PROSPECTRA
an ESPRIT Project

Language Reference Manual for relational TrafoLa-H

Version 1.5

Reinhold Heckmann

Universität des Saarlandes
6600 Saarbrücken
Bundesrepublik Deutschland

PROSPECTRA Deliverable Item S.1.6-DI-14.0

1988-09-05

Distribution: public

ABSTRACT

Relational TrafoLa-H is a language where each expression may have several or even infinitely many values, and each 'function' may produce many results. Its functional predecessor was developed to ease the specification of complex tree transformations starting from a usual functional language similar to HOPE or ML. The non-determinism resulting from its complex patterns was the reason to turn the functional language into a relational one.

Public

Reinhold Heckmann
Universität des Saarlandes

in the Project

PROgram development by
SPECification and
TRAnsformation

sponsored by the

Commission of the European Communities

under the

European
Strategic
Programme for
Research and development in
Information
Technology

Project Ref. No. 390

Table of contents:

Introduction

This manual refers to the backtracking i.e. relational dialect of TrafoLa-H whose interpreter is written in ML. It must not be confused with the functional version implemented in SSL.

The described language has version number 1.5. Version 1.5 differs in many details from its predecessor, version 1.4b, whose interpreter was distributed in June 1988 at Madrid. The language was extended by user defined macros and user definable error messages. Some new operators were introduced, and the set of predefined functions was completely revised and extended.

We shall first define the syntax of the language exactly, then describe its semantics informally.

1. Syntax of relational TrafoLa-H

1.1. Tokens

In this section, we describe the internal structure of the tokens TrafoLa input consists of. The tokens are recognized by a scanner generated in C with a longest match strategy.

The structure of tokens is given by regular expressions where ‘|’ means ‘or’, ‘*’ denotes the Kleene star, and parentheses ‘(’ ‘)’ are used for grouping. Characters are enclosed in quotes. The special expression ‘up to s’ where s is a sequence of characters means ‘any string not containing s followed by an occurrence of s’.

Auxiliary regular expressions:

```
letter      =  ‘a’ |...| ‘z’ | ‘A’ |...| ‘Z’ | ‘_’ | ‘’’
digit1      =  ‘1’ |...| ‘9’                            (* not ‘0’ *)
digit       =  ‘0’ | digit1
alpha       =  letter | digit | quote
```

Tokens:

```
<num>       =  digit  digit *                           (* numbers *)
<hole>      =  ‘@’  digit1 digit * | ‘@’                (* @ == @1 *)
<string>    =  ‘"’ up to ‘"’                            (* strings *)
<const>     =  letter  alpha *                          (* constructors *)
<variable>  =  letter  alpha *                          (* variables *)
<comment>   =  ‘/*’ up to ‘*/’                          (* comments *)
<space>     =  ‘ ’ | TAB | NEWLINE
```

Space and comments are ignored. Between two adjacent terminals, there may be an arbitrary amount of them. Sometimes, space or comments are needed as separators, e.g. in ‘fac 5’.

Remarks:

1) The case of letters is significant, i.e. ‘a’ and ‘A’ are not the same.
2) The length of tokens is not restricted. All characters are significant.
3) Comments may contain newlines, but cannot be nested.
4) Although newlines and control characters are lexically allowed inside strings, they must not be used. Back-slashes must be doubled. There is no means to include ‘"’ into strings (see also User Manual, section 7).
5) Although constructors and variables are different token classes, they are lexically equal. Initially, all identifiers are variables. Variables may be turned into constructors and vice versa by special declarations.
6) The maximum line length is 255.

1.2. Grammar

The syntax of relational TrafoLa-H is given by a context free grammar in Backus-Naur form. In this grammar, non-terminal names are enclosed by angular brackets ‘<’ ‘>’, and keywords and operator symbols by double quotes. In a TrafoLa-H program, keywords may be written completely lower case or completely upper case, e.g. ‘if’ or ‘IF’, but not ‘If’. Considered lexically, all keywords are variables, but they are reserved symbols; declarations such as ‘dec if = 5’ are syntactically illegal.

For reasons of simplicity, only the lower case form of keywords is mentioned in the following grammar.

Meta-symbols of Backus-Naur form:

produces	:
or	|
repetition (0 or more times)	{ }
optional part	[ ]
Parentheses	( )
Comments to syntax	(* ... *)

Productions:

```
<Clause>           :  <DecT>          ";"           (* Declaration at top level *)
                   |  <Expression>    ";"           (* Expression to be evaluated *)

(* Declarations at top level *)
<DecT>             :  "dec"   <NDefList>            (* non-recursive definition list *)
                   |  "rec"   <RDefList>            (* recursive definition list *)
                   |  "con"   <NameList>            (* the names are turned into constructors *)
                   |  "var"   <NameList>            (* the names are turned into variables *)

(* Declarations in ‘let’ expressions; here ‘dec’ is optional.  ‘con’ and ‘var’ are not allowed. *)
<DecL>             :  ["dec"]  <NDefList>           (* non-recursive definition list *)
                   |  "rec"   <RDefList>            (* recursive definition list *)

<NameList>         :  <Name>  { ","  <Name> }
<Name>             :  <variable>  |  <constructor>

<NDefList>         :  <NDef>  { "and"  <NDef> }
<NDef>             :  <ClosedPattern>  "="  <Expression>
                   |  <MacroPattern>  "="  <Expression>
                   |  <FunctionBinder>

<RDefList>         :  <RDef>  { "and"  <RDef> }
<RDef>             :  <FunctionBinder>              (* recursive definitions are more restricted *)

<FunctionBinder>   :  <variable>  <ArgList>  "="  <Expression>

<ArgList>          :  {  <GenClosedPattern>  |  <MacroPattern>  }

(* A function binder contains a special syntax for functions:
‘F x1 ... xn = e’   is equivalent to   ‘F = { x1 => { ... { xn => e } ... }’

Closed patterns are a special form of syntactically closed patterns e.g. ‘(a . b)’, ‘[a, b]’.  Using parentheses,
each pattern may be closed.  Closed patterns do not comprise variables.  ‘GenClosedPattern’ means ‘gener-
alized closed pattern’.  They are more general and comprise variables. *)

<MacroPattern>     :  "$"  <variable>  |  "$"  "_"

<Pattern>          :  <Pattern>  <Binop>  <Pattern>
                   |  <Pattern>  "where"  <Expression>
                   |  <GenClosedPattern>
(* The binary operators, their priority and associativity are described in section 1.3 *)

<GenClosedPattern> :  <SPattern>  [ "+" | "*" ]                     (* uniform sequence *)

<SPattern>         :  <variable>                                    (* binding variable *)
```

```
                   |  <ClosedPattern>

<ClosedPattern>    :  "!"  <SPattern>                              (* not *)
                   |  "{"  <Pattern>  "}"                          (* unbind *)
                   |  "<"  <Pattern>  ">"                          (* cut *)
                   (* <p> matches iff p matches, but produces only one environment. *)
                   |  "["  "]"                                     (* empty sequence *)
                   |  "["  <Pattern>  { ","  <Pattern> }  "]"      (* sequence *)
                   |  "%"  <variable>                              (* importing variable *)
                   (* %A matches v, iff the value bound to A equals v. *)
                   |  <basic-value>                                (* number, hole, etc. *)
                   |  <basic-class>
                   |  "("  <Pattern>  ")"
```

(* Basic classes provide means to select special classes of values. Lexically, they are denoted by keywords. Remember that keywords may be written totally in lower case as well as totally in upper case, e.g. ‘tree’ and ‘TREE’. Semantically, some classes could be replaced by compound patterns; these are only comfortable abbreviations. Others cannot be replaced. *)

```
<basic-class>      :  "_"           (* matches all values                                     *)
                   |  "tree"        (* matches trees;           ==  _ _                       *)
                   |  "seq"         (* matches sequences;       ==  _ *                       *)
                   |  "const"       (* matches constructors                                   *)
                   |  "bool"        (* matches booleans;        ==  false | true              *)
                   |  "num"         (* matches numbers;         ==  0 | 1 | -1 | 2 | ...      *)
                   |  "hole"        (* matches holes;           ==  @1 | @2 | ...             *)
                   |  "string"      (* matches strings;         ==  "" | "a" | "aa" | ...     *)
                   |  "char"        (* matches strings of length 1                            *)
                   |  "fun"         (* matches all functional values                          *)
                   |  "mac"         (* matches all macros (lazy functions)                    *)
                   |  "stream"      (* matches all streams                                    *)
                   |  "<<"  ">>"    (* matches the empty stream only                          *)
                   |  "nil"         (* matches ‘[ ]’, ‘""’, and the empty stream              *)

<Expression>       :  <SExp>  <Binop>  <SExp>                  (* Binary operators see below *)
<SExp>             :  "-"  <SExp>
                   |  "fail"                                       (* no result *)
                   |  <variable>
                   |  <basic-value>
                   |  "["  "]"                                     (* empty sequence *)
                   |  "["  <Expression> { ","  <Expression> } "]"  (* sequence *)
                   |  "case"  <Expression>  "of"  <Expression>
                   |  "if"  <Expression>  "then"  <Expression>  [ "else"  <Expression> ]
                   |  "let"  <DefL>  { ";"  <DefL> }  "in"  <Expression>
                   |  "matches"  <ClosedPattern>
                   |  "{"   <FunOrMacro>   "}"
                   |  "<<"  [ <Expression> ]  ">>"             (* frozen evaluation *)
(* <<e>> evaluates to a stream corresponding to the frozen evaluation of e in the actual environment. *)
                   |  "("  <Binop>         ")"                 (* prefix form of binary operators *)
                   |  "("  <Expression>    ")"

<FunOrMacro>       :  <MacroPattern>  "=>"  <Expression>       (* macro *)
                   |  <F1>  {  "\#"  <F1>  }                   (* function definition by cases *)

<F1>               :  <F2>  {  "#"  <F2>  }

<F2>               :  <Pattern>  "=>"  <Expression>            (* abstraction *)
```

<basic-value>	: <bool> | <num> | <const> | <hole> | <string>	
<bool>	: "false" | "true"	(* Booleans *)

'<num>', '<const>' etc. are token classes as introduced in the previous section.

The syntax of expressions contains some purely syntactic abbreviations that we shall not consider any further.

abbreviation	expanded form	
case E of F	F E	(* application *)
if C then T	if C then T else fail	
let D1; ... ; Dn in E	let D1 in ... let Dn in E	
let B in E	let dec B in E	
matches P	{ P => true \# _ => false }	
<< >>	<< fail >>	

1.3. Binary operators

Binary operators occur in patterns (p1 op p2), in expressions (e1 op e2), and in prefix form ((op)). The prefix form may be used to derive functional values from the operator:

(+) 2 3	is equivalent with	2 + 3
Fold (+) 0 [1, 2, 3]	evaluates to	6

All operators have prefix forms except 'where', ',', '\#', '#', and '=>' that are not real operators, the invisible operator of application, and unary minus since '(-)' denotes binary minus in prefix form.

We now give a table of all binary operators ordered by increasing priority. The symbols ',', '\#', '#', and '=>' are included here for completeness although they are not real operators, but binary syntactic constructors occurring only in special contexts.

Column 1: lexical form of the operator
Column 2: associativity:
- L: associates to the left
- R: associates to the right
- A: operator is conceptually associative
- N: 'e1 op e2 op e3' is not allowed

Column 3: may occur in patterns (P), or in expressions (E), or only between braces ({E})
Column 4: meaning resp. remark
Column 5: intended type of operand(s) and/or results

1	2	3	4	5
,	A	PE	separator in sequences (occurring only between '[' ']')	
\#	A	{E}	function definition by cases with decreasing priority	
#	A	{E}	function definition by cases with equal priority	
=>	N	{E}	abstraction	pattern => expression
\$	A	E	failure switch	any type
$	A	E	non-deterministic choice	any type
\	A	PE	biased union	functions and macros
|	A	PE	union	functions and macros
where	N	P	restriction of a pattern by a Boolean expression	
& :	A	PE	intersection	any type
o	A	E	functional composition	$(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$
\\\|	A	E	lazy sequential "or"	bool $\rightarrow$ bool $\rightarrow$ bool
||	A	E	strict Boolean "or"	bool $\rightarrow$ bool $\rightarrow$ bool
\\&	A	E	lazy sequential "and"	bool $\rightarrow$ bool $\rightarrow$ bool
&&	A	E	strict Boolean "and"	bool $\rightarrow$ bool $\rightarrow$ bool
==	N	E	equality	any types to bool

!=	N	E	inequality	any types to bool
<	N	E	less than	num, bool, hole, or string to bool
>	N	E	greater than	num, bool, hole, or string to bool
<=	N	E	less or equal	num, bool, hole, or string to bool
>=	N	E	greater or equal	num, bool, hole, or string to bool
^	L	PE	insertion	any type
.	A	PE	concatenation	sequences, strings, or streams
::	R	PE	"cons"	for sequences, strings, and streams
+:	R	PE	insert into sequence	any $\rightarrow$ seq $\rightarrow$ seq
+	A	E	addition	num $\rightarrow$ num $\rightarrow$ num
-	L	E	subtraction	num $\rightarrow$ num $\rightarrow$ num
*	A	E	multiplication	num $\rightarrow$ num $\rightarrow$ num
/	L	E	division with failure	num $\rightarrow$ num $\rightarrow$ num
div	L	E	integer division	num $\rightarrow$ num $\rightarrow$ num
mod	L	E	modulus	num $\rightarrow$ num $\rightarrow$ num
-		E	unary minus	num $\rightarrow$ num
	*	PE	application	$(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$

‘&’ and ‘:’ are two completely equivalent forms of the intersection operator.

