$\lambda^a\mathcal{M}^a$ Language Specification

Author
Dmitry BOULYTCHEV

September, 10, 2020

Contents

Chapter 1

Introduction

$\lambda^a\mathcal{M}^a$ is a programming language developed by JetBrains Research for educational purposes as an exemplary language to introduce the domain of programming languages, compilers and tools. Its general characteristics are:

- procedural with first-class functions — functions can be passed as arguments, placed in data structures, returned and "constructed" at runtime via closure mechanism;
- with lexical static scoping;
- strict — all arguments of function application are evaluated before function body;
- imperative — variables can be re-assigned, function calls can have side effects;
- untyped — no static type checking is performed;
- with S-expressions and pattern-matching;
- with user-defined infix operators, including those defined in local scopes;
- with automatic memory management (garbage collection).

The name $\lambda^a\mathcal{M}^a$ is an acronym for λ-ALGOL since the language has borrowed the syntactic shape of operators from ALGOL-68 [10]; HASKELL [2] and OCAML [3] can be mentioned as other languages of inspiration.

The main purpose of $\lambda^a\mathcal{M}^a$ is to present a repertoire of constructs with certain runtime behavior and relevant implementation techniques. The lack of a type system (a vital feature for a real-word language for software engineering) is an intensional decision which allows to show the unchained diversity of runtime behaviors, including those which a typical type system is called to prevent. On the other hand the language can be used in future as a raw substrate to apply various ways of software verification (including type systems) on.

The current implementation contains a native code compiler for X86-32, written in OCAML, a runtime library with garbage-collection support, written in C, and a small standard library, written in $\lambda^a\mathcal{M}^a$ itself. The native code compiler uses GCC as a toolchain.

In addition, a source-level reference interpreter is implemented as well as a compiler to a small stack machine. The stack machine code can in turn be either interpreted on a stack machine interpreter, or used as an intermediate representation by the native code compiler.

Chapter 2

Concrete Syntax and Semantics

In this chapter we describe the concrete syntax of the language as it is recognized by the parser. In the syntactic description we will use extended Backus-Naur form with the following conventions:

- nonterminals are presented in *italics*;
- concrete terminals are `grayed out`;
- classes of terminals are CAPITALIZED;
- a postfix "*" designates zero-or-more repetitions;
- square brackets "[...]" designate zero-or-one repetition;
- round brackets "(...)" are used for grouping;
- alteration is denoted by " | ", sequencing by juxtaposition;
- a colon ":" separates a nonterminal being defined from its definition.

In the description below we will take in-line code samples in blockquotes "..." which are not considered as a part of concrete syntax.

2.1 Lexical Structure

The character set for the language is ASCII, case-sensitive. In the following lexical description we will use the GNU Regexp syntax [6] in lexical definitions.

2.1.1 Whitespaces and Comments

Whitespaces and comments are ASCII sequences which serve as delimiters for other tokens but otherwise are ignored.

The following characters are treated as whitespaces:

- blank character " ";
- newline character "\n";
- carriage return character "\r";
- tabulation character "\t".

Additionally, two kinds of comments are recognized:

- end-of-line comment "--" escapes the rest of the line, including itself;
- block comment "(* ... *)" escapes all the text between "(*" and "*)".

There is a number of specific cases which have to be considered explicitly.

First, block comments can be properly nested. Then, the occurrences of comment symbols inside string literals (see below) are not considered as comments.

End-of-line comment encountered *outside* of a block comment escapes block comment symbols:

```
-- the following symbols are not considered as a block comment: (*
-- same here: *)
```

Similarly, an end-of-line comment encountered inside a block comment is escaped:

```
(* Block comment starts here ...
   -- and ends here: *)
```

2.1.2 Identifiers and Constants

The language distinguishes identifiers, signed decimal literals, string and character literals (see Fig. 2.1). There are two kinds of identifiers: those beginning with uppercase characters (UIDENT) and lowercase characters (LIDENT).

String literals cannot span multiple lines; a blockquote character (") inside a string literal has to be doubled to prevent from being considered as this literal's delimiter.

