The mLite Language

Nils M Holm, 2014

mLite is a general-purpose, functional, lightweight, dynamic programming language. It borrows ideas from both the Scheme [R4RS] and Standard ML [DEFSML] languages, taking most of its syntax from ML and its dynamic nature from Scheme. It extends ML-style pattern matching by adding guarded patterns and also introduces the principle of *implicit guards*. The implementation presented here is intended to be portable and easily realized on top of existing Scheme systems.

Contents

Introduction	**2**
Pattern Matching	3
Syntax and Informal Semantics	4
Function Application	*5*
Conditional Evaluation	*6*
Guarded Patterns	*7*
Exception Handling	*9*
Input and Output	*10*
Declaration Syntax and Semantics	11
Operator Declarations	*14*
Algebraic Types	*15*
The Lite Abstract Machine	**18**
LAM Syntax	18
Scheme-Like Syntax	*19*
Non-Scheme Syntax	*20*
Scheme-Like Functions	*21*
Renamed Functions	*22*
mLite Functions	*22*
Compiling mLite to LAM	**24**
Data Objects	24
Functions and Application	24
Expressions	25
Declarations	26
Local Declarations	28
Derived Forms	29
mLite Reference	**31**
mLite/LAM Data Objects	31
Predefined Infix Operators	32
Pattern Matching Algorithm	32
Implicit Guard Patterns	*33*
mLite Syntax Summary	35
Declarations	*35*
Expressions	*37*
mLite Function Summary	39
Arithmetics	*39*
Structural Operations	*42*
Type Predicates and Conversion	*46*
Char Functions	*47*
Input/Output Functions	*48*
Appendix	**50**
mLite Grammar	50
mLite and Scheme Functions	56
Differences to ML	56
References	58

Introduction

mLite is a functional language with immutable data types (with the exception of vectors), which reduces surprises in the process of program development to a minimum. Its syntax is lightweight and math-oriented. Here is a simple example that creates a list of numbers (*iota*), and then uses a generic *extract* function to extract all multiples of 7 from such a list:

$$\begin{aligned} &\mathit{fun}\ \mathit{iota}\ (0, a) = a \\ &\qquad \mid (x, a) = \mathit{iota}\ (x - 1, x :: a) \\ &\qquad \mid x = \mathit{iota}\ (x, []) \end{aligned}$$

$$\begin{aligned} &\mathit{fun}\ \mathit{extract}\ (f, []) = [] \\ &\qquad\qquad \mid (f, f\ x :: xs) = x :: \mathit{extract}\ (f, xs) \\ &\qquad\qquad \mid (f, x :: xs) = \mathit{extract}\ (f, xs); \end{aligned}$$

$$\mathit{extract}(\mathit{fn}\ x = x\ \mathit{mod}\ 7 = 0, \mathit{iota}\ 123)$$

mLite makes extensive use of *pattern matching*. All functions are unary, and multiple arguments are passed to functions in tuples and destructured by patterns. Tuples are delimited by parentheses and their elements are separated by commas. Lists use square brackets instead of parentheses.

Lists and tuples are built dynamically in expressions (but not in patterns), so $[1+2, 3+4]$ gives $[3, 7]$ and (a, b) would be a tuple containing the values of the variables a and b.

$::$ is the “cons” operator that adds a new element to the front of a list. When it appears in a pattern, it breaks up a list into its head (first element) and tail (rest of elements).

Each function contains a set of pattern/expression pairs of the form
$\mathit{pattern}_1 = \mathit{expression}_1 \mid \cdots$

The vertical bar is used to separate multiple pattern/expression pairs. When a function is applied to a value, the value is matched against each pattern in sequence, binding variables in the pattern to corresponding components in the argument. When the match succeeds, the associated expression is evaluated with the bindings in effect. In case of a mismatch, the remaining patterns are tried. When no more patterns are left to try, an error is reported.

Patterns may contain implicit or explicit guards that place further constraints on the values that match a pattern. For example, the second pattern of *extract* matches only if f applied to x is true.

Functions for destructuring compound data objects, like car/cdr or first/rest in Lisp are rarely used, as pattern matching does the same job, but more intuitively:

$fun\ head\ (h :: t) = h$
$fun\ tail\ (h :: t) = t$

The *head*/*tail* functions are mostly used in higher-order constructs.

Pattern Matching

Function application matches data objects against patterns. Patterns may be any data objects themselves, and they may or may not contain variables. All literal objects match themselves. For instance, the objects

$0 \quad 123.45 \quad "hello" \quad \#"x" \quad () \quad [] \quad true$

would match themselves exactly (with the usual caveats regarding real numbers in mind). Note that *true* is the “truth” constant and not a variable.

Tuples and lists are matched element-wise, so

$(1,2,3)$ matches $(1,2,3)$ and
$[1,2,3]$ matches $[1,2,3]$

Nested lists and tuples as well as lists containing tuples and tuples containing lists are matched recursively, so even this data object would match itself:

$[(1,2),(2,4),(3,9)]$

Things get more interesting, though, when a pattern contains variables. For instance,

$[(1,x)]$

would match any list containing a 2-tuple with a 1 in the first slot of the tuple. *In addition* it would *bind* the x in the pattern to the value in the corresponding data object, so matching

$[(1,x)]$ against $[(1,23)]$

would bind x to 23.

Patterns may also contain *constructors*, like $::$. In an expression, $::$ adds a new element to the front of a list, but in a pattern it deconstructs a list, so matching

$(1,x)::t$ against $[(1,23),(2,34),(3,45)]$

would bind x to 23 and t to $[(2,34),(3,45)]$.

Patterns containing the $::$ constructor are frequently put in parentheses for greater clarity, but this is not a syntactic necessity.

When the identifier _ appears in a pattern, it matches any data object, just like a variable, but no value is bound. For instance, in the $length$ function, which computes the number of elements of a list, we are not interested in the values of the individual list elements, so we use _ instead of a variable:

$$\begin{aligned}&fun\ length\ [\,] = 0\\ &\qquad\qquad\mid (\_ :: t) = 1 + len\ t\end{aligned}$$

The complete pattern matching algorithm can be found in the mLite Reference (pg 32).

Syntax and Informal Semantics

mLite expressions are based upon function application and infix operators. The usual operators exist:

$$\begin{aligned}&x+y;\ \ x-y;\ \ x*y;\ \ x/y;\ \ x\ div\ y;\ \ x\ rem\ y;\\ &x=y;\ \ x<y;\ \ x<>y;\ \ x<=y;\ \ x>=y;\ \ \cdots\end{aligned}$$

There are also operators for “consing” an element to a list (::) and for appending lists and strings (@).

Many operators are *polymorphic*, like @. Other examples include the comparison operators (=, <, <>, etc), which can be used to compare not only numbers, but also characters and strings.

There are case-insensitive counterparts to the comparison operators that begin with a tilde (~). For example, $a\tilde{} = b$ tests whether the strings or characters a and b are the same after folding their case. When applied to numbers, there is no difference between case-sensitive and case-insensitive operators, e.g. $x = y$ is equal to $x\tilde{}{=}y$, if both x and y are numeric.

Function Application

Function application is denoted by juxtaposition and associates to the left. E.g. the expression $f\ x\ y$ is the same as $(f\ x)\ y$. It first applies f to x and the result of $f\ x$ to y.

This notation is particularly useful in combination with higher-order functions. For instance, *map* is a function of a function f to a function which maps f over its argument. So:

$map\ (fn\ x = x * x)\ \rightarrow\ fn\ a$
$map\ (fn\ x = x * x)\ [1,2,3]\ \rightarrow\ [1,4,9]$

($a \rightarrow b$ means "a maps to b" or "a evaluates to b".)

Parentheses can be used to override the precedence and associativity of all operators and function application. See the mLite Reference for a list of all predefined operators (pg 32) and functions (pg 39).

Alternatively, the apply (‘) operator can be used to change the associativity of function application. It is a low-precedence operator that applies its left-hand side to its right-hand side, so

f ‘ g ‘ $x * y$

would be equal to $f\ (g\ (x * y))$. This is the same as Haskell's "dollar" operator.

The *fn* keyword introduces an anonymous function, just like "lambda" in Scheme. Functions defined with *fn* may have multiple patterns, just like functions defined with *fun*:

$fn\ false = true \mid \_ = false$

Curried functions can be created by putting multiple patterns between the *fn* keyword and the = operator introducing the function body. For instance,

$fn\ a\ b\ c = a + b + c$

is a shorthand notation for

$fn\ a = fn\ b = fn\ c = a + b + c$

This works even in *fun* declarations:

$fun\ f\ \ a\ b\ c = a + b + c$

Conditional Evaluation

The *if* syntax is used to evaluate expressions conditionally:

$$
\begin{aligned}
fun\ gcd\ (a, b) = {} & if\ b = 0\ then \\
& \quad a \\
& else\ if\ a = 0\ then \\
& \quad b \\
& else\ \mathrm{if}\ a < b\ then \\
& \quad gcd\ (a, b\ mod\ a) \\
& else \\
& \quad gcd\ (b, a\ mod\ b)
\end{aligned}
$$

Note, though, that it is often more intuitive to replace *if* by pattern matching wherever possible:

$$
\begin{aligned}
fun\ gcd\ & (a, 0) = a \\
& |\ (0, b) = b \\
& |\ (a, b) = \mathrm{if}\ a < b\ then \\
& \qquad\qquad gcd\ (a, b\ mod\ a) \\
& \qquad\quad else \\
& \qquad\qquad gcd\ (b, a\ mod\ b)
\end{aligned}
$$

The *or* and *also* operators implement *short-circuit* logic operations:

$$
\begin{aligned}
& true\ or\ "x" \ \rightarrow\ true \\
& false\ or\ "x" \ \rightarrow\ "x" \\
& true\ also\ "x" \ \rightarrow\ "x" \\
& false\ also\ "x" \ \rightarrow\ false
\end{aligned}
$$

Because of the short-circuit nature of these operators, the following expressions are save:

$\mathit{true}\ \mathit{or}\ 1\ \mathit{div}\ 0$
$\mathit{false}\ \mathit{also}\ 1\ \mathit{div}\ 0$

When the operators are chained, they associate to the left and *also* binds stronger than *or*.

The *case* syntax is merely a thin coating of syntactic sugar on top of *fn*:

$\mathit{case}\ x\ \mathit{of}\ \ \mathit{pattern} = \mathit{expression} \mid \cdots$

is merely an alternative form of the function application

$(\mathit{fn}\ \ \mathit{pattern} = \mathit{expression} \mid \ldots)\ x$

It moves the argument to the beginning of the expression, which can increase readability in some cases.

Guarded Patterns

mLite extends the ML-style pattern matching mechanism by introducing implicit and explicit *guards*. A guard is an expression that is part of a pattern rather than a function body. It is evaluates after matching a pattern, but *before* evaluating the associated body. The body is only evaluated, if the guard evaluates to a “true” (i.e. non-*false*) value. Otherwise the match fails and the next pattern is tried.