The operator of application is invisible. In expressions, application associates to the left; whereas in patterns, it associates to the right. This is because application in patterns only occurs in the restricted form of applying a constructor to a value to form a tree.

Note that the token ‘$’ has two completely different meanings: it denotes the binary operator of non-deterministic choice in expressions, and it also serves as prefix marker of macro patterns.

The meaning of all these operators will be explained in section 2.4.

1.4. Ambiguities

The grammar specified above is highly ambiguous. The ambiguities are resolved along the following strategies:

1) Ambiguities resulting from infix operators in patterns and expressions are resolved by looking at their priorities and associativities, e.g.
   a + b * c $\rightarrow$ a + (b * c)
   a b c $\rightarrow$ (a b) c
2) ‘let’, ‘case’, and ‘if’ expressions are not closed to the right. The general rule is that all patterns and expressions extend to the right as much as possible, e.g.
   let a = 4 in a + a $\rightarrow$ let a = 4 in (a + a)
   if c1 then if c2 then t else e $\rightarrow$ if c1 then (if c2 then t else e)
3) In doubtful expressions, ‘-’ is always parsed as binary, not unary. Thus, if ‘f’ denotes a function from integers, you must write ‘f (-1)’ to apply it to ‘-1’. ‘f -1’ is parsed as difference between the function ‘f’ and the number ‘1’ and gives a run time type error. But ‘-2 * -3’ is parsed as ‘(-2) * (-3)’ since ‘-’ cannot be binary here.

   Unary ‘-’ is not parsed as function; its application has less precedence than function application such that ‘- f x’ means ‘- (f x)’.

2. Semantics of relational TrafoLa-H

We shall introduce the semantics only informally, but with many examples. An exact definition will follow in a separate paper.

One might imagine various levels of abstraction for the semantics of TrafoLa-H:

1) Set semantics: neither the order nor the multiplicity of the values of an expression is specified, only the

set of values.

2) Multi-set semantics: whereas the order of the results is not specified, their multiplicity is considered important.
3) Stream semantics: the values of an expression are ordered to a finite or infinite stream.
4) Operational semantics: the order of evaluation in time and/or the location of objects in the physical storage are specified in addition to the abstract semantics.
5) Concrete semantics of an implementation on a specific machine with machine dependent effects such as the maximum size of integers and the character code.

The values of an expression are presented in a specific order to the user by the TrafoLa-H interpreter. But often, the user is not interested in this order, and thinks in terms of the set or multi-set semantics.

Unfortunately, there is no self-contained set or multi-set semantics of TrafoLa-H because of the stream forming operator and its derivatives e.g. the ‘cut’ operator. Nevertheless, we shall introduce a multi-set semantics of TrafoLa-H first and delay the discussion of the order to section 2.5.

Although TrafoLa-H is claimed to be referentially transparent, there are a few situations where side effects become relevant. These situations and also the dependences on the concrete implementation are indicated during the description below.

2.1. Values and multi-values

In relational TrafoLa-H, several classes of values occur as values of expressions and results of functions:

- basic values (Booleans, numbers, constructors, holes, and strings)
- structural values (sequences and trees)
- abstract values (functions, macros, and streams)
- pseudo values (not yet evaluated expressions)
- error values (exceptional values carrying a string, the error message)

Expressions evaluate to a multi-set of values that may be empty, contain one element, or a finite or infinite number of elements. Such multi-sets of values are also called multi-values. They are displayed interactively in some order (see 2.5).

2.1.1. Basic values

Basic values are stored and handled as atomic entities although there are means to partition them into subunits e.g. the characters of a string.

Booleans: There are only two Booleans ‘false’ resp. ‘true’.

Numbers are integers 0, 1, 2, etc., and -1, -2, etc.
The size of numbers is bounded by the underlying ML system and is likely to depend directly on the hardware. On SUN work stations, the maximal number seems to be 2^{30} = 1 073 741 824.

Holes are objects @1 @2 @3 etc., i.e. there is a one-to-one correspondence between holes and integers greater than 0. The first hole ‘@1’ is often abbreviated to ‘@’.

Constructors are defined by ‘con’ declarations on top level and serve to label the root of a tree.

Strings are (almost) arbitrary sequences of characters enclosed by double quotes ‘"’:
"" "PROSPECTRA" "This is a string."

A back-slash ‘\’ must be entered as doubled back-slash ‘\\’. It is displayed as single back-slash e.g.

Input: "abc\\\\def" → *Output:* "abc\\def"
Input: Len "a\\\\b" → *Output:* 3

There is no possibility to include a double quote into a string.

Among the predefined functions, there are operations on basic values and some conversions between them.

2.1.2. Structural values

Structural values are the main subject of pattern matching. They may be decomposed by patterns into their constituents, and created by corresponding expressions.

Sequences consist of an ordered finite collection of other values: [v_1, ..., v_n].

Examples: [ ] [1] [[ ]] [1, "a", true] [1, 2, 3, 4, 5, 6, 7]

Trees consist of a constructor denoting the root label of the tree, and an arbitrary value denoting the children list. This value will mostly be a sequence.

Examples: con add, op, a, b, c;
add [1, 2] op true a (b c)

2.1.3. Abstract values

Abstract values can neither be entered directly, nor be displayed completely. They may be obtained as result of expressions, e.g. functions as result of pattern abstractions or functional forms.

Functions and **macros** are objects that may be applied to a value resp. a multi-value and then produce a multi-value (see 2.3).

Macros are new and were not contained in version 1.4.

Streams are created by freezing the evaluation of an expression (<<e>>), and thus catching a multi-value (the result of this evaluation) into one value. They may be considered as finite or infinite lazy sequences.

Semantically, there are three kinds of streams: **finite** streams consisting of a finite number of items with definitively known end, **infinite** streams consisting of a countably infinite number of items, and **open** streams where a finite number of items may be produced without problems, but the attempt to produce another one leads to a non-terminating computation. The last kind is the unpleasant one.

A special finite stream is the empty one, and a special open stream is the bottom stream where even the computation of the first item does not terminate.

2.1.4. Pseudo values

Pseudo values are never results of expressions or subexpressions, thus there are no operations for them. They are bound to variables by recursive or macro declarations and correspond to values or multi-values that are not yet evaluated. When such a variable occurs in an expression, the pseudo value it is bound to is evaluated. TrafoLa binding is static; this means that the evaluation is done in the environment of the declaration. Thus, pseudo values are in fact closures consisting of an expression and its environment.

The system does not know anything about pseudo values, not even their type since there is no static type checking.

Recursive pseudo values result from recursive declarations ‘rec ...’. If they are evaluated, they often become functions. Other pseudo values result from bindings done by macro patterns.

In version 1.4, there was a difference between recursive and macro-created pseudo values. Recursive ones were treated less efficiently. This was the reason why there was a loss of up to a factor of 20 when evaluating applications of recursive functions in version 1.4.

Examples:

```
- rec  fac n   =   if  n == 0  then  1  else  n * fac (n-1);
> fac : not yet evaluated

- fac;                          (the pseudo value bound to 'fac' is evaluated to a
> function                       function after it was read from the environment)

- fac 3;                        (the call of 'fac' above is not necessary for this application)
> 6

- rec a = 1;
> a : not yet evaluated
```

```
- a;                              (here, the recursive object evaluates to a number)
> 1
```

2.1.5. Error values

Error values are not ordinary values. They result from erroneous evaluations and mostly correspond to run-time type errors. There are no operations to process error values; once an error value results from the evaluation of a subexpression, it also becomes the result of the whole expression. If an expression contains more than one erroneous part, the result is nevertheless only one error value. It depends on the temporal order of the evaluation of subexpressions.

Each error value carries a string, the error message. Error values are displayed as 'Error: s' where 's' is the message. Many predefined operations issue fixed error messages that are described with the operations in section 2.4. The user may create his/her own error messages with the function 'Error'.

Examples:

```
1 / 0                         →  Error:  1 / 0
1 2                           →  Error:  num applied
if 1 then 1 / 0 else 1 2      →  Error:  if num then ...
dec pred x  =  if x == 0 then Error "pred 0" else x - 1;
pred 2                        →  1
pred 0                        →  Error:  pred 0
pred "a"                      →  Error:  string - num
```

The function 'Error' was not yet available in version 1.4.

2.2. Lazy and eager declarations

From its origins, TrafoLa-H is a 'call-by-value' language i.e. originally, all expressions were evaluated no matter whether their value was needed. Starting with version 1.4, various lazy constructs appeared in the language. Version 1.4 contained lazy declarations and streams as new features, and 1.5 additionally contains macros i.e. lazy functions.

A lazy declaration is introduced by a macro pattern:

    dec $variable = expression

It produces exactly one environment where the variable is bound to the unevaluated expression.

An eager declaration is introduced by a normal pattern:

    dec pattern = expression

The expression is evaluated and its values are matched against the pattern. Thus, generally many environments are produced where the variables occurring in the pattern are bound to ordinary values, not pseudo values. When the declaration occurs at top level, only the first outcoming environment is retained and added to the global environment.

A special case of this is when the pattern degenerates to just a variable:

    dec variable = expression

The expression is evaluated and its values are bound to the variable. Thus, as many environments are produced as there are values of the expression.

A recursive declaration looks like

    rec variable = expression

if all syntactic sugar (argument list) has been removed. It is now handled lazily i.e. 'dec $a = 1' and 'rec a = 1' are semantically equivalent since 'a' does not occur in the defining expression. Operationally, the recursive form is a bit less efficient.

Examples:

In these examples and all following ones, we step through top level results by means of ';' to see the next solution and 'q' to tell the system that it should stop to produce further solutions.

- dec x = 1 $ 2;	*(the expression '1 $ 2' evaluates to 1 or 2)*
x : 1 ;	*(the declaration produces two environments)*
x : 2	
No	
- x;	*('x' was bound to 1 by the first result)*
> 1	
- dec $ y = 1 $ 2;	*(lazy declaration where the expression*
> y : not yet evaluated	*is not evaluated, thus only one environment results)*
- y;	*(Now, the pseudo value is evaluated)*
1 ;	
2	
No	

With local declarations occurring after 'let', there is a big difference between lazy and eager declarations if the 'let' expression has more than one value.

let x = 1 $ 2 in [x, x];

gives results [1, 1] and [2, 2] only since [x, x] is evaluated first with x = 1, then with x = 2.

let $ x = 1 $ 2 in [x, x];

gives results [1, 1], [1, 2], [2, 1], [2, 2] since [x, x] is evaluated with x = (1 $ 2).

This last form is therefore equivalent with '[1 $ 2, 1 $ 2]'. So, if you want to abstract common subexpressions out of a big expression, you must use a lazy local declaration to preserve the meaning.

2.3. Functions, macros, and application

Principally, there are no functions at all in a relational language, but I'm used to call some objects functions due to the functional origins of TrafoLa-H.

A function is applied to a single value and generally produces a multi-value. Many functions are deterministic i.e. they produce at most one value when applied to one value. Deterministic functions are inherited from the functional origins of TrafoLa-H and may often be similarly defined as in other functional languages such as ML.

Functions always evaluate their argument. If it results in many values, the function is applied to each value in turn, and the resulting multi-values are combined into one big multi-value (see section 2.5).

A macro is applied to a whole multi-value and produces a multi-value. Macros do not evaluate their argument. The argument expression is closed by its environment and moved into the body of the macro without evaluation. If the argument expression does not contain free variables, this may be understood as textual substitution, whence the name 'macro'.

Assume a big expression containing two subexpressions 'A' and 'B' without free variables: '...A...B...'. Then 'A' and 'B' may be abstracted out of the big expression and replaced by variables:

dec M $x $y = ...x...y...

