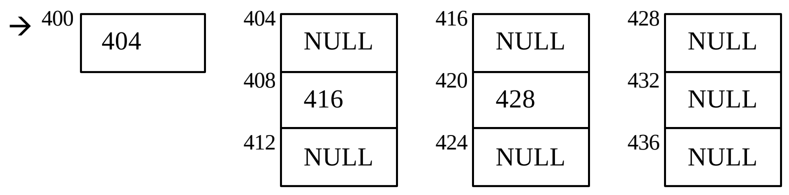
[0, 0, 0]



[[0, 0, 0], 0, 0]



[0, [0, [0, 0, 0], 0], 0]



Arbitrary memory locations have been chosen to illustrate addresses and values.

In C, these ternary lists can be represented by pointers. In fact the **only** variable type we use is a pointer. Either a pointer points to another pointer, or it has the value NULL. A list can be represented by a pointer of type `void **` (a pointer to a pointer of no particular type), or by a `void ***`, or a `void ****`, etc depending on the depth of the list. We define a `list` to be of type `void **`.

Strict type checking is enabled, and to ensure that pointer types for lists are compatible (e.g. `void *` and `void **` are not, but they both represent lists) we frequently use the cast `(list)` to convert a `void *` pointer into a `void **` pointer.

```
typedef void ** list;   // a list is a pointer to a list
                        // The depth depends on list itself
                        // so void **, void ***, void **** etc are all valid
                        // descriptions of a list
                        // choose void ** as the list description
                        // We will often use the cast: (list)
list x;                 // x and y are list variables
list y = NULL;
```

(Definition 1.2) Individual elements of a list can be reached with the help of **destructors**. Destructors return the head, body or tail of a list.

$0_i =_{df} 0 \quad (i = 1,2,3)$ ($=_{df}$ → 'by definition')

$$[X, \ Y, \ Z]_1 =_{df} X$$
$$[X, \ Y, \ Z]_2 =_{df} Y$$
$$[X, \ Y, \ Z]_3 =_{df} Z$$

The original list is not altered. No deep copying is done. The value returned is the address of that part of the list. We use subscripts 1, 2 or 3 to denote which destructor to return.

```
list destructor(list X, int i) {        // i is 1, 2 or 3
   if (X == NULL) return NULL;
   else return (list) X[i-1];           // one of the 3 pointers
}
```

Multiple subscripts to lists are always meaningful.

$x_{ij} = (x_i)_j$ is coded as:

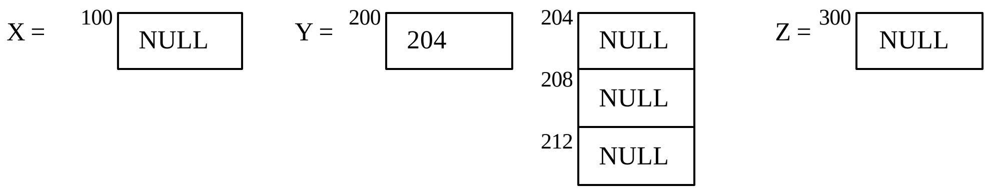
```
   y = destructor(destructor(x, i), j);
```

Three macros `HEAD`, `BODY` and `TAIL` are defined to make our C programs more readable.

```
#define HEAD(x) destructor(x,1)
#define BODY(x) destructor(x,2)
#define TAIL(x) destructor(x,3)
```

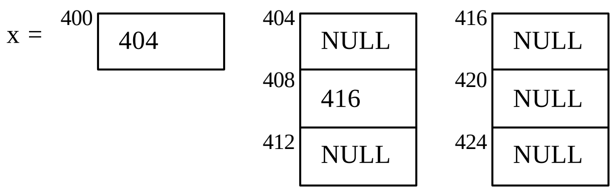
The function `new_list(list X, list Y, list Z)` is one way to create a list in the form $[X,\ \ Y,\ \ Z]$. If X, Y and Z are lists that already exist, `new_list` allocates memory and makes a copy of each of the lists. This is a 'deep' copy – the created list does not contain any of the original pointers.

If X = 0, Y = [0, 0, 0] and Z = 0



then `x = new_list(list X, list Y, list Z);` creates the following structure:

x = [0, [0, 0, 0], 0]



```
list new_list(list X = NULL, list Y = NULL, list Z = NULL) {
   list *M;                          // a list pointer
   M = (list *)calloc(3,sizeof(list)); // point it to 3 lists all set to NULL
   // recurse if necessary
   if (X) M[0] = new_list(HEAD(X), BODY(X), TAIL(X));
   if (Y) M[1] = new_list(HEAD(Y), BODY(Y), TAIL(Y));
   if (Z) M[2] = new_list(HEAD(Z), BODY(Z), TAIL(Z));
   return (list)M;                   // we need the cast
}
```

(Definition 1.3) Lists are referred to by using upper-case characters normally in the range U – Z.

$U,\dots,Z$	are generic lists
x	is a list 'variable'

These can be declared in C by:

```
   list U, V, Z;
   list x;
```

Trailing 0s in a list can be omitted.

$[X,\ \ Y]$	is short for $[X,\ \ Y,\ \ 0]$
$[X]$	is short for $[X,\ \ 0,\ \ 0]$

Default arguments are applied to `new_list()` so trailing NULL's can be omitted.

```
list new_list(list X = NULL, list Y = NULL, list Z = NULL); //default args
list x = new_list(X, Y);
```

(**Definition 1.4)** Numbers in LT_0 are defined to be particular lists:

$$\mathbf{0} =_{df} 0$$
$$\mathbf{1} =_{df} [0,\ \ 0,\ \ 0]$$
$$\mathbf{2} =_{df} [0,\ \ [0,\ \ 0,\ \ 0],\ \ 0] = [0,\ \ \mathbf{1},\ \ 0] = [0,\ \ \mathbf{1}]$$
$$\mathbf{3} =_{df} [0,\ \ [0,\ \ [0,\ \ 0,\ \ 0],\ \ 0],\ \ 0] = [0,\ \ \mathbf{2}]$$

…and in general

$$\mathbf{n} + \mathbf{1} =_{df} [0,\ \ \mathbf{n}]$$

Boolean values are defined as:

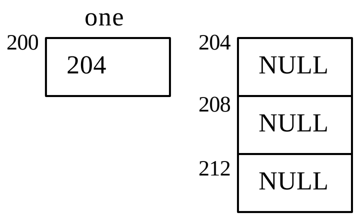
$$0 =_{df} \mathit{True}$$
$$\mathbf{1} =_{df} [0,\ \ 0,\ \ 0] =_{df} \mathit{False}$$

The numbers 0 … 7 and the Boolean values True and False, are used a lot in our programs. It is not space efficient if we have to create a completely new copy of these numbers every time they are used. We create 'constants' when our program starts, and copy only the pointer to a constant, rather than performing a deep copy.

A function `insert_list(list X, int i, list Y)` creates a list using a **shallow** copy of the list `Y` into the list `X` at either the head, body or tail, depending on the value of `i`.
Only the pointer is copied, not the complete list that it points to.

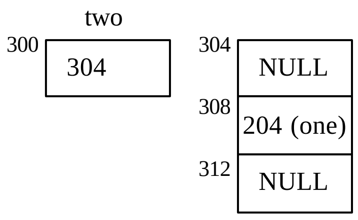
```
list one = new_list();
```

$\mathbf{1} =_{df} [0,\ \ 0,\ \ 0] \quad \rightarrow$



```
list two = insert_list(new_list(), 2, one);
```

$\mathbf{2} =_{df} [0,\ \ \mathbf{1},\ \ 0] \quad \rightarrow$



At the beginning of the program, we initialise the constants zero to seven, TRUE and FALSE.

```
list insert_list(list X, int i, list Y) {    // i is 1, 2 or 3
   X[i-1] = Y;                               // Shallow copy
   return X;
}
```

`insert_list`, like most of our functions, alters the value of its arguments (they are pointers). However, it is useful to also return a value. Most functions return the value of the list that is passed as the first argument.