This is a version of the *gcd* function using a guard:

$$\begin{aligned}
&\mathit{fun}\ \mathit{gcd}\ (a, 0) = a\\
&\qquad\quad \mid (0, b) = b\\
&\qquad\quad \mid (a, b)\ \mathit{where}\ a < b\\
&\qquad\qquad\qquad = \mathit{gcd}\ (a, b\ \mathit{mod}\ a)\\
&\qquad\quad \mid (a, b) = \mathit{gcd}\ (b, a\ \mathit{mod}\ b)
\end{aligned}$$

The *where* keyword introduces the guard $a < b$. So the first pattern (a, b) matches only if $a < b$. The second (a, b), without any guard, matches if the first one did not match.

When a guard deals only with a single variable, it can be placed in the pattern itself. This is called in *implicit* guard. Implicit guards often allow to express functions in a more natural way. For example:

$$\begin{aligned} &sgn\ x < 0 = \tilde{}1 \\ &\quad |\ x > 0 = 1 \\ &\quad |\ \_ = 0 \end{aligned}$$

instead of

$$\begin{aligned} &sgn\ x\ where\ x < 0 = \tilde{}1 \\ &\quad |\ x\ where\ x > 0 = 1 \\ &\quad |\ \_ = 0 \end{aligned}$$

Note: because = is also used to introduce function bodies, the guard expression has to be parenthesized when using this operator, as in

$$fn\ (x\ mod\ 2 = 0) = ...$$

or

$$fn\ (a, b)\ where\ (a = b) = ...$$

Even function application has to appear in parentheses in guard expressions to distinguish it from currying. For instance:

$$fn\ f\ \ x = ... \quad \text{equals} \quad fn\ f = fn\ x = ...$$

but

$$fn\ (f\ \ x) = ... \quad \text{equals} \quad fn\ x\ where\ (f\ \ x) = ...$$

Parentheses can be omitted when an equal operator or function application appears inside of a list or tuple.

Multiple implicit guards can be used in the same tuple or list:

$$fn\ [x < 0, y > 0] = ...$$

would be equal to

$$fn\ [x, y]\ where\ (x < 0\ also\ y > 0) = ...$$

Each guard evaluates inside of the lexical environment of the function containing it, so functions can close over identifiers used in guards. The following *filter* function is a curried version of the *extract* function in the initial example (pg 2):

This function works, because $f2$ closes over f, and $f\ x$ is evaluated inside of the environment of $f2$.

$$
\begin{aligned}
&fun\ \ filter\ \ f =\\
&\quad let\ \ fun\ \ f2\ ([\,], r) = rev\ \ r\\
&\qquad\qquad\quad \mid (f\ \ x :: xs, r) = f2\ (xs, x :: r)\\
&\qquad\qquad\quad \mid (x :: xs, r) = f2\ (xs, r)\\
&\quad in\ \ fn\ \ a = f2\ (a, [\,])\\
&\quad end
\end{aligned}
$$

Exception Handling

The declaration

exception :foo

creates a new exception named *:foo*.

Exceptions are technically constructors, so their names must begin with a colon, just like ::.

The *raise* keyword *raises* an exception, i.e. it notifies the program that something interesting happened. When there is no *handler* for the given exception, the program reacts to the notification by terminating and printing a message. So the expression

raise :foo

will just terminate program execution.

Exceptions are frequently used to indicate *unusual* conditions during program execution. For instance, we might expect a list of single-digit numbers in the *sum* function. We catch the case of a non-digit by raising the *:not_a_digit* exception.

exception :not_a_digit

$$
\begin{aligned}
&fun\ sum\ [\,] = 0\\
&\qquad\quad \mid ((x > 9\ \ or\ \ x < 0) :: \_) = raise\ :not\_a\_digit\\
&\qquad\quad \mid (x :: t) = x + sum\ t
\end{aligned}
$$

An *exception handler* is installed with the *handle* operator. An exception handler is an ordinary function with exception names as patterns. It will handle all exceptions listed as patterns, so the handler in

(*raise :foo*) *handle :foo* = *"caught!"*

will return the value *"caught!"*.

An exception handler offers a way to provide a non-local exit and a default value for a failed computation at the same time. It can be thought of as a “jump” from the *raise* to the handler. For instance, we can exit from *sum* *without* performing all pending + operations caused by recursion:

$sum\ [1,2,3,99,4,5]\ handle\ {:}not\_a\_digit = false$

Exception handling can also be used to implement *backtracking*. Exploring why the following program generates the correct solution

$change(16,[5,2]) \ \rightarrow\ [5,5,2,2,2]$

is left as an exercise to the reader.

```
exception :out_of_coins

fun change (0, _) = []
         | (_, []) = raise :out_of_coins
         | (amount, coin :: coins) =
             if coin > amount then
               change (amount, coins)
             else
               coin :: change (amount - coin, coin :: coins)
               handle :out_of_coins = change (amount, coins)
```

Input and Output

There are the usual functions for reading single characters (*readc*), reading and writing lines of text (*readln*, *println*), as well as for single-character look-ahead (*peekc*).

The *print* and *println* functions (which differ only in the point that *println* emits a final newline sequence) can be used to write any type of data object. The printer will use a human-readable representation for the object. These functions are similar to Scheme’s “display” procedure.

New *I/O streams* are created by the *instream* and *outstream* functions and closed by *close*. The garbage collector will automatically close unused streams.

The $\ll$ (“receive output”) and $\gg$ (“send input”) operators are used to redirect the input/output of the I/O functions to user-created streams.

The following program writes “Hello, World!” to the file “hello.txt”:

$outstream\ "hello.txt" \ll println\ "Hello, World!"$

The $\ll$ operator makes the stream on its left-hand side receive the output of the expression to its right. The $\gg$ operator passes input from the source to its left to the expression on its right.

The following expression re-reads the text written above:

$instream\ "hello.txt" \gg readln\ ()$

The sequence (;) operator orders the effects of I/O expressions (any expressions, in fact). It first evaluates the expression to its left and then the expression to its right. It associates to the left, so chains of ; operators are equal to Scheme’s “begin”.

The following sequence opens a file, writes to it, and closes it:

$val\ f = outstream\ "pi";\ f \ll println\ `\ 355/113;\ close\ f$

Declaration Syntax and Semantics

$val\ pattern = expression$

The *val* declaration matches *one* pattern against *one* expression and binds the variables in the pattern to the corresponding components of the expression. Here are some examples (;; introduces a comment to the end of line):

$val\ x = 1$;; *bind x to* 1
$val\ (x, y) = (1, 2)$;; *bind x to* 1 *and y to* 2
$val\ (1, [x], 3) = (1, [2], 3)$;; *bind x to* 2

When the pattern does not match the expression, an error is reported.

$val\ p_1 = x_1\ [\ and\ p_2 = x_2 \cdots]$

Multiple bindings can be established in parallel by chaining them together with *and* (the brackets indicate an optional part here).

Parallel bindings will be established by first computing *all* expressions and then binding the variables. So the following declaration would in fact swap the values of x and y:

$val\ \ x = y\ and\ y = x$

BTW, variables inside of a single pattern are also bound in parallel, so this version would also swap the values of x and y:

$val\ \ (x, y) = (y, x)$

For sequential bindings, the sequence operator ; is used:

$val\ \ x = 1; val\ \ y = x + 1; val\ \ z = y + 1$

Function declarations have been used throughout this text without ever explaining them in detail.

$fun\ id_1\ \ p_1\ [where\ x_g] = x_1 \mid \cdots [and\ fun\ id_2 \cdots]$
$fun\ id_1\ \ p_1 \cdots [where\ x_g] = x\ [and\ fun\ id_2 \cdots]$

The declaration $fun\ id\ \ p = x$ for any pattern p and expression x is *almost* the same as the declaration $val\ id = fn\ p = x$, i.e. it binds an identifier to a function.

The only difference is that *local* functions defined with fun may be recursive, while *local* functions defined with val may not. At the top level of a program, there is no difference between these declarations.

Functions declared by fun can be curried by using shorthand notations like

$$fun\ add\ \ a\ b = a + b$$

instead of

$$fun\ add\ \ a = fn\ b = a + b$$

Note, however, that currying cannot be combined with alternative patterns, so the typical ML-style, where multiple cases are used with curried patterns, cannot be replicated in mLite.

Like most other declarations, multiple instances of fun may be chained together with and. However, the combination of $and\ fun$ is only useful when defining *local mutually recursive* functions. See let, below, for an example.

Each pattern of a function may have an associated guard x_g (or an implicit guard). In this case, the pattern matches only, if the guard expression delivers a “true” value.

$local\ ldecl_1\ [\ ;\ ldecl_2 \cdots\ ]\ in\ decl_1\ [\ ;\ decl_2 \cdots\ ]\ end$

The *local* construct evaluates the declarations $ldecl_1 \cdots$ and with the bindings established by these declarations in effect, it evaluates $decl_1 \cdots$. After evaluating the *decl*’s, the *ldecl* bindings are removed, but the *decl*’s stay in effect.

local is used to hide local declarations from the top level. For instance, the *revlist* function, which reverses a list, could be implemented as follows:

$$
\begin{array}{l}
local \\
\quad fun\ rev'\ ([], b) = b \\
\qquad\qquad |\ (h :: t, b) = rev'\ (t, h :: b) \\
in \\
\quad fun\ revlist\ a = rev'\ (a, []) \\
end
\end{array}
$$

Here the rev' function, which uses the accumulator b to reverse the list a in linear time, is hidden from the outer context.

Only *fun* and *val* bindings may be used in local declarations.

$let\ ldecl_1\ [\ ;\ ldecl_2 \cdots\ ]\ in\ expr_1\ [\ ;\ expr_2 \cdots\ ]\ end$

let is like *local*, but instead of declaring bindings in its body (the part between *in* and *end*), it evaluates expressions. Also, *let* is a valid factor in expressions, which *local* is not. For instance,

$$
\begin{array}{l}
val\ x = let\ \ fun\ e\ 0 = 1 \\
\qquad\qquad\qquad |\ x = o\ (x - 1) \\
\qquad\qquad and\ o\ 0 = 0 \\
\qquad\qquad\qquad |\ x = e\ (x - 1) \\
\qquad\quad in \\
\qquad\qquad map\ e\ [1, 2, 3, 4, 5] \\
\qquad\quad end
\end{array}
$$

would bind x $[0, 1, 0, 1, 0]$. e maps even numbers to 1 and odd numbers to 0. e and o are local *and* mutually recursive, which is why they must be declared with *fun* ... *and* ...

Operator Declarations

Operators are in fact functions in mLite, which can easily be demonstrated as follows:

$+ (\ *\ (1,2), *\ (3,4)\ ) \ \rightarrow\ 24$

(Make sure to leave a blank between “(” and “*”, because “(*” would begin a block comment.)