Then, the application 'M A B' of the macro is semantically equivalent to the original expression '...A...B...'. With a function definition 'dec F x y =', it would not be equivalent.

An applicative expression 'F X' where 'F' and 'X' are expressions is evaluated by first evaluating 'F' to some values f_1, f_2, ... Then, each result f_i is applied to 'X' in turn.

If f_i is a macro, then 'X' is closed and moved into the body of the macro. If f_i is a function or a constructor, then 'X' is evaluated to some values x_1, x_2, ... Then, f_i is applied to each result x_j in turn, and the results y_{ij1}, y_{ij2}, ... will be combined.

Some examples will clarify the situation.

dec one_f _ = 1   dec one_m $_ = 1

Application to a simple, deterministic expression makes no difference:

```
one_f 5          →  1                   one_m 5          →  1
```

Application to a failing (erroneous, non-terminating) expression makes the function application failing (erroneous, non-terminating). This statement holds for any function, not only for this special example.

```
one_f fail       →  No                  one_m fail       →  1
one_f (1 / 0)    →  Error: 1 / 0        one_m (1 / 0)    →  1
```

The application of 'one_f' to an expression yields as many results as the expression has; the application of 'one_m' gives exactly one:

```
one_f (4 $ 7)    →  1 $ 1               one_m (4 $ 7)    →  1
```

where '$' is the operator of non-deterministic choice i.e. '4 $ 7' evaluates to the multi-value {4, 7}, and the result '1 $ 1' is {1, 1}. Note that this result is not displayed as '1 $ 1', but according to section 8 of the User Manual.

```
dec id_f x = x                          dec id_m $x = x
```

There is no semantic difference between these two. Whenever 'id_f X' is to be evaluated, then 'X' is evaluated before application. Whenever 'id_m X' is evaluated, the evaluation first results in 'X' that is then evaluated.

```
dec pair_f x = [x, x]                   dec pair_m $x = [x, x]
```

Note that 'x' must also be evaluated in the macro case. There is no semantic difference when applied to expressions with at most one value. But multi-valued expressions lead to different results:

```
pair_f (1 $ 2)  →  pair_f 1 $ pair_f 2  →  [1, 1] $ [2, 2]
pair_m (1 $ 2)  →  [1 $ 2, 1 $ 2]  →  [1, 1] $ [1, 2] $ [2 ,1] $ [2, 2]

rec fac_f n    =  if n == 0 then 1 else n * fac_f (n - 1)
rec fac_m $n   =  if n == 0 then 1 else n * fac_m (n - 1)
```

This is similar to the 'pair' example since 'n' must be evaluated in any case due to its occurrence in the condition, and there is more than one occurrence of 'n' in the right hand side. Whereas the function copes quite well with multi-valued arguments, the macro does not since it combines any value with any other value what leads to many strange results and then to non-termination.

```
fac_f (2 $ 3)   →  fac_f 2 $ fac_f 3  →  2 $ 6
fac_m (2 $ 3)   →  if (2 $ 3) == 0 then 1 else (2 $ 3) * fac_m ((2 $ 3) - 1)
                →  2 * fac_m (1 $ 2)  $  3 * fac_m (1 $ 2)
...             →  2 * 1 * fac_m (0 $ 1)  $  ...
                →  2 * 1 * ( if (0 $ 1) == 0 then 1 else (0 $ 1) * fac_m ((0 $ 1) - 1) )  $  ...
                →  2 * 1 * ( 1 $ 0 * fac_m (-1 $ 0) $ 1 * fac_m (-1 $ 0) )  $  ...
```

Final results:

```
fac_m (2$3)     2; 2; 2; 2; 3; 3; <looping>
fac_m (3$7)     6; 6; 6; 6; 14; 14; 6; 6; 14; 14; 6; 6; 14; 14; 18; 18; 14; 14; 42; 42; <looping>
```

See section 2.5 for the order of the results.

2.4. Predefined operations

2.4.1. Introduction

Compared with version 1.4, the predefined operations are completely refined and extended. Specially, the operations on streams are now overloaded with those on sequences.

45 predefined operations are bound to variables in the global environment when the interpreter starts to run. The user might redeclare these variables locally by using them in patterns or 'let' expressions, and also globally. When such a global redeclaration is killed by '#k', the predefined meaning of the variable reappears.

26 predefined operations are denoted by binary infix operators. The user is neither able to redeclare them nor to declare new ones. All these operators are also available in prefix form e.g. '1+2' is equivalent

with ‘(+) 1 2’. Many operators denote functions, but some denote macros. For reasons of uniformity, the prefix form is internally represented as macro for all operators such that the interpreter displays ‘macro’ if only ‘(+)’ is entered.

Some more predefined operations are denoted by special syntax e.g. ‘if’.

We give a list of the predefined operations, their intended type, a short description, user definition if possible, and examples. The user definitions have several purposes: first, they provide an exact specification of the operation in TrafoLa-H, especially with respect to what arguments are evaluated; second, they give examples of useful TrafoLa-H definitions to the reader of this manual. Even if a complete user definition is impossible, we give a partial one with a question mark for the omitted parts.

The operations will be presented in groups of related ones, starting with more basic operations and going on to the derived ones. The given user definitions normally only refer to operations mentioned before.

List of all groups of operations:

(2) Basic relational operations: ‘$’ ‘\$’
(3) Basic functional forms: ‘|’ ‘#’ ‘\’ ‘\#’ ‘o’ I
(4) Pattern operations: ‘|’ ‘\’ ‘&’ ‘!’ ‘{ }’ ‘where’ ‘< >’
(5) Issuing error messages: Type Error
(6) Conversions: HtoN NtoH CtoS NtoS StoN
(7) Boolean operations: ‘if’ ‘||’ ‘&&’ ‘\|’ ‘\&’ Not
(8) Equality operations: ‘==’ ‘!=’ ‘&’
(9) Order operations: ‘<’ ‘>’ ‘<=’ ‘>=’
(10) Numbers and arithmetic: ‘$-_1$’ Abs ‘+’ ‘-’ ‘*’ Succ Pred ‘div’ ‘mod’ ‘/’ Nat
(11) Sequences, streams, and strings: ‘nil’ ‘::’ Empty Hd Tl ‘.’ ‘+:’ Len Nth First Member Rev Open MkSeq MkStream MkSet Implode Substr SeqTo StreamTo
(12) Cutting functionals: Map Fold ForAll Exists Repeat
(13) Useful macros: ‘<< >>’ MClose MDef MCut MSet NthS
(14) Related functionals: Close Def Cut Set
(15) Insertion and extraction: ‘^’
(16) Trace functions: MPrint Print
(17) The special variable ‘It’

2.4.2. Basic relational operations

Non-deterministic choice operator: **‘$’** (untyped)

Description: The multi-set of results of the expression ‘e1 $ e2’ is the union of the results of ‘e1’ and of ‘e2’. See section 2.5 for more details about the order of the results of ‘e1 $ e2’.

User definition: dec ($) $e1 $e2 = ? /* impossible */

Examples: ‘1 $ 2’ gives results ‘1’ and ‘2’.
‘(1 $ 2) $ (101 $ 102)’ gives results ‘1’, ‘101’, ‘2’, ‘102’.
‘1 $ (1/0) $ 2’ yields first ‘1’, then ‘Error: 1 / 0’, then nothing else
(errors break backtracking)

Failure switch ‘\$’ (untyped)

Description: To evaluate ‘e1 \$ e2’, first the expression ‘e1’ is evaluated. If it produces values, these are the result of the whole expression; otherwise, ‘e2’ is evaluated and its results are the final results.

User definition: dec (\$) $e1 $e2 = ? /* impossible */

Examples:
1 \\$ (1/0) → 1
fail \\$ 2 → 2
(1/0) \\$ 5 → Error: 1 / 0

2.4.3. Basic functional forms

Union of functions '|': FUN → FUN → FUN

Description: The result of applying 'f | g' to some value 'x' is the union of the results of applying 'f' to 'x' and 'g' to 'x'.

User definition: dec (|) $f $g = { x => f x $ g x }

Examples: ({ 0 => 0 } | { n => n-1 }) 0 → 0 $ -1

Remark to the relation between '$' and '|':

At first glance, there is no big difference between combining two functions with '$' and with '|':

dec f = {0 => 1}   dec g = {0 => 2}
(f $ g) 0 → f 0 $ g 0 → 1 $ 2
(f | g) 0 → { x => f x $ g x} 0 → f 0 $ g 0 → 1 $ 2

We see that any application of 'f $ g' to a value gives the same result as 'f | g'. But there are contexts where both expressions behave different:

let h = f $ g in [h 0, h 0] → [f 0, f 0] $ [g 0, g0] → [1, 1] $ [2, 2]
let h = f | g in [h 0, h 0] → [(f | g) 0, (f | g) 0] → [1 $ 2, 1 $ 2] → [1,1] $ [1,2] $ [2,1] $ [2,2]

Union of cases in a function definition '#'

Description: This is not a real operator since it cannot occur freely in expressions, but only inside a function definition i.e. between braces. It specifies the union of two cases of the definition. If the two cases overlap, both are used.
Thus, '{p_1 => e_1 # p_2 => e_2 }' is equivalent with '({ p_1 => e_1 } | { p_2 => e_2 })'.

User definition: not an operator

Examples: { 0 => 0 # n => n-1 } 0 → 0 $ -1

Biased union of functions '\\': FUN → FUN → FUN

Description: The result of applying 'f \ g' to some value 'x' is 'f x' if this application is able to produce at least one result, otherwise it is 'g x'.

User definition: dec (\) $f $g = { x => f x \$ g x }

Examples:
({0 => 0} \\ {n => n-1}) 0 → 0
({0 => 0} \\ {n => n-1}) 7 → 6
({0 => fail} \\ {n => n-1}) 0 → -1
('{0 => fail}' applied to '0' gives no result)

Biased union of cases '\#'

Description: This is not a real operator since it cannot occur freely in expressions, but only inside a function definition i.e. between braces. It specifies the union of two cases of the definition where the first case wins if the entry patterns overlap. The decision which case is to be chosen only depends on the patterns, not on the expressions. Thus, '{p_1 => e_1 \\# p_2 => e_2 }' and '({ p_1 => e_1 } \\ { p_2 => e_2 })' are **not** equivalent.

User definition: not an operator

Examples:
{0 => 0 \\# n => n-1} 0 → 0
{0 => 0 \\# n => n-1} 7 → 6
{0 => fail \\# n => n-1} 0 → No
(Pattern '0' matches value '0' such that 'fail' is chosen.)
Note the difference between the last example here and with '\'.

Remark to '|' and '\' (resp. '#' and '\#') in expressions: '\' and '\#' are always preferable since their implementation is more efficient and they give more information to the interpreter.

Example: { [ ] => ... # h :: t => ... } Here, there is no semantic difference between '#' and '\#' because the patterns are disjoint. But '\#' should be used instead of '#'. There is also an operational difference: with '\#' the interpreter knows that there is only one result, with '#' it generally does not know this.

Functional composition ‘**o**’: $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$

Description:	‘f o g’ is applied to ‘x’ by first applying ‘f’, then applying ‘g’ to ‘x’.
User definition:	dec (o) $f $g = { $x => g (f x) }
Remark:	The final argument ‘x’ is not evaluated by ‘o’ such that its evaluation is left to ‘f’ and ‘g’. This guarantees that both functions and macros may be composed, even mixed.

I:	$\alpha \rightarrow \alpha$
Description:	identity function
User definition:	dec I x = x
Examples:	I 1 → 1 I "abc" → "abc"

2.4.4. Pattern operators

In this section, we describe those pattern operators that are not the direct inverse of some operation in expressions. The other pattern operators (‘::’, ‘+:’, ‘.’, and ‘^’) are described together with their positive counterparts.

Union of patterns ‘|’

‘p | q’ matches a value ‘v’ iff ‘p’ matches ‘v’ or ‘q’ matches ‘v’. The result of the match is the union of the results of matching ‘p’ against ‘v’ and ‘q’ against ‘v’.

Example: rec fib = { 0 | 1 => 1 \# n => fib (n-1) + fib (n-2) }

Biased union of patterns ‘\’

‘p \ q’ is equivalent to ‘p | (q & !p)’ i.e. if ‘p’ matches ‘v’ then ‘p \ q’ matches ‘v’ with the same result; otherwise ‘q’ is matched against ‘v’.

Example: rec fib = { 0 \ 1 => 1 \# n => fib (n-1) + fib (n-2) }

Remark: This version with ‘\’ is more efficient than the one using ‘|’. Naturally, a tail-recursive version is much more efficient, and for safe programming, it is recommended to catch the case of negative arguments.