Finally, here are some useful functions for dealing with lists:

```
list delete_list(list X) {        // release allocated memory
   if (X == NULL) return NULL;    // don't delete our constants!
   if ((X == zero) || (X == one) || (X == two) || (X == three)) return NULL;
   if ((X == four) || (X == five) || (X == six) || (X == seven)) return NULL;
   delete_list(HEAD(X));
   delete_list(BODY(X));
   delete_list(TAIL(X));
   free(X);
   return NULL;
}
list copy_list(list X) {          // deep copy of an existing list
   if (X == NULL) return NULL;
   return new_list(HEAD(X), BODY(X), TAIL(X));
}
int ltoi(list X) {                // convert a list into an integer
   int i;                         // for printing. If it is not a number
                                  // return -1
   if (X == NULL) return 0;
   if ((HEAD(X) == NULL) && (TAIL(X) == NULL)) {
      i = ltoi(BODY(X));     // recurse
      if (i == -1) return -1;     // always check for 'non-numbers'
      else return 1+i;
   }
   return -1;
}
void print_list(list X) {          // print a list in numeric form if
   int i;                          // possible e.g. 4
                                   // or in list form e.g. [0,0,[0]]
   if ((i=ltoi(X)) != -1) printf("%d",i);
   else {
      printf("[");
      print_list(HEAD(X));
      if (BODY(X) || TAIL(X)) {         // ignore trailing 0's
         printf(",");
         print_list(BODY(X));
         if (TAIL(X)) {
            printf(",");
            print_list(TAIL(X));
         }
      }
      printf("]");
   }
}
void print_listln(list X) {              // print list and then a new line
   print_list(X);
   printf("\n");
}
list itol(int n) {                       // convert an integer into a list
   switch (n) {                          // use our constants if possible
   case 0:return NULL;                   // this function will be used
   case 1:return one;                    // in input routines
   case 2:return two;
   case 3:return three;
   case 4:return four;
   case 5:return five;
   case 6:return six;
   case 7:return seven;
   }
   return insert_list(new_list(), 2, itol(n-1));
}
```

(Definition 2) Creating **functions** that can be applied to lists.

$x_1 \quad x_{21} \quad x_{31}$ are **basic** functions

E.g.

$$\begin{aligned} \textit{if} \quad x &= [0, \ [\mathbf{1}, \ 0, \ 0], \ 0] \quad \textit{then} \\ x_{21} &= [0, \ [\mathbf{1}, \ 0, \ 0], \ 0]_{21} \\ &= [\mathbf{1}, \ 0, \ 0]_1 \\ &= \mathbf{1} \\ &= [0, \ 0, \ 0] \end{aligned}$$

The basic functions implemented in C are:

```
list x1(list X) {
   return HEAD(X);
}

list x21(list X) {
   return HEAD(BODY(X));
}

list x31(list X) {
   return HEAD(TAIL(X));
}
```

The closure of the basic functions under the destructors are the **initial** functions.

E.g. $x_{1212321}$ is an initial function

or formally: $x_p\,, \quad p \in \{1,2,3\}^+$

The C implementation of the initial functions is `xnnn(list X, char *s)`:

```
list xnnn(list X, char *s) {  // s is a string of the form "1212321"
   int i = 0;
   while (s[i] != '\0') {
      X = destructor(X, s[i]-'0');
      i++;
   }
   return X;
}
```

(Definition 4) Creating **schemes** that can be applied to lists.

Complex functions are built from **schemes**.
Schemes are either **basic** or **compiled**.

The basic schemes are:

assignment: given $f(x)$ then substitute Y for x, $f(Y)$

equality: $f(x) \simeq g(x) =_{df} 0$ iff $f(x) = g(x)$ for some particular x

$f(x) \simeq g(x) =_{df} \mathbf{1}$ iff $f(x) \neq g(x)$ for some particular x

```
list eq(list X, list Y) { // deep test of each list
   int i;

   if ((X == NULL) && (Y == NULL)) return TRUE;
   if ((X == NULL) || (Y == NULL)) return FALSE;
   for (i=0;i<3;i++) {
      if ((X[i] == NULL) && (Y[i] != NULL)) return FALSE;
      if ((X[i] != NULL) && (Y[i] == NULL)) return FALSE;
      if ((X[i] != NULL) && (Y[i] != NULL) &&
       (eq((list) X[i], (list) Y[i]) != TRUE)) return FALSE;
   }
   return TRUE;
}
```

negation: $\mathit{not}\ f(x) =_{df} 0$ iff $f(x) \neq 0$ for some particular x

$\mathit{not}\ f(x) =_{df} \mathbf{1}$ iff $f(x) = 0$ for some particular x

```
list not(list X) {
   if (X == TRUE) return FALSE;
   else return TRUE;
}
```

disjunction: $f(x)\ \mathit{or}\ g(x) =_{df} 0$ iff $f(x) = 0$ or $g(x) = 0$ for some particular x

$f(x)\ \mathit{or}\ g(x) =_{df} \mathbf{1}$ iff $f(x) \neq 0$ and $g(x) \neq 0$ for some particular x

```
list or(list X, list Y) {
   if ((X == TRUE) || (Y == TRUE)) return TRUE;
   else return FALSE;
}
```

Schemes or functions can be thought of as "computer programs" It is important to realise that a particular function $f(x)$ may not produce an answer (is undefined) – i.e. the program never **halts**. These functions are called "Partial Functions".

(Definition 5) Creating **compiled schemes**.

For each Y, the constant function Y(x) is defined to be $x_1([Y, \quad 0, \quad 0])$.
When we write Y we refer to a function that produces Y as its result.

Compiled schemes are created from basic schemes:

$$f(x)\ \mathit{impl}\ g(x) =_{df} \mathit{not}\ f(x)\ \mathit{or}\ g(x)$$

```
list impl(list X, list Y) {
   return or(not(X), Y);
}
```

$$f(x)\ and\ g(x) =_{df}\ not\ \left(not\ f(x)\ or\ not\ g(x)\right)$$

```
list and(list X, list Y) {
   return not(or(not(X), not(Y)));
}
```

$$f(x)\ iff\ \ g(x) =_{df}\ \left(not\ f(x)\ or\ g(x)\right) and\ \left(not\ g(x)\ or\ f(x)\right)$$

```
list iff(list X, list Y){
   return and(or(not(X), Y), or(not(Y), X));
}
```

(Definition 6) Creating **recursive schemes**, schemes that are applied recursively to each element of a list.

If we are given a function $g(x)$ and an **initial** function $h(x)$ (so $h(x) = x_p$, $p \in \{1,2,3\}^+$), we can create a function $f(x) = R(g,h)(x)$.

R is a 'Boolean recursion scheme' in the 'step function' g and the 'substituted function' h.

R is given in terms of another function $f^*(x)$.

$$f^*(0) = 0$$

$$f^*(x) = g(x)\ and\ f^*(x_2)\ and\ f^*(x_3) \quad x \neq 0$$

$$f(x) = f^*(h(x))$$

where x_2 and x_3 are the destructors

(slight change to the original paper)

So if $h(x)$ is x_1 then

$$f(x) = f^*(x_1) = g(x_1)\ and\ f^*(x_{12})\ and\ f^*(x_{13})$$

```
list R(list X, list g(list Y)) {
   if (X == NULL) return NULL;
   else return and(g(X), and(R(g, BODY(X)), R(g, TAIL(X))));
}
```

The substituted function h is often set to be the identity function $h(x) = x$ (although the identity function does not itself belong to LT_0)

The above recursive scheme then simplifies to:

$$f(0) = 0$$

$$f(x) = g(x)\ and\ f(x_2)\ and\ f(x_3) \quad x \neq 0$$

This version of the recursive scheme is used in later sections.

(Example 1) The goal of this paper is to implement a complete (structured) Turing Machine. A Turing Machine needs a 'list representation' for both its paper tape and its sequence of instructions. This first example shows how to represent the sequence of binary numbers on the tape as a list.