However, not all functions are operators:

$max(5,7) \ \rightarrow\ 7$
$5\ max\ 7 \ \rightarrow\ error$

The *infix* and *infixr* keywords are used to declare functions as infix operators. The *nonfix* keyword removes an operator declaration. Note that these declarations actually change the syntax of the mLite language, so they should be used with care.

The mLite system maintains an internal parse table that contains the precedence and associativity values of all operators. This table controls the part of the mLite parser that analyzes infix expressions. Operator declarations modify this table. The initial table can be found in the mLite Reference (pg 32).

$infix\ id_1\ \{<, =, >\}\ id_2$
$infixr\ id_1\ \{<, =, >\}\ id_2$

An *infix* declaration adds the identifier id_1 to the parse table. The precedence will be either less than, equal to, or greater than the precedence of id_2, depending on the operator used in between. E.g.

$$infix\ max = +$$

would assign the precedence of the + operator to *max*. The new operator associates to the left. After the above declaration, the expression

$$a + b\ max\ c + d$$

would parse as

$$((a + b)\ max\ c) + d$$

An *infixr* declaration works in exactly the same way, but makes id_1 associate to the right.

$nonfix\ id\ [\ , \cdots\ ]$

A *nonfix* declaration removes the given identifiers from the parse table. After removing an identifier from the table, it no longer works as an infix operator.

While *nonfix* can theoretically be applied to predefined operators, such as + and @, doing so is not recommended.

$op\ f$

Any infix operator can be used as a function by prefixing it with the keyword *op*. For instance:

$op + (5, 7)$

or

$fold\ (op\ +, 0)\ [1, 2, 3, 4, 5]$

Note that the *op* keyword is merely a hint for the parser and can often be omitted.

Algebraic Types

$type\ {:}id = constructor \mid \cdots$

The *type* declaration creates a new data type called *:id*, which may be constructed *and deconstructed* by any of the constructors following the equal sign.

The following declaration defines a “list” type, although this is actually redundant, because mLite already has a primitive list type:

$type\ {:}list = {:}nil \mid {:}cons\ (x, {:}list)$

It means, “a *:list* is either *:nil* or a *:cons* of some object and another *:list*”. This is the archetypal definition of the list, as it can be found in pretty much every introduction-level computer science textbook.

A new *:list* can then be created by *:cons*:

${:}cons\ (1,\ {:}cons\ (2,\ {:}cons\ (3,\ {:}nil)))$

and functions may use the *:cons* constructor in patterns in order to *destructure:lists*:

$$fun\ length\ {:}nil = 0$$
$$\quad\quad | \ {:}cons\ (\_, t) = 1 + length\ t$$

Of course, *type* constructors can be infix operators, so after declaring

$$infixr\ {:}cons = ::$$

the following expression can be used to create a list:

$$1\ {:}cons\ 2\ {:}cons\ 3\ {:}cons\ {:}nil$$

and *length* can be written like this:

$$fun\ length\ {:}nil = 0$$
$$\quad\quad | \ (\_\ {:}cons\ t) = 1 + length\ t$$

The following *type* defines a binary tree:

$$type\ {:}tree = {:}leaf\ (x) \mid {:}node\ ({:}tree, {:}tree)$$

So a *:tree* is either a *:leaf* of a value x or a *:node* of two *:tree*'s. Here is a sample tree:

```
:node
 (:node
   (:leaf 1,
    :leaf 2),
  :node
   (:node
     (:leaf 3,
      :leaf 4),
    :node
     (:leaf 5,
      :leaf 6)))
```

And this is a function computing the depth (the longest path from the root to a leaf) of a tree:

$$fun\ depth\ {:}leaf\ (\_) = 1$$
$$\quad\quad | \ {:}node\ (l, r) = 1 + max\ (depth\ l, depth\ r)$$

Of course, algebraic types can be combined:

$$\mathit{type}\ {:}\mathit{vtree} = {:}\mathit{leaf}\,(x) \mid {:}\mathit{node}\,({:}\mathit{tlist})$$
$$\mathit{type}\ {:}\mathit{tlist} = {:}\mathit{nil} \mid {:}\mathit{tcons}\,({:}\mathit{vtree}, {:}\mathit{tlist})$$

A *:vtree* is either a *:leaf* of a value or a *:node* of a *:tlist* (tree list), and a *:tlist* is either *:nil* or a *:tcons* of a *:vtree* and a *:tlist*. The types are mutually recursive and together define a variadic tree.

$$\mathit{exception}\ {:}\mathit{id}\ [\ \mathit{and} \cdots\ ]$$

An *exception* declaration is technically equal to a *type* declaration of the form

$$\mathit{type}\ {:}\mathit{id} = {:}\mathit{id}$$

It defines a *constant constructor*, i.e. a constructor that evaluates to itself. However, the *exception* declaration should be used for clarity.

The Lite Abstract Machine

mLite compiles internally to a Scheme-based domain-specific language (DSL) that is easily implemented on top of a Scheme system by means of functions and macros, allowing an mLite environment to make use of a mature Scheme system as its back-end.

The Lite Abstract Machine (LAM) language differs from ordinary Scheme in the following aspects.

- Lambda abstraction is replaced by a pattern matching function abstraction.
- There are additional special forms to support algebraic types and exceptions.
- There is a new "tuple" data type.
- All non-variadic primitive functions are unary, using tuples and pattern matching to receive multiple arguments.
- Some special forms and most of the built-in procedures have been removed or renamed in order to make LAM more similar to the mLite language.
- File input/output works in a different way.

LAM Syntax

Most LAM data objects look exactly like Scheme data objects, with the following exceptions:

- Truth values are represented by the identifiers *true* and *false*.
- Char literals are represented by #"*c*" where *c* may be a single character or *space* or *newline*.
- Lists use square brackets instead of parentheses and :: instead of the dot (.).
- There is no external representation for vectors. Vectors are created by the *newvec* function.

There is a comprehensive list of mLite/LAM data objects in the mLite Reference (pg 31).

In the following summary, x denotes *any* type. In syntactical forms ("special forms"), x denotes an unevaluated expression and $eval(x)$ denotes its normal form, i.e. its value after evaluation.

A summary of all other type designators can also be found in the mLite Reference.

Scheme-Like Syntax

$(quote\ x)\ \rightarrow\ x$

Return the value x.

$(begin\ x_1 \cdots x_n)\ \rightarrow\ eval(x_n)$

Evaluate each x_i in sequence, return the value of x_n.

$(if\ x_1\ x_2\ x_3)\ \rightarrow\ eval(x_2 \mid x_3)$

If $eval(x_1)$ is not $false$, evaluate to $eval(x_2)$, else evaluate to $eval(x_3)$. There is no two-argument variant of *if*.

$(or\ x_1 \cdots)\ \rightarrow\ eval(x_i)$

Return the value of the first x_i that does not evaluate to $false$, else return $false$.

$(also\ x_1 \cdots x_n)\ \rightarrow\ eval(x_i)$

Evaluate to the value of the first x_i that evaluates to $false$. Evaluate to x_n, if all prior x_i's evaluated to a non-$false$ value. This is like Scheme's "and". It is called *also*, because *and* is reserved for chaining declarations.

$(set!\ id\ x)\ \rightarrow\ ()$

Change the value bound to id to $eval(x)$. Used to implement recursive bindings.

(Note: the name *set!* does not have to be a valid mLite identifier, because it is only used internally.)

Non-Scheme Syntax

$(fn\ (p_1\ x_1) \cdots) \ \rightarrow\ f$

Create a new function of patterns $p_1 \cdots$ to expressions $x_1 \cdots$. See function application, below, for further details.

$((fn\ (p_1\ x_1) \cdots)\ x_a) \ \rightarrow\ eval(x_1)$

Attempt to match the argument expression x_a against the pattern p_1, thereby binding variables in the pattern to matching values in the argument. When the match succeeds, evaluate x_1 with the bindings in effect and return $eval(x_1)$.

When x_a does not match p_1, try the other patterns p_i, until a pattern matches or no more patterns can be found. A function application running out of patterns is an error.

The exact pattern matching algorithm is be explained in the section on the mLite language itself and, more formally, in the mLite Reference (pg 32).

$(letrec\ ((id_1\ (p_1\ x_1) \cdots) \cdots)\ x_{b1} \cdots x_{bn}) \ \rightarrow\ eval(x_{bn})$

Bind each identifier id_i to the corresponding function $(fn\ (p_i\ x_i) \cdots)$. With those bindings in effect, evaluate $x_{b1} \cdots x_{bn}$ and finally return x_{bn}.

$(\%raise\ {:}id) \ \rightarrow\ undefined$

Raise the exception *:id*. This operator never returns. Exception handling is discussed in the section on the mLite language (pg 9).

$(define\ id\ x) \ \rightarrow\ ()$

Create the new variable *id* and bind it to the value $eval(x)$. The new binding will be always added to the *top level environment*, so the binding created by the form

$$((fn\ (x\ (define\ foo\ x)))\ "bar")$$

will persist even after the function returns. In other words, *define* binds variables to values *globally*, no matter what context it appears in.

$(define\_type\ {:}id\ constructor \cdots) \ \rightarrow\ ()$

Create a new algebraic data type named *:id*. Each *constructor* is either a name beginning with a colon (a constructor name) or a tuple containing a constructor name in the first slot. Atomic constructors create atomic objects, tuples create constructor functions.

For instance, the following data type declaration creates the typical Lisp list type:

$$(define\_type\ {:}list\ {:}nil\ ({:}cons\ x\ {:}list))$$

Algebraic types are explained in detail in the introduction to the mLite language (pg 15).

$(\gg\ instream\ x) \ \rightarrow\ eval(x)$

Evaluate x with program input redirected to the given *instream*, where x is typically an expression whose effect is to read input from a stream. Return $eval(x)$.

$(\ll\ outstream\ x) \ \rightarrow\ eval(x)$

Evaluate x with program output redirected to the given *outstream* where x is typically an expression whose effect is to write output to a stream. Return $eval(x)$.

Scheme-Like Functions

The following functions are almost identical to their Scheme counterparts:

$$* \ \ + \ \ - \ \ / \ \ < \ \ <= \ \ = \ \ > \ \ >= \ \ abs \ \ call/cc \ \ max \ \ min \ \ not$$

They implement multiplication, addition, subtraction, the usual comparison operations, magnitude (*abs*), call-with-current-continuation, the minimum and maximum of two numbers, and the logical “not”.

Unlike their Scheme counterparts, all of these functions are *unary*, though, and multiple values are passed to them via tuples so, for instance, two numbers x and y are added by the form

$$(+\ (tuple\ x\ y))$$

Also, none of these functions are variadic. The arithmetic and comparison functions (except for *abs*) all expect 2-tuples as arguments. *call/cc* expects a single value. The *not* function is type-agnostic, as in Scheme.