Character literals as a rule are comprised of a single ASCII character; if this character is a quote (') it has to be doubled. Additionally two-character abbreviations "\t" and "\n" are recognized and converted into a single-character representation.

```
UIDENT  =  [A-Z][a-zA-Z_0-9]*
LIDENT  =  [a-z][a-zA-Z_0-9]*
DECIMAL =  -?[0-9]+
STRING  =  "([^\"]|"")*"
CHAR    =  '([^']|''|\n|\t)'
```

Figure 2.1: Identifiers and constants

2.1.3 Keywords

The following identifiers are reserved for keywords:

```
after   array   at      before  boxed   case    do      elif    else
esac    eta     false   fi      for     fun     if      import  infix
infixl  infixr  lazy    length  local   od      of      public  repeat
return  sexp    skip    string  string  syntax  then    true    unboxed
until   when    while
```

2.1.4 Infix Operators

Infix operators defined as follows:

INFIX = `[+*/%$#@!|&^ ?<>:=\-]+`

There is a predefined set of built-in infix operators (see Fig. 2.4); additionally an end-user can define custom infix operators (see Section 3.1). Note, sometimes additional whitespaces are required to disambiguate infix operator applications. For example, if a custom infix operator "+−" is defined, then the expression "`a +- b`" can no longer be recognized as "`a +`(−`b`)". Note also that a custom operator containing "`--`" can not be defined due to lexical conventions.

2.1.5 Delimiters

The following symbols are treated as delimiters:

```
.   ,   (   )   {   }
;   #   →   |
```

Note, custom infix operators can coincide with delimiters "#", "|", and "→", which can sometimes be misleading.

2.2 Compilation Units

Compilation unit is a minimal structure recognized by the parser. An application can contain multiple units, compiled separately. In order to use other units they have to be imported. In particular, the standard library is comprised

```
compilationUnit : import* scopeExpression
import : import UIDENT ;
```

Figure 2.2: Compilation unit concrete syntax

```
scopeExpression : definition* [ expression ]
definition : variableDefinition
           | functionDefinition
           | infixDefinition
variableDefinition : ( local | public ) variableDefinitionSequence ;
variableDefinitionSequence : variableDefinitionItem ( , variableDefinitionItem )*
variableDefinitionItem : LIDENT [ = basicExpression ]
functionDefinition : [ public ] fun LIDENT ( functionArguments )
                     functionBody
functionArguments : [ LIDENT ( , LIDENT )* ]
functionBody : { scopeExpression }
```

Figure 2.3: Scope expression concrete syntax

of a number of precompiled units, which can be imported by an end-user application.

The concrete syntax for compilation unit is shown on Fig. 2.2. Besides optional imports a unit must contain a *scopeExpression*, which may contain some definitions and computations. Note, a unit can not be empty. The computations described in a unit are performed at unit initialization time (see Chapter 4).

2.3 Scope Expressions

Scope expressions provide a mean to put expressions is a scoped context. The definitions in scoped expressions comprise of function definitions and variable definitions (see Fig. 2.3). For example:

```
local x, y, z; -- variable definitions

fun id (x) {x} -- function definition
```

As scope expressions are expressions, they can be nested:

```
local x;

{ -- nested scope begins here
  local y;
```

```
  skip
} -- nested scope ends here
```

The definitions on the top-level of compilation unit can be tagged as "**public**", in which case they are exported and become visible by other units which import the given one. Nested scopes can not contain public definitions.

The nesting relation has the shape of a tree, and in a concrete node of the tree all definitions in all enclosing scopes are visible:

```
local x;

{local y;
  {local z;
    skip -- x, y, and z are visible here
  };
  {local t;
    skip -- x, y, and t are visible here
  };
  skip -- x and y are visible here
};
skip -- only x is visible here
```

Multiple definitions of the same name in the same scope are prohibited:

```
local x;
fun x () {0} -- error
```

However, a definition is a nested scope can override a definition in an enclosing one:

```
local x;

{
  fun x () {0} -- ok
  skip         -- here x is associated with the function
};

skip -- here x is associated with the variable
```

A function can freely use all visible definitions; in particular, functions defined in the same scope can be mutually recursive:

```
local x;
fun f () {0}

{
  fun g () {f () + h () + y} -- ok
  fun h () {g () + x}        -- ok
  local y;
  skip
};
skip
```

A variable, defined in a scope, can be attributed with an expression, calculating its initial value. These expressions, however, are evaluated in the order of variable declaration. Thus, while technically it is possible to have forward references in the initialization expression, their behavior is undefined. For example:

```
local x = y + 2; -- undefined, as y is not yet initialized at this point
local y = x + 2;
skip
```

2.4 Expressions

The syntax definition for expressions is shown on Fig. 2.6. The top-level construct is *sequential composition*, expressed using right-associative connective "`;`". The basic blocks of sequential composition have the form of *binaryExpression*, which is a composition of infix operators and operands. The description above is given in a highly ambiguous form as it does not specify explicitly the precedence and associativity of infix operators. The precedences and associativity of predefined built-in infix operators are shown on Fig. 2.4 with the precedence level increasing top-to-bottom.

infix operator(s)	description	associativity
:=	assignment	right-associative
:	list constructor	right-associative
!!	disjunction	left-associative
&&	conjunction	left-associative
==, !=, <=, <, >=, >	integer comparisons	non-associative
+, −	addition, subtraction	left-associative
∗, /, %	multiplication, quotient, remainder	left-associative

Figure 2.4: The precedence and associativity of built-in infix operators

Apart from assignment and list constructor all other built-in infix operators operate on signed integers; in conjunction and disjunction any non-zero value is treated as truth and zero as falsity, and the result respects this convention.