Suppose we have a paper tape *T*, which contains binary numbers e.g. 011010111.
The binary code on the tape is represented as a list $\overline{T}$ in the following way:

$$\overline{0T} =_{df} [\mathbf{1},\quad \overline{T}]$$
$$\overline{1T} =_{df} [\mathbf{2},\quad \overline{T}]$$

If the tape is absent then $\overline{T}$ is 0.

E.g.	the tape 0	is coded as:	$\overline{0}$	=	$[\mathbf{1}]$
	the tape 101	is coded as:	$\overline{101}$	=	$[\mathbf{2},\quad [\mathbf{1},\quad [\mathbf{2}]]]$

A recursive scheme can be used to show whether a list is a valid representation of a tape.
Given an arbitrary list $x = [x_1,\quad x_2,\quad x_3]$, is it valid?

E.g.	is $[\mathbf{2},\quad [\mathbf{1},\quad [\mathbf{1},\quad \mathbf{2}]]]$	a valid tape? (no).
	is $[\mathbf{2},\quad [\mathbf{1},\quad [\mathbf{1}]]]$	a valid tape? (yes).

A code is valid if:
the first element of the list is either **1** or **2** (remember these are lists)
the head of the second element of the list is either **1** or **2** or 0
and the head of the third element is 0

and then apply the above conditions recursively to the second and third elements of the list.

$tp(x)$ is a (simplified) recursive scheme that returns TRUE if $\overline{T}$ is valid and FALSE otherwise.

$$tp(x) = g(x) \quad and \quad tp(x_2) \quad and \quad tp(x_3)$$

For tapes the step function $g(x)$ is:

$$g(x) = ((x_1 \simeq \mathbf{1}\ or\ x_1 \simeq \mathbf{2})\ and\ (x_{21} \simeq \mathbf{1}\ or\ x_{21} \simeq \mathbf{2}\ or\ x_{21})\ and\ x_{31})$$

Hence the recursive scheme $tp(x)$ is:

$$tp(x) = ((x_1 \simeq \mathbf{1}\ or\ x_1 \simeq \mathbf{2})\ and\ (x_{21} \simeq \mathbf{1}\ or\ x_{21} \simeq \mathbf{2}\ or\ x_{21})\ and\ x_{31}) \quad and \quad tp(x_2) \quad and \quad tp(x_3)$$

For the list $[\mathbf{2},\quad [\mathbf{1},\quad [\mathbf{2}]]]$:

$$\begin{aligned} tp([\mathbf{2},\quad [\mathbf{1},\quad [\mathbf{2}]]]) &= ((\mathbf{2} \simeq \mathbf{1}\ or\ \mathbf{2} \simeq \mathbf{2})\ and\ (\mathbf{1} \simeq \mathbf{1}\ or\ \mathbf{1} \simeq \mathbf{2}\ or\ \mathbf{1})\ and\ 0) \quad and \quad tp([\mathbf{1},\quad [\mathbf{2}]]) \quad and \quad tp(0) \\ &= 0 \quad and \quad ((\mathbf{1} \simeq \mathbf{1}\ or\ \mathbf{1} \simeq \mathbf{2})\ and\ (\mathbf{2} \simeq \mathbf{1}\ or\ \mathbf{2} \simeq \mathbf{2}\ or\ \mathbf{2})\ and\ \mathbf{0}) \quad and \quad tp(0) \\ &= 0 \quad and \quad 0 \quad and \quad tp(0) \\ &= 0 \quad and \quad 0 \quad and \quad 0 \\ &= 0 \\ &= TRUE \end{aligned}$$

```
list tp(list X) {  // recursive scheme to validate tapes
   list t1,t2,t3;

   if (X == NULL) return NULL;
   t1 = or(eq(x1(X),one), eq(x1(X),two));
   t2 = or(eq(x21(X),one), or(eq(x21(X),two),x21(X)));
   t3 = and(t1, and(t2,x31(X)));
   return and(t3, and(tp(BODY(X)), tp(TAIL(X))));
}
```

The following function `read_tape` allows a description of a tape to be input by the user.

The user types in a sequence of 0's and 1's. This sequence either ends with a #, in which case the # character is discarded, or it ends with another character (neither 0 or 1), in which case the character is put back into the input stream and can be read by another input function. All white spaces are ignored.

```
list read_tape(void) {
   int c;

   while (isspace(c = getchar())); //skip white space
   switch (c) {
   case '0':return insert_list(insert_list(new_list(),1,one),2,read_tape());
   case '1':return insert_list(insert_list(new_list(),1,two),2,read_tape());
   default:if ((c != '#') && (c != EOF)) ungetc(c, stdin);return NULL;
   }
}
```

(Definition 8)

$cons_{Y,Z}(x) =_{df} [Y, \quad x, \quad Z]$. This is implemented by the function `new_list()`.

(**Definition 9)**

The height of a list:

$ht(0) =_{df} 0$

$ht([X, \quad Y, \quad Z]) =_{df} 1 + \max(ht(X), ht(Y), ht(Z))$

```
int ht(list X) {
   int a,b,c,m;

   if (X == NULL) return 0;
   a = ht(HEAD(X));
   b = ht(BODY(X));
   c = ht(TAIL(X));
   m = a;
   if (b > m) m = b;
   if (c > m) m = c;
   return m;
}
```

The length of a list:

$|0| =_{df} 1$

$|[X, \quad Y, \quad Z]| =_{df} 1 + |X| + |Y| + |Z|$

```
int length(list X) {
   if (X == NULL) return 1;
   return 1 + length(HEAD(X)) + length(BODY(X)) + length(TAIL(X));
}
```

Turing Machines
In this section the description of a Turing Machine is completed, by representing the sequence of instructions as a list. 'Structured Turing Machines' [1] are used.

Elementary Turing Machines have as instructions:

l	move left
r	move right
w	write a 1
e	erase – write a 0

A general Turing Machine is expressed with the following grammar:

<TM> := l | r | w | e | <TM><TM> | (<TM>)

where *<TM><TM>* is the **'sequence** composition' of two Turing Machines, that is if M is a Turing Machine and $M = M_1M_2$, then M_1 is executed before M_2

If M is a Turing Machine then (M) is the **repetition** of M while the observed symbol is a 1. This is represented by *(<TM>)*.

Examples of Turing Machines are:

l
lw
llw
(lw)
(lw)rr

A Turing Machine M is coded by $\overline{M}$.

For **elementary** Turing Machines:

$M = l$	then	$\overline{M}$	$=_{df}$	$[\,\overline{l}\,]$	$=_{df}$	[**2**]
$M = r$	then	$\overline{M}$	$=_{df}$	$[\,\overline{r}\,]$	$=_{df}$	[**3**]
$M = w$	then	$\overline{M}$	$=_{df}$	$[\,\overline{w}\,]$	$=_{df}$	[**4**]
$M = e$	then	$\overline{M}$	$=_{df}$	$[\,\overline{e}\,]$	$=_{df}$	[**5**]

For **sequence** composition:

$M = M_1M_2$	then	$\overline{M}$	$=_{df}$	$[\,\overline{\circ}\,, \overline{M}_1, \overline{M}_2\,]$	$=_{df}$	[**7**, $\overline{M}_1$, $\overline{M}_2$]
e.g.						
if $M = lr$	then	$\overline{M}$	$=_{df}$	$[\,\overline{\circ}\,, [\,\overline{l}\,], [\,\overline{r}\,]\,]$	$=_{df}$	[**7**, [**2**], [**3**]]
if $M = lrr$	then	$\overline{M}$	$=_{df}$	$[\,\overline{\circ}\,, [\,\overline{l}\,], [\,\overline{\circ}\,, [\,\overline{r}\,], [\,\overline{r}\,]\,]\,]$	$=_{df}$	[**7**, [**2**], [**7**, [**3**], [**3**]]]

For **repetition**:

$M = (M_1)$	then	$\overline{M}$	$=_{df}$	$[\,\overline{while}\,, \overline{M}_1\,]$	$=_{df}$	[**6**, $\overline{M}_1$]
e.g.						
if $M = (l)$	then	$\overline{M}$	$=_{df}$	$[\,\overline{while}\,, [\,\overline{l}\,]\,]$	$=_{df}$	[**6**, [**2**]]
if $M = (lr)$	then	$\overline{M}$	$=_{df}$	$[\,\overline{while}\,, [\,\overline{\circ}\,, [\,\overline{l}\,], [\,\overline{r}\,]\,]\,]$	$=_{df}$	[**6**, [**7**, [**2**], [**3**]]]

(Notation 11)

1. Tapes consist of a sequence of 1's and 0's. Example 1 represents a tape as a list.

2. A Tape T consists of two parts T_1T_2. The observed symbol is the **left-most** bit of $\boldsymbol{T_2}$.
All other bits on the tape, outside of T_1T_2, will be 0.