The domain of the comparison operators has been extended, so they can be applied to *chars* and *strings* in addition to numbers.

Renamed Functions

Some functions have been renamed to make them fit better in the mLite language. A full map of these functions can be found in the appendix (pg 56).

mLite Functions

The following identifiers denote LAM-level mLite functions:

:: @ <> ~< ~<= ~<> ~= ~> ~>= *close*
len *println* *readln* *ref* *rev* *set* *setvec* *sub* ~

Some of these functions are explained in detail in the introduction to the mLite language. A full list can be found in the mLite Reference.

The following identifiers denote functions that are used internally in code compiled from mLite programs:

$(list\ x_1 \cdots) \ \rightarrow\ L$

Create a list.

$(tuple\ x_1 \cdots x_k) \ \rightarrow\ T_k$

Create a k-tuple.

$(\%register\ k\ f) \ \rightarrow\ ()$

Register exception handler f with exit point k (k is a continuation).

$(\%unregister\ x) \ \rightarrow\ x$

Unregister the most recently registered exception handler and return x. This is an identity function with the effect of unregistering a handler.

(*%typecheck :id* x L) $\rightarrow$ x

Check whether x is contained in L, where L is a list of all patterns representing the type *:id*. For instance, given the definition

(*define_type :list :nil* (*:cons* x *:list*))

the *:cons* constructor would use *%typecheck* to make sure its second argument is of the form *:nil* or (*:cons* x y).

Compiling mLite to LAM

This chapter defines the semantics of the mLite language in terms of the Lite Abstract Machine.

Data Objects

Atomic data objects and identifiers compile to themselves; $a \rightarrow b$ means "a compiles to b" in this chapter.

$$\begin{array}{rcl}
[] & \rightarrow & [] \\
() & \rightarrow & () \\
bool & \rightarrow & bool \\
int & \rightarrow & int \\
real & \rightarrow & real \\
char & \rightarrow & char \\
str & \rightarrow & str \\
id & \rightarrow & id \\
op\ id & \rightarrow & id
\end{array}$$

Lists and tuples:

$$[x_1, \cdots] \rightarrow (list\ x_1 \cdots)$$

$$(x_1, x_2, \cdots) \rightarrow (tuple\ x_1\ x_2 \cdots)$$

$$\#int(x_1, x_2, \cdots) \rightarrow (ref\ (tuple\ x_1\ x_2 \ldots)\ int)$$

Algebraic types:

$$:type \rightarrow :type$$

$$:type\ (x_1, \cdots) \rightarrow (:type\ x_1 \cdots)$$

Functions and Application

$$fn\ p = x \mid \cdots \rightarrow (fn\ (p\ x) \cdots)$$

$$fn\ p_1\ p_2 = x \rightarrow (fn\ (p_1\ (fn\ (p_2\ x))))$$

$$(fn\ p = x \mid \cdots)\ x_a \rightarrow ((fn\ (p\ x) \cdots)\ x_a)$$

$$f\ x \rightarrow (f\ x)$$

$$f\ g\ x \rightarrow ((f\ g)\ x)$$

$f\ (g\ x)\ \rightarrow\ (f\ (g\ x))$

$f\ \text{‘}\ g\ x\ \rightarrow\ (f\ (g\ x))$

R denotes an infix operator and R_n an infix operator of precedence n, where higher values indicate stronger binding. R_L and R_R indicate left- and right associative operators, respectively (with equal precedence). Then:

$x_1\ R\ x_2 = R\ (x_1, x_2),$ so $x_1\ R\ x_2\ \rightarrow\ (R\ (\mathit{tuple}\ x_1\ x_2))$

Furthermore:

$x\ R_L\ y\ R_L\ z\ \rightarrow\ (R_L\ (\mathit{tuple}\ (R_L\ (\mathit{tuple}\ x\ y))\ z))$

$x\ R_R\ y\ R_R\ z\ \rightarrow\ (R_R\ (\mathit{tuple}\ x\ (R_R\ (\mathit{tuple}\ y\ z))))$

$x\ R_2\ y\ R_1\ z\ \rightarrow\ (R_1\ (\mathit{tuple}\ (R_2\ (\mathit{tuple}\ x\ y))\ z))$

$x\ R_1\ y\ R_2\ z\ \rightarrow\ (R_1\ (\mathit{tuple}\ x\ (R_2\ (\mathit{tuple}\ y\ z))))$

$(x)\ \rightarrow\ x$

$(x\ R_1\ y)\ R_2\ z\ \rightarrow\ (R_2\ (\mathit{tuple}\ (R_1\ (\mathit{tuple}\ x\ y))\ z))$

$x\ R_2\ (y\ R_1\ z)\ \rightarrow\ (R_2\ (\mathit{tuple}\ x\ (R_1\ (\mathit{tuple}\ y\ z))))$

Expressions

Sequences:

$x_1\ \mathit{or}\ x_2\ \mathit{or} \cdots\ \rightarrow\ (\mathit{or}\ x_1\ x_2 \cdots)$

$x_1\ \mathit{also}\ x_2\ \mathit{also} \cdots\ \rightarrow\ (\mathit{also}\ x_1\ x_2 \cdots)$

$x_1\ ;\ x_2\ ; \cdots\ \rightarrow\ (\mathit{begin}\ x_1\ x_2 \cdots)$

Conditional evaluation:

$\mathit{if}\ x_1\ \mathit{then}\ x_2\ \mathit{else}\ x_3\ \rightarrow\ (\mathit{if}\ x_1\ x_2\ x_3)$

$\mathit{case}\ x_0\ \mathit{of}\ \ p_1 = x_1 \mid \cdots\ \rightarrow\ ((\mathit{fn}\ (p_1\ x_1) \cdots)\ x_0)$

I/O redirection:

$x_1 \ll x_2\ \rightarrow\ (\ll x_1\ x_2)$

$x_1 \gg x_2\ \rightarrow\ (\gg x_1\ x_2)$

Exceptions

x_0 *handle* $:exn_1 = x_1 \mid \cdots$
$\quad \to$ $(call/cc\ (fn\ (k\ (\%register\ k\ (fn\ (:exn_1\ x_1)\cdots))$
$\qquad\qquad\qquad\qquad (\%unregister\ x_0))))$

$raise\ :exn \ \to\ (\%raise\ :exn)$

Local values:

$let\ val\ p_1 = x_1\ in\ x_2\ end \ \to\ ((fn\ (p_1\ x_2))\ x_1)$

$let\ val\ p_1 = x_1\ and\ p_2 = x_2\ in\ x_3\ end$
$\quad \to\ ((fn\ ((p_1\ p_2)\ x_3))\ x_1\ x_2)$

Note that:
$\quad ((fn\ ((p_1\ p_2\cdots)\ x_b))\ x_1\ x_2\ldots)$
$= ((fn\ ((p_1\ p_2\cdots)\ x_b))\ (tuple\ x_1\ x_2\ldots))$

$let\ val\ p_1 = x_1;\ val\ p_2 = x_2\ in\ x_3\ end$
$\quad \to\ ((fn\ (p_1\ ((fn\ (p_2\ x_3))\ x_2)))\ x_1)$

$let\ val\ p_1 = x_1\ in\ x_2;\ x_3\ end \ \to\ ((fn\ (p_1\ (begin\ x_2\ x_3)))\ x_1)$

Local functions:

$let\ fun\ f_1\ p_1 = x_1\ in\ x_2\ end \ \to\ (letrec\ ((f_1\ (p_1\ x_1)))\ x_2)$

$let\ fun\ f_1\ p_1 = x_1\ and\ f_2\ p_2 = x_2\ in\ x_3\ end$
$\quad \to\ (letrec\ ((f_1\ (p_1\ x_1))\ (f_2\ (p_2\ x_2)))\ x_3)$

$let\ fun\ f_1\ p_1 = x_1;\ fun\ f_2\ p_2 = x_2\ in\ x_3\ end$
$\quad \to\ (letrec\ ((f_1\ (p_1\ x_1)))\ (letrec\ ((f_2\ (p_2\ x_2)))\ x_3))$

$let\ fun\ f_1\ p_1 = x_1\ in\ x_2;\ x_3\ end$
$\quad \to\ (letrec\ ((f_1\ (p_1\ x_1)))\ (begin\ x_2\ x_3))$

Declarations

Values:

$val\ v = x \ \to\ (define\ v\ x)$

$val\ p = x \ \to\ ((fn\ (p\ (define\ v_1\ v_1)\cdots))\ x)$

where each v_i is a variable of the pattern p.

For example:

$val\ (x, y) = (1, 2)$
$\quad \rightarrow\ ((fn\ ((x\ \ y)\ (define\ \ x\ \ x)\ (define\ \ y\ \ y)))\ (tuple\ 1\ 2))$

Because *define* binds variables at the top level, the construct $(define\ v\ v)$ copies the binding of the identifier v to the top-level (global) environment.

$val\ \ p_1 = x_1\ and\ x_2 = p_2$
$\quad \rightarrow\ ((fn\ ((p_1\ \ p_2)\ (define\ v_1\ \ v_2)\cdots))\ x_1\ \ x_2)$

Functions:

$fun\ id\ \ p_1 = x_1 \mid \cdots\ \rightarrow\ (define\ id\ (fn\ (p_1\ \ x_1)\cdots))$

$fun\ id\ \ p_1\ \ p_2 = x\ \ \rightarrow\ (define\ id\ (fn\ (p_1\ (fn\ (p_2\ \ x)))))$

$fun\ id_1\ \ p_1 = x_1\ and\ id_2\ \ p_2 = x_2$
$\quad \rightarrow\ (letrec\ ((id_1\ (p_1\ \ x_1))\ (id_2\ (p_2\ \ x_2)))$
$\qquad\qquad (define\ id_1\ id_1)$
$\qquad\qquad (define\ id_2\ id_2))$

Guarded functions:

$fun\ id\ \ p\ where\ x_g = x$
$\quad \rightarrow\ (define\ id\ (fn\ ((GUARD\ \ p :: x_g)\ x)))$

$fun\ id\ \ x_{ig} = x$
$\quad \rightarrow\ (define\ id\ (fn\ ((GUARD\ P(\rho\ x_{ig}) :: G(\rho\ x_{ig}))\ x)))$

Here x_{ig} denotes an implicit guard. *GUARD* is a primitive type of the Lite Abstract Machine. See the section on Implicit Guard Patterns (pg 33) for definitions of the ρ, P, and G functions.

Example:
$fun\ id\ \ f\ \ x_1 = x_2\ \ \rightarrow\ (define\ id\ (fn\ ((GUARD\ x_1 :: (f\ \ x_1))\ x_2)))$

Type declarations:

$type\ {:}id = cons_1 \mid \cdots\ \rightarrow\ (define\_type\ {:}id\ cons_1 \cdots)$

where each *cons* denotes a *type constructor*.