Intersection of patterns ‘**&**’ or ‘**:**’

‘p & q’ matches iff both ‘p’ and ‘q’ match. This is a generalization of the multi-level pattern i.e. the special case where ‘p’ is a variable.

Example: ‘s & (h :: t)’ matched against ‘[1, 2, 3]’ binds ‘s’ to ‘[1, 2, 3]’, ‘h’ to ‘1’, and ‘t’ to ‘[2, 3]’.

Not-pattern ‘**!**’

‘! p’ matches iff ‘p’ does not match, and then produces the empty environment.

Example: ‘! [A, A]’ matches ‘[2, 3]’ but does not create a binding.

Unbind pattern ‘**{ }**’

‘{p}’ matches iff ‘p’ matches, but only produces the empty environment. It is equivalent to ‘!!p’.

Example: ‘[a, b] & {[b, a]}’ matches ‘[1, 2]’ with ‘a = 1’ and ‘b = 2’. This pattern is strange since it is equivalent to ‘[a, b]’.

Restriction of patterns ‘**where**’

This is not a typical binary operator since it combines patterns with Boolean expressions. ‘p where e’ is matched against ‘v’ by matching ‘p’ against ‘v’ and keeping only those environments which make the expression ‘e’ evaluate to a multi-set ‘ms’ containing ‘true’ as an element. It does not matter whether ‘ms’ contains several occurrences of ‘true’, or contains ‘false’ or any other value beneath ‘true’. If ‘ms’ is infinite and does not contain ‘true’, this cannot be recognized in finite time, and the construct does not terminate.

Examples: ‘[a, b] where a < b’ matches ‘[1, 2]’ but does not match ‘[2, 1]’.
‘a where (1 $ a $ false)’ matches ‘true’, but does not match any other value; thus, it is equivalent to the pattern ‘true’.

Cut pattern **‘< >’**

‘<p>’ matches iff ‘p’ does, but it produces one environment only.

Example: ‘<a . b>’ matched against ‘[1, 2]’ produces only the environment where ‘a’ is bound to ‘[ ]’ and ‘b’ to ‘[1, 2]’; nothing else.

All these operators were already contained in version 1.4. The only difference is that the cut operator was written ‘<<p>>’ instead of ‘<p>’ there.

2.4.5. Issuing error messages

Both functions described here are new; in version 1.4, the user could not program error messages. The first function, ‘Type’, is described here since it is often used to give diagnostics for run-time type errors.

Type: $\alpha \rightarrow$ string

Description: Result is a string containing the type of the argument.

User definition:

```
dec  Type  =  {  num        =>  "num"
              \#  bool      =>  "bool"
              \#  string    =>  "string"
              \#  const     =>  "const"
              \#  hole      =>  "hole"
              \#  tree      =>  "tree"
              \#  seq       =>  "seq"
              \#  stream    =>  "stream"
              \#  fun       =>  "fun"
              \#  mac       =>  "mac"  }
```

Examples: Type 3 $\rightarrow$ "num"   Type Type $\rightarrow$ "fun"

Remark: This function is also used to set up the error messages of the predefined functions

Error: string $\rightarrow$?

Description: Creates an error value with its argument as message

User definition:

```
rec  Error  =  { s: string   =>  ?  /* impossible */
               \# x          =>  Error ("Error ". Type x) }
```

where ‘.’ means concatenation of strings.

Examples:

```
Error "run-time error"    ->  Error: run-time error
Error 5                   ->  Error: Error num
Error ("1" $ "2")         ->  Error: 1
```

2.4.6. Conversions

HtoN: hole $\rightarrow$ num

Description: hole to number

User definition:

```
dec HtoN  =     { h: hole    =>  ?  /* impossible */
                \# x         =>  Error ("HtoN ". Type x) }
```

Examples: HtoN @5 $\rightarrow$ 5   HtoN @ $\rightarrow$ 1

NtoH: num $\rightarrow$ hole

Description: number to hole

User definition:

```
dec NtoH  =     { n: num     =>  if  n > 0     then  ?  /* impossible */
                                               else  Error "NtoH (n where n < 1)"
                \# x         =>  Error ("NtoH ". Type x) }
```

Examples: NtoH 5 $\rightarrow$ @5   NtoH 1 $\rightarrow$ @

CtoS: const $\rightarrow$ string

Description: constructor to string

User definition:

```
dec CtoS  =     { c: const   =>  ?  /* impossible */
                \# x         =>  Error ("CtoS ". Type x) }
```

```
Examples:           con equal, ’a’;
                    CtoS  equal  →  "equal"          CtoS  ’a’  →  "’a’"
```

NtoS: num → string
Description: number to string

```
User definition:    dec NtoS  =      { n: num      =>  ?  /* difficult */
                                     \# x          =>  Error ("NtoS ". Type x) }
Examples:           NtoS  4711  →  "4711"          NtoS  (-57)  →  "-57"
```

StoN: string → num
Description: string to number

```
User definition:    dec StoN  =      { s: string   =>  ?  /* difficult */
                                     \# x          =>  Error ("StoN ". Type x) }
Examples:           StoN  "4711"  →  4711                 StoN  "-57"  →  -57
                    StoN  ""  →  Error:  StoN ""          StoN  5  →  Error:  StoN num
                    StoN  "1a"  →  Error:  StoN "...a..."
```

All these functions were already contained in version 1.4. ‘CtoS’ was called ‘OtoS’ there.

2.4.7. Boolean operations

Conditional expressions ‘**if**’ : bool → any → any → any
Special syntax: if c then t else e
Description: The condition is evaluated to a multi-value ‘mv’. The ‘then’ part is evaluated as often as there are ‘true’s in ‘mv’, and the ‘else’ part as there are ‘false’s. All results are combined.

```
User definition:    dec  (if)  c  $t  $e   =     {  true        =>  t
                                                 \#  false      =>  e
                                                 \#  x          =>  Error ("if ". Type x ." then ...")  }  c
Examples:           if  true  then  1  else  1/0             →  1
                    if  true $ false  then  1  else  2       →  1 $ 2
                    if  2 < 1  then  17                      →  No
                    if  1  then  2 else  3                   →  Error: if num then ...
```

For the user definition of the following binary operations, we need an auxiliary macro:
dec curry $f $x $y = f [x, y]

Note that we could have written ‘f’ instead of ‘$f’ since there is exactly one occurrence of ‘f’ on the right hand side whose evaluation is forced by its context. If ‘curry’ is applied to a function, not a macro, then the evaluation of ‘x’ and ‘y’ is also forced.

Eager Boolean ‘or’ denoted by ‘**||**’ and ‘and’ denoted by ‘**&&**’ : bool → bool → bool
Description: These operations are only defined for Boolean arguments. Both arguments are evaluated even if the first one could determine the result.

```
User definitions:   dec  (||)      = curry     {  [x: bool, y: bool]     =>  if x then true else y
                                               \#  [x, y]                =>  Error (Type x ." || ". Type y) }
                    dec  (&&)  =  curry        {  [x: bool, y: bool]     =>  if x then y else false
                                               \#  [x, y]                =>  Error (Type x ." && ". Type y) }
Examples:           (false $ true) || (false $ true)  →  (false || false) $ (false || true) $ (true || false) $ (true || true)
                       →  false $ true $ true $ true
                    (false $ true) && (false $ true)  →  false $ false $ false $ true
```

Lazy sequential Boolean ‘or’ denoted by ‘**\\|**’ and ‘and’ denoted by ‘**\\&**’ : bool → any → any
Description: These operations are only defined if the first argument is Boolean. The second

argument is not evaluated and either dropped or propagated.

User definitions:
```
dec (\|) =   {  x: bool    => { $y  =>  if x then true else y }
                \#  x      => { $y  =>  Error (Type x ." \\| ...") } }
dec (\&) =   {  x: bool    => { $y  =>  if x then y else false }
                \#  x      => { $y  =>  Error (Type x ." \\& ...") } }
```
(remember that back-slashes must be doubled in strings)

Examples:
```
(false $ true) \\| (false $ true)  →  (false \\| (false $ true)) $ (true \\| (false $ true))
   →  (false $ true) $ true
(false $ true) \\& (false $ true)  →  false $ (false $ true)
```

Negation **Not**: bool → bool

Description: logical negation

User definition:
```
dec Not =   {  false      =>  true
               \#  true   =>  false
               \#  x      =>  Error ("Not ". Type x) }
```
Examples: Not false → true   Not 5 → Error: Not num

Version 1.4 merely contained ‘if’, ‘||’, and ‘&&’; ‘\|’, ‘\&’, and ‘Not’ are new.

The order operations ‘<’ etc. also work for Booleans.

2.4.8. Equality operations

Operational **equality** ‘**==**’ : any → any → bool

Description: Operational equality is defined for all pairs of values and returns either ‘true’ or ‘false’. Values of different types are never operationally equal. Basic values are operationally equal iff they are equal. Operational equality on structural values is defined recursively:

$[u_1, \ldots, u_n] == [v_1, \ldots, v_m]$ iff $n = m$ and $u_1 == v_1$ and ... and $u_n == v_n$

$op_1\ v_1 == op_1\ v_2$ iff $op_1 == op_2$ and $v_1 == v_2$

Abstract values (functions and streams) are usually not operationally equal to each other. They are only operationally equal if they were created by the same evaluation of the same occurrence of an expression. You might think of a time stamp or unique number added to abstract values when they are created.

User definition: dec (==) x y = ?

Examples:
```
{ x => x } == { x => x }   →   false
```
(There are two different occurrences of the functional expression)
```
let f = { x => x } ; g = f  in  f == g   →   true
```
(One occurrence, once being evaluated)
```
<<1 $ 2>> == <<1 $ 2>>   →   false
```
(Streams are also abstract)

Summarizing, we conclude that abstract values are only claimed to be equal if there is no doubt on its equality. The user should avoid comparing abstract values since the results are not reliable.

There is a special function ‘Empty’ (see below) to check whether a stream is empty. The test ‘s == << >>’ would always return ‘false’.

The equality defined here is also used when non-linear patterns are matched. This implies that the pattern ‘[a, a]’ does not match the value ‘[<< >>, << >>]’.

All these strange effects are for the sake of efficiency. Operational equality of two TrafoLa-H values is implemented by comparing the ML data structures representing these values by the efficient ML equality check. In case of abstract values, their representing ML data structure contains more information than just the argument-result relation of functions resp. the contents of a stream.

Operational **inequality** ‘**!=**’ : any → any → bool

Description: This is just the negation of operational equality.
‘u != v’ yields ‘true’ iff ‘u == v’ yields ‘false’ and vice versa.

All remarks given there are also valid here.

User definition: dec (!=) x y = Not (x == y)

Intersection of multi-sets ‘**&**’ or ‘**:**’ (untyped)

Description: Applied to two values, it returns the value if it is the same, otherwise, it fails. This implies that ‘e1 & e2’ returns the ‘intersection’ of the multi-sets resulting from evaluating ‘e1’ and ‘e2’. This operator is not very useful, even dangerous, since the emptiness of the ‘intersection’ cannot be recognized by the interpreter if one of the multi-sets is infinite.

User definition: dec (&) x y = if x == y then x

Examples:
1 & 2 → No   3 & 3 → 3
(1 $ 3 $ 5 $ 7) & (2 $ 4 $ 3 $ 1) → 1 $ 3
(2 $ 1 $ 2) & (3 $ 2 $ 4 $ 2) → 2 $ 2 $ 2 $ 2
Nat → 1 $ 2 $ 3 $ ... (see below)
-1 & Nat → <looping>

These three operations were already contained in version 1.4 with the same properties. == != &

2.4.9. Order operators

There is a total order defined on numbers, holes, Booleans, and strings. Values may be compared according to this order by the usual operations.

‘<’ ‘>’ ‘<=’ ‘>=’ (overloaded)

Description:

These operators are defined only for arguments of some basic types. Both arguments must have the same type.

- numbers: usual order
- Booleans: false < true
- holes: @i < @j iff i < j
- strings: lexical order derived from some order on characters

User definition: partially possible, but lengthy and not too illuminating

Warning: The exact meaning of ‘<’ for strings might depend on the representation of characters used by the underlying hardware.

2.4.10. Numbers and arithmetic

Numbers and their operations directly correspond to the numbers in ML. The exact range of numbers and the exact semantics of the arithmetic operations are therefore the same as in ML except ‘div’ and ‘mod’ that are slightly updated. Thus, some bugs of the underlying ML implementation are inherited e.g. it is possible that the sum of two large positive numbers becomes negative. On SUN work stations, the greatest positive number seems to be 2^{30} = 1 073 741 824.