3. The action of the Turing Machine is to perform the **left-most** instruction in the sequence of instructions M.

   We say that M is **placed over** T_1T_2.

   The current instruction is then removed from the sequence, and the action repeated.

   At any point in the execution of the Turing Machine, we can say that it is in a certain state. We call this an **instantaneous description** or **ID**.
   An ID is a triple T_1, M, T_2.

   If M is absent the ID is said to be **terminal**, i.e. the Turing Machine has halted.

4. An **atom** is in the form $T_1\ M\ T_2 \models T_1^*\ M^*\ T_2^*$
M over the input T_1T_2 yields the same output as M^* over the input $T_1^*T_2^*$
if M^* is absent then M over the input T_1T_2 yields $T_1^*T_2^*$.

We progress from one instance (ID) to another by executing an instruction.

The actions of the Turing Machine are fixed, and obey the following rules:

Rules:

1a:	T_1blMT_2	|=	T_1MbT_2
1b:	$blMT_2$	|=	$0MbT_2$
2a:	T_1rMbT_2	|=	T_1bMT_2
2b:	T_1rMb	|=	T_1bM0
3:	T_1wMbT_2	|=	T_1M1T_2
4:	T_1eMbT_2	|=	T_1M0T_2
5a:	$T_1(M_1)M1T_2$	|=	$T_1M_1(M_1)M1T_2$
5b:	$T_1(M_1)M0T_2$	|=	T_1M0T_2

Examples:

1a:	$11lr00$	|=	$1r100$
1b:	$1lr00$	|=	$0r100$
2a:	$11rl10$	|=	$111l0$
2b:	$11rl1$	|=	$111l0$
3:	$11wr000$	|=	$11r100$
4:	$11er100$	|=	$11r000$
5a:	$11(l)r100$	|=	$11l(l)r100$
5b:	$11(l)r000$	|=	$11r000$

An ID, T_1MT_2 is coded by $\overline{T_1MT_2}$ $=_{df}$ [$\overline{T_1}$, $\overline{M}$, $\overline{T_2}$]

Where $\overline{T_2}$ is the code for a tape (See Example 1).

$\overline{T_1}$ is the code for a tape read in **reverse order**. (The right-most bit is the first element in the list).

```
list reverse_tape(list X) {
   list p=NULL;
   while (X != NULL) {
      p = insert_list(new_list(HEAD(X)),2,p);
      X = BODY(X);
   }
   return p;
}
```

Examples of ID's are:

$01l01$	x_1	=	$[\,[\,\bar{1}, [\,\bar{0}\,]\,], [\,\bar{l}\,], [\,\bar{0}\,, [\,\bar{1}\,]\,]\,]$	$=_{df}$	[[**2**, [**1**]], [**2**], [**1**, [**2**]]]
T_1blMT_2	x_1	=	$[\,[\,\bar{b}\,,\ \overline{T_1}\,], [\,\bar{\circ}\,, [\,\bar{l}\,],\ \overline{M}\,],\ \overline{T_2}\,]$	$=_{df}$	$[\,[\,\bar{b}\,,\ \overline{T_1}\,],$ [**7**, [**2**], $\overline{M}$], $\overline{T_2}$]
T_1MbT_2	x_{21}	=	$[\ \overline{T_1}\,,\ \overline{M}\,, [\,\bar{b}\,,\ \overline{T_2}\,]\,]$	$=_{df}$	$[\overline{T_1}\,,\ \overline{M}\,, [\,\bar{b}\,,\ \overline{T_2}\,]\,]$

The definition of the function $id(x)$ that recognises ID's is:

$$id(x) \quad =_{df} \quad tm(x_2)\ and\ tp(x_1)\ and\ tp(x_2)$$

```
list id(list X) {
   return and(tm(BODY(X)), and(tp(HEAD(X)), tp(TAIL(X))));
}
```

The rules 1a-5b in the form I |= J will be coded by [$\bar{I}$, [$\bar{J}$, U], W]

The function $nx(x)$ defined below recognises atoms:

$$nx(x) \quad = \quad id(x_1)\, and\ id(x_{21})\, and\ (\, st_{1a}(x\ )\ or\ \ st_{1b}(x\ )\ or\ \ st_{2a}(x) \ldots or\ \ st_{5b}(x\ )\,)$$

```
list nx(list X) {
   list t1;

   t1 = or(st1a(X), or(st1b(X), or(st2a(X), or(st2b(X), or(st3(X), or(st4(X),
           or(st5a(X), st5b(X))))))));

   return and(id(x1(X)), and(id(x21(X)), t1));
}
```

Finally functions `st1a(X)` etc can be defined that recognise each of the rules:

Rule 1a: $T_1 blMT_2 \quad |= \quad T_1MbT_2$

$st_{1a}(x\ )$ is given by:

$x = [\,[\,[\,\bar{b}\,,\ \overline{T_1}\,],[\,\bar{\circ}\,,[\,\bar{l}\,],\ \overline{M}\,],\ \overline{T_2}\,],[\,[\,\overline{T_1}\,,\ \overline{M}\,,[\,\bar{b}\,,\ \overline{T_2}\,]\,],\mathrm{U}],\mathrm{W}]$

extracting terms:

$x_1 = [\,[\,\bar{b}\,,\ \overline{T_1}\,],[\,\bar{\circ}\,,[\,\bar{l}\,],\ \overline{M}\,],\ \overline{T_2}\,]$

 $x_{11} = [\bar{b}\,,\ \overline{T_1}\,]$

  $x_{111} = \bar{b}$

  $x_{112} = \overline{T_1}$

 $x_{12} = [\,\bar{\circ}\,,[\,\bar{l}\,],\ \overline{M}\,]$

  $x_{121} = \bar{\circ}$

  $x_{122} = [\,\bar{l}\,]$

   $x_{1221} = \bar{l}$

  $x_{123} = \overline{M}$

 $x_{13} = \overline{T_2}$

$x_2 = [\,[\,\overline{T_1}\,,\ \overline{M}\,,[\,\bar{b}\,,\ \overline{T_2}\,]\,],\mathrm{U}]$

 $x_{21} = [\,\overline{T_1}\,,\ \overline{M}\,,[\,\bar{b}\,,\ \overline{T_2}\,]\,]$

  $x_{211} = \overline{T_1}$

  $x_{212} = \overline{M}$

  $x_{213} = [\,\bar{b}\,,\ \overline{T_2}\,]$

   $x_{2131} = \bar{b}$

   $x_{2132} = \overline{T_2}$

 $x_{22} = \mathrm{U}$

$x_3 = \mathrm{W}$

To recognise this expression we have to ensure that each of the terms in x_1 and x_{21} are equivalent:

$st_{1a}(x\ )$	$=$	x_{111}	$\simeq$	x_{2131}	*and*	(equality of $\bar{b}$)
		x_{112}	$\simeq$	x_{211}	*and*	(equality of $\overline{T_1}$)
		x_{123}	$\simeq$	x_{212}	*and*	(equality of $\overline{M}$)
		x_{13}	$\simeq$	x_{2132}	*and*	(equality of $\overline{T_2}$)
		x_{121}	$\simeq$	$\bar{\circ}$	*and*	
		x_{1221}	$\simeq$	$\bar{l}$		

(which is slightly different than the formula in the paper).

However, this is not the complete recogniser for rule 1a. It may be the case that ***M* is absent**.