Exceptions:

$exception\ {:}id\ \ \rightarrow\ (define\_type\ {:}id\ {:}id)$

Fixity declarations ($R \in \{<, =, >\}$):

$infix\ id_1\ R\ id_2, \cdots \ \rightarrow\ ()$

$infixr\ id_1\ R\ id_2, \cdots \ \rightarrow\ ()$

$nonfix\ id, \cdots \ \rightarrow\ ()$

These declarations do not compile to any code, but modify the internal infix operator table.

Local Declarations

Local values:

$local\ val\ v_1 = x\ in\ val\ v_2 = v_1\ end$
$\quad \rightarrow\ ((fn\ (v_1\ (define\ v_2\ v_1)))\ x)$

$local\ val\ v_1 = x_1\ and\ v_2 = x_2\ in\ val\ v_3 = v_2\ end$
$\quad \rightarrow\ ((fn\ ((v_1\ v_2)\ (define\ v_3\ v_2)))\ x_1\ x_2)$

$local\ val\ v_1 = x_1;\ val\ v_2 = v_1\ in\ val\ v_3 = v_2\ end$
$\quad \rightarrow\ ((fn\ (v_1\ ((fn\ (v_2\ (define\ v_3\ v_2)))\ v_1)))\ x_1)$

$local\ val\ v_1 = x_1\ in\ val\ v_2 = v_1\ and\ v_3 = v_1\ end$
$\quad \rightarrow\ ((fn\ (v_1\ ((fn\ ((v_2\ v_3)\ (define\ v_2\ v_2)\ (define\ v_3\ v_3)))$
$\qquad\qquad\qquad v_1\ v_1)))$
$\qquad x_1)$

$local\ val\ v_1 = x_1\ in\ val\ v_2 = v_1;\ val\ v_3 = v_2\ end$
$\quad \rightarrow\ ((fn\ (v_1\ (begin\ (define\ v_2\ v_1)\ (define\ v_3\ v_2))))\ x_1)$

Local functions:

$local\ fun\ f\ p = x\ in\ fun\ g\ p = x\ end$
$\quad \rightarrow\ (letrec\ ((f\ (p\ x)))\ (define\ g\ (fn\ (p\ x))))$

$local\ fun\ f\ p = x\ and\ g\ p = x\ in\ fun\ h\ p = x\ end$
$\quad \rightarrow\ (letrec\ ((f\ (p\ x))\ (g\ (p\ x)))\ (define\ h\ (fn\ (p\ x))))$

$local\ fun\ f\ p = x;\ fun\ g\ p = x\ in\ fun\ h\ p = x\ end$
$\quad \rightarrow\ (letrec\ ((f\ (p\ x)))$
$\qquad\quad (letrec\ ((g\ (p\ x)))$
$\qquad\qquad (define\ h\ (fn\ (p\ x)))))$

$$\begin{array}{l} \mathit{local}\ \ \mathit{fun}\ f\ \ p = x\ \mathit{in}\ \ \mathit{fun}\ g\ \ p = x\ \mathit{and}\ h\ \ p = x\ \mathit{end} \\ \quad \rightarrow\ (\mathit{letrec}\ ((f\ \ (p\ \ x))) \\ \qquad\qquad (\mathit{letrec}\ ((g\ (p\ \ x))\ (h\ (p\ \ x))) \\ \qquad\qquad\quad (\mathit{define}\ g\ g) \\ \qquad\qquad\quad (\mathit{define}\ h\ h))) \end{array}$$

$$\begin{array}{l} \mathit{local}\ \ \mathit{fun}\ f\ \ p = x\ \mathit{in}\ \ \mathit{fun}\ g\ \ p = x;\ \mathit{fun}\ h\ \ p = x\ \mathit{end} \\ \quad \rightarrow\ (\mathit{letrec}\ ((f\ \ (p\ \ x))) \\ \qquad\qquad (\mathit{begin}\ (\mathit{define}\ g\ (\mathit{fn}\ (p\ \ x)))\ (\mathit{define}\ h\ (\mathit{fn}\ (p\ \ x))))) \end{array}$$

Derived Forms

The *letrec* and *define_type* forms are *derived forms*, i.e. they are composed of simpler LAM forms internally.

letrec translates to *fn* and *set*!:

$$\begin{array}{lcl} \mathit{(letrec} & \rightarrow & \mathit{((fn\ ((f1\ f2)} \\ \quad \mathit{((f1\ (pat1\ expr1)} & & \qquad \mathit{(set!\ f1\ (fn\ (pat1\ expr1)} \\ \qquad \mathit{(pat2\ expr2)} & & \qquad\qquad \mathit{(pat2\ expr2))} \\ \qquad \cdots\mathit{)} & & \qquad\qquad \cdots\mathit{)} \\ \quad \mathit{(f2\ (pat1\ expr1)} & & \qquad \mathit{(set!\ f2\ (fn\ (pat1\ expr1)} \\ \qquad \mathit{(pat2\ expr2)} & & \qquad\qquad \mathit{(pat2\ expr2)} \\ \qquad \cdots\mathit{))} & & \qquad\qquad \cdots\mathit{)))} \\ \quad \mathit{expr}\cdots\mathit{)} & & \qquad \mathit{((fn\ ((\,)\ expr}\cdots\mathit{))))} \\ & & \quad \mathit{undefined} \\ & & \quad \mathit{undefined)} \end{array}$$

define_type translates to a set of *define*'s, the first one defining the type itself, the subsequent ones the constructors of the type:

$$\begin{array}{l} (\mathit{define\_type}\ \mathit{:id\ constructor}\cdots) \\ \quad \rightarrow\ (\mathit{begin}\ (\mathit{define}\ \mathit{:id}\,'(\mathit{:id\ constructor}\cdots)) \\ \qquad\qquad\quad (\mathit{define}\ \mathit{:id}\cdots) \\ \qquad\qquad\quad \cdots) \end{array}$$

Atomic constructors:

$\mathit{:id}\ \rightarrow\ (\mathit{define}\ \mathit{:id}\ \mathit{':id})$

Parametric constructors (carrying a variable object):

$(\mathit{:id\ id})\ \rightarrow\ (\mathit{define}\ \mathit{:id}\ (\mathit{fn}\ (\mathit{id}\ (\mathit{list}\ \mathit{':id\ id}))))$

Parametric constructors (carrying a typed object):

$(:id_1\ :id_2)$
$\quad\rightarrow\ (define\ :id_1$
$\qquad\quad (fn\ (x\ (list\ ':id_1\ (\%typecheck\ :id_1\ \ x\ :id_2)))))$

Example:

$(define\_type\ :list\ :nil\ (:cons\ x\ :list))$
$\quad\rightarrow\ (begin\ (define\ :list\ '(:list\ :nil\ (:cons\ x\ :list)))$
$\qquad\qquad\quad (define\ :nil\ ':nil)$
$\qquad\qquad\quad (define\ :cons$
$\qquad\qquad\qquad (fn\ ((x\ list_1)$
$\qquad\qquad\qquad\quad (list\ ':cons\ x\ (\%typecheck\ ':cons\ list_1\ :list))))))$

mLite Reference

mLite/LAM Data Objects

Type	Examples	Designator
boolean	*true; false*	b
integer	$0; 123$	n
real	$0.0; 3.14; 1.23e6$	r
char	*#x; #space*	c
string	*"hello"; "\"\""*	s
unit	()	()
tuple	(x, y, z)	T_k
list	$[x, y, z]; [x::[y]]; []$	L
vector	*newvec* $(10, x)$	V
function	$fn\ x = x + 1$	f
instream	*instream "file"*	*SI*
outstream	*outstream "file"*	*SO*
identifier	*foo;* f'*;* *=/=*	*id*
constructor	*:cons*	*:id*

Note: in LAM programs, there are no commas in lists and tuples, and function application is Scheme-like, e.g.:

(*newvec* (*tuple* 10 x)) instead of *newvec* $(10, x)$.

As in Scheme, the sequences “\\” and “\"” can be used to escape the corresponding characters inside of string literals. There are no other escape sequences.

The *unit* object () is used as an unspecific value. For example, it is passed to procedures not expecting any specific arguments and returned by procedures not returning any meaningful value.

T_k denotes a k-tuple, i.e. a tuple of k elements. There are no tuples with $k = 1$, because these would just denote single values, i.e. $(x) = x$ for any value x. The 0-tuple is equal to *unit*.

There are two groups of identifiers. One consists of letters, decimal digits, and the following special characters: underscore (_), apostrophe (′), colon (:). These identifiers must begin with a non-digit character. They are case-sensitive.

The other group (“operator identifiers”) consist of sequences of the following special characters:

! @ $ % ^ & * – + = < > / ~ ‘

Predefined Infix Operators

Precedence	Operators	Associativity
high	*o*	right
	^	right
	* *div* *mod* *rem* /	left
	+ −	left
	:: @	right
low	< <= <> = > >= ~< ~<= ~<> ~= ~> ~>=	left

Pattern Matching Algorithm

- () matches ().
- *true* matches *true* and *false* matches *false*.
- a number x matches a number y, if $x = y$.
- a char c_1 matches a char c_2, if $c_1 = c_2$.
- a string s_1 matches a string s_2, if $s_1 = s_2$.
- the special pattern _ matches any object.
- an identifier *id* matches any object and binds *id* to that object.
- a list L_1 matches a list L_2, if $len\ L_1 = len\ L_2$ and each element a_i of L_1 matches the corresponding element b_i of L_2.
- the pattern $h :: t$ matches a list of at least one element and binds the identifier h to the first element of the list and the identifier t to the rest of the list.
- a k-tuple $T_{k,1}$ matches a k-tuple $T_{k,2}$, if each $\#iT_{k,1}$ matches $\#iT_{k,2}$ for $0 \le i < k$.
- a constant constructor *:id* matches itself.

- an exception *:exn* matches itself.
- a parametric constructor $:id\,(id_1, \cdots)$ matches any object created by that constructor and binds any variables in the pattern to the corresponding values in the object.

Implicit Guard Patterns

Informally: an implicit guard is an expression inside of a pattern. Function application, infix operators, *or*, and *also* are allowed inside of implicit guard expressions. Guards using function application or = operators must be parenthesized to distinguish them from currying and function bodies.

The function ρ converts each implicit guard pattern to a tuple (P, G) consisting of an (unguarded) pattern P and a guard expression G. $P\ t$ is the pattern component of the (P, G) tuple t, and $G\ t$ is its guard component. $\varnothing$ means "no guard expression".

$unit \ \rightarrow\ (unit, \varnothing)$

$bool \ \rightarrow\ (bool, \varnothing)$

$int \ \rightarrow\ (int, \varnothing)$

$real \ \rightarrow\ (real, \varnothing)$

$str \ \rightarrow\ (str, \varnothing)$

$id \ \rightarrow\ (id, \varnothing)$

Function application is the combination $(\cdot)$ of two identifiers, where the rightmost identifier is the pattern.

