

# **The pLucid Programming Manual**

**By**

**A.A.Faustini**

**S.G. Matthews**

**A.AG. Yaghi**

**SEPTEMBER 1984**

Copies of this manual and a pLucid distribution tape are available from A. Faustini, Department of Computer Science, Arizona State University, Tempe 85287, Arizona, USA (Please send a tape with any request)

© A. Faustini 1983

**The pLucid Programmer's manual**

*A.A. Faustini*  
*S G Matthews (\*)*  
*A AG Yaghi (\*\*)*

Department of Computer Science

Arizona State University  
Tempe 85287  
Arizona  
USA

### ABSTRACT

!pLucid (pronounced "pellucid") is a member of the Lucid family of functional dataflow programming languages. A program in !pLucid defines a network of continuously operating autonomous processing stations (or filters). Computation in this network is controlled by the flow of data along arcs that interconnect the nodes, thus a !pLucid program defines a dataflow net. At the outermost level a !pLucid program is an expression that defines a functional relationship between the sequence of data values that correspond to the program's entire input activity and the sequence of values that corresponds to the program's entire output activity. Statements in !pLucid are equations, the left hand side of each equation defining the output of the functional filter defined by the expression on the right hand side of the equation. Thus the !pLucid programmer writes programs in terms of filters and streams. The data values that make up streams are similar to those of Pop2, namely finite lists, strings, words, reals and integers.

The current implementation of !pLucid runs under Berkley UNIX<sup>†</sup> 4.1bsd on a VAX 11/780. The !plucid evaluator simulates a Lucid dataflow machine and consequently !pLucid programs do not run as efficiently on the Vax as those written in more conventional von Neumann languages. On the other hand pLucid programs can be developed and debugged quickly and are much easier to maintain than programs written in conventional languages.

---

(\*) Department of Computer Science, University of Victoria,  
P.O. Box 1700, Victoria BC, Canada V8W 2Y2.

(\*\*) Department of Computer Science, University of Warwick,  
Coventry CV4 7AL, England.

<sup>†</sup> UNIX is a registered trademark of The Open Group in the U.S. and other countries.

## CONTENTS

Abstract

Contents

1 Introduction

2 Lucid Expressions

2.1 The @where clause

2.2 The operators @next, @fby and @first

2.3 User Defined Functions

2.4 The operators @asa, @whenever and @upon

2.5 The @@is current& declaration

3 p-Lucid Data Types

3.1 Numeric Expressions

3.2 Non-Numeric Data Processing

3.2.1 Word Processing

3.2.2 Boolean Expressions and Predefined Variables

3.2.3 String Processing

3.2.4 List Processing

3.3 The objects @eod and @error

3.4 pLucid Conditional Expressions

4 Scope Rules

5 Running pLucid under UNIX

5.1 The Basics

5.2 The @filter and @arg operators

5.3 Runtime errors

5.4 The @include facility

5.5 A UNIX manual entry for pLucid

- 6 Tables and Rules
  - 6.1 Tables of Operators
  - 6.2 Associativity and Precedence Rules
  - 6.3 Reserved Words
- 7 Miscellaneous
  - 7.1 pLucid Grammar
  - 7.2 Syntax Diagrams
  - 7.3 Programming examples

## 1 INTRODUCTION

The language pLucid can best be described as a functional dataflow programming language. We use the term !dataflow because a pLucid program defines a dataflow net and we use the term !functional because for each node in the net, the entire output history is a function of the entire input history.

A brief look at !pLucid's pedigree shows that its block structuring mechanism is the @where clause from P.J.Landin's ISWIM, and that its data types are those of Pop2, i.e. integers, reals, booleans, words, character strings and finite lists. The language pLucid is !typeless and so there are neither type declarations nor compile-time type checking. Those programmers who are familiar with programming the UNIX shell will feel at home programming in pLucid. The reason for this is that basic to both pLucid and UNIX are the concepts of !filter and !!data stream&.

A pLucid program is !!an expression&, and a program execution is an !!expression evaluation&. For example, consider the simple expression  $x + y$  which constitutes a complete pLucid program. When evaluated this program repeatedly demands values for @x and @y and outputs their sum. Note that the free variables in a pLucid expression are assumed to be inputs. The above program induces an endless interaction between the user and the evaluator that might proceed as follows:

```
x( 0): 2      !!The first demand for a value of& x!!.&
y( 0): 3      !!The first demand for a value of& y!!.& output( 0): 5      !!The first
value of the sum.&
x( 1): 1      !!The second .....&
y( 1): ~8 output( 1): ~7
x( 2): 2.73   !!The third .....&
y( 2): 1 output( 2): 3.73
x( 3):        !!A fourth demand for x, for which the user&
```

!!has not yet supplied a value.& In section 5 we explain how these apparently endless computations can be made to terminate in a graceful manner.

Another example program is the following: `hd( tl( L ) )` When this program is evaluated it repeatedly demands values for L (which are assumed to be finite lists) and outputs the head of their tail (i.e. the second element of the input list). This is another example of a continuously operating program evaluation. For this program a listing of the interaction between the user and the evaluator of might look as follows:

```
L( 0): [42 7 5] output( 0): 7
L( 1): [3 [2.4 8] 9] output( 1): [2.4 8]
L( 2): [2] output( 2): ?      !!(the pLucid error object) &
```

```
L(3): [5 1 2] output(3): 1
. .
. .
. .
```

Note that, given an input which is not a finite list or a finite list consisting of at least one object, the resulting output is the error value @?&.

One important property shared by the above examples is that they can all be thought of as filters. The first example can be thought of as an addition filter, i.e. it takes two streams of inputs and produces the stream of their sums. The following sequence of snapshots of the dataflow computation:

```

| | | | |
2| |3 | | 1| |~8 2.73| |1
V V V V V V V V
+-----+ +-----+ +-----+ +-----+
| plus |==>| plus |==>| plus |==>| plus |==> continued
+-----+ +-----+ +-----+ +-----+
| | |5 | | |7
| | |5 | | |5
V V V V
V note in pLucid negative numbers are prefixed by the
symbol ~ as in ~8 (minus 8).
```

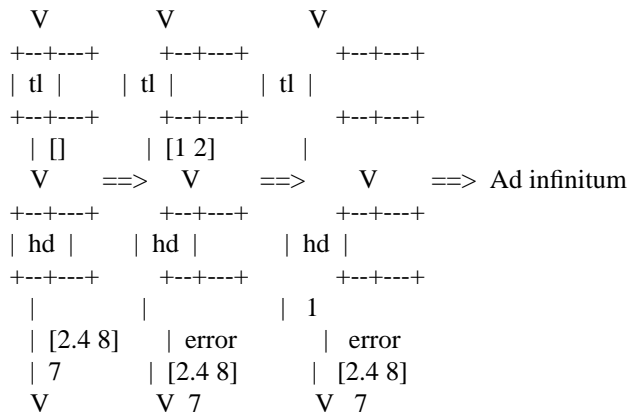
```

| | | |
| | | |
V V V V
+-----+ +-----+
| plus |==>| plus |==> ..... Ad infinitum
+-----+ +-----+
| 3.73 | 3.73
| ~7 | | ~7
V 5 V 5
```

Similarly the second example is a combination of two filters. The first called the tail filter, takes one input stream and produces the stream of tails. The output of this filter is piped to the input of the second filter, namely the head filter. This second filter produces as its output the heads of the sequence of finite lists input. The following is a sequence of snapshots of the computation:

```

| | |
|[42 7 5] | | [3 [2.4 8] 9] | [2]
V | |
+-----+ +-----+ +-----+
| tl | | | tl | | | tl |
+-----+ +-----+ +-----+
| ==> | [7 5] ==> | [[2.4 8] 9] ==> continued
V V V
+-----+ +-----+ +-----+
| hd | | | hd | | | hd |
+-----+ +-----+ +-----+
| | | |7
V V V
| | |
|[5 1 2] | | |
```



Simple filters like the above are usually memoryless, but there is no reason why this should be the case for more complex filters. It is possible for a filter to produce as output values that depend upon values already processed. An example of such a filter is one that outputs the running total of the numbers that it has received as input. Another example of a filter with memory is one that takes as input a sequence of numbers, one at a time, and produces as output the smallest and the largest of the numbers read so far. Initially the program reads the first number and gives this number as both the smallest and the largest read so far. Then it asks for the next input and if this input is smaller than the smallest it replaces the old smallest and in addition is output as the current smallest. If this input is larger than the largest it becomes the largest and is output as the current largest. In the case of it being the same value as the current smallest or the current largest the input is ignored. Note that if the input is anything but a number, then the output will be the special error value. This program can be written in pLucid as follows : [% "smallest", s , "largest", h %]

where

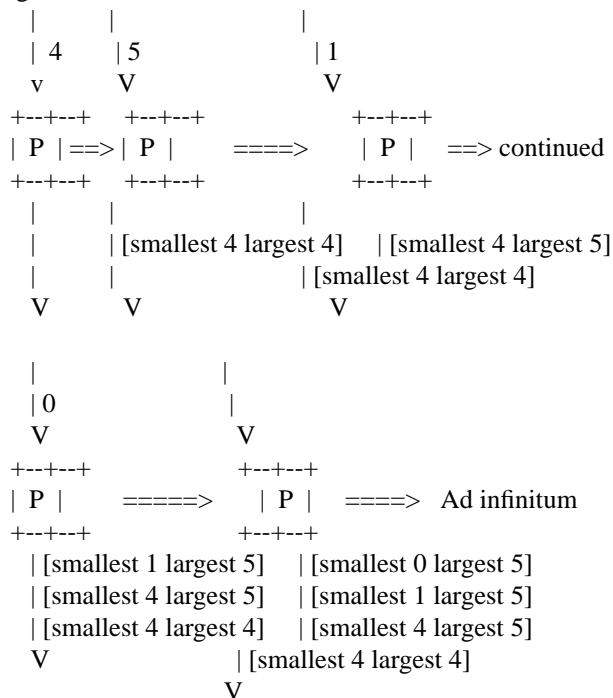
```

s = x fby if next x < s then next x else s fi;
h = x fby if next x > h then next x else h fi;

```

end A sequence of snapshots of a computation is illustrated in the following

diagram:



## 2 LUCID EXPRESSIONS

### 2.1 THE where CLAUSE

The @where clause is pLucid's means of structuring programs, just as the !block is the means in Algol and Pascal. As mentioned earlier a pLucid program is an expression, and is thus a composition of subexpressions. To avoid horrendously long and complicated expressions we use the @where clause to replace subexpressions by variables. For example, the following three programs are all equivalent.

- i. `@@(x ** y) / (x**y div z)&`
- ii. `@@temp / (temp div z) where temp = x ** y; end&`
- iii. `@@temp1 / temp2  
 @@where  
     @@temp1 = x ** y;  
     @@temp2 = temp1 div z;  
 @@end`

Moreover, @where clauses may be nested to arbitrary depths. For example the following two programs are equivalent:

- i. `@@(x-2)*(x+3)+y+z&`
- ii. `@@a+b  
 @@where  
     @@a = w1*w2  
     @@where  
         @@w1 = x-2;  
         @@w2 = x+3;  
     @@end;  
     @@b = y+z;  
 @@end&`

In this last program, the expression @a+b is called the !subject part& of the (outermost) @where clause. The !body of this @where clause consists of two !definitions&, the first defining @@a&, and the second defining @@b&. The subject part of the innermost @where clause is @@w1\*w2&, and its body consists of two definitions, namely those of @w1 and @w2&.

### 2.2 THE OPERATORS next, fby and first

Variables in pLucid are similar to variables in most conventional programming languages. They are dynamic objects in the sense that their values change as the program execution proceeds. Often, in the execution of a particular @where clause, it is possible to think of an execution as a sequence of steps in which the values of the local variables of the @where are updated simultaneously. For example, suppose we wish to write a program to read in a stream of numbers and output the partial sums i.e. after each number is read the sum of the numbers read so far is output. For this we can use two variables, one called @@i&, which at any step holds the last number read in, and another called @@s&, which at any step holds the sum of the numbers read so far. At each step in the program execution the variables @i and @s are updated. At any step the next value of @i is the next value to be read, while the next value of @s is the present value of @s plus the next value of @@i&. In pLucid @s is defined as follows: `s = i fby s+next i` This definition is read: The first value of @s is the first value of @@i&, while at any step in the program execution, the next value of @s is the present value of @s plus the next value

of `@i&`. The complete pLucid program to generate the stream `@s` of partial sums is: `s` where

```
s = i fby s+next i;
```

end This program uses the two pLucid operators `@next` and `@fby` (pronounced "followed by") which we will now introduce. `@next` is a prefix operator which, when applied to a variable at any step in the execution, returns the value which that variable has after the next step of the execution. Of course in a conventional language we do not know what the next value of a variable will be; however in pLucid this value can be computed from the definition. For example, suppose that a variable `@x` is defined in terms of a variable `@y` by the pLucid definition, `x = next y` then at any step in the program execution the value of `@x` will be the next value of `@y` i.e. the value of `@y` after the next execution step. Hence if, as we go through the execution steps, `@y` takes on successive values from the stream 2,4,6,8,10,..., then `@x` takes on the successive values from the stream 4,6,8,10,... Thus, `@x` is 4 when `@y` is 2, `@x` is 6 when `@y` is 4, and so on.