The assignment operator is unique among all others in the sense that it requires its left operand to designate a *reference*. This property is syntactically ensured using an inference system shown on Fig. 2.5; here $\mathcal{R}(e)$ designates the property "e is a reference". The result of assignment operator coincides with its right operand, thus

```
x := y := 3
```

assigns 3 to both "`x`" and "`y`".

$$\mathcal{R}(x),\ x \text{ is a variable} \qquad \frac{\mathcal{R}(e)}{\mathcal{R}(e[\ldots])}$$

$$\frac{\mathcal{R}(e_i)}{\mathcal{R}(\mathbf{if}\ldots\mathbf{then}\ e_1\ \mathbf{else}\ e_2\ \mathbf{fi})} \qquad \frac{\mathcal{R}(e_i)}{\mathcal{R}(\mathbf{case}\ldots\mathbf{of}\ \ldots \to e_1\ \ldots\ \ldots \to e_k\ \mathbf{esac})}$$

$$\frac{\mathcal{R}(e)}{\mathcal{R}(\ldots; e)}$$

Figure 2.5: Reference inference system

2.4.1 Postfix Expressions

There are four postfix forms of expressions:

- function call, designated as postfix form "$(arg_1,\ldots,arg_k)$";
- array element selection, designated as "$[index]$";
- built-in primitive "`.string`", returning the string representation of a value;
- built-in primitive "`.length`", returning the length of a boxed value.

Multiple postfixes are allowed, for example

```
x () [3] (1, 2, 3) . string
x . string [4]
x . length . string
x . string . length
```

The basic form of expression is *primary*. The simplest form of primary is an identifier or constant. Keywords **`true`** and **`false`** designate integer constants 1 and 0 respectively, character constant is implicitly converted into its ASCII code. String constants designate arrays of one-byte characters. Infix constants allow to reference a functional value associated with corresponding infix operator (however, a value associated with builtin assignment operator ":=" can not be taken), and functional constant (*lambda-expression*) designates an anonymous functional value in the form of closure.

expression : *basicExpression* (`;` *expression*)
basicExpression : *binaryExpression*
binaryExpression : *binaryOperand* INFIX *binaryOperand* |
binaryOperand
binaryOperand : *binaryExpression* |
[`-`] *postfixExpression*
postfixExpression : *primary* |
postfixExpression `(` [*expression* (`,` *expression*)*] `)` |
postfixExpression `[` *expression* `]` |
postfixExpression `.` length |
postfixExpression `.` string
primary : DECIMAL |
STRING |
CHAR |
LIDENT |
true |
false |
infix INFIX |
fun `(` *functionArguments* `)` *functionBody* |
skip |
return [*basicExpression*] |
`{` *scopeExpression* `}` |
listExpression |
arrayExpression |
S-expression |
ifExpression |
whileExpression |
repeatExpression |
forExpression |
caseExpression |
`(` *expression* `)`

Figure 2.6: Expression concrete syntax

2.4.2 `skip` and `return` Expressions

Expression **`skip`** can be used to designate a no-value when no action is needed (for example, in the body of unit which contains only declarations). **`return`** expression can be used to immediately complete the execution of current function call; optional return value can be specified.

2.4.3 Arrays, Lists, and S-expressions

arrayExpression : [[*expression* (, *expression*)*]]
listExpression : { [*expression* (, *expression*)*] }
S-expression : UIDENT [(*expression* [(, *expression*)*])]

Figure 2.7: Array, list, and S-expressions concrete syntax

There are three forms of expressions to specify composite values: arrays, lists and S-expressions (see Fig. 2.7). Note, it is impossible to specify one-element list as "{*e*}" since it is treated as scope expression. Instead, the form "*e*:{}" can be used; alternatively, a standard unit "`List`" (see Section 6.9) defines function "`singleton`" which serves for the same purpose.