Rule 1a (revised): when M is absent:

$$x \quad = \quad [\,[\,[\,\bar{b}\,,\ \overline{T_1}\,],\ [\,\bar{l}\,],\ \overline{T_2}\,],\ [\,[\,\overline{T_1}\,,\ \mathrm{NULL}\,,\ [\,\bar{b}\,,\ \overline{T_2}\,]\,],\ U],\ W]$$

$\rightarrow$ x_{111}	=	$\bar{b}$		x_{211}	=	$\overline{T_1}$
x_{112}	=	$\overline{T_1}$		x_{212}	=	NULL
x_{121}	=	$\bar{l}$		x_{213}	=	$[\bar{b}\,,\ \overline{T_2}\,]$
x_{13}	=	$\overline{T_2}$		x_{2131}	=	$\bar{b}$
				x_{2132}	=	$\overline{T_2}$

The final recogniser for rule 1a is:

$st_{1a}(x\ )$	=	x_{111}	$\simeq$	x_{2131}	*and*	(equality of $\bar{b}$)
		x_{112}	$\simeq$	x_{211}	*and*	(equality of $\overline{T_1}$)
		x_{13}	$\simeq$	x_{2132}	*and*	(equality of $\overline{T_2}$)
		$(\,x_{123}$	$\simeq$	x_{212}	*and*	(equality of $\overline{M}$)
		x_{121}	$\simeq$	$\bar{\circ}$	*and*	
		x_{1221}	$\simeq$	$\bar{l}\ )$		
		or				
		$(\,x_{121}$	$\simeq$	$\bar{l}$	*and*	$x_{212}\ )$

```
list st1a(list X) {
   list t1,t2,t3,t4,t5,t6;

   t1 = eq(xnnn(X,"111"),xnnn(X,"2131"));
   t2 = eq(xnnn(X,"112"),xnnn(X,"211"));
   t3 = eq(xnnn(X,"13"),xnnn(X,"2132"));
   t4 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),left);
   t5 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),left);
   t2 = xnnn(X,"212");
   t6 = and(t1, t2);

   return and(t4, or(t5,t6));
}
```

Rule 1b: $blMT_2 \quad |= \quad 0MbT_2$

$$x \quad = \quad [\,[\,[\,\bar{b}\,], [\,\bar{\circ}\,, [\,\bar{l}\,], \;\overline{M}\,], \;\overline{T_2}\,], [\,[\,[\,\bar{0}\,], \;\overline{M}\,, [\,\bar{b}\,, \;\overline{T_2}\,]\,], \mathrm{U}], \mathrm{W}]$$

$$\begin{array}{lllllll}
\rightarrow & x_{11} & = & [\,\bar{b}\,] & x_{211} & = & [\,\bar{0}\,] \\
 & x_{121} & = & \bar{\circ} & x_{212} & = & \overline{M} \\
 & x_{121} & = & [\,\bar{l}\,] & x_{2131} & = & \bar{b} \\
 & x_{123} & = & \overline{M} & x_{2132} & = & \overline{T_2} \\
 & x_{13} & = & \overline{T_2} & & &
\end{array}$$

when *M* is absent:

$$x \quad = \quad [\,[\,[\,\bar{b}\,], [\,\bar{l}\,], \;\overline{T_2}\,], [\,[[\,\bar{0}\,], \mathrm{NULL}\,, [\,\bar{b}\,, \;\overline{T_2}\,]\,], \mathrm{U}], \mathrm{W}]$$

$$\begin{array}{lllllll}
\rightarrow & x_{11} & = & [\,\bar{b}\,] & x_{211} & = & [\,\bar{0}\,] \\
 & x_{12} & = & [\,\bar{l}\,] & x_{212} & = & \mathrm{NULL} \\
 & x_{13} & = & \overline{T_2} & x_{2131} & = & \bar{b} \\
 & & & & x_{2132} & = & \overline{T_2}
\end{array}$$

$$\boxed{\begin{array}{llllllll}
st_{1b}(x\;) & = & x_{111} & \simeq & x_{2131} & \textit{and} & & [\,\bar{b}\,, 0] \\
 & & x_{112} & & & \textit{and} & & \\
 & & x_{2111} & \simeq & \mathbf{1} & \textit{and} & & (\bar{0} \;=_{\mathrm{df}} \mathbf{1}) \\
 & & x_{13} & \simeq & x_{2132} & \textit{and} & & (\,\overline{T_2}\,) \\
 & & & & & & & \\
 & & (\,x_{123} & \simeq & x_{212} & \textit{and} & & (\,\overline{M}\,) \\
 & & x_{121} & \simeq & \bar{\circ} & \textit{and} & & \\
 & & x_{1221} & \simeq & \bar{l}\;) & & & \\
 & & & & & & & \\
 & & \textit{or} & & & & & \\
 & & & & & & & \\
 & & (\,x_{121} & \simeq & \bar{l} & \textit{and} & x_{212}\,) &
\end{array}}$$

```
list st1b(list X) {
   list t1,t2,t3,t4,t5,t6,t7;
   t1 = eq(xnnn(X,"111"),xnnn(X,"2131"));
   t2 = eq(xnnn(X,"2111"),one);
   t3 = eq(xnnn(X,"13"),xnnn(X,"2132"));
   t4 = xnnn(X,"112");
   t5 = and(t1, and(t2, and(t3, t4)));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),left);
   t6 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),left);
   t2 = xnnn(X,"212");
   t7 = and(t1, t2);

   return and(t5, or(t6,t7));
}
```

Rule 2a: $T_1rMbT_2 \quad \models \quad T_1bMT_2$

$$x = [\,[\,\overline{T_1}, [\,\bar{\circ}, [\,\bar{r}\,], \overline{M}\,], [\,\bar{b}, \overline{T_2}\,]\,], [\,[\,[\,\bar{b}, \overline{T_1}\,], \overline{M}, \overline{T_2}\,], U], W]$$

$$\rightarrow \quad \begin{array}{lll lll}
x_{11} & = & \overline{T_1} & x_{2111} & = & \bar{b} \\
x_{121} & = & \bar{\circ} & x_{2112} & = & \overline{T_1} \\
x_{122} & = & [\,\bar{r}\,] & x_{212} & = & \overline{M} \\
x_{123} & = & \overline{M} & x_{213} & = & \overline{T_2} \\
x_{131} & = & \bar{b} & & & \\
x_{132} & = & \overline{T_2} & & &
\end{array}$$

when M is absent:

$$x = [\,[\,\overline{T_1}, [\,\bar{r}\,], [\,\bar{b}, \overline{T_2}\,]\,], [\,[\,[\,\bar{b}, \overline{T_1}\,], \text{NULL}, \overline{T_2}\,], U], W]$$

$$\rightarrow \quad \begin{array}{lll lll}
x_{11} & = & \overline{T_1} & x_{2111} & = & \bar{b} \\
x_{12} & = & [\,\bar{r}\,] & x_{2112} & = & \overline{T_1} \\
x_{131} & = & \bar{b} & x_{212} & = & \text{NULL} \\
x_{132} & = & \overline{T_2} & x_{213} & = & \overline{T_2}
\end{array}$$

$$\begin{array}{lllllll}
st_{2a}(x) & = & x_{131} & \simeq & x_{2111} & \mathit{and} & \text{(equality of } \bar{b}\text{)} \\
 & & x_{11} & \simeq & x_{2112} & \mathit{and} & \text{(equality of } \overline{T_1}\text{)} \\
 & & x_{132} & \simeq & x_{213} & \mathit{and} & \text{(equality of } \overline{T_2}\text{)} \\
 & & & & & & \\
 & & (x_{123} & \simeq & x_{212} & \mathit{and} & \text{(equality of } \overline{M}\text{)} \\
 & & x_{121} & \simeq & \bar{\circ} & \mathit{and} & \\
 & & x_{1221} & \simeq & \bar{r}\;) & & \\
 & & & & & & \\
 & & \mathit{or} & & & & \\
 & & & & & & \\
 & & (x_{121} & \simeq & \bar{r} & \mathit{and} & x_{212}\;)
\end{array}$$

```
list st2a(list X) {
   list t1,t2,t3,t4,t5,t6;

   t1 = eq(xnnn(X,"131"),xnnn(X,"2111"));
   t2 = eq(xnnn(X,"11"),xnnn(X,"2112"));
   t3 = eq(xnnn(X,"132"),xnnn(X,"213"));
   t4 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),right);
   t5 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),right);
   t2 = xnnn(X,"212");
   t6 = and(t1, t2);

   return and(t4, or(t5,t6));
}
```