As well as being able to talk about the next value of a variable we can also talk about the next value of an expression. For example, suppose `@x` and `@y` are as above, then at any step the next value of `@@x+y&` will be the sum of the next values of `@x` and `@@y&`. So, if `@z` is a variable such that at any step `@z` is the next value of `@@x+y&`, then in pLucid `@z` is defined by: `z = next(x+y)`

Let us now turn our attention to the infix operator `@@fby&`. As described earlier, in pLucid we regard variables as dynamic objects, dynamic in the sense that their values change as the program execution proceeds. In the programs we have introduced so far, the values of all the variables are simultaneously updated at each computation step. In fact, for each variable we can talk about the "stream of values" it assumes during the course of a program execution. For example, we can interpret statements such as, "the variable `@x` takes the values 2 followed by 4, followed by 6 etc.", to mean that after the first execution step `@x` is 2, after the second step `@x` is 4, after the third step `@x` is 6, and so on.

In general the infix operator `@fby` may have the following form: `x = <!!expression1&> fby <!!expression2&>` This can be read as follows : The stream of values of `@x` is the initial value of the `<!!expression1&>` followed by each of the successive values of the `<!!expression2&>`. An alternative reading is: The initial value of `@x` is the initial value of the `<!!expression1&>`, and at any step in the program execution, the next value of `@x` is the present value of `<!!expression2&>`.

The final operator to be introduced in this section is the prefix operator called `@@first&`. For any expression `<!!expr&>`, if the variable `@x` is defined by

```
@x @ = @first <!!expr&>
```

then at any step in the program execution, the value of `@x` is the first (i.e. initial) value of `<!!expr&>`. For example, suppose `@int` is a variable having the values 0 followed by 1, followed by 2, followed by 3, etc. Then the expression, `@@first int&`, takes the values, 0 followed 0, followed by 0 ,etc. in other words, `@@first int&` is equivalent to the constant `@@0&`.

Now that `@@next&`, `@fby` and `@first` have been introduced we consider examples of their use. The first example is a filter that produces as output the stream of integers 0,1,2,..etc. `int` where

```
int = 0 fby 1+int;
```

end The program execution begins by producing the initial value of `@@int&`, i.e. 0. From then on, the execution repeatedly adds 1 to `@int` and outputs the result. The filter just described is so useful that the current Unix implementation includes a predefined variable called `@index` that produces the same values as the above filter. Thus the following program: `index` is equivalent to the one above. Moreover each `@where` clause

has its own private @index variable.

The next example is a filter that produces as output the stream of squares 0,1,4,9,16,...etc. sq where

```
int = 0 fby 1+int;
sq = 0 fby sq+2*int+1;
```

end As in the previous program the variable @int takes on the successive values 0,1,2,3,... The first value of @sq (i.e. the square of 0) is 0, while at any step the next value of @sq is the present value of @sq plus two times the present value of @int plus 1. Note that we have used the fact that for any n

$$(n+1)*(n+1) = n*n+2*n+1$$

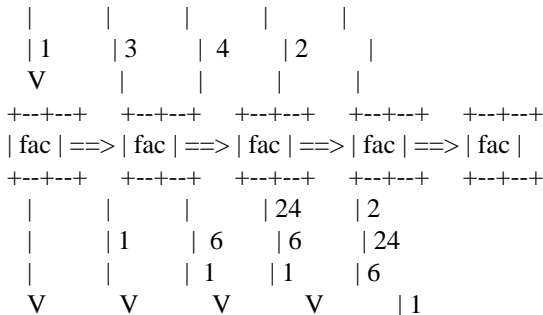
The next example is a filter that uses Newton's algorithm to output a stream of approximations to the square root of the input. Roughly speaking the algorithm goes as follows, to calculate the square root of a number n, take the first approximation to be 1, and thereafter take each successive approximation to be  $(x + n/x)/2$ . In pLucid we might code this up as follows: approx

```
where
approx = 1 fby (approx+first n/approx)/2;
end (For improvements on this example see sections 2.4 & 2.5)
```

### 2.3 USER DEFINED FUNCTIONS (UDF'S)

Pointwise functions (or operators) in pLucid are similar to pure functions (i.e. functions without side effects) in other programming languages. For example, the following program repeatedly reads in integers and outputs the corresponding factorial. fac(n)

```
where
fac(n) = if n eq 0 then 1 else n*fac(n-1) fi;
end The function @fac defined in the body of the @where clause can be thought of as a machine or !!black box& which continuously consumes values of @n and produces values of @@fac(n)&. In other words it is a filter in the UNIX sense, the only difference being that this !!black box& filters integers rather than characters. The following is a sequence of snapshots for a particular sequence of inputs:
```



V The filter produces each of the factorials @fac(n)& 1,6,24,2 as @n takes each of the values 1,3,4,2. However pLucid programmers are not restricted to only pointwise filters. A pLucid filter may have internal memory and/or it may produce outputs at a different to that at which it consumes inputs.

The following non-recursive udf has internal memory. It computes the average of the last three numbers input.

```
avg3(input)
where
avg3(n) = avg
where
avg = (one+two+three)/3;
```

```

one = n;
two = next n;
three = next next n;
// This is a comment
// A simpler way of writing this whole
// where clause would be as the following
// expression
// (n + next n + next next n)/3
end;
end

```

Another example of a program using a udf is one which reads in a stream of finite lists and outputs a stream of the atoms in those lists, e.g. if the list [1, [3.7], true] is input, then the atoms 1, 3.7, and true will be output. flatseq(x)

```

where
flat(x) = if x eq nil then nil
         elseif isatom( x ) then x fby nil
         else join(flat(hd x),flat(tl x))
fi;
join(x,y) = if first x eq nil then y
            else x fby join(next x,y)
fi;
flatseq(x) = join(flat(first x),flatseq(next x));

```

The function @flatseq consumes a stream of finite lists and produces a stream of all the atoms in those lists, while the function @flat consumes just one list and produces its atoms. The function join consumes two streams of atoms for example 1,2,[],[],... and 3,4.75,[],[],... and produces the stream 1,2,3,4.75,[],[],... The filters @@flat&, @join and @flatseq could in principle be implemented as three concurrently running processes. Note that @flatseq produces values faster than it consumes them, as for each list going into @flatseq many atoms may be produced as output.

Another example of a simple program with functions is the following. The program generates the prime numbers using the !!sieve of Eratosthenes&. sieve(n)

```

where
n = 2 fby n+1;
sieve(i) = i fby sieve(i whenever i mod first i ne 0);
end

```

## 2.4 THE OPERATORS asa,whenever,upon and attime

### asa

If @asa were not a predefined infix operator it could be defined as follows:

```

asa(x,y) =if first y then first x
         else asa(next x,next y) fi

```

Although this is a concise and precise way of defining @asa it may be better at this stage to give a more operational description, such as the following. The @asa operator computes by repeatedly reading in pairs of values of @x and @y until (if ever) @y has the value @@true&. Assuming a @true is eventually read then the value taken by the @@asa& operator will be the value of @x corresponding to the @@true& value of @@y&. Moreover the @asa takes on this value everywhere. In this respect the @asa operator can be thought of as a constant. For example, if @x takes the values 0,1,2 and @y the values false, false, true then @@x asa y& takes the values 2,2,2,... Another example is if @x takes the values abc, 3, [d,e] and @y the values false, true, false, then @@x asa y& produces as output the stream of values 3,3,3,...

The following program illustrates a use of the @asa operator. The program is a slightly different version of the Newton's algorithm program of Section 2.2. (approx asa count eq 10) fby eod

where

```
approx = 1 fby (approx+first n/approx)/2;
count = 1 fby count+1;
```

end Given a number 42 say as input (i.e. the variable @n is a free variable in this program and is therefore assumed to be an input variable), the program outputs the 10th approximation to its square root i.e. 6.48.

**whenever**

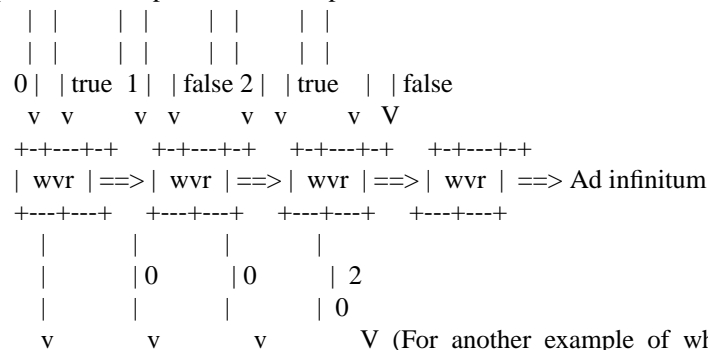
The predined infix operator @whenever (sometimes written @@wvr&) could be defined recursively as follows: whenever(x,y) = if first y then x fby Z else Z fi

where

```
Z = whenever(next x,next y);
```

end Again, at this stage, a more operational description will be of more

benefit to the reader. The operator @whenever is best thought of as a process that filters out some elements (possibly none) from the stream @@x& according to a control stream (i.e. booleans) @@y&. In terms of machines (see the factorial example in section 2.3) @whenever repeatedly takes pairs of values for @x and @@y&; if @y is true then @x is output and if @y is false then @x is discarded. For example suppose at each step in the computation @@y& takes on successive values from the stream true,false,true,false,... i.e. y = true fby false fby y In addition suppose at each step in the computation @@x& takes on successive values from the stream 0,1,2,... i.e. x = 0 fby x+1 If @x and @y take on successive values as described above then, @@x whenever y& will produce as output successive values from the stream 0,2,4,6,... of even numbers. The following is a pictorial description of the computation:

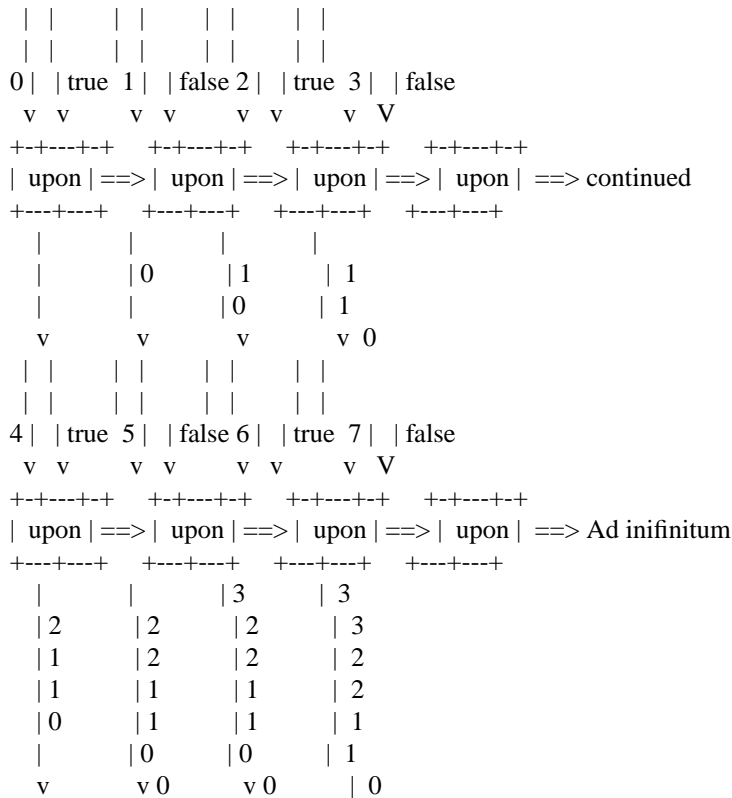


**upon**

The @whenever operator is best thought of as a filter that selectively passes on values that it receives as its input. The Lucid operator @upon is a dual to @@whenever& in the sense that it selectively stretches out the values that it receives on its input. Again we can give a concise definition of @@upon& in terms of a recursive user defined function as follows: upon(x,y) = x fby if first y then upon(next x,next y) else upon(x,next y) fi If we think of @upon as a black box then we can describe its behaviour as follows. The first output of the box is whatever shows up as the first value of @x. If the corresponding value for @y is @true then the box will advance looking for the next value of @@x&. However if the corresponding value of @y is @false the next value output will again be the current value of @@x&. Thus our black box' output is controlled by the boolean values sent along @@y&. The following is an example of a program to stretch out the stream 0,1,2,3,.. of natural numbers.

stretch where

```
x = 0 fby x+1;
y = true fby false fby y;
stretch = x upon y;
end; The following is a pictorial description of the computation:
```



v 0 Another example of a program that uses @upon is the following merge program. The streams to be merged (i.e @xx and @@yy&) are assumed to be streams of numbers in increasing order. Let define the stream associated with @xx to be 2,4,8,16,32,64..., and the stream associated with @@yy& to be 3,9,27,81,.... Then the ordered merge of these two streams will be will be the stream 2,3,4,8,9,16,27,32,64,81,.... The following is the pLucid program that computes this ordered merge:

```
merge
where
  merge = if a<b then a else b fi;
  a = xx upon a eq merge;
  b = yy upon b eq merge;
  xx = 2**i;
  yy = 3**i;
  i = 1 fby i+1;
end;
```

### attime

The last operator (filter) to be described @attime. If this was not a predefined infix operator we could define it with the following udf:

```
attime(x,y) = (x asa index eq first y ) fby attime(x, next y); Note @index has been defined in section 2.2. In fact using @attime it is possible to give the following non-recursive definition for @upon and @whenever.
```

```
wvr(x,p) = x attime t2
where
```

```

    t1 = if p then index else next t1 fi;
    t2 = t1 fby t1 attime t2+1;
end;
upon(x,p) = x attime t1
where
    t1 = 0 fby if p then t1+1 else t1 fi;
end;

```

## 2.5 THE is current DECLARATION

In this section we introduce a pLucid construct that allows the programmer to write programs which involve nested iteration. This construct is the @is @current declaration.