Unary minus ‘-’ : num → num

Description: additive inverse of a number

User definition:
```
dec ‘-’ =   { n: num   =>  ?
              \# x     =>  Error ("- ". Type x) }
```

Examples: - 5 → -5   -(-1) → 1

Absolute value **Abs** : num → num

Description: absolute value of a number

User definition:
```
dec Abs =   { n: num   =>  if n < 0 then -n else n
              \# x     =>  Error ("Abs ". Type x) }
```

Examples: Abs 5 → 5   Abs (-1) → 1

Integer operations ‘+’ ‘-’ ‘*’ : num → num → num
Description: usual binary arithmetic operations on integers
User definition:
```
dec (+) = curry { [a: num, b: num]   =>  ?
                 \# [x, y]           =>  Error (Type x ." + ". Type y) }
```
Examples: 3 + 7 → 10   3 - 7 → - 4   3 * 7 → 21

Successor **Succ** : num → num | hole → hole
Description: successor function for integers and also holes
User definition:
```
dec Succ = { n: num     =>  n + 1
           \# h: hole   =>  NtoH (HtoN h + 1)
           \# x         =>  Error ("Succ ". Type x) }
```
Examples: Succ 3 → 4   Succ (-1) → 0   Succ @ → @2

Predecessor **Pred** : num → num | hole → hole
Description: predecessor function for integers and also holes
fails when applied to @ = @1 since there is no @0
User definition:
```
dec Pred = { n: num     =>  n - 1
           \# @         =>  fail
           \# h: hole   =>  NtoH (HtoN h - 1)
           \# x         =>  Error ("Pred ". Type x) }
```
Examples:
```
Pred 3    →  2          Pred 0    →  -1
Pred @2   →  @          Pred @    →  No
```

Division and remainder ‘**div**’ ‘**mod**’ : num → num → num
Description: ‘div’ (division with remainder) and ‘mod’ (remainder upon division, called ‘modulus’) are two strongly related operations on integers. Division by zero gives an error value.
User definition:
```
dec (div) = curry { [a: num, 0]          =>  Error (NtoS a ."div 0")
                  \# [a: num, b: num]    =>  ?
                  \# [x, y]              =>  Error (Type x ." div ". Type y) }
```
Specification: Let d = a div b and m = a mod b where $b \neq 0$.
1) sign of m = sign of b
2) $|m| < |b|$
3) a = d * b + m
Conclusion: m = 0 iff b divides a

Examples:
```
7     div  5     →  1          7     mod  5     →  2
(-7)  div  5     →  -2         (-7)  mod  5     →  3
7     div  (-5)  →  -2         7     mod  (-5)  →  -3
(-7)  div  (-5)  →  1          (-7)  mod  (-5)  →  -2
```

Integer division with failure ‘/’ : num → num → num
Description: Gives the quotient provided it is an integer; otherwise, no result is produced. Division by zero yields an error.
User definition:
```
dec (/) = curry { [a: num, 0]          =>  Error (NtoS a ." / 0")
                \# [a: num, b: num]    =>  if a mod b == 0 then a div b
                \# [x, y]              =>  Error (Type x ." / ". Type y) }
```
Examples: 12 / 4 → 3   17 / 4 → No   1 / 0 → Error: 1 / 0

Nat : num
Description: not a function, but the multi-set of all natural numbers
User definition:
```
rec Nat = 1 $ (Nat + 1)
```
or: `dec $Nat = let rec from n = n $ from (n+1) in from 1`

Example: Nat → 1 $ 2 $ 3 $ 4 $ 5 $ ...

2.4.11. Sequences, streams, and strings

In version 1.4, many predefined operations worked equally for sequences and strings, but not for streams, and there were special operations for streams. This was unified, now most operations work for sequences, strings, and streams equally. The user may also specify such functions easily by using the unspecific pattern operators 'nil' and '::'.

To have a short name, we call these three kinds of values s-values since the names 'sequence', 'string', and 'stream' all begin with 's'.

For this section, remember the subdivision of streams into three classes: finite streams with a definite end-of-stream indicator, infinite streams, and open streams ending by a non-terminating computation. The empty stream is a special finite one, and the bottom stream a special open one.

String constants are allowed in a pattern since strings are considered basic values. A special string constant is '""' denoting the empty string. Sequences may be specified in patterns using sequential patterns '[p_1, ..., p_n]'. A special case is '[ ]' denoting the empty sequence. There is no direct notation for streams in patterns except '<< >>' denoting the empty stream. When matching '<< >>' against a stream value, the operational equality '==' is **not** used internally since it would not work correctly. A special internal test is used instead that evaluates the stream value until a first item or an end-of-stream indicator is found. This works well for all streams except the bottom stream that starts with an infinite computation. The pattern match will not terminate either in this case.

Finite streams may be specified in patterns indirectly by use of '::' and '<< >>' e.g. '<<1 $ 2>>' as '1 :: 2 :: << >>'. Such a pattern works well for all stream values except short open streams (in our example open streams with less than three items).

The pattern operator 'nil'

Description: 'nil' matches empty s-values
User definition: [ ] | "" | << >>
Examples: see '::'

The pattern operator '::'

Description: '::' partitions s-values into their first item and the remainder. A '::' pattern does not match empty s-values or any other value.
User definition: ?
Examples:
'1 :: _' matches [1, 2, 3], <<1 $ 2>>, <<Nat>>, ...
'"a" :: _' matches ["a", "b"], <<"a">>, "abc", ...
'h :: t' matched against [1, 2, 3] binds h to 1 and t to [2, 3]
matched against <<Nat>> binds h to 1 and t to <<2 $ 3 $ ...>>
matched against "BKB" binds h to "B" and t to "KB"
'1 :: 2 :: nil' matches [1, 2], <<1 $ 2>> and nothing else
'"a" :: "b" :: nil' matches ["a", "b"], <<"a" $ "b">>, "ab" and nothing else

'::' in expressions

Description: This is the inverse of '::' in patterns. It attaches its left operand as new first item to its right operand that must be an s-value. In case of strings, the left operand must be a character i.e. a string of length 1.

User definition:
```
dec (::) = curry { [h, t: seq]            => ?
                \# [h, t: stream]         => ?
                \# [h: char, t: string]   => ?
                \# [h, t: string]         => Error "(! char) :: string"
                \# [x, y]                 => Error (Type x ." :: ". Type y) }
```

Examples:
```
1 :: [2, 3]       →  [1, 2, 3]
1 :: <<2 $ 3>>    →  <<1 $ 2 $ 3>>
0 :: <<Nat>>      →  <<0 $ 1 $ 2 $ 3 $ ... >>
"a" :: "bc"       →  "abc"
"ab" :: "c"       →  Error: (! char) :: string
```

Empty: (sequence | string | stream) → bool

Description: check whether s-value is definitely empty

User definition:
```
dec  Empty  =    {  nil       =>  true
                 \#  _ :: _   =>  false
                 \#  x        =>  Error ("Empty ". Type x) }
```
Examples:
Empty [ ] = true — Empty [1, 2] = false
Empty "" = true — Empty "TrafoLa-H" = false
Empty << >> = true — Empty << 1 $ 2 >> = false

Remarks: For sequences and strings, ‘s == [ ]’ resp. ‘s == ""’ also works, but for streams, ‘s == << >>’ always gives ‘false’.
‘Empty’ does not terminate when applied to the bottom stream.

Hd: α * → α | string → char | stream of α → α

Description: first item of s-values, no result for empty s-values

User definition:
```
dec  Hd  =    {  nil       =>  fail
              \#  h :: _   =>  h
              \#  x        =>  Error ("Hd ". Type x) }
```
Examples:
Hd [1, 2, 3] → 1 — Hd "TrafoLa" → "T"
Hd <<Nat>> → 1 — Hd [ ] → No

Remark: ‘Hd’ does not terminate when applied to the bottom stream.

Tl: seq → seq | string → string | stream → stream

Description: remainder after removing first item, no result for empty s-values

User definition:
```
dec  Tl  =    {  nil       =>  fail
              \#  _ :: t   =>  t
              \#  x        =>  Error ("Tl ". Type x) }
```
Examples:
Tl [1, 2, 3] → [2, 3] — Tl "TrafoLa" → "rafoLa"
Tl <<Nat>> → <<2 $ 3 $ ...>> — Tl [ ] → No

Remark: ‘Tl’ does not terminate when applied to the bottom stream.

Concatenation / append: ‘**.**’ in expressions

Description: Two s-values are concatenated to a new s-value. In case of streams, this makes only sense if the left operand is finite. If not, the result is the unchanged left operand.

User definition:
```
rec  append  =      {  [nil, y]        =>  y
                    \#  [h :: t, y]    =>  h :: append [t, y] }
rec  (.)  =  curry  {  p: ([seq, seq] | [stream, stream] | [string, string])  =>  append p
                    \#  [x, y]         =>  Error (Type x ." . ". Type y) }
```
Examples:
[1, 2] . [3, 4, 5] → [1, 2, 3, 4, 5]
"PRO" . "SPECTRA" → "PROSPECTRA"
<<-1 $ 0>> . <<Nat>> → <<-1 $ 0 $ 1 $ 2 $ ...>>
<<Nat>> . <<-1 $ -2>> → <<Nat>>

Decomposition into left and right part: ‘**.**’ in patterns

Description: The pattern ‘p . q’ matches an s-value ‘u’ by decomposing it into two values ‘v’ and ‘w’ such that ‘v . w = u’ holds. For infinite streams, there is an infinite number of possible decompositions. The first part is always a finite stream.

Examples:
‘a . b’ matches [1, 2] with a = [ ], b = [1, 2], or
a = [1], b = [2], or a = [1, 2], b = [ ]
‘a . b’ matches "xy" with a = "", b = "xy", or
a = "x", b = "y", or a = "xy", b = ""
‘a . b’ matches <<Nat>> with a = << >>, b = <<1 $ 2 $ ...>>, or
a = <<1>>, b = <<2 $ 3 $ ...>>, or ...

Insertion into a sequence: ‘**+:**’ in expressions

Description: The left operand is inserted somewhere into the right operand that must be a sequence

User definition:
```
rec  insert  =    { [x, [ ]]                 =>  [x]
                  \#  [x, s & (h :: t)]       =>  x :: s  $  h :: insert [x, t] }
dec  (+:)  x  =   { y: seq                   =>  insert [x, y]
                  \#  y                       =>  Error (Type x ." +: ". Type y) }
```

Example:
```
100  +:  [1, 2, 3]  →  [100, 1, 2, 3]  $  [1, 100, 2, 3]  $  [1, 2, 100, 3]  $  [1, 2, 3, 100]
```

Choosing and removing an item: ‘**+:**’ in patterns

Description: The left operand is bound to an item of the sequence, the right one to the remainder

Example: ‘x +: y’ matches [1, 2, 3] with x = 1 and y = [2, 3], or x = 2 and y = [1, 3], or x = 3 and y = [1, 2]

Len: (seq | string | stream) → num

Description: length of s-values. Does not terminate when applied to open or infinite streams

User definition:
```
rec  Len  =    {  nil        =>  0
               \#  _ :: t     =>  1 + Len t
               \#  x          =>  Error ("Len ". Type x) }
```

Examples:
```
Len  [1, 2, 3]  →  3              Len  "TrafoLa"  →  7
Len  <<1 $ 2>>  →  2              Len  <<Nat>>  →  <looping>
```

Nth: num → (α * → α | stream of α → α | string → char)

Description: The nth element, counted from 1. If there is no nth element, no result is produced

User definition:
```
rec  Nth  =    {  n: num    =>   {  s: (seq | string | stream)  =>
                                        if  n == 1    then  Hd s  else
                                        if  n > 1     then  Nth (n-1) (Tl s)
                                    \#  y    =>  Error ("Nth ". NtoS n ." ". Type y) }
               \#  x         =>   Error ("Nth ". Type x) }
```

Examples:
```
Nth  2  [7, 8, 9]  →  8         Nth  3  "TrafoLa"  →  "a"
Nth  128  <<Nat>>  →  128       Nth  0  "TrafoLa"  →  No         Nth  3  [1]  →  No
```

First: num → (seq → seq | stream → stream | string → string)

Description: ‘First n s’ delivers the first n items of s, or s itself if it is smaller than n. If n is less than 1, an empty s-value is returned.