Rule 2b: $T_1 rMb$ $\models$ $T_1 bM0$

$$x = [\,[\,\overline{T_1}, [\,\overline{\circ}, [\,\overline{r}\,], \overline{M}\,], [\,\overline{b}\,]\,], [\,[\,[\,\overline{b}, \overline{T_1}\,], \overline{M}, [\,\overline{0}\,]\,], U], W]$$

$\rightarrow$ x_{11}	=	$\overline{T_1}$	x_{2111}	=	$\overline{b}$
x_{121}	=	$\overline{\circ}$	x_{2112}	=	$\overline{T_1}$
x_{122}	=	$[\,\overline{r}\,]$	x_{212}	=	$\overline{M}$
x_{123}	=	$\overline{M}$	x_{213}	=	$[\,\overline{0}\,]$
x_{13}	=	$[\,\overline{b}\,]$			

when M is absent:

$$x = [\,[\,\overline{T_1}, \overline{r}, [\,\overline{b}\,]\,], [\,[\,[\,\overline{b}, \overline{T_1}\,], \text{NULL}, [\,\overline{0}\,]\,], U], W]$$

$\rightarrow$ x_{11}	=	$\overline{T_1}$	x_{2111}	=	$\overline{b}$
x_{12}	=	$[\,\overline{r}\,]$	x_{2112}	=	$\overline{T_1}$
x_{13}	=	$[\,\overline{b}\,]$	x_{212}	=	NULL
			x_{213}	=	$[\,\overline{0}\,]$

$st_{2b}(x\ )$	=	x_{131}	$\simeq$	x_{2111}	*and*		(equality of $[\,\overline{b}$, 0])
		x_{132}			*and*		
		x_{11}	$\simeq$	x_{2112}	*and*		(equality of $\overline{T_1}$)
		x_{2131}	$\simeq$	**1**	*and*		($\overline{0}$ $=_{df}$ **1**)
		$(x_{123}$	$\simeq$	x_{212}	*and*		(equality of $\overline{M}$)
		x_{121}	$\simeq$	$\overline{\circ}$	*and*		
		x_{1221}	$\simeq$	$\overline{r}$)			
		or					
		$(x_{121}$	$\simeq$	$\overline{r}$	*and*	x_{212})	

```
list st2b(list X) {
   list t1,t2,t3,t4,t5,t6,t7;

   t1 = eq(xnnn(X,"131"),xnnn(X,"2111"));
   t2 = xnnn(X,"132");
   t3 = eq(xnnn(X,"11"),xnnn(X,"2112"));
   t4 = eq(xnnn(X,"2131"),one);
   t5 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"), seq);
   t3 = eq(xnnn(X,"1221"),right);
   t6 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),right);
   t2 = xnnn(X,"212");
   t7 = and(t1, t2);

   return and(t5, or(t6,t7));
}
```

Rule 3: $T_1wMbT_2 \qquad \models \qquad T_1M1T_2$

$$x = [\,[\,\overline{T_1}\,,[\,\overline{\circ}\,,[\,\overline{w}\,],\ \overline{M}\,],[\,\bar{b}\,,\ \overline{T_2}\,]\,],[\ \overline{T_1}\,,\ \overline{M}\,,[\bar{1},\ \overline{T_2}\,]\,],\mathrm{U}],\mathrm{W}]$$

$$\rightarrow \quad \begin{array}{lll@{\qquad}lll}
x_{11} & = & \overline{T_1} & x_{211} & = & \overline{T_1} \\
x_{121} & = & \overline{\circ} & x_{212} & = & \overline{M} \\
x_{122} & = & [\,\overline{w}\,] & x_{2131} & = & \bar{1} \\
x_{123} & = & \overline{M} & x_{2132} & = & \overline{T_2} \\
x_{131} & = & \bar{b} & & & \\
x_{132} & = & \overline{T_2} & & &
\end{array}$$

when M is absent:

$$x = [\,[\,\overline{T_1}\,,[\,\overline{w}\,],[\,\bar{b}\,,\ \overline{T_2}\,]\,],[\ \overline{T_1}\,,\mathrm{NULL}\,,[\bar{1},\ \overline{T_2}\,]\,],\mathrm{U}],\mathrm{W}]$$

$$\rightarrow \quad \begin{array}{lll@{\qquad}lll}
x_{11} & = & \overline{T_1} & x_{2112} & = & \overline{T_1} \\
x_{12} & = & [\,\overline{w}\,] & x_{212} & = & \mathrm{NULL} \\
x_{131} & = & \bar{b} & x_{2131} & = & \bar{1} \\
x_{132} & = & \overline{T_2} & x_{2132} & = & \overline{T_2}
\end{array}$$

$$\boxed{\begin{array}{llllll}
st_3(x\ ) = & x_{11} & \simeq & x_{211} & \mathit{and} & (\text{equality of } \overline{T_1}\,) \\
 & x_{132} & \simeq & x_{2132} & \mathit{and} & (\text{equality of } \overline{T_2}\,) \\
 & x_{2131} & \simeq & \mathbf{2} & \mathit{and} & (\bar{1} =_{\mathrm{df}} \mathbf{2}) \\
 & & & & & \\
 & (\,x_{123} & \simeq & x_{212} & \mathit{and} & (\text{equality of } \overline{M}\,) \\
 & x_{121} & \simeq & \overline{\circ} & \mathit{and} & \\
 & x_{1221} & \simeq & \overline{w}\ ) & & \\
 & & & & & \\
 & \mathit{or} & & & & \\
 & & & & & \\
 & (\,x_{121} & \simeq & \overline{w} & \mathit{and} & x_{212}\ )
\end{array}}$$

```
list st3(list X) {
   list t1,t2,t3,t4,t5,t6;

   t1 = eq(xnnn(X,"11"),xnnn(X,"211"));
   t2 = eq(xnnn(X,"132"),xnnn(X,"2132"));
   t3 = eq(xnnn(X,"2131"),two);
   t4 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),write);
   t5 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),write);
   t2 = xnnn(X,"212");
   t6 = and(t1, t2);

   return and(t4, or(t5,t6));
}
```

Rule 4: T_1eMbT_2 |= T_1M0T_2

$$x = [\,[\,\overline{T_1}\,,[\,\bar{\circ}\,,[\,\bar{e}\,],\ \overline{M}\,],[\,\bar{b}\,,\ \overline{T_2}\,]\,],[\,[\,\overline{T_1}\,,\ \overline{M}\,,[\,\bar{0}\,,\ \overline{T_2}\,]\,],\mathrm{U}],\mathrm{W}]$$

$$\begin{array}{lllllll}
\rightarrow & x_{11} & = & \overline{T_1} & x_{211} & = & \overline{T_1} \\
 & x_{121} & = & \bar{\circ} & x_{212} & = & \overline{M} \\
 & x_{122} & = & [\,\bar{e}\,] & x_{2131} & = & \bar{0} \\
 & x_{123} & = & \overline{M} & x_{2132} & = & \overline{T_2} \\
 & x_{131} & = & \bar{b} & & & \\
 & x_{132} & = & \overline{T_2} & & &
\end{array}$$

when M is absent:

$$x = [\,[\,\overline{T_1}\,,[\,\bar{e}\,],[\,\bar{b}\,,\ \overline{T_2}\,]\,],[\,[\,\overline{T_1}\,,\mathrm{NULL}\,,[\,\bar{0}\,,\ \overline{T_2}\,]\,],\mathrm{U}],\mathrm{W}]$$