Suppose we wish to write a program which repeatedly takes pairs of non-negative integers and raises the first to the power of the second, e.g. 2 to the power of 3 is 8. A typical approach to designing such a program would be firstly to write a smaller program to produce the power of just one pair of inputs, and secondly to embed this program in another that executes the first program indefinitely. In pLucid the first program could be written as follows: p asa index eq first n

```

where
    p = 1 fby p * first x;

```

end It is a simple loop program where, each time round the loop, @p is multiplied by the first value of @@x&. The loop continues until a variable that we have chosen to act as the loop counter, namely @@i&, equals the first value of @@n&, after which the value of @p is the first value of @x raised to the power of the first value of @@n&. We now attempt to place this program within a nonterminating loop program. The resulting program has two loops, one nested within the other; however whenever the inner loop (i.e. the above program) is executed, the expressions @@first n& and @@first x& would always have the same value. Consequently the power of only the first pair of inputs would be repeatedly computed. Obviously this is not what we require. We can try to resolve the situation by replacing @@first n& and @@first x& by @n and @x respectively. Unfortunately this does not help either since we require @n and @x to remain constant during each execution of the inner loop. Thus we require that @n and @x be outer loop variables which only change between executions of the inner loop. This is clearly impossible when we think of @@x&, @@n&, @@index&, and @p as variables which are all updated simultaneously. To overcome this problem we use the @@is current declaration with @n and @x in the above program. The effect of this is to set up an outer loop in which @x and @n only get updated between executions of the inner loop. The resulting program is the following: p asa index eq first N

```

where
    X is current x;
    N is current n;
    p = 1 fby p * first X;

```

end Note the informal convention used i.e. that the variables that are introduced by an @is @current declaration use upper case. Although any variable name could have been used, the use of upper case letters makes programs involving nested iterations easier to understand. The inner loop variable @X only ever has one value which is the current value of @@x&, hence as @index and @p get updated, @X stays constant. Similarly @N, the current value of @@n&, stays constant during the execution of the inner loop. Remember, @x and @n are outer loop variables, while @X and @N are inner loop variables which restart life anew at the beginning of each inner loop execution. In general, the effect of the @@is current& declaration is to set up a nested iteration.

An improvement in the above program is to replace @@first X& by @X and @@first N& by @N. This does not change the meaning of the program as @X and @N remain constant during each execution of the inner loop. This results in the

following program: p asa index eq N

```

where
  X is current x;
  N is current n;
  p = 1 fby p * X;
end

```

Now, suppose we had wanted to write a program which took a power (the first value of @n) and then raised each of a sequence of inputs to that power e.g. it might compute the squares of all its inputs. The program would then use @ is current with @x but not with @n. p asa index eq first n

```

where
  X is current x;
  p = index fby p*X;
end

```

Note that the expression @first n cannot be replaced by @n as n is a variable that is active in the inner loop, and we require n to be the same for each execution of the inner loop. In a similar way we can write a program which takes an integer (the first value of @x) and raises it to each of a sequence of powers e.g. it might compute powers of 2. The program uses @ is current with @n and not @x. p asa index eq N

```

where
  N is current n;
  p = 1 fby p * first x;
end

```

As with @n in the previous program, we cannot replace @first x by @x.

Another example of a program using @ is current is one to calculate the exponential  $e^x$  using the series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

The program is expsum asa next i eq 10

```

where
  X is current x;
  i = next index;
  term = 1 fby (term/i)*X;
  expsum = 0 fby expsum+term;
end

```

The body of the outer loop asks for the present value of @x to be input, and then outputs an approximation to  $e^x$ . The inner loop takes the current value of @x i.e. @X and computes an approximation to  $e^x$  by adding the first ten terms of the above series. The body of the inner loop computes the next term of the series and adds it to @expsum. Hence each time the inner loop is called it executes its body ten times. If the stream of values 1, 2, 0.5, -1 are the inputs for @x then the stream 2.718, 7.389, 1.649, 0.369 will be output.

The next example is an improvement of Newton's algorithm for computing a square root given in section 2.1. Given a stream of numbers we compute approximations to their roots by applying Newton's algorithm to each of them 10 times. sqroot where

```

X is current x;
sqroot = approx asa count eq 10
  where
    approx = 1 fby (approx + X/approx)/2;
    count = 1 fby count+1;
  end;
end

```

Inputting the numbers 2, 2.213, 26.7 produces the stream 1.414, 1.487, 5.167 as output.

The general form of the `@@is current&` declaration is:  
`<!!variable&> @@is @current <!!expression&>` Our next example uses `@@is current&` in its more general form, namely with an expression on the rhs. It computes the roots of the polynomial

$$a x^2 + b x + c = 0$$

in the obvious way using the previous example. The roots are output as a two element list. `r1::r2::nil`

```

where
  r1 = (-b+sqrt)/(2*a);
  r2 = (-b-sqrt)/(2*a);
  sqrt = approx asa count eq 10
        where
          X is current b*b-4*a*c;
          approx = 1 fby (approx+X/approx)/2;
          count = 1 fby count+1;
        end ;
end

```

### 3 THE pLucid DATA TYPES

So far we have described the Lucid operators which are available in pLucid. Such operators are common to all members of the Lucid family of languages. Each particular member of the family is distinguished by the choice of data types on which the Lucid operators are applied. However, the members of the Lucid family are typeless, in the sense that there are no type declarations and no type checking at compile-time.

The data types of pLucid are integers, reals, words, strings, finite lists, together with the two special data types `!error` and `!!eod&`. In this section we give a brief informal introduction to pLucid expressions using these types (for more formal details see Section 6).

#### 3.1 Numeric Expressions

Numeric expressions in pLucid are basically the same as in most other programming languages supporting mixed integer and real arithmetic. Numbers, i.e. reals and integers, are represented by numeric constants. As with APL, pLucid distinguishes between the sign of a number, and the operator applied on that number. The binary subtraction operator on numeric expressions is represented by the symbol `@@-&`; but the negative sign of a numeric constant is represented by the symbol `@@~&`. The symbol `@@+` is the binary addition operator on numerics, but the positive sign for numeric constants should not be written, as it is assumed by default. For example the following are numeric constants

```

1 42 ~6 1.0 ~5.324 42.127 the following are not numeric constants
+2 -4 -1.0 .12

```

At present there is no provision in pLucid for the exponent representation of reals.

Apart from the additional operator `@@isnumber&`, pLucid's numeric operators are essentially the same as those in most other programming languages.

`@isnumber` is an operator which tests its argument to see if it is numeric. It returns the word `@true` if its argument is either an integer or a real, and returns the word `@false` otherwise. For example, the following program checks whether its input is a number. If it is, it outputs the square of that number, then asks for the next input, and so on. If the input is not a number the program outputs an error message:

```

if isnumber(x) then x*x else 'The input was not numeric\n' fi

```

To finish this section on numeric expressions we will, by way of examples, describe the pLucid division and

modulus operators. The remaining numeric operators need no explanation beyond that given in Section 6.

pLucid has two division operators; @div for integer division, and @/ for real division. For any numbers !n and !m, !n @div !m is the largest integer such that (!n& @div !m&)\*!m& is not greater than !n&. The following are example pLucid expressions together with their equivalent integer constants.

12 @div 5 is equivalent to 2  
60 @div ~5 is equivalent to ~12

The second division operator is for real division. @/ accepts numeric operands and yields a numeric result, e.g.

123 @/ 5 is equivalent to 24.6  
0.123 @/ 0.123 is equivalent to 1  
~1.0 @/ 3 is equivalent to ~0.33333

Remember that the accuracy of @/ is dependent upon the accuracy of the pLucid implementation.

The modulus operator @mod is the mathematical modulus function. For any numbers !n and !m&,

!n @mod !m equals !n @-!m&@\*&(!n& @div !m&)

For example,

9 @mod 5 is equivalent to 4  
~9 @mod 5 is equivalent to ~4  
4.5 @mod 1.2 is equivalent to 0.9

### 3.2 Non-Numeric Data Processing

Besides numeric computation, pLucid allows non-numeric data processing. This is done using the types word, string and finite list. These types, which are similar to those in POP-2, are discussed in this section.

#### 3.2.1 Word Processing:

A word is determined by a sequence of characters of one of the following forms,

- an arbitrarily long sequence of alphanumerics starting with a letter  
(e.g. this t23r Warwick )
- an arbitrarily long sequence of signs, where a sign is one of  
+ - \* / \$ & = < > : # ^
- a bracket, where a bracket is one of,  
( ) ( % %) [% %]
- a separator, where a separator is either ; or ,
- a period
- a double quote

The following are examples of words

yes true . " %] ##\$\$& , false A word constant is simply a word enclosed in double quotes. For example, the pLucid word constants representing the above words are "yes" "true" "." "" "" "%]" ", " "##\$\$&" The distinction between words and word constants is important and should must be clearly understood. The following example is

instructive:

"fred" is a word constant representing the word fred. On the other hand fred is an identifier (i.e. a variable).

Word constants, like other constants, are pLucid expressions, and can be used to define values for variables in programs. For example, the definition

x = "dog"; defines the value of the variable @x to be the word @@dog&.

The boolean values in pLucid are the words @true and @false, i.e they obey the rules for the type word i.e. @@isword (true)=true& and @@isword(false)=true&. Note the programmer may assume that the variables @true and @false have been predefined to the words @true and @@false&. Thus it is as if every pLucid program has the following definitions included automatically:

true = "true";

false = "false"; However, for the sake of clarity, we shall talk about these two special words separately in section 3.2.2.

Beside those operators described in section 3.2.2, which apply to the words @true and @@false&, there are four other operators applicable to words in general. These operators are @@isword&, @mkword and the polymorphic operators @eq and @ne which are used to compare words.

The operator @isword recognizes words, it yields @"true" if its argument is a word, and @"false" otherwise. For example :

@@isword ("pLucid")& is the word @"true" &

@@isword (123)& is the word @"false" &, as the argument is a number.

The operator @@mkword&, read !make word&, takes a string as an argument. If the characters in this string make up a legal word it returns that word. Otherwise, it returns the error object. For the definition of what a string is see section 3.2.3. The following are some examples of the use of the operator @@mkword&:

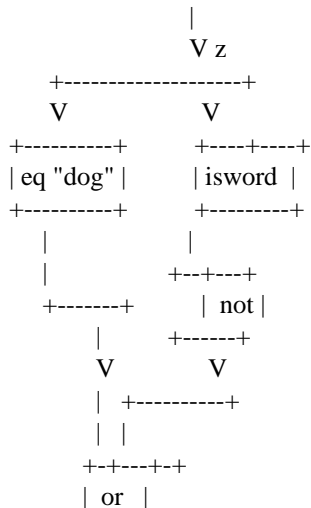
@@mkword ('this')& has the same value as the word constant @"this" &

@@mkword ('this ')& returns the error object,

@@mkword ('123')& returns the error object,

The binary operators @eq& and @ne are for comparing two words. @eq returns the word @true if its arguments are equal, and returns @false otherwise. @ne returns @true if the arguments are different, and @false otherwise. It is worth mentioning here that these operators can be applied to any data objects in pLucid data types. For example :

(z eq "dog") or not (isword (z)) can be viewed as a filter with three inputs.



```

+-----+
|

```

V The above machine continuously asks for inputs. If the input is neither a word nor the word @dog, then the output is the word @@true&, otherwise the output is @@false&.

### 3.2.2 Boolean Expressions and Predefined Variables

It is assumed in each pLucid program that there are five variables with predefined values. These variables are @@true&, @@false&, @@nil&, @error and @@eod&. The variables @true and @false have as values the words @true and @false respectively. The value of the variable @nil is the empty list. It is as if the pLucid programmer has the following definitions included in his program.

```

true = "true"
false = "false"

```

nil = [] The variable @error has as its value the error object and the variable @eod has as its value the eod (end of data) object. For the definitions of these last two objects see section 3.3.

pLucid has the usual boolean operators, namely @@or&, @@not& and @and. These operators yield their usual boolean results when applied to the words @@true& and @@false&. However when one of their arguments is neither @true nor @false the operators may yield the error object. For further details see section 3.3.