User definition:
```
rec  First  =  {  n: num  =>  {  s: (seq | string | stream)  =>
                                    if  n < 1  then  case s of
                                        {  seq       =>  [ ]
                                        \#  string   =>  ""
                                        \#  stream   =>  << >> }
                                    else  case s of
                                        {  nil       =>  s
                                        \#  h :: t   =>  h :: First (n-1) t }
                              \#  y     =>          Error ("First ". NtoS n ." ". Type y) }
               \#  x        =>  Error ("First ". Type x) }
```

Examples:
```
First  2  [7, 8, 9]  →  [7, 8]                 First  5  [1, 2]  →  [1, 2]
First  3  "PROSPECTRA"  →  "PRO"               First  (-1)  [1, 2]  →  [ ]
First  3  <<Nat>>  →  <<1 $ 2 $ 3>>
```

Member: α → (α * | stream of α) → bool | string → string → bool

Description: ‘Member x s’ checks whether the item x is contained in the s-value s. If s is a string and x is not a character, the answer is ‘false’. If s is an infinite or open stream not containing x, ‘Member x s’ does not terminate.

User definition:
```
rec member x  =     {  nil          =>  false
                    \#  h :: t       =>  x == h  \|  member x t }
dec  Member  =  curry    { [x, s]: ([ _, (seq | stream)] | [string, string])  =>  member x s
```

```
                                 \#  [x, y]   =>   Error ("Member ". Type x ." ". Type y) }
Examples:          Member  3  [1, 2, 3]  →  true          Member  "l"  "TrafoLa"   →  false
                   Member  7  <<Nat>>  →  true           Member  0  <<Nat>>   →  <looping>
                   Member  "12"  "12"   →  false         Member  1  "12"   →  Error:  Member num string

Rev:               seq  →  seq   |   string  →  string   |   stream  →  stream
Description:       reverse of an s-value
                   Does not terminate for infinite and open streams
User definition:   dec  single  =     {  seq          =>   { x  =>  [x] }
                                      \#  string      =>   I
                                      \#  stream      =>   { x  =>  <<x>> } }
                   rec  Rev   =       {  s : nil      =>   s
                                      \#  h :: t      =>   Rev t . single t h
                                      \#  x           =>   Error ("Rev ". Type x) }
Examples:          Rev  [1, 2, 3]   →   [3, 2, 1]               Rev  "TrafoLa"   →   "aLofarT"
                   Rev  <<1 $ 2>>   →   <<2 $ 1>>               Rev  <<Nat>>   →   <looping>

Open:              (seq | string | stream)  →  _       (multiple results)
Description:       enumeration of the items of an s-value
User definition:   rec  Open  =       {  nil          =>   fail
                                      \#  h :: t      =>   h $ Open t
                                      \#  x           =>   Error ("Open ". x) }
Examples:          Open  [1, 2, 3]   →   1 $ 2 $ 3              Open  "BKB"   →   "B" $ "K" $ "B"
                   Open  <<1 $ 2>>   →   1 $ 2                  Open  <<Nat>>   →   1 $ 2 $ 3 $ ...

MkSeq:             (seq | string | stream)  →  seq
Description:       s-value is transformed into a sequence consisting of the same items
                   Does not terminate for infinite or open streams
User definition:   rec  MkSeq   =     {  nil          =>   [ ]
                                      \#  h :: t      =>   h :: MkSeq t
                                      \#  x           =>   Error ("MkSeq ". Type x) }
Examples:          MkSeq  "TrafoLa"     →   ["T", "r", "a", "f", "o", "L", "a"]
                   MkSeq  [1, 2, 3]     →   [1, 2, 3]
                   MkSeq  <<1 $ 2>>     →   [1, 2]                 MkSeq  <<Nat>>   →   <looping>

MkStream:          (seq | string | stream)  →  stream
Description:       s-value is transformed into a stream consisting of the same items
User definition:   rec  MkStream   =    {  nil          =>   << >>
                                        \#  h :: t      =>   h :: MkStream t
                                        \#  x           =>   Error ("MkStream ". Type x) }
Examples:          MkStream  "TrafoLa"    →   <<"T" $ "r" $ "a" $ "f" $ "o" $ "L" $ "a">>
                   MkStream  [1, 2, 3]    →   <<1 $ 2 $ 3>>
                   MkStream  <<1 $ 2>>    →   <<1 $ 2>>       MkStream  <<Nat>>   →   <<Nat>>

MkSet:             seq  →  seq   |   string  →  string   |   stream  →  stream
Description:       removes multiple occurrences of items
                   Transforms some infinite into open streams
User definition:   rec  help   =     {  [items, s: nil]      =>   s
                                    \#  [items, h :: t]     =>
                                              if  Member h items          then  help [items, t]
                                                                          else  h :: help [h :: items, t] }
                   dec  MkSet   =     {  s: (seq | stream | string)      =>   help [[ ], s]
                                      \#  x                              =>   Error ("MkSet ". Type x) }
Examples:          MkSet  "TrafoLa"   →   "TrafoL"
```

```
MkSet  [1, 2, 2, 1, 3, 2, 4, 1]   →   [1, 2, 3, 4]
MkSet  <<1 $ 1>>   →   <<1>>                        MkSet  <<Nat>>   →   <<Nat>>
let rec  ones  =  1 $ ones   in   MkSet  <<ones>>
    →  MkSet  <<1 $ 1 $ 1 $ ...>>   →   <<1 $ <looping> >>
```

Implode: string * → string

Description: multiple concatenation of the strings in the sequence (inverse of ‘MkSeq’ for strings)

User definition:

```
dec  Implode  =      {  [ ]                         =>   ""
                     \#  (h: string) :: (t: seq)    =>   h . Implode t
                     \#  h :: seq                   =>   Error ("Implode [..., ". Type h . ", ...]")
                     \#  x                          =>   Error ("Implode ". Type x) }
```

Examples:

```
Implode ["PRO", "SPEC", "TRA"]   →   "PROSPECTRA"
Implode  [1, 2, 3]   →   Error:  Implode [..., num, ...]
```

Substr: num → num → string → string

Description: selection of a substring with given start and stop index

User definition:

```
rec  substr  a  b  s   =
        if  s == ""      then  ""  else
        if  a > 1        then  substr (a-1) (b-1) (Tl s)  else
        if  a == 1       then  First b s
        else  ""
dec  Substr   =
        {  a: num   =>
                {  b: num   =>
                        {  s: string =>   substr a b s
                        \#  x         =>   Error ("Substr ". NtoS a ." ". NtoS b ." ". Type x) }
                \#  b         =>   Error ("Substr ". NtoS a ." ". Type b) }
        \#  a         =>   Error ("Substr ". Type a) }
```

Examples:

```
Substr  3  4  "TrafoLa"        →   "af"
Substr  (-1)  7  "abs"         →   "abs"
Substr  11  15  "x"            →   ""
```

SeqTo: num → num *

Description: sequence from 1 to n

User definition:

```
dec  SeqTo  =        {  n: num   =>   let rec  seqto’ [a, s]   =
                                               if  a < 1  then  s  else  seqto’ [a - 1, a :: s]
                                      in  seqto’ [n, [ ]]
                     \#  x        =>    Error ("SeqTo ". Type x) }
```

Examples:

```
SeqTo  10      →   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
SeqTo  (-5)    →   [ ]
```

StreamTo: num → stream of num

Description: stream from 1 to n

User definition:

```
dec  StreamTo  =     {  n: num   =>   let rec  streamto’ [a, s]  =
                                               if  a < 1  then  s  else  streamto’ [a - 1, a :: s]
                                      in  streamto’ [n, << >>]
                     \#  x        =>   Error ("StreamTo ". Type x) }
```

Examples:

```
StreamTo  10      →   <<1 $ 2 $ 3 $ 4 $ 5 $ 6 $ 7 $ 8 $ 9 $ 10>>
StreamTo  (-5)    →   << >>
```

Remark to infinite streams:

It is recommended to create infinite streams by catching an infinite multi-value e.g.

```
dec  ones    =  let rec  help  =  1 $ help   in   <<help>>
ones    →   <<1 $ 1 $ 1 $ ...>>
dec  cycle  =  let rec  help  =  1 $ 2 $ 3 $ help   in   <<help>>
cycle   →   <<1 $ 2 $ 3 $ 1 $ 2 $ 3 $ 1 $ ...>>
```

The direct construction

```
rec  ones    =  1 :: ones
rec  cycle   =  <<1 $ 2 $ 3>> . cycle
```

does not work (ones → <looping>) since both '::' and '.' must know the type of their right operand, but there is no way to determine the type of an expression except evaluating it. Using 'MkStream'

```
rec  ones    =  1  ::  MkStream ones
rec  cycle   =  <<1 $ 2 $ 3>>  .  MkStream cycle
```

does also not work since 'MkStream' is a function and evaluates its argument when called. The trick is to delay the evaluation of the recursive variable such that it produces the information that it denotes a stream without evaluating any item of the stream:

```
dec  delay  $s  =  <<Open s>>                  or:   dec  delay  =  MClose  o  Open
rec  ones    =  1  ::  delay ones
rec  cycle   =  <<1 $ 2 $ 3>>  .  delay cycle
```

2.4.12. Cutting functionals

In this section, we introduce 5 functions taking a functional object i.e. a function, a constructor, or a macro as their first argument. For the sake of efficiency, they all cut down this argument to be a deterministic function i.e. only the first result of applying the argument to a value is further considered. Without this cutting, the corresponding predefined ML functions could not be used in the implementation.

'Repeat' was already cutting in version 1.4, 'ForAll' and 'Exists' did not exist, and 'Map' and 'Fold' were only defined for sequences and not cutting, thus inefficient in 1.4.

Map: fun → (seq → seq | string → string | stream → stream)

Description: mapping a function over an s-value

- Sequences: The length of the sequence is retained. 'Map F' fails if 'F' fails for an item.
- Streams: The length of finite streams is not retained. Items where 'F' fails, silently disappear.
- Strings: The function must be string valued. All resulting strings are concatenated.

User definition:

```
rec  mapseq f  =      {  [ ]        =>  [ ]
                      \#  h :: t    =>  f h  ::  mapseq f t }
rec  mapstring f  =
        {  ""        =>  ""
        \#  h :: t   =>  case f h of
                             {  fh: string    =>  fh  .  mapstring f t
                             \#  x            =>  Error ("Map (char -> ". Type x .") string") } }
rec  mapstream f s  =  << let  si  =  Open s   in   f si >>
```

(not very expressive since the implicit mapping capability of the semantics of application and 'let' is used)

```
dec  Map f  =    {  s: seq           =>  mapseq (Cut f) s
                 \#  s: string       =>  mapstring (Cut f) s
                 \#  s: stream       =>  mapstream (Cut f) s
                 \#  x               =>  Error ("Map ". Type f ." ". Type x) }
```

('Cut' is predefined and explained below)

Examples:

```
Map  {x  =>  x * x}  [1, 2, 3]   →   [1, 4, 9]
Map  {x  =>  x-1  $  x+1}  [1, 2, 3]   →   [0, 1, 2]
Map  {x  =>  2 * (x/2)}  [2, 4, 5, 6]   →   No
                 (since the argument function fails for 5)
Map  {x  =>  x * x}  << Nat >>   →   << 1  $  4  $  9  $  16  $  ... >>
Map  {x  =>  2 * (x/2)}  << 2  $  4  $  5  $  6 >>   →   << 2  $  4  $  6 >>
Map  {x  =>  x . x}  "TrafoLa"   →   "TTrraaffooLLaa"
Map  StoN  ["1", "2", "3"]   →   [1, 2, 3]
Map  StoN  "123"   →   Error:  Map (char -> num) string
```

```
Map  StoN  ""  →  ""
```

Fold: $(\alpha \to \beta \to \beta) \ \to \ \beta \ \to \ \alpha * \ \to \ \beta$ | ...

Description: folding an s-value with a binary operation
Does not terminate for infinite or open streams

User definition:

```
rec  foldseq f x0  =      {  [ ]            =>  x0
                          \#  h :: t        =>  f h (foldseq f x0 t) }
dec  Cut2 f x y  =  Cut (Cut f x) y     or     dec  Cut2 f  =  Cut f  o  Cut
dec  Fold f x0  =    {  s: (seq | string | stream)  =>  foldseq (Cut2 f) x0 (MkSeq s)
                     \#  s  =>  Error ("Fold ". Type f ." ". Type x0 ." ". Type s) }
```

Examples:

```
Fold  (+)  0  [1, 2, 3]   →   1 + (2 + (3 + 0)))   →   6
```

Operational quantifiers **ForAll** and **Exists** : $(\alpha \to \text{bool}) \ \to \ \alpha * \ \to$ bool | ...

Description: The first argument should be a predicate (a bool-valued function) and the second argument an s-value. The predicate is applied to the items of the s-value in turn, until it fails (then the whole computation fails) or has ‘false’ resp. ‘true’ as its first result. This is then the final answer. Applied to open or infinite streams, ‘ForAll’ loops if it were abstractly true, and ‘Exists’ if it were false.