$f\ x \ \rightarrow\ (f, \varnothing) \cdot (x, \varnothing) \ \rightarrow\ (x, f\ x)$

Higher order application:

$f\ g\ x \ \rightarrow\ (g, f\ g) \cdot (x, \varnothing) \ \rightarrow\ (x, (f\ g)\ x)$

Function composition:

$f\ (g\ x) \ \rightarrow\ (f, \varnothing) \cdot (x, g\ x) \ \rightarrow\ (x, f\ (g\ x))$

More generally (f and x may be be any expression):

$\rho\ (f\ x) \ \rightarrow\ (P(\rho\ x), f'\ x')$

where $f' = G\ (\rho\ f)$, if $G\ (\rho\ f) \neq \emptyset$ else $f' = P\ (\rho\ f)$.
and $x' = G\ (\rho\ x)$, if $G\ (\rho\ x) \neq \emptyset$, else $x' = P\ (\rho\ x)$.

Tuples:

$$(a_1, a_2, \cdots) \\ \rightarrow \big((P\ (\rho\ a_1), P\ (\rho\ a_2), \cdots), G\ (\rho\ a_1)\ also\ G\ (\rho\ a_2)\ also \cdots\big)$$

I.e. a tuple is converted to a tuple with each guarded pattern replaced by the corresponding unguarded pattern. The guard of the tuple is the conjunction of all guards in the tuple.

Lists work analogously:

$$[a_1, \cdots] \ \rightarrow\ \big([P\ (\rho\ a_1), \cdots], G\ (\rho\ a_1)\ also \cdots\big)$$

The guard $G\ (\rho\ p)$ of an unguarded pattern p may be thought of as $true$, although an actual implementation would simply omit the guard in the conjunction.

Infix expressions: for each infix operator R,

$$id\ R\ x \ \rightarrow\ (id, R\ (id, x))$$

$$x\ R\ id \ \rightarrow\ (id, R\ (x, id))$$

$$id_1\ R\ id_1 \ \rightarrow\ (id_1, R\ (id_1, id_1))$$

Also note that:

$$id_1\ R\ id_2 \ \rightarrow\ error$$

That is, only one single identifier may be contained in an infix expression x, and that identifier will be the resulting unguarded pattern $P\ (\rho\ x)$. However, the single identifier may appear multiple times, e.g.:

$$\rho\ (10 < id\ also\ id < 20) \ \rightarrow\ (id, 10 < id\ also\ id < 20)$$

The detailed conversion rules for infix expression are rather elaborate. For a complete formal description see [HOL14].

mLite Syntax Summary

Declarations

$val\ \ p = x\ and \cdots$

Bind variables of pattern p to corresponding components of expression x. Expressions in bindings chained with and evaluate *before* binding any values.

$fun\ id\ \ p\ [where\ \ x_g] = x \mid \cdots\ and \cdots$
$fun\ id\ \ p \cdots [where\ \ x_g] = x\ and \cdots$

Bind each identifier id to the corresponding function

$$fn\ \ p = x \mid \cdots$$

Multiple patterns $p, q \cdots$ indicate currying, e.g.:

$$fn\ \ p\ q \cdots = x \quad \text{equals} \quad fn\ \ p = fn\ q = \cdots x$$

Note that multiple patterns (separated by |) *cannot* be combined with currying. Each function is *either* curried *or* has multiple cases.

Multiple fun declarations chained together with and may be mutually recursive, even in local (let, $local$) contexts.

The $where$ keyword introduces an explicit *guard expression*. A pattern matches only if x_g evaluates to a non-$false$ value.

Each pattern p may contain *implicit guards*; see the corresponding section in this reference (pg 33) for details.

$type\ {:}id = cons \mid \cdots$

Define algebraic type $:id$ as union of the subsequent constructors. Each constructor may be of the form $:id$, defining an atomic instance of the type, or of the form

$$:id\ (id \cdots)$$

where each id may be an (untyped) variable or a type name. A constructor containing a type name in a formal argument expects a value of the given type in the corresponding actual argument. E.g., the constructor $:c$ defined in

$$type\ {:}L = {:}N \mid {:}c\ (x, {:}L)$$

would accept any type of argument in the place of x, but only values of the type $:L$ in the second slot of its argument.

$local\ ldecl_{1,1}; \cdots in\ ldecl_{2,1}; \cdots\ end$

First evaluate the $ldecl_1$'s, giving a new environment E, then evaluate the $ldecl_2$'s in that environment, adding them to the top level environment. Eventually remove the environment E, but keep the bindings established by $ldecl_2$.

I.e.: define $ldecl_2$ in a local context containing $ldecl_1$, hiding the $ldecl_1$'s from the top level.

Each $ldecl$ may be a val or a fun declaration.

$infix\ id_1\ \{<, =, >\}\ id_2, \cdots$

Add the identifier id_1 to the internal operator precedence table as a left-associative operator. After adding the identifier to the table, it will be recognized as an infix operator, so $x\ id_1\ y$ will map to $id_1\ (x, y)$.

The less/equal/greater sign together with the second identifier id_2 defines the precedence of the id_1 operator:

$id_1 < id_2$	id_1 has lower precedence than id_2
$id_1 = id_2$	id_1 has the same precedence as id_2
$id_1 > id_2$	id_1 has higher precedence than id_2

$infix$ itself does not define the operator, it merely makes it known to the mLite parser.

$infixr\ id_1\ \{<, =, >\}\ id_2, \cdots$

The $infixr$ operator works exactly as the $infix$ operator (above), but adds id_1 as a *right-associative* operator.

$nonfix\ id, \cdots$

The $nonfix$ declaration removes the given identifiers from the internal operator precedence table. After $nonfix$-ing an identifier, it will no longer parse as an infix operator (but may still be used as a function).

Expressions

$unit, bool, int, real, char, str$

The atomic data types evaluate to themselves.

$(x_1, \cdots, x_k)$

The k-tuple notation (a tuple of k elements) evaluates to a k-tuple with all its elements in normal form, i.e.: $(eval\ (x_1), \cdots, eval\ (x_k))$.

$[x_1, \cdots]$

List notation evaluates to a list with all elements in normal form, i.e.: $[eval\ (x_1), \cdots]$.

$fn\ p_1 \cdots = x_1 \mid \cdots$

The fn syntax evaluates to a function from patterns p_i to expressions x_i. The vertical bar separates individual cases. Multiple patterns between fn and = indicate currying. See fun for details.

In a curried function, an explicit guard is associated with the innermost function, e.g.:
$fn\ a\ b\ c\ where\ g = x$ equals $fn\ a = fn\ b = fn\ c\ where\ g = x$.

$f\ x$

Juxtaposition of a function f and an object x applies the function f to x. Furthermore:

$$f\ g\ x\ =\ (f\ g)\ x$$

$$f\ (g\ x)\ =\ f\ (g\ x)$$

$$f\ \text{`}\ g\ x\ =\ f\ (g\ x)$$

$x_1\ R\ x_2$

Functions defined as infix operators (R; see $infix$, pg 36) may be used as operators in infix expressions. See the operator table on page 32 for a full list of pre-defined operators. The usual precedence and associativity rules apply, e.g.:

$$a - b - c\ =\ ((a - b) - c)$$

$$a * b + c * d\ =\ (a * b) + (c * d)$$

Also note:

$$f\ x\ R\ y\ =\ (f\ x)\ R\ y$$

$$f\ `\ x\ R\ y\ =\ f\ (x\ R\ y)$$

$\#intT_k$

This is a shorthand form of $ref\ (T_k, int)$.

$(x_1; x_2; \cdots)$

Evaluate the given expression in sequence, from left to right. Usually used for effect.

$x_1\ also\ \cdots\ also\ x_n$

Evaluate to the first $false$ expression x_i or to x_n if all prior expressions evaluated to non-$false$ values. $false\ also\ x$ never evaluates x.

$x_1\ or\ x_2\ or\ \cdots$

Evaluate to the first non-$false$ expression x_i or to $false$, if all expressions are false. $true\ or\ x$ never evaluates x.

$x_0\ handle\ {:}id_1 = x_1 \mid \cdots$

Evaluate x_0 with the exception handler $fn\ {:}id_1 = x_1 \mid \cdots$ in effect. Raising any ${:}id_i$ defined in the exception handler will evaluate and return the corresponding x_i.

$raise\ {:}id$

Raise the exception ${:}id$. Raising an exception aborts the current computation and invokes the innermost handler handling the given exception. Exceptions without a handler terminate program execution.

$if\ x_1\ then\ x_2\ else\ x_3$

First evaluate x_1. If it evaluates to a non-$false$ result, evaluate and return x_2 else evaluate and return x_3.

$case\ x_0\ of\ p_1 = x_1 \mid \cdots$

An alternative form of $(fn\ p_1 = x_1 \mid \cdots)\ x_0$.

$x_1 \gg x_2$

Evaluate expression x_2 with program input read from x_1. x_1 must evaluate to an *instream* object.

$x_1 \ll x_2$

Evaluate expression x_2 with program output written to x_1. x_1 must evaluate to an *outstream* object.

mLite Function Summary

Arithmetics

$r_1 * r_2 \rightarrow r$

Return the product of r_1 and r_2.

$r_1 + r_2 \rightarrow r$

Return the sum of r_1 and r_2.

$r_1 - r_2 \rightarrow r$

Return the difference between r_1 and r_2.

$r_1 / r_2 \rightarrow r$

Return the quotient of r_1 and r_2.

$r_1 < r_2 \mid r_1 <= r_2 \mid r_1 <> r_2 \mid r_1 = r_2 \mid r_1 > r_2 \mid r_1 >= r_2 \rightarrow b$

These functions implement the “less-than”, “less/equal”, “not-equal”, “equal”, “greater-than”, and “greater/equal” predicates, respectively. They return *true*, if their conditions apply.

$c_1 < c_2 \mid c_1 <= c_2 \mid c_1 <> c_2 \mid c_1 = c_2 \mid c_1 > c_2 \mid c_1 >= c_2 \rightarrow b$

The domain of the above predicates also covers the char type. When applied to two chars,

$$c_1 \ R \ c_2$$

holds, if and only if

$$ord \ c_1 \ R \ ord \ c_2$$

for any operator R of the above set.

$s_1 < s_2 \mid s_1 <= s_2 \mid s_1 <> s_2 \mid s_1 = s_2 \mid s_1 > s_2 \mid s_1 >= s_2 \;\rightarrow\; b$

The domain of the above predicates also covers the string type. The final character of each string is assumed to be NUL. So, $s_1 < s_2$, if and only if there is a position p so that

$$ref\ (s_1, p) < ref\ (s_2, p)$$

and each

$$ref\ (s_1, i) = ref\ (s_2, i)$$

for $0 \le i < p$.

$s_1 > s_2$, if $s_2 < s_1$. $s_1 <= s_2$, if not $s_2 > s_1$. $s_1 >= s_2$, if not $s_1 < s_2$.