Apart from many of the usual logical operators found in other languages, pLucid includes the operators @@isnumber&, @@isword&, @@isstring&, @@isnull&, @@isatom&, @@iserror&, and @@iseod&. These will be defined throughout the definition of the different data types of pLucid.

### 3.2.3 String Processing:

A string, in pLucid, is determined by a sequence of zero or more ASCII characters. Strings must not be confused with words, they form a completely different data type. Any attempt to apply a string operator on a word, or vice versa will yield the error object.

Strings in pLucid are string constants. The string constant representing a given string is obtained by first replacing each character by its pLucid representation, and secondly enclosing the resulting sequence between @' and @'&. For example, the following are string constants:

```

':: yes, ~ #'
'this is a string ? '
'pLucid is non procedural '

```

For string processing in pLucid, we have the following operators: @isstring takes one argument and returns the word @true whenever this argument is of type string; otherwise it returns the word @@false&.

#### ^(string concatenation)

The operator @^ is the string concatenation operator. It takes two strings and forms a new one by concatenating the second to the end of the first. However, if one of its arguments is not a string it yields the error object.

#### substr

The operator @substr read as !!substring&, takes three arguments. The first of which is a string, and the other two are integers. These two integers determine a range for extracting a substring from the first argument of @substr (the first argument must be a string). Hence, the first integer should be less than or equal to the second which should be less than or equal to the length of the string. Mathematically, the operator @substr can be

defined as follows:

If the value of the string  $S$  is the sequence of characters  $\{S_1\} \{S_2\} \dots \{S_{n-1}\}$  where  $k$  and  $m$  are integers such that  $!k \leq !m \leq !n$ , then  $@@\text{substr}(!S, !k, !m) = \{S_k\} \dots \{S_m\}$ . Otherwise, the result will be the error object.

### **mkstring**

The operator `@mkstring`, read as `!!make string`, takes its argument, which should be a word, and makes a string out of it. If the argument is not a word, the result will be the error object. Consider the following example pLucid expressions and their equivalent string constants

```
@@mkstring("hello")& gives @@'hello'&
@@mkstring("t1239")& gives @@'t1239'&
@mkstring(123) yields the error object
```

### **length**

The operator `@length` returns the length of a string, that is the number of characters contained in the string. Note that internally characters are stored unescaped. That is to say that `\b`, backspace is not stored as two characters `backslash` and `b` it is stored as the character `backspace`.

NOTE `@length` is polymorphic with respect to types, that is given an expression that evaluates to a type that is measurable in length then `@length` will return its length otherwise it will return the error object.

### **Examples**

The following is an example program that takes one input `@@x`, and checks whether or not it is a string.

```
if isstring x
then x ^ ' is a string well done'
else 'Sorry its not a string, try again'
fi If @@x is a string the program outputs @@x concatenated to the string
' is a string well done' Otherwise the program outputs
'Sorry its not a string, try again'
```

### **3.2.4 List Processing:**

The third non-numeric type in pLucid are the finite lists. A list in pLucid is essentially a sequence of zero or more items, each of which is either a number, a word, a string, or a list. Syntactically, there are two ways to represent a list in pLucid; as a list constant, and as a list expression.

#### **list constant**

A list constant is a sequence of items enclosed in square brackets. Each of these items is either a numeric constant, a word constant less the quotes, a string constant, or a list constant. Quotes, single and double, around string and word constants act as delimiters. However it is obligatory in pLucid to write the string quotes, it is necessary to drop the quotes of the word constants when written as items in a list constant. Whenever ambiguity occurs a space can be used to separate two successive words. So the following two list constants have the same value

- i. @@" dog "
- ii. @@"dog"

but the following do not

- i. @@"dog"
- ii. @@"dog]

Moreover, the following two list constants are different

- i. @@" this is pLucid ]
- ii. @@"this' is pLucid]

because the sequence of letters, this, occurs in the first list as a word constant, while in the second it is a string constant.

The second way for representing lists is by list expressions. A list expression is either an expression build up using the @:: (cons) operator or a sequence of zero or more pLucid expressions enclosed in %-square brackets (@@[%& and @@%]&) and separated by commas. To make this clearer, consider the following examples:

1- The followings are expressions in pLucid, as they are all constants,

"dog" 'c a t' [this is pLucid] The list expression, built from these expressions, can be written as

"dog" :: 'c a t' :: ("this" :: "is" :: "pLucid" :: nil) :: nil

[% "dog", 'c a t', [this is pLucid] %] The value of either list expression is equal to the value of the list constant

[ dog 'c a t' [this is pLucid] ]

2- The value of the list expression

[% i+3, tl (y), 'S T R', [% "programming" %] %] depends on the values of @@i&, and @@y&. Assuming that @i equals 2, and @y is the list @@[ hello world ]&, then the list expression above will be equivalent to the constant

[ 5 [world] 'S T R' [programming] ]

3- The sequence of characters @@[% "dog" %]& in the list constant

[ the [% "dog" %] sat on the mat ] is not a list expression, it is a sequence of five word constants, each of which is considered as an item in the main list. Note that every item in a list constant should be a constant.

pLucid facilitates list processing by means of the following four operators.

**hd**

The prefix operator @@hd&, read as !!head&, yields as its result first element of the argument list. This may be a word, a number, a string, or a list. When the argument to @hd is a list of one element the result will be that element. If the argument to @hd is either the empty list or a non-list then @hd yields the error object.

**tl**

The prefix operator @@tl&, read as !!tail&, yields as its result the argument list but with its head (i.e the first object in the list) removed. Unlike @@hd&, the tail operator yields the empty list when its argument is a one object list. If the argument is either the empty list or a non-list @tl yields the error object.

### <> (append)

The infix operator `@@<>&`, read as `!!append&`, takes two lists and yields a new list constructed by appending the second list to the end of the first. If one of the arguments is a non list it yields the error object.

### :: (cons)

the infix operator `@@::&`, read `!!construct&`, yields (constructs) a new list from its two arguments, the right argument must be a list. The new list has as its head the left hand argument and as its tail the right hand argument. If the right argument is not a list the result will be the error object.

`@eq` and `@ne`

The operators `@eq` and `@ne` are for comparing data items in pLucid, and have been explained before.

### isnull

The operator `@isnull` which takes a list and returns the word `@true` if the list is the empty list, and `@false` if it is not. It returns the error object if the argument is not a list.

### isatom

The operator `@isatom` takes an expression as its argument, if the expression evaluates to an object of type number, string or word the value returned by `@isatom` is `@true` otherwise it is the value `@false`.

### islist

The operator `@islist` takes an expression as its argument, if the expression evaluates to an object of type list then the value returned by `@islist` is `@true` otherwise it is the value `@false`.

### Examples

The following examples illustrate the use of `@hd`

`@@hd([hello world])&` has the same value as the word constant `@@"hello"&`

`@@hd([% [this 'is' ] pLucid %])&`  
has the same value as the list constant `@@[this 'is']&`

`@@hd([ [a b] c] d [e] )&`  
has the same value as the list constant `@@[a b] c]&`

The following expressions return the error object

`@hd([])`      `@hd` is applied to the empty list,  
`@hd("this")`    the argument is not a list,  
`@hd(nil)`      the argument is the empty list.

The following examples illustrate the use of `@tl`

@@tl([hello world])& has the same value as the list constant @[world]

@@tl([hello [world] ])& has the same value as the constant @@[[world]]&

@@tl([ programming ])& returns the empty list nil

@@tl([% 1+3, 'S T R', [% "programming" %]&  
has the same value as the list constant  
@@['S T R' [programming]]&

The following expressions return the error object

@tl("what") the argument is not a list  
@tl(nil) the argument is the empty list

The following examples illustrate the use of the operator @<>

@@[[pLucid] is a] <> [ [non] procedural language]&  
has the same value as the list constant  
@@[[pLucid] is a [non] procedural language]&

@@[Lucid ] <> 'with POP2'& returns the error object as one of the  
arguments is not a list

The following examples illustrate the use of the construction operator @::

@@[programming] :: [languages]&  
has the same value as  
@@[ [programming] languages ]&

@@['pLucid is']::[ Lucid]& has the same value as @@['pLucid is' Lucid]&

@@[the language] :: [% "Iswim" ,700 , 'iteration' %]&  
has the same value as  
@@[[the language] Iswim 700 'iteration']&

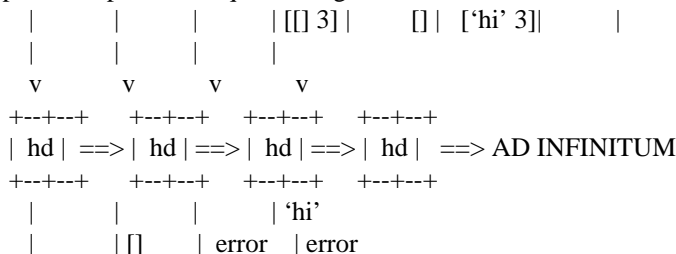
@@['##87'] :: nil& returns the list constant @['##87']

@@[pascal] :: 'triangle'& returns the error object, because  
the second argument is not a list.

**Example**

Any expression above whether it is a constant or an operator with some operands can be written as a program by itself, for example the expression

hd(x) is a program which repeatedly asks for values for the variable @@x&, expecting each to be a list, and producing as output the head of each. It yields the object error if the input is not a list. The program could be thought of as a filter for which a typical computation sequence might be as follows:





**Example**

When the program rotate where

```

t = x fby tl t <> z ;
z = hd(t) :: nil ;
rotate =if t eq first x and counter ne 0 then eod
        else t fi;
counter = 0 fby counter + 1 ;

```

end is run on the machine, it will ask for a value for the variable @@x&, expecting it to be a list. Then it produces all the possible rotations for this list. Instead of asking for the next value of @@x&, it gives the object eod, which terminates the whole program. So supplying the list constant @[a b c d]& as an input, the program will give the result

```

[a b c d]
[b c d a]
[c d a b]
[d a b c] and then terminates.

```

**3.3 The objects eod and error :**

Among the data types of pLucid there are two special types which are completely distinct. These are the types error and eod.

**The object error**

We have mentioned, while defining the data types of pLucid, the cases where an operator yields the object error. Informally speaking, the error object results from applying an operator of certain type to operands which are of types different than the ones which such an operator expects. For example, trying to add two words, find the head of a string, or divide a number by a list.

In a program we represent the object error by the predefined identifier @@error&. Hence, the value of the expression

error is the stream of error objects. It is worth noting here that, the above expression differs from the expression

"error" The first represents the error object, while the second is the word constant representing the word error.

While the object error is represented, in a program, by the predefined identifier @@error&, its representation as an input, or as an output, is dependent on the implementation of the language. In the Warwick implementation it is represented by the symbol @@?&. (for further details see section 5).

**the object eod**

In a program, this object will be represented by the predefined identifier @@eod&. Unlike the object error, trying to output the object eod will cause the machine to terminate the execution of the program. For example, the program

1 fby 2 fby eod will output the numbers 1, 2 and then terminates. The representation of the eod object as an input is again dependent on the implementation. In the Warwick implementation it is represented by the character control-D or the !at character (See section 5 for more details). Again, the distinction between the eod object represented by the reserved identifier @@eod&, and the word eod represented by the word constant @@"eod"&, should be clear.

Furthermore, the special objects error and eod are considered as expressions in pLucid. They are represented by predefined identifiers, but they do not have constant representations in the language. Hence, they cannot be written as items in list constants, as such items must be constants. So the items in the list constants

[ error ] [ eod ] are, respectively, the words error and eod. On the other hand being expressions, they can be written as items in list expressions.

**Examples**

The value of the list expression

[% error %] is the object error. This is because the above list expression is really an abbreviation of the expression

error :: nil which evaluates to the object error. However, the value of the list expression

[% "error" %] is a list of one item. This item is the word error.

Similarly, the value of the list expression

[% eod %] is the object eod.

[% "eod" %] is a list of one item. This item is the word @@"eod"&.

If the argument to an operator is the object eod (or the object error) then, except for certain operators, the object yielded by that operator will be the object eod (or the object error). The following are the only prefix operators that yield values other than @eod or @error when one of their arguments is @eod or @error are @@iserror& and @@iseod&. These operators are for recognizing the objects error and eod. When the argument of the operator @iserror is the object error, it yields the word @@true&. If the argument is the object eod it yields the object eod. Otherwise it yields the word @@false&. On the other hand, when the operator @iseod is applied to the object eod it yields the word @@true&. Otherwise it yields the word @@false&.