User definition:

```
dec  ForAll  p  =    {  nil       =>  true
                     \#  h :: t   =>  case  Cut p h  of
                                          {  false    =>  false
                                          \#  true    =>  ForAll p t
                                          \#  y       =>  Error ("ForAll (... -> ". Type y .")" ) }
                     \# x         =>  Error ("ForAll ". Type p ." ". Type x) }
```

Examples:

```
dec  even n   =   n mod 2 == 0
ForAll  even  [2, 4, 6]   →   true
ForAll  even  <<Nat>>   →   false
ForAll  even  <<2 * Nat>>   →   ForAll  even  <<2 $ 4 $ 6 $ ...>>   →   <looping>
ForAll   {n  =>  even (n/2)}   [4, 8, 9]   →   No             (9/2 fails)
ForAll   {n  =>  even (n/2)}   [2, 3, 4]   →   false          (2 produces ‘false’)
```

Repeat: $(\alpha \to \alpha) \ \to \ \alpha \ \to \ \alpha$

Description: f is repeatedly applied until it is no more defined i.e. fails

User definition:

```
rec  Repeat   =      {  f: (fun | mac)      =>   Cut f   o   Repeat f   \   I
                     \#  x                  =>   Error ("Repeat ". Type x) }
```

Examples:

```
Repeat   {n  =>  n / 2}   48    →    3
```

2.4.13. Useful macros

There are some predefined macros to handle the multi-set of values derived from an expression. None of these macros except the basic ‘<<...>>’ was contained in version 1.4.

Catching a multi-value into a stream: **<<** e **>>**

Description: The evaluation of an expression ‘<<e>>’ is easy: the type tag for streams is added to the closure formed from the unevaluated expression ‘e’. An application of ‘Open’ to this stream removes the type tag and reproduces the original multi-value. The stream may also be processed with ‘::’ patterns, or ‘Hd’ and ‘Tl’. The order of the elements in the stream ‘<<e>>’ is the same as the order of appearance of the values of ‘e’ if it were evaluated on top level.

User definition: ?

Examples:

```
<<let (a . b)  =  [1, 2]  in  a>>   →   << [ ]  $  [1]  $  [1, 2] >>
```

Macro catching a multi-value into a stream: **MClose**

Description: prefix form of ‘<<...>>’

User definition:

```
dec  MClose  $e   =   <<e>>
```

```
Example:            MClose  (1 $ 2)   →   <<1 $ 2>>
```

Checking whether expression is defined i.e. does not fail: **MDef**

```
User definition:    dec  MDef  $e   =   Not  (Empty  <<e>>)
                    or:   dec  MDef   =   MClose  o  Empty  o  Not
Example:            MDef  (3 / 2)     →  false          MDef  (4 / 2)     →  true
                    MDef  (1 / 0)     →  Error: 1 / 0   MDef  (1 $ 1/0)   →  true
                    MDef  Nat         →  true           MDef  (let rec  a  =  a  in  a)  →  <looping>
```

Cutting an expression to produce at most one value: **MCut**

```
Description:        'MCut e' fails iff 'e' fails, otherwise it returns the first value of 'e'
User definition:    dec  MCut  $e   =   Hd  <<e>>
                    or:   dec  MCut   =   MClose  o  Hd
Example:            MCut  (3 / 2)     →  No             MCut  (4 / 2)     →  2
                    MCut  (1 / 0)     →  Error: 1 / 0   MCut  (1 $ 1/0)   →  1
                    MCut  Nat         →  1              MCut  (let rec  a  =  a  in  a)  →  <looping>
```

Removing multiple occurrences from of a value: **MSet**

```
User definition:    dec  MSet  $e   =   Open  (MkSet  <<e>>)
                    or:   dec  MSet   =   MClose  o  MkSet  o  Open
Example:            MSet  (1 $ 2 $ 2 $ 1 $ 3)   →   1 $ 2 $ 3
                    MSet  (let rec  a  =  a  in  a)   →   <looping>
                    MSet  (let rec  ones  =  1 $ ones  in  ones)  →  MSet  (1 $ 1 $ 1 $ ...)  →  1 $ <looping>
```

Selecting the nth solution: **NthS**

```
User definition:    dec NthS  =      { n: num      =>   MClose  o  (Nth n)
                                     \#  x         =>   Error ("NthS ". Type x) }
Examples:           NthS  27  Nat   →   27        NthS  4  (1 $ 2)   →   No
```

2.4.14. Related functionals

The macros 'MClose', 'MDef', 'MCut', and 'MSet' possess related functionals '**Close**', '**Def**', '**Cut**', and '**Set**' with the schematic user definition:

dec X f x = MX (f x)   for X = Close, Def, Cut, and Set

Note that 'MX (f x)' and 'X f x' are not equivalent due to the fact that there is no '\$' before 'f' and 'x' i.e. the expressions 'f' and 'x' are evaluated before they come into the domain of the macro.

Remember that an application 'f x' is evaluated by evaluating 'f' to functions f_i and 'x' to values x_i and that the application of f_i to x_j returns y_{ijk}. In 'MX (f x)', the macro applies to all values y_{ijk} together, whereas in 'X f x', it applies for each fixed i_0 and j_0 to the values $y_{i_0 j_0 k}$ only.

Examples:

```
MClose  ( ({x  =>  x + 101  $  x + 102}  $  {y  =>  y + 201  $  y + 202})  (10 $ 20) )
     →  << 111 $ 211 $ 121 $ 221 $ 112 $ 212 $ 122 $ 222 >>
Close  ({x  =>  x + 101  $  x + 102}  $  {y  =>  y + 201  $  y + 202})  (10 $ 20)
     →  << 111 $ 112 >>  $  << 211 $ 212 >>  $  << 121 $ 122 >>  $  << 221 $ 222 >>
Close  I  (1 $ 2)   →   <<1>>  $  <<2>>
Close  Open  [1, 2, 3]   →   << 1 $ 2 $ 3 >>

Cut  Open  [1, 2, 3]   →   1                  Cut  I  (1 $ 2)   →   1 $ 2
Cut  Hd  [ ]   →   No

Def  {0 => 1}  0   →   true                   Def  {0 => 1}  2   →   false
Def  {0 => 1}  (1/0)   →   Error:  1 / 0      Def  {0 => 1}  fail   →   No
Def  {0 => 1}  let rec  a  =  a  in  a   →   <looping>
Def  Hd  [1, 2, 3]   →   true                 Def  Hd  [ ]   →   false
```

Def Hd 17 → Error: Hd num

Set Open ([1, 1] $ [1, 2]) → MSet (1 $ 1) $ MSet (1 $ 2) → 1 $ 1 $ 2

2.4.15. Insertion and extraction

The classical TrafoLa-H operator '^' denotes the insertion of a value into a hole when it occurs in an expression, and the extraction of a subvalue from a big value when it occurs in a pattern. Insertion is done by replacing all occurrences of '@1' in the left operand by the right operand, and changing all occurrences of '@n' where n > 1 into '@(n-1)'. Extraction is done by traversing the big value and replacing some subtree by '@1' and changing all occurrences of '@n' into '@(n+1)'. It is not the exact inverse of insertion since extraction generates left results containing exactly one occurrence of '@1' whereas insertion is possible for any left operand.

In version 1.4, insertion/extraction could not cope with streams. Extraction of a subvalue from a stream gives an infinite number of matches.

User definition of insertion:

```
rec (^) =  { @1                 =>  { w  =>  w }
           \# h: hole           =>  { _  =>  Pred h }
           \# op v              =>  { w  =>  op (v ^ w) }
           \# v: (seq | stream) =>  { w  =>  Map { vi => vi ^ w } v }
           \# v                 =>  { _  =>  v }
           }
```

The expression 'u ^ v' means: replace all occurrences of '@' = '@1' in 'u' by 'v', and decrease the number of other holes by 1. The pattern 'p ^ q' is matched against a value 'u' by decomposing it into two values 'v' and 'w' such that 'v ^ w = u' holds and 'v' contains exactly one occurrence of '@'. By the last condition, this is not the exact inverse of insertion.

Examples:

con add, sub, mul, a, b, c;

add [@1, @2] ^ a ^ b → add [a, @1] ^ b → add [a, b]
add [@1, @1] ^ a → add [a, a]

add [@1, @2] ^ sub [@1, @2] ^ mul [a, @1] ^ b ^ c →
  add [sub [@1, @2], @1] ^ mul [a, @1] ^ b ^ c →
  add [sub [mul [a, @1], @1], mul [a, @1]] ^ b ^ c →
  add [sub [mul [a, b], b], mul [a, b]] ^ c →
  add [sub [mul [a, b], b], mul [a, b]]

The pattern 'x ^ y' matches the value 'add [a, a]' and gives 4 solutions:
x = @ and y = add [a, a], or x = add @ and y = [a, a], or
x = add [@, a] and y = a, or x = add [a, @] and y = a.
The solution ' x = add [@1, @1] and y = a ' is **not** produced.

The pattern 'x ^ y' matches the stream '<<Nat>>' and produces an infinite number of solutions:
x = @, y = <<Nat>>
x = <<@ $ 2 $ 3 $ ...>>, y = 1
x = <<1 $ @ $ 3 $ ...>>, y = 2 etc.

2.4.16. Trace functions

There is no means to observe the evaluation of a TrafoLa-H expression directly. But it is possible to decorate subexpressions with the predefined functions '**MPrint**' and '**Print**' that print a message on the screen as side effect when they are evaluated.

The expression 'MPrint s e' where 's' is a string prints the message '{s}' when it is evaluated and causes the evaluation of the expression 'e'.

The expression 'Print s e' where 's' is a string prints the message '{s, v_i}' whenever its context needs a new value where v_1, v_2, etc. are the values of the expression ‘e’.

User definitions specifying the declarative meaning of the trace functions without their side effect:

```
dec  MPrint  =   { s: string      =>   { $v     =>   /* write ("{". s ."}") */    v }
                 \#  x            =>   Error ("MPrint ". Type x) }
dec  Print   =   { s: string      =>   { v      =>   /* write ("{" . s . ", " . mkstring v . "}") */    v }
                 \#  x            =>   Error ("Print ". Type x) }
```

Examples:

```
- 1;
> 1
- MPrint "a" 1;
{a}> 1
- Print "b" 1;
{b, 1}> 1
- dec  $ pnat   =   Print  "p"  Nat;
> pnat : not yet evaluated
- pnat;
{p, 1} 1 ;
{p, 2} 2 q
- NthS  5  pnat;
{p, 3}{p, 4}{p, 5}> 5
- pnat;
1 ;   2 ;   3 ;   4 ;   5 ;   {p, 6} 6 ;   {p, 7} 7 q
```

Notice the lazy evaluation of ‘pnat’!

2.4.17. The special variable ‘It’

The multi-set resulting from the evaluation of an expression at top level is automatically bound to the variable ‘It’. An expression ‘e’ on top level is handled as if it were given in a declaration ‘dec $ It = e’.

Examples:

```
- 4;
> 4
- It;
> 4
- It + 1;
> 5
- It;
> 5

- Close  I  (1 $ 2);                  (the result is  <<1>> $ <<2>>, but this cannot be seen)
<<...>> ;
<<...>> ;
No
- It;                                 (Calling ‘It’ always gives the same results
<<...>> ;                              as the previous expression)
<<...>> ;
No
- dec  $temp   =   It;                (The solutions are temporarily saved in ‘temp’)
> temp : not yet evaluated
- Open  (NthS 1 temp);                (Now we can regard the contents ...)
> 1
- Open  (NthS 2 temp);                (... of the two streams in turn)
> 2
```

```
- #k temp                  (the variable 'temp' is no longer needed)
> Binding killed
```

2.5. Order of results

When a complex expression gets the request to produce the next result, it submits this request to its subexpressions. When the subexpressions produce their next result or definitely fail, these positive or negative results are combined by the expression.

The distribution of the requests to the subexpressions is deterministic, but fair. Each subexpression gets the chance to produce a value. The only problem results from subexpressions starting an infinite computation without delivering results or definitely failing. Their computation is not stopped (except by '^C' that stops everything) such that other subexpressions never again get the chance to produce a result.

Thus, there are three possibilities for the multi-set of results:

1) It is finite and completely enumerated: v_1; v_2; ... No
2) It is infinite and potentially completely enumerated, i.e. if there is a special result 'v', the user needs only to demand the next result often enough to see 'v' eventually. Because of the fair distribution of requests, it is impossible that the multi-set of results is split into two parts where only one is enumerated.
3) It is finite or infinite, but not completely enumerated, since the interpreter after producing some results begins to loop forever without producing 'No' or the next result.