$s_1 = s_2$, if $len\ s_1 = len\ s_2$ and all $ref\ (s_1, i) = ref\ (s_2, i)$ for $0 \le i < k$, where $k = len\ s_1$. $s_1 <> s_2$, if not $s_1 = s_2$.

$x = b \mid x = [] \mid x = () \;\rightarrow\; b$

The = operator can be safely applied to any type of object, as long as the other object is a bool, or the empty list, or unit. In this case, it returns *true*, when the two objects are identical. It returns *false*, if the objects are not identical — this includes the case that the objects have incompatible types, e.g. $123 = () \rightarrow false$.

$abs\ r \;\rightarrow\; r$

Return the magnitude $|r|$ of r.

$ceil\ r \;\rightarrow\; n$

Return $\lceil r \rceil$ (ceiling of r, the smallest integer that is not smaller than r).

$n_1\ div\ n_2 \;\rightarrow\; n$

Return $\lfloor n_1/n_2 \rfloor$ (the floored quotient of n_1 and n_2).

$floor\ r \;\rightarrow\; n$

Return $\lfloor r \rfloor$ (the largest integer that is not larger than r).

$gcd\ (n_1, n_2) \;\rightarrow\; n$

Return the *greatest common divisor* of n_1 and n_2, i.e. the largest integer that divides both n_1 and n_2.

$lcm\ (n_1, n_2) \ \rightarrow\ n$

Return the *least common multiple* of n_1 and n_2, i.e. the smallest integer that is a multiple of both n_1 and n_2.

$\max\ (r_1, r_2) \ \rightarrow\ r$

Return the larger one of the numbers r_1 and r_2. Return a real number, if at least one of the numbers is a real number, else return an integer.

$\min\ (r_1, r_2) \ \rightarrow\ r$

Return the smaller one of the numbers r_1 and r_2. Return a real number, if at least one of the numbers is a real number, else return an integer.

$n_1\ mod\ n_2 \ \rightarrow\ n$

Return $n_1 - \lfloor n_1/n_2 \rfloor \cdot n_2$ (the modulus of n_1 and n_2).

$f_1\ o\ f_2 \ \rightarrow\ f$

Return the function composition $fn\ x = f_1\ (f_2\ x)$.

$n_1\ rem\ n_2 \ \rightarrow\ n$

Return $n_1 - trunc\ (n_1/n_2) \cdot n_2$ (the truncated division remainder of n_1 and n_2).

$sgn\ r \ \rightarrow\ n$

Return the *sign* of r: $r < 0 \rightarrow -1$, $r > 0 \rightarrow 1$, and $r = 0 \rightarrow 0$.

$trunc\ r \ \rightarrow\ n$

Return $sgn(r) \cdot \lfloor |r| \rfloor$ (in other words: remove the fractional part of r and return the integer part only).

$sqrt\ r \ \rightarrow\ r$

Return $\sqrt{r}$.

$r_1\ \hat{}\ r_2 \ \rightarrow\ r$

Return x^y, where $x = r_1$ and $y = r_2$.

$\tilde{}r \;\rightarrow\; r$

Return $-r$, the negative value of r.

$r_1\tilde{}{<}r_2 \mid r_1\tilde{}{<=}r_2 \mid r_1\tilde{}{<>}r_2 \mid r_1\tilde{}{=}r_2 \mid r_1\tilde{}{>}r_2 \mid r_1\tilde{}{>=}r_2 \;\rightarrow\; b$

For numbers, these operators are identical to $<$, $<=$,, $=$, etc.

$c_1\tilde{}{<}c_2 \mid c_1\tilde{}{<=}c_2 \mid c_1\tilde{}{<>}c_2 \mid c_1\tilde{}{=}c_2 \mid c_1\tilde{}{>}c_2 \mid c_1\tilde{}{>=}c_2 \;\rightarrow\; b$

These are shorthand notations for

$$c\_downcase\ c_1\ R\ c\_downcase\ c_2$$

where R is in $\{<\,, <=, =\,, <>\,, >\,, >=\}$.

$s_1\tilde{}{<}s_2 \mid s_1\tilde{}{<=}s_2 \mid s_1\tilde{}{<>}s_2 \mid s_1\tilde{}{=}s_2 \mid s_1\tilde{}{>}s_2 \mid s_1\tilde{}{>=}s_2 \;\rightarrow\; b$

These are equal to their counterparts ($<$, $<=$, $=$, etc), but use the above case-folding variants of the comparison operators to compare individual characters.

Structural Operations

Many functions in this section are curried higher-order functions. The signatures of these functions are specified like this:

$$map\ f\ L \;\rightarrow\; L$$

which makes the function appear to take more than one argument. However, the above is merely a shorthand notation for

$$map\ f \;\rightarrow\; L \;\rightarrow\; L$$

So, for example, map is always applied to a single unary function, giving another function $L \rightarrow L$, which may then be applied on the spot to a list. E.g.: $map\ (fn\ x = 2\hat{}x)\ [1, 2, 3, 4, 5]$.

$x :: L \;\rightarrow\; L$

Attach a new element x to the front of an existing list L, giving a new list.

$L_1 @ L_2 \;\rightarrow\; L$

Return a new list that consists of the concatenation of L_1 and L_2.

$s_1 @ s_2 \ \rightarrow \ s$

Return a new string that consists of the concatenation of the strings s_1 and s_2.

$append\_map \ f \ L \ \rightarrow \ L$

append_map is like *map*, but appends the elements of its result rather than consing them, i.e.:

$$append\_map \ f \ [a, b, c]$$

would be equal to

$$f \ a \ @ \ f \ b \ @ \ f \ c$$

$clone \ x \ \rightarrow \ x$

Return an exact copy of the object x. Note that only vectors are actually cloned by this function, because all other data types are immutable.

$x_1 \ eql \ x_2 \ \rightarrow \ b$

The *eql* function is an extension of the = operator that covers all data types of the mLite language and also allows to compare incompatible types, resulting in *false*.

Lists, vectors, and tuples are compared element-wise and recursively. This function is similar to Scheme's "equal?" procedure.

$explode \ s \ \rightarrow \ L$

Return a new list that contains the same characters as the string s in the same order. This is the reverse operation of *implode*.

$filter \ f \ L \ \rightarrow \ L$

Return a new list containing all elements x_i satisfying the condition $f \ x_i$ from the list L.

$fold \ (f, x) \ L \ \rightarrow \ L$

Fold the list L by applying f to the neutral element x and the first element of L (giving a result R) and then applying f to R and the second element of L, etc.

Formally, $fold\ (f, x)\ [a, b, c]$ is equal to $f\ (f\ (f\ (x, a), b), c)$.

$foldr\ (f, x)\ L\ \rightarrow\ L$

The $foldr$ function is like $fold$, but folds the list L to the right, so

$foldr\ (f, x)\ [a, b, c]$ is equal to $f\ (a, f\ (b, f\ (c, x)))$.

$foreach\ f\ L\ \rightarrow\ ()$

The $foreach$ function is like map, but calls f only for effect. It does not collect any result and always returns ().

$head\ L\ \rightarrow\ x$

Return the first element of a list.

$implode\ L\ \rightarrow\ s$

Return a new string that contains the same characters as the list L in the same order. This is the reverse operation of $explode$.

$iota\ n\ \rightarrow\ L$

Return a list $[1, 2, \cdots, n]$.

$iota\ (n_1, n_2)\ \rightarrow\ L$

Return a list $[n_1, n_1 + 1, \cdots, n_2]$.

$len\ T_k \mid len\ L \mid len\ V \mid len\ s\ \rightarrow\ n$

Return the length of the given data object. For lists and vectors, this is the number of elements, for strings the number of characters, for tuples, their order.

$map\ f\ L\ \rightarrow\ L$

Map the function f over the list L, generating a new list

$$[f\ a_1, f\ a_2, \cdots]$$

where each a_i is an element of L.

$newstr\ (n, c)\ \rightarrow\ s$

Return a new string of the length n containing the character c in all positions of the string.

newvec $(n, x) \rightarrow V$

Return a new vector of the length n containing the element x in all slots of the vector.

order $x \rightarrow n$

Return the order of x. The order of each "true" tuple T_k is k. The order of () is zero, and the order of all other objects is one, i.e. all singular objects are 1-tuples.

ref (L, n) | *ref* (T_k, n) | *ref* $(V, n) \rightarrow x$

Extract the n'th element of the given data object. Extracting an element from a vector is guaranteed to be an $O(1)$ operation.

ref $(s, n) \rightarrow c$

Extract the n'th char of the string s.

rev $L \rightarrow L$

Return a new list containing the elements of L in reverse order.

rev $s \rightarrow s$

Return a new string containing the characters of the string s in reverse order.

set $(T_k, n, x) \rightarrow T_k$

Return a copy of the tuple T_k with the n'th element replaced by x.

set $(s, n, c) \rightarrow s$

Return a copy of the string s with the n'th char replaced by c.

set $(V, n, x) \rightarrow V$

Change the n'th element of the vector V to x.
Note: this will mutate V!

setvec$(V, n_1, n_2, x) \rightarrow V$

Change the elements in the slots $n_1 \cdots n_2 - 1$ of the vector V to x.
Note: this will mutate V!

$sub\ (L, n_1, n_2)\ \rightarrow\ L$

Return a new list containing the elements at positions $n_1 \cdots n_2 - 1$ of the original list L.

$sub\ (s, n_1, n_2)\ \rightarrow\ s$

Return a new string containing the characters at positions $n_1 \cdots n_2 - 1$ of the original string s.

$tail\ L\ \rightarrow\ L$

Return the tail (all but the first element) of a list.

$zip\ L_1\ L_2\ \rightarrow\ L$

Combine the lists L_1 and L_2 pairwise, returning a list of tuples:

$$zip\ [1,2,3]\ [4,5,6]\ \rightarrow\ [(1,4),(2,5),(3,6)]$$

This is identical to $zipwith\ (fn\ x = x)$.

$zipwith\ f\ L_1\ L_2\ \rightarrow\ L$

Combine the lists L_1 and L_2 pairwise using the function f, so that

$$zipwith\ f\ [1,2,3]\ [4,5,6]$$

would be equal to

$$[f\ (1,4), f\ (2,5), f\ (3,6)]$$

Type Predicates and Conversion

$bool\ x\ \rightarrow\ b$

Return $true$, if x is a boolean.

$char\ x\ \rightarrow\ b$

Return $true$, if x is a char.

$chr\ n\ \rightarrow\ c$

Return the character at the code point n. Inverse operation: ord.