The boolean operators @and and @@or& can also return values other than @eod and @error when given these two objects as arguments. The result of applying these operators to the objects error and eod are given in the following table. It is worth mentioning here that these operators are commutative, i.e. the following two rules hold for any P and Q

$$P \text{ or } Q = Q \text{ or } P$$
$$P \text{ and } Q = Q \text{ and } P$$

P	Q	P and Q	P or Q
true	error	error	true
false	error	false	error
true	eod	eod	true
false	eod	false	eod
error	eod	eod	eod
error	error	error	error
eod	eod	eod	eod

### The operator if then else fi

The operator @if @then @else @fi is the usual. For any expressions !X&, !Y&, and !Z&, the value of the expression

```
@if !X @then !Y @else !Z @fi
```

is defined as follows. If the value of !X is the word @@true& then the value of the expression is equivalent to the value of !Y&, no matter what the value of !Z is. If the value of !X is @@false&, the value of the expression is the value of !Z&, no matter what the value of !Y is. However, if the value of !X is the object error then the value of the expression is the object error. Similarly, if the value of !X is the object eod, then the value of the expression is the object eod. For example when the program

```
if iseod x then 'I will terminate \n' fby eod
  elseif isnumber x
  then 'it is a number \n'
  elseif islist x
  then 'it is a list \n'
  else "itisaword" fi
```

is run on a machine then the program will ask for values for the identifier @x&. If the input is the input representation of the object eod in that implementation, the execution of the program will terminate after outputting the string @@'I will terminate'& followed by a new line. If @x is a number, it will output the string @@'it is a number'& as a message, then asks for the next input value of @x&. If it is a list, or a word, it will notify that by similar messages as the ones before, then asks for the next value of @x&, and so on.

### 3.4 pLucid Conditional Expressions :

The simplest conditional expression in pLucid is the expression @if @then @@else&. The expression

```
if !!<expr1>& then !!<expr2>& else !!<expr3>& fi
```

 expects the values of !!<expr1>& to be of type boolean, while the values of !!<expr2>& and !!<expr3>& could be of any type. The pLucid conditional expression works as the logical function @if @then @@else&. It evaluates the expression !!<expr1>&, if it is the word @true the result will be the value of the expression !!<expr2>&, and if it is @false the result is the value of !<expr3>&. If the expression !!<expr1> is not a boolean, i.e its value is not a truth value, the @if @then @else expression will output the error object. The word @fi appearing at the end in the above expression is a terminator for the @if @then @else expression.

#### Example

The pLucid program

```
if X <= Y then Y else X fi
```

 asks for the values of @X and @Y as inputs. It evaluates the expression @@X <= Y&, if it is @true it returns the value of @@Y&, if @false it returns the value of @@X&. If one of the inputs is not numeric it outputs the error object. Then it asks for new values for @X and @@Y&, and so on. This expression could be used to define a function which takes two arguments and yields their maximum by writing

```
max (X,Y) = if X <= Y then Y else X fi ;
```

pLucid conditional expressions could be nested to many levels by using the reserved word @elseif instead of @@else&. The word @elseif acts as @else for the outer @if expression and as @if for the new one. i.e there is no need to write the word @if after @elseif. Hence the expression

```
if x then y elseif
z then m elseif
```

```

n then q else r fi is equivalent to the expression
if x then y
  else if z then m
    else if n then q
      else r
        fi
      fi
    fi
  fi
fi

```

**Example**

The following is a legitimate expression in pLucid, it expresses a simple salaries system,

```

if ( status eq "single" ) then basicsalary
elseif ( status eq "married") then basicsalary * 1.30 +
  children * 40

```

else specialsalar y fi When this program is run the evaluator asks for a value for the variable @@status&. If the value of @status is the word @@single&, it asks for the value of @basicsalary and outputs it. If the value of @status is the word @@married&, it asks for the values of @basicsalary and @@children&, then outputs the value of the expression @@basicsalary \* 1.30 + children \* 40&. Otherwise, it asks for the value @specialsalary and outputs it. Then it asks for the next value of @@status&, and so on.

**pLucid case Expression :**

Another way of writing conditional expressions in pLucid is by using the pLucid @case expression. The @case expression provides a clearer way for writing long nested @if expressions. The example above could be written using the @case expression as follows:

```

case status of
  "single" : basicsalary ;
  "married" : basicsalary * 1.30 + children * 40 ;
  default : specialsalar y ;

```

end The @case expression consists of two parts, the !selector !part and the @case body. The selector part is the expression which comes between the words @case and @@of&, see the above example. The @case body is a list of switches, terminated by the word end. Each switch is of the form

expression : expression and must be terminated by a semicolon. One of these switches should be the default switch, which is a switch whose left hand side is the word @default. Evaluation of the @case expression proceeds as follows: the selector part, which is an expression, is evaluated, then its value is compared with the value of the left hand side expression of each switch in the @case body. If there is an expression which is equal to the selector, then the value of the right hand side expression of that switch is the value of the @case expression. Otherwise, the value of the @case expression is the value of the expression to the right of the word default. In the event of two expressions in the @case body with the same selector value, the first one to appear in the list is chosen.

The case expression, like the if expression, could be nested to many levels. This is done by writing another case expression as the right hand side of a switch in the outer case expression. For example, suppose we want to split the case of being married in the above example, so we write

```

case status of
  "single" : basicsalary ;
  "married" : case arechildren of
    "none" : basicsalary * 1.20 ;
    "many" : basicsalary * 1.30 +
      numchildren * 40 ;

```

```

        default : 'there is an error in your data\n';
    end;
    default : specialsalary ;
end

```

### pLucid cond Expression :

Another way of writing conditional expressions in pLucid is by using the pLucid @cond expression. The @cond expression provides a clean mechanism for the writing of long nested if-expressions. The example above could be written in terms of the @cond expression:

```

cond
    status eq "single" : basicsalary ;
    status eq "married" : basicsalary * 1.30 + children * 40 ;
    default           : specialsalary ;
end

```

The @cond expression consists of a body which is a list of switches, terminated by the word end. Each switch is of the form

expression : expression and must be terminated by a semicolon. One of these switches should be the default switch, which is a switch whose left hand side is the word @default. Evaluation of the @cond expression proceeds as follows: the expressions to the left of the @: are evaluated from top to bottom and if one is found to be true then the value of the @cond is the expression on the right hand side of the @: and if no expression to the left of a @: is true then the value of the @cond is the default value. In the case of there being two expressions in the @cond body that are true then the first encountered is chosen.

The @cond expression, like the @if and @case expressions, can be nested to many levels.

### 4 Scope Rules

The scope of an occurrence of a variable, in a pLucid @where clause, is either !local or !!global&. It is local if it is either defined or declared in that clause. The only declaration we have in pLucid is the @@is current& declaration. The variable occurring to the left of the declaration is local to that clause, while any variable occurring in the expression to the right is global. Moreover, if the variable is neither declared nor defined in the clause then its occurrence is global.

If an occurrence is global to a clause its value is expected to come from an outer clause, the first outer clause in which that variable is local. This also applies on the outermost clause. If a variable is left global in that clause, its value is supposed to come from an outer clause, which is the user environment. Consequently, the machine asks for that value as an input. For example:

```

X + y + z where
    X is current x + y ;
    y = 12 + z ;
end

```

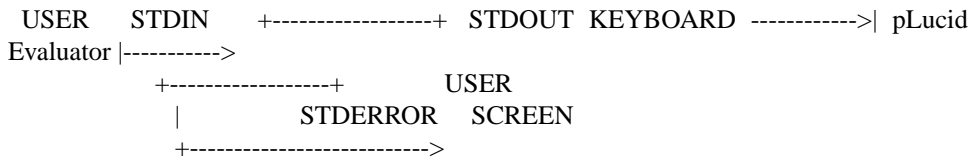
The occurrences of @X and @y in @@X + Y + Z& are local to the @where clause as is the first occurrence of @X in @@X is current x + y&. Also @y in @@Y = 12 + z& is local. The occurrence of @y and the second occurrence of @x in @@X is current x + y& are both global to the clause, as is @z in @@y = 12 + z&. Note pLucid binds variables dynamically.

### 5 Running pLucid under UNIX

### 5.1 The Basics

This section of the manual explains how a pLucid program runs in the UNIX environment.

The pLucid evaluator is itself a UNIX process and as such has associated with it an input stream (the standard input), an output stream (the standard output) and an output stream for reporting of errors (the standard error). On most implementations of UNIX the standard input is associated with the user's keyboard and the standard output and standard error with the user's screen. It is possible in UNIX to redirect input and output to places other than the screen and keyboard and the reader is referred to the UNIX manual. The following diagram shows how the pLucid evaluator interfaces with the user:



Many useful pLucid programs never need use more than these three basic streams. The @filter operator described in the next section explains how more input and output streams may be associated with a pLucid program.

Let us begin with the most trivial of pLucid programs namely the expression

x In this simple program there is no definition for @x, in other words @x is a free variable of the expression. In pLucid each free variable occurring in an expression is associated with its own pLucid input stream and all pLucid input streams take there input from the standard input (see above). Thus if we run the above program the following prompt will appear on the screen

x(0): (Prompts for input and output are sent along the standard error thus they can be discarded if they are of no help) If we input a constant such as @@4.3&, @[ 2 3 [ 4 ]]&, '@a string'&, or the word @@fred&; then our program will echo the input e.g.

```

x(0): 4.3
output(0): 4.3
x(1): [ 2 3 [ 4 ] ]
output(1): [ 2 3 [ 4 ] ]
x(2): 'a string'
output(2): 'a string'
... ..
... ..

```

Note that when words are input, such as the input for x(2), they must be input unquoted. Thus if we input @nil then we are inputting the word @nil and not the empty list. To input the empty list we use square bracket notation, i.e. @[ ]&.

Similarly, if we want to input the special object error we shall use the symbol @@?&. e.g.

```

x(3): ?
output(3): ? The special object eod is input as control-D. (or whatever the local UNIX
convention is for end of file) e.g.

```

```

x(4): ^D
% where % is the UNIX shell prompt. Thus the special object eod can be used to
gracefully terminate the evaluation of a continuously operating program. What is in
effect happening is that the pLucid input stream associated with @x is closed after an
@eod has been read. All future inputs on that particular pLucid stream are assumed to be
@@eod&.

```

### Pop input versus character input

In UNIX processes communicate to each other by sending streams of characters. For example the date command sends a stream of characters on its standard output to the user's screen. This stream of characters are those which make up the current date and time. In pLucid there are no objects of type character, so how does pLucid manage to run in the UNIX environment ?

When a pLucid program is run, it normally expects all of its inputs and outputs to be either numbers, words, strings or finite lists, successive inputs and outputs being separated by white space. However there are options available that allow input or output or both to take a different form.

### Character input

In this mode the values read from the standard input are assumed to be raw characters. As far as the pLucid program is concerned a character is a string of length one. Note that these characters may include control characters and non-printable characters. These characters are represented in pLucid strings by the following escape sequences.

```
@'\b'   for backspace
@'\n'   for newline
@'\t'   for tab
@'\f'   for formfeed
@'\r'   for carriage return
@'\\'   for \
@'\''   for '

```

In addition an arbitrary byte-sized bit pattern can be generated by writing

```
@\DDD
```

where @DDD is one, two or three octal digits. For example:

```
@'\014' is formfeed
@'\33'  is the character escape

```

The option for running the evaluator in character input mode is given in the UNIX manual entry (the -c option). Note that Pop and character input cannot be mixed. Programs that use only the basic input stream must be evaluated either in Pop input mode or character input mode.

### Character output

This mode is similar to the normal output mode except that there is no white space to separate successive outputs and string constants are stripped of their quotes and escaped characters such as @\n and @\33 are output as the characters they denote. The option for running the evaluator in character output mode is given in the UNIX manual entry (the -s option). Again programs must be evaluated either in Pop output mode or in character output mode.

### A simple program

The following is an example of a pLucid program (or filter) to compress the blanks in a file:

input whenever not (input eq ' ' and next input eq ' ') reads from the standard input character by character taking each character to be a string of length one. Similarly the output must be in character form and so the -s option must be used. If we type the above program into a file called compressblanks. We can then compile this program to get the pLucid intermediate code in the file compressblanks.i. When we run the evaluator as

follows:

`lual -c -s compressblanks.i < text` the result is that the file text will be printed on the screen with all blanks compressed.

### More than one free variable

In the following program

`x` or `y` there are two free variables, `@x` and `@y` and thus the evaluator associates with each variable it's own pLucid input stream. To evaluate the program successive values for `@x` and `@y` must be obtained from the user. The first value required is the first value for `@x`, this is because the `@` or operator evaluates its argument left to right. e.g

`x(0): true`

`y(0): false`

`output(0): true`

`x(1): ...` If the object `eod` is input as a value for one of these variables, say `@x`, the effect will be to close the input stream for that variable. Nevertheless, this does not terminate the program, as according to the algebra associated with pLucid

`eod` or `true = true` Thus if the evaluator receives `eod` as input for `@x` it will close the stream associated with `@x` (i.e. all future input on `@x` are `@@eod&`). However since the input stream for `@y` is still open and since, given `eod` as it's first argument, the `@` or operator need not terminate, the evaluator will demand the next input for `@@y&`. If the input is `@true` the evaluator outputs `@true` and continues evaluation. Note that values of `@x` will no longer be required since the input stream for `@x` is closed. Thus only values of `@y` are demanded and so long as these values are `@true` the computation continues, any other value for `@y` will cause the program to terminate.