$$\begin{array}{lllllll}
\rightarrow & x_{11} & = & \overline{T_1} & x_{211} & = & \overline{T_1} \\
 & x_{12} & = & [\,\bar{e}\,] & x_{212} & = & \mathrm{NULL} \\
 & x_{131} & = & \bar{b} & x_{2131} & = & \bar{0} \\
 & x_{132} & = & \overline{T_2} & x_{2132} & = & \overline{T_2}
\end{array}$$

$$\boxed{\begin{array}{lllllll}
st_4(x\ ) & = & x_{11} & \simeq & x_{211} & \mathit{and} & \text{(equality of } \overline{T_1}\text{)} \\
 & & x_{132} & \simeq & x_{2132} & \mathit{and} & \text{(equality of } \overline{T_2}\text{)} \\
 & & x_{2131} & \simeq & \mathbf{1} & \mathit{and} & (\bar{0}\ =_{\mathrm{df}} \mathbf{1}) \\
 & & (\,x_{123} & \simeq & x_{212} & \mathit{and} & \text{(equality of } \overline{M}\text{)} \\
 & & x_{121} & \simeq & \bar{\circ} & \mathit{and} & \\
 & & x_{1221} & \simeq & \bar{e}\ ) & & \\
 & & \mathit{or} & & & & \\
 & & (\,x_{121} & \simeq & \bar{e} & \mathit{and} & x_{212}\ )
\end{array}}$$

```
list st4(list X) {
   list t1,t2,t3,t4,t5,t6;

   t1 = eq(xnnn(X,"11"),xnnn(X,"211"));
   t2 = eq(xnnn(X,"132"),xnnn(X,"2132"));
   t3 = eq(xnnn(X,"2131"),one);
   t4 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),erase);
   t5 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),erase);
   t2 = xnnn(X,"212");
   t6 = and(t1, t2);

   return and(t4, or(t5,t6));
}
```

Rule 5a: $T_1(M_1)M1T_2 \quad |= \quad T_1M_1(M_1)M1T_2$

$x = [\,[\,\overline{T_1}, [\,\bar{\circ}, [\,\overline{while}, \overline{M_1}\,], \overline{M}\,], [\,\bar{1}, \overline{T_2}\,]], [\,[\,\overline{T_1}, [\,\bar{\circ}, \overline{M_1}, [\,\bar{\circ}, [\,\overline{while}, \overline{M_1}\,], \overline{M}\,]\,], [\,\bar{1}, \overline{T_2}\,]], U], W]$

→	x_{11}	=	$\overline{T_1}$	x_{211}	=	$\overline{T_1}$
	x_{121}	=	$\bar{\circ}$	x_{2121}	=	$\bar{\circ}$
	x_{1221}	=	$\overline{while}$	x_{2122}	=	$\overline{M_1}$
	x_{1222}	=	$\overline{M_1}$	x_{21231}	=	$\bar{\circ}$
	x_{123}	=	$\overline{M}$	x_{212321}	=	$\overline{while}$
	x_{131}	=	$\bar{1}$	x_{212322}	=	$\overline{M_1}$
	x_{132}	=	$\overline{T_2}$	x_{21233}	=	$\overline{M}$
				x_{2131}	=	$\bar{1}$
				x_{2132}	=	$\overline{T_2}$

when M is absent:

$x = [\,[\,\overline{T_1}, [\,\overline{while}, \overline{M_1}\,], [\,\bar{1}, \overline{T_2}\,]\,], [\,[\,\overline{T_1}, [\,\bar{\circ}, \overline{M_1}, [\,\overline{while}, \overline{M_1}\,]\,], [\,\bar{1}, \overline{T_2}\,]\,], U], W]$

→	x_{11}	=	$\overline{T_1}$	x_{211}	=	$\overline{T_1}$
	x_{121}	=	$\overline{while}$	x_{2121}	=	$\bar{\circ}$
	x_{122}	=	$\overline{M_1}$	x_{2122}	=	$\overline{M_1}$
	x_{131}	=	$\bar{1}$	x_{21231}	=	$\overline{while}$
	x_{132}	=	$\overline{T_2}$	x_{21232}	=	$\overline{M_1}$
				x_{2131}	=	$\bar{1}$
				x_{2132}	=	$\overline{T_2}$

$$\boxed{\begin{array}{lllllllll}
st_{5a}(x) & = & x_{131} & \simeq & x_{2131} & \simeq & \mathbf{2} & and & (\bar{1} =_{df} \mathbf{2}) \\
 & & x_{11} & \simeq & x_{211} & and & & & (\text{equality of } \overline{T_1}) \\
 & & x_{132} & \simeq & x_{2132} & and & & & (\text{equality of } \overline{T_2}) \\
 & & (x_{123} & \simeq & x_{21233} & and & & & (\text{equality of } \overline{M}) \\
 & & x_{1222} & \simeq & x_{2122} & \simeq & x_{212322} & & (\text{equality of } \overline{M_1}) \\
 & & x_{121} & \simeq & x_{2121} & \simeq & x_{21231} & \simeq & \bar{\circ} \quad and \\
 & & x_{1221} & \simeq & x_{212321} & \simeq & \overline{while}\ ) & & \\
 & & or & & & & & & \\
 & & (x_{122} & \simeq & x_{2122} & \simeq & x_{21232} & and & (\text{equality of } \overline{M_1}) \\
 & & x_{121} & \simeq & x_{21231} & \simeq & \overline{while} & and & \\
 & & x_{2121} & \simeq & \bar{\circ} & & & & \\
 & & x_{212}\ ) & & & & & &
\end{array}}$$

```
list st5a(list X) {
   list t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11;

   t1 = eq(xnnn(X,"131"),xnnn(X,"2131"));
   t2 = eq(xnnn(X,"131"),two);
   t3 = eq(xnnn(X,"11"),xnnn(X,"211"));
   t4 = eq(xnnn(X,"132"),xnnn(X,"2132"));
   t9 = and(t1, and(t2, and(t3, t4)));

   t1 = eq(xnnn(X,"123"),xnnn(X,"21233"));
   t2 = eq(xnnn(X,"1222"),xnnn(X,"2122"));
   t3 = eq(xnnn(X,"1222"),xnnn(X,"212322"));
   t4 = eq(xnnn(X,"121"),xnnn(X,"2121"));
   t5 = eq(xnnn(X,"121"),xnnn(X,"21231"));
   t6 = eq(xnnn(X,"121"),seq);
   t7 = eq(xnnn(X,"1221"),xnnn(X,"212321"));
   t8 = eq(xnnn(X,"1221"),repeat);
   t10 = and(t1, and(t2, and(t3, and(t4, and(t5, and(t6, and(t7, t8)))))));

   t1 = eq(xnnn(X,"122"),xnnn(X,"2122"));
   t2 = eq(xnnn(X,"122"),xnnn(X,"21232"));
   t3 = eq(xnnn(X,"121"),xnnn(X,"21231"));
   t4 = eq(xnnn(X,"121"),repeat);
   t5 = eq(xnnn(X,"2121"),seq);
   t11 = and(t1, and(t2, and(t3, and(t4,t5))));

   return and(t9, or(t10,t11));
}
```

Rule 5b: $T_1(M_1)M0T_2$ |= T_1M0T_2

$x = [\,[\,\overline{T_1}, [\,\overline{\circ}, [\,\overline{while}, \overline{M_1}\,], \overline{M}\,], [\,\overline{0}, \overline{T_2}\,]\,], [\,[\,\overline{T_1}, \overline{M}, [\,\overline{0}, \overline{T_2}\,]\,], U], W]$

→	x_{11}	=	$\overline{T_1}$	x_{211}	=	$\overline{T_1}$
	x_{121}	=	$\overline{\circ}$	x_{212}	=	$\overline{M}$
	x_{1221}	=	$\overline{while}$	x_{2131}	=	$\overline{0}$
	x_{1222}	=	$\overline{M_1}$	x_{2132}	=	$\overline{T_2}$
	x_{123}	=	$\overline{M}$			
	x_{131}	=	$\overline{0}$			
	x_{132}	=	$\overline{T_2}$			

when *M* is absent:

$x = [\,[\,\overline{T_1}, [\,\overline{while}, \overline{M_1}\,], [\,\overline{0}, \overline{T_2}\,]\,], [\,[\,\overline{T_1}, \text{NULL}, [\,\overline{0}, \overline{T_2}\,]\,], U], W]$

→	x_{11}	=	$\overline{T_1}$	x_{211}	=	$\overline{T_1}$
	x_{121}	=	$\overline{while}$	x_{212}	=	NULL
	x_{122}	=	$\overline{M_1}$	x_{2131}	=	$\overline{0}$
	x_{131}	=	$\overline{0}$	x_{2132}	=	$\overline{T_2}$
	x_{132}	=	$\overline{T_2}$			

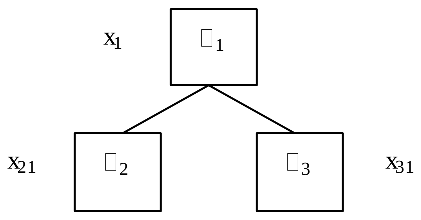
$st_{5b}(x)$	=	x_{131}	$\simeq$	x_{2131}	$\simeq$	**1**	*and*	$(\overline{0} =_{\text{df}} \mathbf{1})$
		x_{11}	$\simeq$	x_{211}	*and*			(equality of $\overline{T_1}$)
		x_{132}	$\simeq$	x_{2132}	*and*			(equality of $\overline{T_2}$)
		$(x_{123}$	$\simeq$	x_{212}	*and*			(equality of $\overline{M}$)
		x_{121}	$\simeq$	$\overline{\circ}$	*and*			
		x_{1221}	$\simeq$	$\overline{while}\,)$				
		or						
		$(x_{121}$	$\simeq$	$\overline{while}$	*and*	$x_{212}\,)$		

```
list st5b(list X) {
   list t1,t2,t3,t4,t5,t6,t7;

   t1 = eq(xnnn(X,"131"),xnnn(X,"2131"));
   t2 = eq(xnnn(X,"131"),one);
   t3 = eq(xnnn(X,"11"),xnnn(X,"211"));
   t4 = eq(xnnn(X,"132"),xnnn(X,"2132"));
   t5 = and(t1, and(t2, and(t3, t4)));

   t1 = eq(xnnn(X,"123"),xnnn(X,"212"));
   t2 = eq(xnnn(X,"121"),seq);
   t3 = eq(xnnn(X,"1221"),repeat);
   t6 = and(t1, and(t2, t3));

   t1 = eq(xnnn(X,"121"),repeat);
   t2 = xnnn(X,"212");
   t7 = and(t1, t2);

   return and(t5, or(t6,t7));
}
```

```
list read_tm(void) {               //function to read a TM input by the user
   int c,d;
   static int parenthesis_count = 0;
   list tm = NULL;

   while (isspace(c = getchar())); //skip white space
   if (strchr("lrwe()",c) == NULL) {
      if ((c != '#') && (c != EOF)) ungetc(c,stdin);
      return NULL;                 // no tape at all!
   }
   while (isspace(d = getchar())); //see what the next character is
   if (c == '(') {
      ungetc(d,stdin);
      parenthesis_count++;
      tm = insert_list(insert_list(new_list(), 1, repeat), 2, read_tm());
      c = getchar();               // get the trailing ')'
      while (isspace(d = getchar()));
   }
   if (c == ')') {
      parenthesis_count--;
   }
   if (strchr("lrwe(",d) == NULL) {
      if ((d != '#') && (d != -1)) ungetc(d,stdin); // gobble up a #
      if (d == ')') {
         if (parenthesis_count <= 0) {
            printf("Error mis-matched parentheses\n");
            exit(1);
         }
      } else if (parenthesis_count != 0) {
         printf("Error mis-matched parentheses\n");
         exit(1);
      }
      switch (c) {
      case 'l':return insert_list(new_list(),1,left);
      case 'r':return insert_list(new_list(),1,right);
      case 'w':return insert_list(new_list(),1,write);
      case 'e':return insert_list(new_list(),1,erase);
      case ')':return tm;
      default:
         printf("Error in input tm format(1) c=%c(%d)\n",c,c);
         exit(1);
      }
   } else {
      list t = insert_list(new_list(), 1, seq);
      ungetc(d,stdin);
      switch (c) {
      case 'l':insert_list(t, 2, insert_list(new_list(),1,left));break;
      case 'r':insert_list(t, 2, insert_list(new_list(),1,right));break;
      case 'w':insert_list(t, 2, insert_list(new_list(),1,write));break;
      case 'e':insert_list(t, 2, insert_list(new_list(),1,erase));break;
      case ')':insert_list(t, 2, tm);break;
      default:
         printf("Error in input tm format(2) c=%c(%d)\n",c,c);
         exit(1);
      }
      return insert_list(t, 3, read_tm());
   }
}

list read_id(void) {               //function to read an ID input by the user
   list ID = new_list();

   ID = insert_list(ID, 1, reverse_tape(read_tape()));
   ID = insert_list(ID, 2, read_tm());
   ID = insert_list(ID, 3, read_tape());
   print_listln(ID);
   print_listln(id(ID));
   return ID;
}
```

Lists of the form [$□_1$, [$□_2$], [$□_3$]] can represent binary trees.

[root node, [left sub-tree node], [right sub-tree node]]

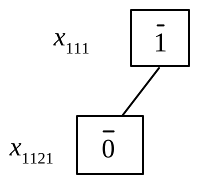


(which is one reason for choosing x_1, x_{21}, x_{31} as the basic functions).

Tapes and Turing Machines can be displayed in this binary tree form e.g.

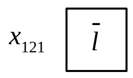
Tapes:

$\overline{10}$ -> x_{11} = $[\bar{1}, [\bar{0}]]$

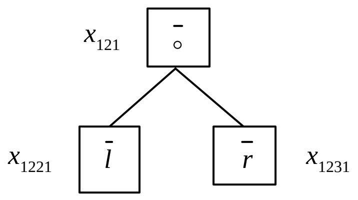


Turing Machines:

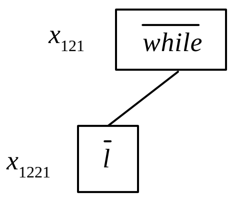
l -> x_{12} = $[\bar{l}]$



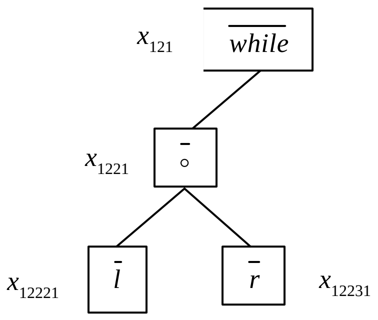
lr -> x_{12} = $[\bar{\circ}, [\bar{l}], [\bar{r}]]$



(*l*) -> x_{12} = $[\overline{while}, [\bar{l}]]$



(*lr*) -> x_{12} = $[\overline{while}, [\bar{\circ}, [\bar{l}], [\bar{r}]]]$



Conclusion

This report covers the C/C++ implementation of the first two sections of the paper [1]. The implementation is at a stage where, Turing Machines and Tapes may be input and verified, and Instantaneous Descriptions of the execution history of a Turing Machine checked for correctness.

Future reports will detail the implementation of the later sections of the paper, and further applications of this system.

References

[1] Caparosa, A., Pani, G. and Covino, E. Incompleteness in Linear Time, *Private communication*.

[2] Chaitin, G. J. Algorithmic Information Theory, *Cambridge University Press*, 1987

[3] Chaitin, G. J. Randomness in arithmetic and the decline and fall of reductionism in pure mathematics, *Bulletin of the European Association for Theoretical Computer Science*, No **50**, pp. 314-328, 1993

[4] Gödel, K. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, *Monatshefte für Mathematik und Physik* **38**, 1931
Gödel, K. On formally undecidable propositions of Principia Mathematica and related systems, *Trans. B. Meltzer. New York: Basic Books Inc.*, 1962.

[5] Turing, A. M. On computable numbers, with an application to the Entscheidungs problem, *Proc. Lond. Math. Soc.* (ser. 2), **42**, pp. 230-265, 1937