Examples (we sometimes write the solutions into one line to save space):

```
- Nat;                                  (The natural numbers are enumerated.
1 ;                                      To see e.g. 1000, the user only has to
2 ;                                      hit 999 times the ';' key.)
3 q

- { x => x * x } Nat;                   (The square numbers are completely enumerated.)
1 ; 4 ; 9 q

- 0 $ Nat $ - Nat;                      (enumerating the integers)
0 ; 1 ; -1 ; 2 ; -2 ; 3 ; -3 q          (fair enumeration)

- [Nat, Nat];                           (all pairs of naturals are enumerated)
[1, 1] ; [1, 2] ; [2, 1] ; [1, 3] ; [3, 1] ; [2, 2] ; [4, 1] q

- { x => if x < 5 then x } Nat;         (a dangerous algorithm)
1 ; 2 ; 3 ; 4 ;                         (now, the interpreter starts to loop,
^C                                       without producing the next result.
Interrupt                                In fact, there is none.)

- - Nat $ { x => if x < 3 then x } Nat;
-1 ; 1 ; -2 ; 2 ; -3 ;                  (looping without producing '-4')
^CInterrupt

- rec From n = n $ From (n + 1);
> From : not yet evaluated

- From 5;
5 ; 6 ; 7 ; 8 q                         (no problems)

- rec From n = From (n + 1) $ n;        (This does not work!)
> From : not yet evaluated

- From 1;                               (loops without results)
^CInterrupt
```

The new function ‘From’ did not work because the ‘$’ operator always gives the first request to produce a value to its first operand, and does so in each new recursive incarnation.

From an abstract point of view, the order of appearance of results is immaterial since the results are considered to form a multi-set. But due to the stream-forming operator ‘<< >>’ and its derivatives ‘MCut’ and ‘Cut’, the order greatly influences the final multi-set of results and cannot be ignored. Practically, the exact order is hard to predict even for the creator of the system although it is deterministic.

Examples:
(Nat $ 10 * Nat) $ 100 * Nat → 1 $ 100 $ 10 $ 200 $ 2 $ 300 $ 20 $ 400 $ ...
Nat $ (10 * Nat $ 100 * Nat) → 1 $ 10 $ 2 $ 100 $ 3 $ 20 $ 4 $ 200 $ 5 $ ...
[Nat, Nat, Nat] → [1, 1, 1] $ [1, 1, 2] $ [2, 1, 1] $ [1, 2, 1] $ [3, 1, 1] $ [2, 1, 2] $ ...

dec pair1 x y = [x, y]
pair1 (10 $ 20) (1 $ 2) → [10, 1] $ [20, 1] $ [10, 2] $ [20, 2]

dec pair2 $x $y = [x, y]
pair2 (10 $ 20) (1 $ 2) → [10, 1] $ [10, 2] $ [20, 1] $ [20, 2]

This is a trade-off: in Prolog, the order is easy to predict, but often, many solutions are lost, whereas in TrafoLa-H, the order is obscure, but solutions may only be lost by looping subexpressions.

3. Known deficiencies of the language

The following enumeration does not pretend to be complete. All users are invited to complain about further deficiencies.

- There are differences between the language of this interpreter and the language of the interpreter written in SSL. Naturally, the main difference is the absence of backtracking there.
- A looping subexpression prevents other subexpressions from producing results.
- There are some restrictions for strings that should either be overcome, or the user should be hindered to enter such strings.
- There are no input / output functions in the language itself.
- The precedence of ‘mod’ is higher than that of ‘div’ that is again higher than that of ‘/’. These three operators should have the same precedence. This and other changes would also make the parser more compact.
- The user cannot define new infix operators. In addition, there is no means to introduce the concrete syntax of the object language.
- There is no compile-time checking, i.e. unbound identifiers or type errors are not detected before the actual evaluation.
  Example: dec f = { a => b }
  where ‘b’ is unbound. This is not detected, and ‘f’ is bound to a function that always produces an error when applied.
- Equality on abstract values is strange.
- There are no means to recognize error values in the language itself. The production of an error value breaks backtracking e.g. ‘1 $ 2 / 0 $ 3’ only produces ‘1’ and the error message ‘2 / 0’, but not ‘3’.
- The order of appearance of results is strange and cannot be influenced by the user.
- The incomplete symmetry between patterns and expressions should be more thoroughly investigated with respect to user defined classes and pattern operators and/or automatic inversion of functions in patterns (as is presently done with ‘::’, ‘.’, ‘+:’ and roughly with ‘^’).

4. Updates with respect to version 1.4

Version 1.5 differs in many details from its predecessor, version 1.4b, whose interpreter was distributed in June 1988 at Madrid. The language was extended by user defined macros i.e. lazy functions, and user definable error messages. Some new operators were introduced, and the set of predefined functions was completely revised and extended.

4.1. Syntax

Due to the introduction of user defined macros, the syntax of declarations had to be revised. In version 1.4, there were declarations introduced by ‘$’. They are now generalized to lazy resp. macro declarations, and introduced by ‘dec’ or ‘rec’ as any other declaration. The ‘$’ sign was retained as marker for macro patterns. Macro patterns can now occur in declarations, formal argument lists, and before ‘=>’ in an abstraction.

```
    { x => x + x }            and            { $x => x + x }
```

have completely different semantics; see section 2.3.

Constructors are now lexically equal to variables; they are introduced by ‘con’ declarations and removed by ‘var’ declarations.

There are some minor updates in 1.5 that are not compatible with 1.4:

The basic class ‘empty’ in patterns is now called ‘<< >>’, and there are new basic classes ‘nil’ and ‘mac’.

The cut operator in patterns was changed from <<p>> to <p> to avoid confusion with the freezing operator <<e>> in expressions. The former ‘$’ operator in patterns was removed and its semantics was added to the ‘::’ operator. The priorities of the operators ‘|’ and ‘\’ and also of the related ones ‘#’ and ‘\#’ were exchanged. ‘\#’ is now no longer a syntactic abbreviation for an expression involving ‘\’; its semantics was slightly changed. There are a few new operators: ‘\$’, ‘\|’, ‘\&’, ‘+:’, ‘/’; the former ‘/’ was renamed into ‘div’.

Semantic changes of the kernel language, the operators, and the predefined values will be explained below.

Some rules of the 1.4 syntax were unnaturally restrictive; they were relaxed and generalized:

In 1.4, definitions by a pattern and non-recursive function definitions could not be intermixed e.g.

```
    dec (a.b) = [1,2] and f x = x + 1
```

was illegal. Now, both kinds of definitions may be freely intermixed after ‘dec’.

In version 1.4, the pattern in a definition had to be parenthesized, and similarly, the pattern after ‘matches’ had to be enclosed within braces. This is now relaxed; the pattern only has to be syntactically closed.

```
    dec [a,b] = [1,2]            matches [a,b]
```

were illegal in 1.4; you had to write

```
    dec ([a,b]) = [1,2]          matches {[a,b]}
```

Note that these updates are compatible since the new form comprises the old one.

4.2. Semantics

From its origins, TrafoLa-H is a ‘call-by-value’ language i.e. originally, all expressions were evaluated no matter whether their value was needed. Starting with version 1.4, various lazy constructs appeared in the language. Version 1.4 contained lazy declarations and streams as new features, and 1.5 additionally contains macros i.e. lazy functions.

In version 1.4, there was a difference between recursive and lazily created pseudo values. Recursive ones were treated less efficiently. This was the reason why there was a loss of up to a factor of 20 when evaluating applications of recursive functions in version 1.4.

Compared with version 1.4, the predefined operations are completely refined and extended. Specially, the operations on streams are now overloaded with those on sequences.

Basic relational operations: *‘$’ ‘\$’*

‘$’ in expressions is unchanged. ‘$’ in patterns was removed and its semantics added to the ‘::’ operator. ‘\$’ is new in 1.5.

Basic functional forms: *‘|’ ‘#’ ‘\’ ‘\#’ ‘o’ I*

‘|’, ‘#’, ‘\’, and ‘I’ are not changed. The result of ‘f o g’ is a macro now, i.e. ‘f o g’ does not evaluate its argument; this is left to ‘f’ and ‘g’.

Concerning the operator ‘\#’ for function definitions by cases with increasing priority, the decision which case is to be selected depended on both patterns and expressions in 1.4. Now, it depends only on the patterns such that ‘\#’ and ‘\’ are no longer analogous.

Examples:

({ 0 => fail } \\ { x => 5 }) 0 → 5	*in 1.4 and 1.5*
{ 0 => fail \\# x => 5 } 0 → 5	*in 1.4*
{ 0 => fail \\# x => 5 } 0 → No	*in 1.5*

Pattern operations: *‘|’ ‘\’ ‘&’ ‘!’ ‘{ }’ ‘where’ ‘< >’*

All these operators were already contained in version 1.4. The only difference is that the cut operator was written ‘<<p>>’ instead of ‘<p>’ there.

Issuing error messages: *Type Error*

Both functions described here are new; in version 1.4, the user could not program error messages.

Conversions: *HtoN NtoH CtoS NtoS StoN*

All these functions were already contained in version 1.4. ‘CtoS’ was called ‘OtoS’ there, and it had a slightly different meaning since it omitted the leading quote that was obligatory for constructors in 1.4.

Boolean operations: *‘if’ ‘||’ ‘&&’ ‘\|’ ‘\&’ Not*

Version 1.4 merely contained ‘if’, ‘||’, and ‘&&’; ‘\|’, ‘\&’, and ‘Not’ are new. ‘if’ was not user definable in 1.4 due to the lack of macros.

Equality operations: *‘==’ ‘!=’ ‘&’*

These three operations were already contained in version 1.4 with the same properties. Though, some concrete instances of equality checks might now produce other results than in 1.4 since the equality check is not completely functional and depends on implementation details.

Order operations: *‘<’ ‘>’ ‘<=’ ‘>=’*

No changes with respect to 1.4.

Numbers and arithmetic: *‘$-_1$’ Abs ‘+’ ‘-’ ‘*’ Succ Pred ‘div’ ‘mod’ ‘/’ Nat*

Unary minus, ‘Abs’, ‘+’, ‘-’, and ‘*’ are unchanged, ‘Succ’, ‘Pred’, ‘/’, and ‘Nat’ are new. The former ‘/’ was renamed into ‘div’. A bug in the implementation of ‘div’ and ‘mod’ was corrected.

Sequences, streams, and strings: *‘nil’ ‘::’ Empty Hd Tl ‘.’ ‘+:’ Len Nth First Member Rev Open MkSeq MkStream MkSet Implode Substr SeqTo StreamTo*

In version 1.4, many predefined operations worked equally for sequences and strings, but not for streams, and there were special operations for streams. This was unified, now most operations work for sequences, strings, and streams equally. The user may also specify such functions easily by using the unspecific pattern operators ‘nil’ and ‘::’.

‘empty’ of 1.4 was renamed into ‘<< >>’.

‘nil’ is new.

‘::’ matched sequences only, and ‘$’ was used for streams. The new ‘::’ matches sequences, streams, and strings.

‘Empty’ was extended to sequences and strings.

‘Hd’ was defined for sequences and strings only, and ‘Head’ for streams. They were combined to the new ‘Hd’. Analogously, ‘Tl’ is the union of the old ‘Tl’ and ‘Tail’. Both functions produced an error when applied to empty s-values; now, they fail.

‘.’ in patterns and expressions was extended to streams.

‘+:’ in patterns and expressions is new.

‘Len’, ‘Nth’, and ‘Member’ were extended to streams. ‘First’ is new.

‘Rev’ was extended to streams. The user definition given in the 1.4 manual was wrong.

‘Open’ worked for streams only; it was extended to sequences and strings.

‘MkSeq’ was called ‘Explode’ and worked for strings only.

‘MkStream’ and ‘MkSeq’ were not yet contained in 1.4.

‘Implode’ and ‘Substr’ are unchanged.

‘SeqTo’ was called ‘Iota’ in 1.4. ‘StreamTo’ is new.

Cutting functionals: *Map Fold ForAll Exists Repeat*

‘Repeat’ was already cutting in version 1.4, ‘ForAll’ and ‘Exists’ did not exist, and ‘Map’ and ‘Fold’ were only defined for sequences and not cutting, thus inefficient in 1.4.

Useful macros: *‘<< >>’ MClose MDef MCut MSet NthS*

None of these macros except the basic ‘<<...>>’ was contained in version 1.4.

Related functionals: *Close Def Cut Set*

‘Set’ is new, the others were not changed.

Insertion and extraction: ‘^’

In version 1.4, insertion/extraction could not cope with streams. Extraction of a subvalue from a stream gives an infinite number of matches.

Trace functions: *MPrint Print*

New in 1.5.

The special variable ‘It’

No changes.