A simpler example of a program with two free variables is

`x + y` When this program is run, it will ask for values for the variables `@x` and `@y`. If one of the input values for `@x` or `@y` is `@eod` then the program terminates. This is because in the algebra associated with pLucid,

`eod + z = eod` for any value `z`.

### pLucid programs as standard UNIX processes

As pLucid programs run as standard UNIX processes we can do all the usual things:

`lual lex.i < filename` if the file `lex` contains the program

`x` then this will read in constants (i.e. `@@[ 1 [2 3] 4]&`, `@@'a string'&`, `@@3.33&`, `@@~9&`) from file `filename` and output them to the terminal. Similarly

`lual lex.i > filename` will read constants one by one from the terminal and output them to the file `@@filename&`. We can also combine these to read constants from one file and output then to another. Our program is then in effect a filter.

`lual lex.e < file1 > file2`

### 5.2 The filter and arg operators

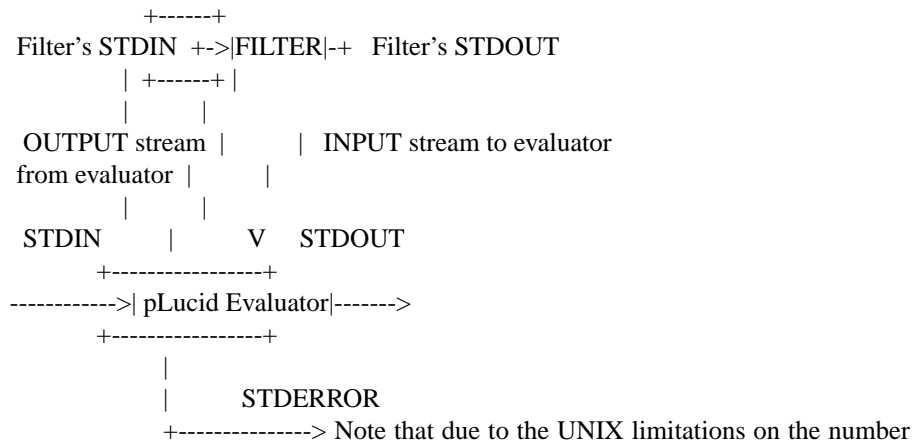
The `@filter` function enables pLucid programs to interact in a simple manner with many useful UNIX utilities. The `@filter` function takes the following form:

`filter(expression1,expression2,expression3)` The evaluation of a filter expression entails the following. Firstly evaluate `expression1`, which should evaluate to a string. This string is taken to be the name of a UNIX process. The pLucid evaluator then spawns a UNIX process named by this first argument. The second argument to the function `@filter` is the stream of values that are to be sent to the standard input of the newly created process. Values produced by the newly created process are passed along its standard output back into the pLucid program and appear as the result of the function `@filter`. The third argument of filter should again be of type string. This string is used to specify how the pLucid program is to interact the process spawned by `@@filter&`. The

options for this are as follows:

- @'s' Write to the stream associated with output to the filter in Character output mode.
- @'c' Read the stream associated with input from the filter as a stream of raw charaters. That is use Character input to read the stream produced by the process spawned by filter.
- @'i' The filter requires no inputs.
- @'p' For every input sent to the filter it will guarantee to return one value.  
i.e the filter is pointwise.

The third argument can be the empty string @'' which means that input and output to this filter should be in pop mode. Note that the above options can be combined in the same string e.g. @@'csi'&. The last two options have been included for reasons of efficiency, it is beyond the scope of this manual to explain why they are needed. The following diagram illustrates what happens when a filter is spawned by a pLucid program:



Note that due to the UNIX limitations on the number of processes and pipes that can be active at any one time, it is not advisable to include a @filter function in a recursive function definition where it can be called upon to spawn large numbers of processes.

Note: the @filter function may also be used to interface programs written in other languages with a pLucid program. The only requirement is that the the process written in the other language acts as a filter with respect to its input/output behavoiur. This means that programs written in assembly language, Pascal, Fortran, PL/1 ... can be used in conjunction with a pLucid program.

### An application of the filter function

In pLucid it is possible supply a program with input from a file via the standard input. For many applications this method (i.e. luval prog.i < filename) is not satisfactory because it forces the standard input to be the named file. One way around this problem is to use the @filter function which does not interfere with the evaluator's standard input. The following function can be included in a pLucid program. The function @file expects its argument to evaluate to a string which it takes to be the name of a file. The function then returns the stream of characters contained in the named file terminated by the eod object.

file(name) = filter( 'cat <' ^ filename, '' , 'ci')  
where filename = first name; end; This facility means that we can choose dynamically the name of the files from which we require input.

### 5.3 Runtime errors

If a pLucid program compiles then the only errors that can occur at runtime will be type clash errors or errors that occur from giving a program strange inputs. A runtime type error will produce an error message which will give the exact location in the original source (i.e file name , line number and cursor position of the error). If we run the following program: moment where

```
avg(x) = s / n
  where
    s = x fby hd s + next x;
    // We have included a deliberate mistake
    // The above definition should be
    // s = x fby s + next x;
    n = 1 fby n + 1;
  end;
```

```
a = 7 fby 2 fby 4;
```

```
moment = avg((a-m)*(a-m))
  where
    m is current m;
  end;
```

m = avg(a); end the evaluator will give use the following runtime error messages:  
Evaluation begins .....

```
output( 0) : 0 ----- Evaluation
time error on line 4, of file bug
      s = x fby hd s + next x;
.....^ arg of hd must be a list, not 7
```

```
Evaluation time error on line 4, of file bug
      s = x fby hd s + next x;
.....^ left arg of numeric binary operator or condition of wvr or upon is
?
```

```
Evaluation time error on line 2, of file bug
      avg(x) = s / n
.....^
```

... and so on Error messages like the above will appear on the screen if a program containing type clashes is evaluated. Notice the propagation of errors. It is possible to turn off error reporting (see the UNIX manual entry).

Note: if the source program is changed the program must be recompiled otherwise the runtime error messages given by the evaluator will be with reference to the source program used to produce the current .i file.

If a program has a bug in it which is not a type error, that is some form of logical error then the @filter function can be used to probe variables in much the same way as a technician uses a scope and probe to debug a piece of faulty hardware. The function @probe can be defined as follows:

probe(name,expr) = filter('tee '^filename,expr,')

where filename = first name; end; Note that in the above example the values of the suspect variable have been tapped and stored in a file. On a system that allows a screen to be divided into multiple windows then the values of the suspect variable could be displayed in one of the windows.

### The arg function

When invoking the evaluator it is often desirable to supply the evaluator with the names of one or more files as arguments. For example if we wanted to write the UNIX cat filter in pLucid we would need to supply the name of the file that we wanted to cat. To achieve this pLucid has the prefix operator @arg. The @arg operator expects its argument to evaluate to a positive integer which is used to select a string from the argument list associated with the invocation of the evaluator. To invoke the evaluator it is necessary to type the lval command which has the following syntax:

lval <option list> <filename>.i <argument list> ( See UNIX manual entry for more details). All that we need to say here is that the @arg function will select a string from the <argument list> of an evaluator invocation. If the argument of @arg is not a positive integer or there is no argument with that index then @arg yields the error object otherwise it yields the appropriate string in the argument list.

### A facility for file inclusion

A simple file inclusion facility is provided. To include a file (containing perhaps some function definitions) in another, just say

```
include "filename";
```

NOTE that any other statement on the same line as the @include is ignored.

Included files may also have @include statements, up to a level of ten.

There is also a standard inclusion directory where many useful functions are kept. In order to include files from this directory in your program, use angle brackets instead of quotes, that is, say

```
include <filename>;
```

where file name is a library file found in /ud/cscfac/faustini/lu/lib (this directory name will probably be different on your machine).

## 6 Tables and Rules

### 6.1 Tables of Operators

<b>Numeric operators</b>	<b>: !operation</b>	<b>!syntax</b>	<b>!type !of !operands</b>	<b>!type !of !result</b>
<b>!addition</b>	<b>+</b>		<b>!numeric</b>	<b>!numeric</b>
<b>!subtraction</b>	<b>-</b>		<b>!numeric</b>	<b>!numeric</b>
<b>!multiplication</b>	<b>*</b>		<b>!numeric</b>	<b>!numeric</b>
<b>!integer !division</b>	<b>div</b>		<b>!integer</b>	<b>!integer</b>
<b>!real !division</b>	<b>/</b>		<b>!numeric</b>	<b>!numeric</b>
<b>!exponentiation</b>	<b>**</b>		<b>!numeric</b>	<b>!numeric</b>

**!modulus**      **mod**      **!integer**      **!integer**  
**!numeric**      **isnumber**      **!anything**      **!boolean**  
**!less !than**      **<**      **!numeric**      **!boolean**  
**!greater !than**      **>**      **!numeric**      **!boolean**  
**!equal**      **eq**      **!numeric**      **!boolean**  
**!less !than !or !equal**      **<=**      **!numeric**      **!boolean**  
**!greater !than !or !equal**      **>=**      **!numeric**      **!boolean**  
**!not !equal**      **ne**      **!numeric**      **!boolean**  
**!sine**      **sin**      **!numeric**      **!numeric**  
**!cosine**      **cos**      **!numeric**      **!numeric**  
**!logarithm**      **log**      **!numeric**      **!numeric**  
**!tangent**      **tan**      **!numeric**      **!numeric**  
**!square !root**      **sqrt**      **!numeric**      **!numeric**  
**!absolute !value**      **abs**      **!numeric**      **!numeric**  
**!log10**      **log10**      **!numeric**      **!numeric**

**Word Operators:**

**!operation**      **!syntax**      **!operand !type**      **!type !of !result**  
**!make !a !word !out !of !a !string**      **mkword**      **!string**      **!word**  
**!recognize !a !word**      **isword**      **!anything**      **!boolean**

**Boolean operators :** **!operation**      **!syntax**      **!type !of !operands**

**!conjunction**      **and**      **!boolean**  
**!disjunction**      **or**      **!boolean**  
**!negation**      **not**      **!boolean**

**String Operators:** **!operation**      **!syntax**      **!type !of !operands**      **!type !of !result**

**!make !a !string !out !of !a !word**      **mkstring**      **!word**      **!string**  
**!string !recognition**      **isstring**      **!anything**      **!boolean**  
**!string !concatenation**      **^**      **!string**      **!string**

**!form !a !substring substr (!string,integer,integer) !string**

**!length length !anything !number**

**@List @operators :**

**!operation !syntax !operand1 !operand2 !result**

**!the !head !of !a !list hd !list -- !anything**

**!the !tail !of !a !list tl !list -- !list**

**!appending !two !lists <> !list !list !list**

**!construction !operator :: !anything !list !list**

**!is !the !list !empty? isnull !list -- !boolean**

**!is !it !an !atom !(not !a !list) isatom !anything -- !boolean**

**!is !it !a !list islist !anything -- !boolean**

**Conditional expressions :**

**if expression :**

**if !boolean !expression then !expression else !expression fi**

**nested if expression: if !boolean !expression then !expression**

**elseif !boolean !expression then !expression else ... fi**

**@case @expression :**

**case !expression of**

**!expression ! : !expression ;**

**!expression ! : !expression ;**

**!....**

**!expression ! : !expression ;**

**default : !expression ;**

**end**

**cond expression : cond**

**!boolean !expression : !expression ;**

**!boolean !expression : !expression ;**

**!....**

**!boolean !expression : !expression ;**

**default : !expression ;**

**end**

**Lucid operators : !operator !syntax !operand 1 !operand 2 !result**

**!first first !anything --- !anything**

```

!next      next  !anything  ---  !anything
!followed !by   fby    "    !anything  "
!whenever  whenever "    !boolean   "
!whenever  wvr   "    !boolean   "
!attime    attime "    !integer   "
!as !soon !as   asa    "    !boolean   "
!upon      upon  "    !boolean   "

```

**current declaration :**  
**!identifier is current !expression**

## 6.2 Associativity and Precedence Rules

### Associativity of Operators:

An infix operator is said to be 'right associative', e.g @@fby&, if for any expressions E1, E2, and E3, the expression

X fby Y fby Z is always interpreted as

X fby ( Y fby Z ) . Similarly, an infix operator is said to be 'left associative',

e.g @@asa&, if for the expressions E1, E2, and E3, the expression

!E1 asa !E2 asa !E3 is always interpreted as

( !E1 asa !E2 ) asa !E3 The following table gives the associativity of infix

operators in pLucid: !Associativity !Operators

```

!left      + , - , * , / , div , mod , or , and , ** ,
           asa , attime , whenever , wvr , upon , if then else ,
           case

```

```

!right     :: , <> , fby , ^

```

### Precedence Rules :

These are rules to avoid clogging up programs with unnecessary brackets. For example if we say that '\*' has higher precedence than '+' then an expression like @@2 + 4 \* 5& is always interpreted as @@2 + ( 4 \* 5 )&.