$int\ x\ \rightarrow\ b$

Return $true$, if x is an integer.

not x $\rightarrow$ *b*

Return *true*, if *x* is *false*, else return *false* (logical “not”; all values but *false* are considered to be “true”).

ntos r $\rightarrow$ *s*

Return a string representation of the number *r*. Negative numbers will have a “-” prefix.

ord c $\rightarrow$ *n*

Return the code point of the character *c*. Inverse operation: *chr*.

real x $\rightarrow$ *b*

Return *true*, if *x* is a real number. Note: *int x* implies *real x*.

ston s $\rightarrow$ *r* | *false*

Convert a numeric string to a number. If the string *s* contains a decimal point (“.”), return a real number, otherwise return an integer. Both “–” and “~” will be accepted as a leading minus sign. When *s* does not represent a valid number, *ston* will return *false*.

str x $\rightarrow$ *b*

Return *true*, if *x* is a string.

vec x $\rightarrow$ *b*

Return *true*, if *x* is a vector.

Char Functions

c_alphabetic c $\rightarrow$ *b*

Return *true*, if *c* is a letter of the English alphabet.

c_downcase c $\rightarrow$ *c*

If *c_upper c*, return the lower-case variant of *c*, else return *c*.

c_lower c $\rightarrow$ *b*

Return *true*, if *c* is a lower-case letter of the English alphabet.

c_numeric c $\rightarrow$ *b*

Return *true*, if *c* is a decimal digit.

c_upcase c $\rightarrow$ *c*

If *c_lower c*, return the upper-case variant of *c*, else return *c*.

c_upper c $\rightarrow$ *b*

Return *true*, if *c* is an upper-case letter of the English alphabet.

c_whitespace c $\rightarrow$ *b*

Return *true*, if *c* is a non-printing character (blank, or ASCII HT, LF, CR, or FF).

Input/Output Functions

append_stream s $\rightarrow$ *SO*

Open the file *s* for writing and return an *outstream* object for accessing that file. When the file already exists, append output to the existing file.

close SI | *close SO* $\rightarrow$ ()

Close the given *instream* or *outstream*. Note that streams are also closed automatically (by the garbage collector), so there is normally no need to use *close*, except for the rare case where output must be committed at a specific point during program execution.

eof x $\rightarrow$ *b*

Return *true*, if *x* is an end-of-file indicator (EOF), as delivered by *readc*, *peekc*, and *readln*.

instream s $\rightarrow$ *SI*

Open the file *s* for reading and return an *instream* object for accessing that file. An error is reported, if the file does not exist.

load s $\rightarrow$ ()

Load the program in the file *s* as if typed in at the mLite prompt. When an error occurs, stop loading.

outstream $s \;\rightarrow\; SO$

Open the file s for writing and return an *outstream* object for accessing that file. When the file already exists, it is truncated to zero length.

peekc $() \;\rightarrow\; c$

Read a character from the current input stream and return it. Do *not* consume the input character, i.e. a subsequent read operation will yield the same character again. When there are no (more) characters to be read from the stream, return EOF.

print $x \;\rightarrow\; ()$

Write a suitable representation of the expression x to the current output stream.

println $x \;\rightarrow\; ()$

Shorthand for (*print* x; *print* #"*newline*").

readc $() \;\rightarrow\; c$

Read a character from the current input stream and return it. When there are no (more) characters to be read from the stream, return EOF.

readln $() \;\rightarrow\; s$

Read one line of characters from the current input stream and return it. A line is delimited by a platform-specific newline sequence or the EOF. When there are no (more) characters to be read from the stream, return EOF.

Appendix

mLite Grammar

; ; *comment*	is a comment to the end of line.
(* *comment* *)	is a nestable block comment.

The & grammar operator indicates that no spaces are allowed between two sequences, i.e. it defines lexemes rather than sentences. For instance, 1 2 would match 1 2 with any number of blanks in between, but 1 & 2 would only match 12 (with no blanks between the digits).

top :=
    *decl (';' top) **
  | *expr (';' top) **
  | *ε*

program := decl + expr

decl :=
    *'val' pat '=' expr ('and' pat '=' expr) **
  | *'fun' id curried_match ('and' id curried_match) **
  | *'type' id '=' cons ('|' cons) **
  | *'exception' :id ('and' :id) **
  | *'local' ldecls 'in' ldecls 'end'*
  | *'infix' fdecl (',' fdecl) **
  | *'infixr' fdecl (',' fdecl) **
  | *'nonfix' id (',' id) **

fdecl :=
    id '=' id
  | *id '<' id*
  | *id '>' id*

cons :=
    :id
  | *:id '(' id (',' id) * ')'*

*ldecls := ldecl (‘;’ ldecls) **

ldecl :=
    *‘val’ pat ‘=’ expr (‘and’ pat ‘=’ expr) **
  | *‘fun’ id curried_match (‘and’ id curried_match) **

curried_match :=
        guarded_pat + ’=’ expr
  *(‘|’ guarded_pat + ‘=’ expr) **

*match := guarded_pat ‘=’ expr (‘|’ guarded_pat ‘=’ expr) **

guarded_pat :=
    pat
  | *pat ‘where’ or_expr*

pat := pat_or

pat_or :=
    pat_also
  | *pat_or ’or’ pat_also*

pat_also :=
    pat_infix
  | *pat_also ’or’ pat_infix*

pat_infix :=
    pat_funapp
  | *pat_infix id pat_infix*

pat_funapp :=
    pat_primary
  | *pat_funapp pat_primary*

pat_primary :=
    unit
  | *bool*
  | *int*

| *real*
| *char*
| *str*
| *id*
| *'_'*
| *list_pat*
| *tuple_pat*
| *:id id*
| *:id tuple_pat*
| *'(' pat ')'*

list_pat :=
'[' ']'
| '[' *pat* (',' *pat*) * ']'
| '[' *pat* '::' *id* ']'
| '[' *pat* '::' *list_pat* ']'

tuple_pat := '(' *pat* (',' *pat*) + ')'

expr :=
case_expr
| *case_expr* '<<' *case_expr*
| *case_expr* '>>' *case_expr*

sequence := *case_expr* (';' *caseexpr*) *

case_expr :=
ifexpr
| *'case' expr 'of' match*

ifexpr :=
raise_expr
| *'if' expr 'then' expr 'else' expr*

raise_expr :=
handle_expr
| *'raise' :id*

handle_expr :=
or_expr
| *or_expr 'handle' match*

or_expr :=
also_expr
| *or_expr 'or' also_expr*

also_expr :=
apply
| *also_expr 'also' apply*

apply :=
infix
| *apply "' infix*

infix :=
funapp
| *infix id infix*

funapp :=
primary
| *funapp primary*

primary :=
unit
| *bool*
| *int*
| *real*
| *char*
| *str*
| *tuple*
| *list*
| *id*
| *'#' int tuple*
| *'#' int id*
| *'fn' curried_match*

| ‘let’ ldecls ‘in’ sequence ‘end’
| ‘(’ sequence ‘)’

list :=
‘[’ ‘]’
| ‘[’ expr (‘,’ expr) * ‘]’
| ‘[’ expr ‘::’ list ‘]’

tuple := ‘(’ expr (‘,’ expr) + ‘)’

unit := ‘(’ ‘)’

bool :=
‘true’
| ‘false’

int :=
natural
| ‘~’ & natural

natural :=
digit
| digit & natural

real :=
positive_real
| ‘~’ & positive_real

positive_real :=
int & ‘.’ & int
| int & ‘e’ & int
| int & ‘.’ & int & ‘e’ & int

char := ‘#"’ & <c> & ‘"’

str := ‘"’ & <c> * & ‘"’

:id := ’:’ & name

```
id :=
      name
    | opname

name :=
      sym_char
    | sym_char & sym_chars

op_name :=
      op_char
    | op_char & op_name

sym_char :=
      ‘a’ | ... | ‘z’
    | ‘A’ | ... | ‘Z’
    | ‘:’ | ‘_’ | ‘”

sym_chars :=
      sym_char
    | digit
    | sym_chars & sym_chars

op_char :=
      ‘!’ | ‘@’ | ‘$’ | ‘%’ | ‘^’
    | ‘&’ | ‘*’ | ‘–’ | ‘+’ | ‘<’
    | ‘=’ | ‘>’ | ‘/’ | ‘~’ | “’

digit :=
      ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’
    | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

mLite and Scheme Functions

mLite	**Scheme**
c_alphabetic	char-alphabetic
c_downcase	char-downcase
c_lower	char-lowercase
c_numeric	char-numeric
c_upcase	char-upcase
c_upper	char-uppercase
c_whitespace	char-whitespace
chr	integer->char
div	quotient
eof	eof-object?
explode	string->list
instream	open-input-file
newstr	make-string
newvec	make-vector
ord	char->integer
outstream	open-output-file
peekc	peek-char
print	display
readc	read-char
rem	remainder

Differences to ML

- The module language is completely absent.
- There is no static type system. This *may* change, though.
- Lists are heterogenous, this *may* also change.
- Function names are not repeated in *fun* definitions, e.g.:
  fun not true = false | _ = true
  instead of
  fun not true = false | not _ = true.
- An equal sign separates the pattern from the body in *fn*, e.g.:
  fn x = x instead of *fn x => x*.

- Currying cannot be combined with multiple patterns, e.g. the function
  $fun\ p\ x\ 0 = 1 \mid x\ y = p\ x\ (y - 1)$
  would have to be written using a tuple as argument:
  $fun\ p\ (x, 0) = 1 \mid (x, y) = p\ (x, y - 1)$
- There are implicit and explicit guards, e.g.:
  $fun\ f\ (a, b)\ where\ a < b = ...$
  or
  $fun\ f\ x < 0 = ...$
- All user-defined types and constructors must begin with a colon, e.g.: *:cons*, *:node*, etc.
- Algebraic types are declared with the *type* keyword instead of *datatype*, and the declaration syntax is different, e.g.
  $type\ {:}list = {:}nil \mid {:}cons\ (x, {:}list)$
  instead of
  $datatype\ 'a\ list = nil \mid cons\ of\ 'a * 'a\ list$
- Exceptions are just atomic types, so *exception :exn* is equal to $type : exn = : exn$.
- Only *val* and *fun* are allowed in the declaration parts of *let* and *local*.
- There are no *val rec* declaration, *fun* must be used instead.
- To concatenate strings, the overloaded @ operator is used instead of the ^ operator (which is used for exponentiation).
- Logical or is named *or* instead of *orelse*.
- Logical and is named *also* instead of *andalso*.
- The $<$, $\leq$, $<>$, $=$, $>$, and $\geq$ operators implement case-insensitive lexical comparison.

See the mLite Function Summary (pg 39) for an overview of predefined mLite functions. The function library differs largely from ML's.

References

[DEFSML] Robin Milner, et al.
"The Definition of Standard ML (Revised)"
MIT Press, 1997

[HOL14] Nils M Holm
"Implicit Guard Expressions in
Functional Programming Languages"
Self-published, 2014

[R4RS] William Clinger and Jonathan Rees (Editors)
"Revised(4) Report on the
Algorithmic Language Scheme"
ACM Lisp Pointers IV (July-September 1991)