We list here the hierarchy of precedences amongst pLucid operators. Operators with lowest precedences are at the top of the list, and ones with highest precedences are at the bottom.

- 1 asa , upon , whenever , wvr , attime
- 2 fby
- 3 if then else fi , case , cond
- 4 :: , <>
- 5 or
- 6 and
- 7 not
- 8 eq , ne , < , <= , > , >=

- 9 + , -
- 10 \* , div , / , mod
- 11 \*\*
- 12 ^
- 13 first, next, sin, cos, tan, log, log10, hd, tl, isnull, isnumber, isatom, isword, isstring, mkword, mkstring, iserror, iseod, sqrt, abs, arg , islist

**The where-clause**

The where-clause has the lowest precedence amongst other constructs in pLucid, so if E1, E2, and E3 are expressions, then for any operators in pLucid, say fby and next, the expression

```

!E1 fby !E2 fby next !E3 where
    ...
    ...
    ...
end is always interpreted as
(!E1 fby !E2 fby next !E3) where
    ...
    ...
    ...
end
    
```

**6.3 Reserved Words:**

These identifiers are reserved as keywords if

tl	fi	current	error	eq	substr	mkword	arg	false	else	isnull	not	iserror	whenever	upon	include	isatom	log	mod	first	ne	tan	length	include	hd	sin	elseif	nil	eod	where	or	iseod	asa	mkstring	wvr	filter	true	then	isnumber	is	of	and	isstring	next	log10	abs	attime
----	----	---------	-------	----	--------	--------	-----	-------	------	--------	-----	---------	----------	------	---------	--------	-----	-----	-------	----	-----	--------	---------	----	-----	--------	-----	-----	-------	----	-------	-----	----------	-----	--------	------	------	----------	----	----	-----	----------	------	-------	-----	--------

**6 Tables and Rules**

**6.1 Tables of Operators**

Numeric operators : operation		syntax	type of operands	type of result
<b>addition</b>	<b>+</b>	<b>numeric</b>	<b>numeric</b>	<b>numeric</b>
<b>subtraction</b>	<b>-</b>	<b>numeric</b>	<b>numeric</b>	<b>numeric</b>
<b>multiplication</b>	<b>*</b>	<b>numeric</b>	<b>numeric</b>	<b>numeric</b>
<b>integer division</b>	<b>div</b>	<b>integer</b>	<b>integer</b>	<b>integer</b>
<b>real division</b>	<b>/</b>	<b>numeric</b>	<b>numeric</b>	<b>numeric</b>
<b>exponentiation</b>	<b>**</b>	<b>numeric</b>	<b>numeric</b>	<b>numeric</b>
<b>modulus</b>	<b>mod</b>	<b>integer</b>	<b>integer</b>	<b>integer</b>

numeric	isnumber	anything	boolean
less than	<	numeric	boolean
greater than	>	numeric	boolean
equal	eq	numeric	boolean
less than or equal	<=	numeric	boolean
greater than or equal	>=	numeric	boolean
not equal	ne	numeric	boolean
sin	sin	numeric	numeric
cos	cos	numeric	numeric
log	log	numeric	numeric
tan	tan	numeric	numeric
sqrt	sqrt	numeric	numeric
abs	abs	numeric	numeric
log10	log10	numeric	numeric

**Word Operators:**

operation	syntax	operand type	type of result
make a word out of a string	mkword	string	word
recognize a word	isword	anything	boolean

**Boolean operators :**

operation	syntax	type of operands
conjunction	and	boolean
disjunction	or	boolean
negation	not	boolean

**String Operators:**

operation	syntax	type of operands	type of result
make a string out of a word	mkstring	word	string
string recognition	isstring	anything	boolean
string concatenation	^	string	string
form a	string, integer, substring	substr (string, integer	string

**List operators : operation    syntax   operand1   operand2   result**

**the head of a list    hd   list    --   anything**

**the tail of a list    tl   list    --   list**

**appending two lists    <>   list    list   list**

**construction operator    ::   anything   list   list**

**is the list empty ?    isnull   list    --   boolean**

**is it an atom (not a list)   isatom   anything   --   boolean**

**Conditional expressions :**

**if expression :**

**if boolean expression then expression else expression fi**

**nested if expression: if boolean expression then expression**

**elseif boolean expression then expression else ... fi**

**case expression : case expression of**

**case 1 : expression 1 ;**

**case 2 : expression 2 ;**

**....**

**case n : expression n ;**

**default : expression ;**

**end**

**Lucid operators : operator    syntax   operand 1   operand 2   result**

**first    first   anything   ---   anything**

**next    next   anything   ---   anything**

**followed by    fby   "   anything   "**

**whenever    whenever   "   boolean   "**

**as soon as    asa   "   boolean   "**

**upon    upon   "   boolean   "**

**current declaration :**

**identifier is current expression**

## **6.2 Associativity and Precedence Rules**

*Associativity of Operators:*

An infix operator is said to be 'right associative', e.g fby, if for any expressions E1, E2, and E3, the expression

X fby Y fby Z

is always interpreted as

X fby ( Y fby Z ) .

Similarly, an infix operator is said to be 'left associative', e.g. *asa*, if for the expressions E1, E2, and E3, the expression

E1 *asa* E2 *asa* E3

is always interpreted as

( E1 *asa* E2 ) *asa* E3

The following table gives the associativity of infix operators in pLucid:

<i>Associativity</i>	<i>Operators</i>
left	+ , - , * , / , div , mod , or , and , <i>asa</i> , whenever , wvr , upon , if then else, case
right	:: , <> , fby , ^

*Precedence Rules :*

These are rules to avoid clogging up programs with unnecessary brackets.

e.g:

If we say that '\*' has higher precedence than '+' then an expression like '2 + 4 \* 5' is always interpreted as '2 + ( 4 \* 5 )'.

We list here the hierarchy of precedences amongst pLucid operators. Operators with lowest precedences are at the top of the list, and ones with highest precedences are at the bottom.

fby  
*asa* , upon , whenever , wvr  
if then else fi, case  
or  
and  
not  
eq, ne, lt, le, gt, ge, < , <= , > , >=  
+ , -  
\* , div , / , mod  
:: , <>  
first, next, sin, cos, log, hd, tl, isnull, isnumber,  
isatom, isword, isstring, mkword, mkstring,  
substr, iserror, iseod,  
^

The *where-clause* has the lowest precedence amongst other constructs in pLucid, so if E1,

E2, and E3 are expressions, then for any operators in pLucid, say fby and next, the expression

```
E1 fby E2 fby next E3 where
    ...
    ...
    ...
end
```

is always interpreted as

```
(E1 fby E2 fby next E3) where
    ...
    ...
    ...
end
```

### 6.3 Reserved Words:

These identifiers are reserved as keywords

if	hd	true
then	tl	false
else	isatom	sin
elseif	isnumber	cos
fi	isnull	log
case	nil	is
of	div	current
default	mod	eod
where	and	error
end	not	isword
first	or	isstring
next	eq	iserror
fby	ne	iseod
asa	lt	substr
whenever	le	mkstring
wvr	gt	mkword
upon	ge	arg
attime	log10	tan
length	abs	sqrt

## 7 Miscellaneous

### 7.1 Plucid grammar

We define here the pLucid syntax using the BNF formalism, where

`::=` is read as <meta variable> is defined as <meta variable> ,

`|` is read as <meta variable> or <meta variable> ,

`{ }` denotes possible repetition zero or more times of the enclosed construct .

<program> ::= <expression>

<expression> ::= <constant>  
| <identifier>  
| error  
| eod  
| <prefix operator> <expression>  
| <expression> <infix operator> <expression>  
| filter ( <expression> , <expression> , <expression> )  
| substr ( <expression> , <expression> , <expression> )  
| length <expression>  
| arg <expression>  
| <list expression>  
| <if expression>  
| <case expression>  
| <cond expression>  
| <function call>  
| <where clause>

<constant> ::= <numeric constant>  
| <word constant>  
| <string constant>  
| <list constant>

<numeric constant> ::= <integer constant>  
| <real constant>

<integer constant> ::= <digit> { <digit> }  
| <n-sign> <integer constant>

<real constant> ::= <integer constant> . { <digit> }

<n-sign> ::= ~

<word constant> ::= <quote> <word constant less the quotes> <quote>

<word constant less the quotes> ::= <letter> { <alphanumeric> }  
| <sign> { <sign> }  
| <bracket>  
| <period>  
| <separator>  
| <quote>

<sign> ::= + | - | \* | \$ | & | = | < | > | : | # | ^ <quote> ::= " <bracket> ::= ( | ) | [% | %] |  
(% | %) <period> ::= . <separator> ::= , | ;

<string constant> ::= '{<character>}'

<character> ::= <Any ASCII character except the closing single quote ' >

<list constant> ::= nil | []  
| [ {<list constant element> } ]

<list constant element> ::= <numeric constant>  
| <word constant less the quotes>  
| <string constant>  
| <list constant>

<alphanumeric> ::= <digit> | <letter>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M  
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z  
| a | b | c | d | e | f | g | h | i | j | k | l | m  
| n | o | p | q | r | s | t | u | v | w | x | y | z

<identifier> ::= <letter> { <alphanumeric> }

<prefix operator> ::= <p-numeric operator>  
| <p-word operator>  
| <p-string operator>  
| <p-list operator>  
| <p-lucid operator>  
| <p-special operator>

<p-numeric operator> ::= sin | cos | tan | sqrt | abs | log10 | log | isnumber

<p-word operator> ::= isword | not | mkstring

<p-string operator> ::= isstring | mkword

<p-list operator> ::= hd | tl | isatom | isnull | islist

<p-lucid operator> ::= first | next

<p-special operator> ::= iseod | iserror

<infix operator> ::= <i-numeric operator>  
| <i-word operator>  
| <i-string operator>  
| <i-list operator>  
| <i-lucid operator>

<i-numeric operator> ::= + | - | \*\* | \* | div | mod | /  
| eq | ne | <= | < | > | >=

<i-word operator> ::= and | or | eq | ne

<i-string operator> ::= ^ | eq | ne

<i-list operator> ::= < | :: | eq | ne

<i-lucid operator> ::= fby | whenever | wvr | upon | asa | attime

<list expression> ::= [%%]  
| [% {<expressions list>} %]



```
+-->--- !identifier ----->-----+
|
+-->--- error ----->-----+
|
+-->--- eod ----->-----+
|
+-->--- !prefix !operator --->--- !expression ----->-----+
|
+-->--- !expression -->--- !infix !operator -->--- !expression -->---+
|
+-->--- !list !expression ----->-----+
|
+-->--- !if !expression ----->-----+
|
+-->--- !case !expression ----->-----+
|
+-->--- !cond !expression ----->-----+
|
+-->--- !function !call ----->-----+
|
+-->--- !where !clause ----->-----+
|
+-->--- substr -->--- ( -->--- !3expression -->--- ) ----->---+
|
+-->--- filter -->--- ( -->--- !3expression -->--- ) ----->---+
|
+-->--- length ----->--- !expression ----->-----+
|
+-->--- arg ----->--- !expression ----->-----+ !constant :
```

```
-->---+-->--- !numeric !constant --->---+--->---
|
+--->--- !word !constant --->-----+
|
+-->--- !string !constant --->-----+
|
+--->--- !list !constant --->-----+ !numeric !constant :
```

```
-->---+-->--- !integer !constant -->---+--->---
|
+--->--- !real !constant --->-----+ !integer !constant :
```

```
-->---+-->---+-----+--->---+--->--- !digit ----+--->---
|
+-->--- ~ -->---+ +---<-----+ !real !constant :
```

```
-->--- !integer !constant -->--- . -->---+--->--- !digit ----+--->---
|
+---<-----+ !word !constant :
```

```
-->--- " --->--- !letter -->---+----->-----+--->--- " -->---
|
| +--- !alphanumeric --<---+ |
|
```

```

+-->+-->-- !sign --->+----->-----+
| | | | | | | | | | | | | | | | | |
| +-----<-----+ | |
| | | | | | | | | | | | | | | | | |
+-->+-----+-----+-----+-----+-----+ |
| | | | | | | | | | | | | | | | | |
| ( ) ( % % ) [ % % ] |
| | | | | | | | | | | | | | | | | |
| +-----+-----+-----+-----+-----+>-----+
| | | | | | | | | | | | | | | | | |
+--->+-----+-----+-----+-----+ |
| | | | | | | | | | | | | | | | | |
| : , ; " |
| | | | | | | | | | | | | | | | | |
+-----+-----+-----+-----+>-----+ !sign:

```

```

-->+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| | | | | | | | | | | | | | | | | |
+ - * / & = < > : # ^
| | | | | | | | | | | | | | | | | |
+-----+-----+-----+-----+-----+-----+-----+>-- !string !constant :

```

```

-->-- ' -->+----->-----+-----+-----+ ' -->--
| |
+--- !ascii !character !except --<--+
!the !closing !single !quote !list !constant :

```

```

-->+-->-- nil ----->-----+-----+-----+>--
| |
| |
+-->-- [ ----->-----+-----+-----+ ] ----+
| |
+---<-- !list !element --<--+ !list !element :

```

```

-->+----->-- !numeric !constant -->+----->--
| |
+--->-- !word !constant --->+---+
| !less !the !quotes |
| |
+--->-- !string !constant --->+---+
| |
+--->-- !list !constant ----->+---+ !alphanumeric : !letter:

```

```

-->+----->-- !digit --->+----->-- -->+-----+-----+-----+-----+
| | | | | | | | | | | | | | | | | |
+--->-- !letter -->+---+ ANY UPPER CASE OR
LOWER CASE LETTER
| | | | | | | | | | | | | | | | | |
+-----+-----+-----+-----+>-- !digit:

```

```

-->+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| | | | | | | | | | | | | | | | | |
0 1 2 3 4 5 6 7 8 9
| | | | | | | | | | | | | | | | | |

```

```
+----->-- !identifier:

-->-- !letter -->+----->-----+-->--
      |           |
      |           |
      +--- !alphanumeric --<--+ !prefix !operator :

-->+-->-- !p-numeric !operator -->+-->--
      |           |
      +-->-- !p-word !operator ---->----+
      |           |
      +-->-- !p-string !operator -->----+
      |           |
      +-->-- !p-list !operator ---->----+
      |           |
      +-->-- !p-lucid !operator ---->----+
      |           |
      +-->-- !p-special !operator -->--+ !p-numeric !operator :

-->+-->-- sin ---->----+-->--
      |           |
      +-->-- cos ---->----+
      |           |
      +-->-- tan ---->----+
      |           |
      +-->-- sqrt ---->----+
      |           |
      +-->-- abs ---->----+
      |           |
      +-->-- log ---->----+
      |           |
      +-->-- log10 ---->----+
      |           |
      +-->-- isnumber -->--+ !p-word !operator:

-->+-->-- not ---->----+-->--
      |           |
      |           |
      +-->-- isword ---->----+
      |           |
      |           |
      +-->-- mkstring -->--+ !p-string !operator:

-->+-->-- isstring -->+-->--
      |           |
      +-->-- mkword -->--+ !p-list !operator :           !p-lucid !operator:

-->+-->-- hd ---->----+-->--           -->+-->-- first -->+-->--
      |           |           |           |
      +-->-- tl ---->----+           +-->-- next ---->----+
      |           |
      +-->-- islist -->--+
      |           |
      +-->-- isatom -->--+
```

```

|
+-->-- isnull --->--+ !p-special !operator

--->---+--->--- iseod ---->+---->---
|
+--->-- iserror --->--+ !infix !operator :

--->---+--->--- !i-numeric !operator --->+---->---
|
+-->-- !i-word !operator ---->+----+
|
+-->-- !i-string !operator --->+----+
|
+-->-- !i-list !operator ---->+----+
|
+-->-- !i-lucid !operator ---->+----+ !i-numeric !operator :           !i-word !operator:

--->---+--->--- + --->+---->--- --->+---->--- or --->+---->---
|
+--->--- - --->+----+           +--->--- and --->+----+
|
+--->--- * --->+----+           +--->--- eq --->+----+
|
+--->--- div --->+----+           +--->--- ne --->+----+
|
+--->--- mod --->+----+
|
+--->--- / --->+----+
|
|           !i-string !operator:
+--->--- ** --->+----+
|
|           --->+---->--- ^ --->+---->---
+--->--- eq --->+----+           |           |
|           |           +--->--- eq --->+----+
+--->--- ne --->+----+           |           |
|           |           +--->--- ne --->+----+
+--->--- < --->+----+
|           |           !i-list !operator:
+--->--- > --->+----+
|           |           -->+---->--- < --->+---->---
+--->--- >= --->+----+           |           |
|           |           +--->--- :: --->+----+
+--->--- <= --->+----+           |           |
|           |           +--->--- eq --->+----+
|           |           |
+--->--- ne --->+----+ !i-lucid !operator :

--->---+--->--- fby ---->+---->+---->---
|
+-->-- whenever --->--+
|
+-->-- wvr ---->+---->+----+
|
+-->-- upon ---->+---->+----+
|

```

```

+-->-- asa ----->-----+
|           |
+-->-- attime --->-----+ !3expression

```

--> !expression --> , --> !expression --> , --> !expression --> !list !expression:

```

+---->----->-----+
|                               | -->-- [% -->----->-- !expression !item
----+>----->-- [%] ->
|                               |
+-----<-- , -----<-----+ !expression !item:

```

```

-->----->-- !expression ---->----->--
|           |
+-->-- !list !expression -->-- !if !expression :

```

-->-- if -->-- !expression -->-- then -->-- !expression -->-- !endif -->-- !endif:

```

-->-->-- else -->-- !expression -->-- fi ----->----->-----+>--
|                               |
|                               |
V                               |
elseif                          ^
|                               |
|                               |
+-->-- !expression -->-- then -->-- !expression -->-- !endif ---+ !case !expression:

```

-->-- case -->-- !expression -->-- of -->-- !case !body -->-- end -->-- !case !body:

```

-->+----->-----+>-----+>-----+
|           |           |
+--<-- ; --<-- !expression --<-- : --<-- !expression --<-- + |
|                               |
V                               |
--<-- end --<-- ; --<-- !expression --<-- : --<-- default --<-- + !cond !expression:

```

-->-- cond ----->----->----- !cond !body -->-- end -->-- !cond !body:

```

-->+----->-----+>-----+>-----+
|           |           |
+--<-- ; --<-- !expression --<-- : --<-- !expression --<-- + |
|                               |
V                               |
--<-- end --<-- ; --<-- !expression --<-- : --<-- default --<-- + !function !call :

```

-->-- !identifier -->-- ( -->-- !actuals !list -->-- ) -->-- !actuals !list :

```

-->-- !expression ---+----->-----+>-----+>--
|           |
+--- !expression --<-- , --<-- + !where !clause :

```

-->-- !expression -->-- where -->-- !body -->-- end -->-- !body :

```
--->+----->-----+>+----->-----+>--
|           | |           |
+---<-- !declaration --<--+ +---<-- !definition ---<--+ !declaration :

-->-- !identifier -->-- is current -->-- !expression -->-- ; -->-- !definition :

--->+----->-- !simple !definition ----->+----->--
|           |
+--->-- !function !definition --->--+ !simple !definition :

--->-- !identifier --->-- = --->-- !expression --->-- ; --->-- !function !definition :

-->-- !identifier -->-- ( -->+----->-- !identifier -->+----->-- ) -->--+
|           |           |
+---<----- , ---<-----+ |
                        V
                        |
---<-- ; --<-- !expression ---<-- = ---<--+
```

### 7.3 Programming Example

The following is an example of how a large program would be organised in pLucid. The example is of a screen editor and it illustrates many of the features of the pLucid system.

The macro used to run the screen editor trap 'reset' 2 stty -echo cbreak; luval -t10 -c -s screen.i S1 2>prompts; stty echo -cbreak

```
// viscid - a vi-like-screen-editor-in-Lucid
```

```
chargen(cdecode(C))
```

```
where
```

```
cddecode(C) = //turn raw stream of chars into stream of commands cmd
```

```
where
```

```
include "cdecoder";
```

```
end;
```

```
chargen(cmd) = //generate the control chars from the commands chars
```

```
where
```

```
include "chargen";
```

```
include "workingtext"; //variables representing current state of working text
```

```
include "escseqs"; // control sequences for the pt100b
```

```
include "strings"; // useful string operations
```

```
include "gather"; // functions for turning lists into streams
```

```
// and vice versa
```

```
end;
```

```
end
```

The file @cdecoder&.

```
namedcmd =                //the command named by C
  case C of
    'i':"beginsert"; //begin inserting text
    'h':"left";      //move cursor left
    ' ': "right";   //move cursor right
    'k':"up";       //move cursor up one line
    'j':"down";     //move cursor down one line
    'o':"open";     //open a line below the cursor
    'x':"chdel";    //delete character
    'w':"write";    //write to file
    'X':"linedel";  //delete current line
    'D':"ldel";    //delete line to right of cursor
    'Z':"finish";   //write and exit
    default:"error";
  end;

cmd = namedcmd fby
  case cmd of
    "beginsert": if next C eq '33' then "endinsert"
                  else "moreinsert" fi;
    "moreinsert": if next C eq '33' then "endinsert"
                  else "moreinsert" fi;
    "open"      : "moreinsert";
    "finish"    : "quit";
    default     : next namedcmd;
  end;
```

The file @chargen&. //generate the stream of screen control characters

```
chars = //stream of strings sent to terminal
CLEAR^HOME^lineconc(initialtext)^HOME
fby
case cmd of
"moreinsert": C^rhs^back(length(rhs));
"right":RIGHT;
"left":LEFT;
"up":UP;
"down":DOWN;
"chdel":stl(rhs)^` ``back(length(rhs));
"open": DSCROLL(lineno+1);
"ldel":space(length(rhs))^POSITION(lineno,colno-1);
"linedel":USCROLL(lineno) ^ if llines eq [] then UP else `` fi;
"write": if iseod output then BEEP else BEEP fi;
"quit":if iseod output then eod else eod fi;
"close":NULL;
"finish":BOTTOM;
default: ``;
end;

lineno = length(ulines)+1;

colno = 1 fby case cmd of
"left": colno-1;
"right": colno+1;
"endinsert":colno+length(insert);
"ldel":colno-1;
"open":1;
"linedel": if llines eq [] then lineno-1 else lineno fi;
default:colno;
end;

insert = `` fby if cmd eq "moreinsert" then insert^C else `` fi;

rhs = srear(ccline,colno-1) ;

initialtext = if filename eq `` then [] else linesof(filename) fi;

filename = arg(1);

output = first filter(`cat >`^filename,TEXT fby eod,`s`)
where TEXT is current text;end;

text = lineconc(ulines<>[%ccline%]<>llines);
```

The file @workingtext&. //the definitions of the variables which represent the //the current working text

ulines = //list of lines above the cursor

```
[] fby
case cmd of
  "up":tl(ulines);
  "down":cline :: ulines;
  "open":cline :: ulines;
  "linedel":if llines ne [] then ulines else tl(ulines) fi;
  default : ulines;
end;
```

llines = //list of lines below the cursor

```
tl(initialtext)
fby
case cmd of
  "up": cline :: llines;
  "down":tl(llines);
  "linedel": if llines ne [] then tl(llines) else [] fi;
  default : llines;
end;
```

cline = // the line the cursor is on

```
hd(initialtext)
fby case cmd of
  "up": hd(ulines);
  "down":hd(llines);
  "endinsert":sfront(cline,colno-1)^insert^srear(cline,colno-1);
  "chdel":sfront(cline,colno-1)^srear(cline,colno);
  "ldel":sfront(cline,colno-1);
  "linedel":if llines ne [] then hd(llines) else hd(ulines) fi;
  "open":“”;
  default:cline;
end;
```

The file @escseqs&. //the "escape sequences" for the pt100b

BEEP = '\7';

CLEAR = '\33[2J'; //clear the screen

HOME = '\33[H'; // send cursor to upper right hand corner

RIGHT = '\33[C'; //move right

LEFT = '\b'; //move left

UP = '\33M'; //up one row

DOWN = '\33D'; //down one row

BOTTOM = '\33[24;1H'; //go to bottom left hand corner

DSCROLL(i) = // control chars to scroll down lines i thru 24

// cursor left at beginning of line i (now blank)

'\33[' ^ n ^ ';' ^ 24r '\33[' ^ n ^ ';' ^ 1H '\33M '\33[1;24r '\33[' ^ n ^ ';' ^ 1H'

where

n = numeral(i);

end;

USCROLL(i) = // control chars to scroll up lines i thru 24

// cursor at beginning of line i

'\33[' ^ n ^ ';' ^ 24r '\33[24;1H '\33D '\33[1;24r '\33[' ^ n ^ ';' ^ 1H'

where

n = numeral(i);

end;

POSITION(r,c) = // move cursor to the rth row and cth column

'\33[' ^ numeral(r) ^ ';' ^ numeral(c) ^ 'H';



The file @gather&. //filters for transforming finite lists into streams and vice versa,  
//streams 'terminated' by eod

```
element(x) = hd(tx) //the stream of elements of the list x
  where
    tx = if first x eq [] then eod else first x fi fby
        if tl(tx) ne [] then tl(tx) else eod fi;
  end;
```

```
gather(x) = first L //a list of the values in the stream x
  where
    L = if iseod x then [] else x :: next L fi;
  end;
```

```
linkup(x) = first L //concatenate a stream of lists
  where
    L = if iseod x then [] else x <> next L fi;
  end;
```

```
ever(p) = first q //true if p is ever true
  where
    q = if iseod p then false elseif
        p then true else
        next q fi;
  end;
```