2.4.4 Conditional Expressions

ifExpression : if *expression* then *scopeExpression* [*elsePart*] fi
elsePart : elif *expression* then *scopeExpression* [*elsePart*] |
else *scopeExpression*

Figure 2.8: If-expression concrete syntax

Conditional expression branches the control depending in the value of a certain expression; the value zero is treated as falsity, nonzero as truth. An extended form

`if` c_1 **`then`** $e1$
`elif` c_2 **`then`** e_2
...
`else` e_{k+1}
`fi`

is equivalent to a nested form

if c_1 **then** $e1$
else if c_2 **then** e_2
...
else e_{k+1}
fi

2.4.5 Loop Expressions

There are three forms of loop expressions — "**while**", "**repeat**", and "**for**", among which "**while**" is the basic one (see Fig. 2.9). In "**while**" expression the evaluation of the body is repeated as long as the evaluation of condition provides a non-zero value. The condition is evaluated before the body on each iteration of the loop, and the body is evaluated in the context of condition evaluation results.

The construct "**repeat** e **until** c" is derived and operationally equivalent to

e ; **while** $c == 0$ **do** e **od**

However, the top-level local declarations in the body of "**repeat**"-loop are visible in the condition expression:

```
repeat local x = read () until x
```

The construct "**for** i , c , s **do** e **od**" is also derived and operationally equivalent to

i ; **while** c **do** e ; s **od**

However, the top-level local definitions of the the first expression ("i") are visible in the rest of the construct:

```
for local i; i := 0, i < 10, i := i + 1 do write (i) od
```

2.4.6 Pattern Matching

Pattern matching is introduced into the language by the mean of *case-expression* (see Fig. 2.11). A case-expression evaluates an expression, called *scrutinee*, and performs branching depending on its structure. This structure is specified by means of *patterns* (see Fig. 2.10). If succeeded, a matching against a pattern delivers a set of bindings — variables with their bindings to the (sub)values of the scrutinee.

The semantics of patterns is as follows:

- a pattern "p_1:p_2" matches a list with a head matched with p_1 and a tail matched with p_2;
- wildcard pattern "_" matches every value;

whileExpression : while *expression* do *scopeExpression* od
repeatExpression : repeat *scopeExpression* until *basicExpression*
forExpression : for *expression* , *expression* , *expression*
do *scopeExpresssion* od

Figure 2.9: Loop expressions concrete syntax

- S-expression pattern "$C\ (p_1,\ldots,p_k)$" matches a value with corresponding top-level tag ("C") and arguments matched by subpatterns p_i respectively; note, patterns can discriminate on the number of arguments for the same constructor, thus the same tag with different number of arguments can be used in different branches of the same case expression (see below);
- array and list patterns match arrays and lists of the specified length with each element matched with corresponding subpattern;
- an identifier matches every value and binds itself to that value in the corresponding branch of case-expression (see below);
- a "$x@p$"-pattern matches what pattern p matches, and additionally binds the matched value to the identifier x;
- constant patterns match corresponding constants;
- six "#"-patterns match values of corresponding shapes (boxed, unboxed, string, array, S-expression or closure) regardless their content;
- round brackets can be used for grouping.

All identifiers, occurred in a pattern, have to be pairwise distinct.

The matching against patterns in case-expression is performed deterministically in a top-down manner: a pattern is matched against only if all previous matchings were unsuccessful. If no matching pattern is found, the execution of the program stops with an error.

pattern : *consPattern* | *simplePattern*
consPattern : *simplePattern* `:` *pattern*
simplePattern : *wildcardPattern* |
S-exprPattern |
arrayPattern |
listPattern |
LIDENT [`@` *pattern*] |
[`-`] DECIMAL |
STRING |
CHAR |
`true` |
`false` |
`#` `boxed` |
`#` `unboxed` |
`#` `string` |
`#` `array` |
`#` `sexp` |
`#` `fun` |
`(` *pattern* `)`
wildcardPattern : `_`
S-exprPattern : UIDENT [`(` *pattern* (`,` *pattern*)* `)`]
arrayPattern : `[` [*pattern* (`,` *pattern*)*] `]`
listPattern : `{` [*pattern* (`,` *pattern*)*] `}`

Figure 2.10: Pattern concrete syntax

caseExpression : `case` *expression* `of` *caseBranches* `esac`
caseBranches : *caseBranch* [(`|` *caseBranch*)*]
caseBranch : *pattern* $\rightarrow$ *scopeExpression*

Figure 2.11: Case-expression concrete syntax

Chapter 3

Extensions

There are some extensions for the core language defined in the previous chapters. These extensions add some syntactic sugar, which makes writing programs in $\lambda^a\mathcal{M}^a$ a less painful task.

3.1 Custom Infix Operators

Besides the set of builtin infix operators (see Fig. 2.4) users may define custom infix operators. These operators may be declared at any scope level; when defined at the top level they can be exported as well. However, there are some restrictions regarding the redefinition of builtin infix operators:

- redefinitions of builtin infix operators can not be exported;
- the assignment operator ":=" can not be redefined;
- infix definitions can not be mutually recursive.

The syntax for infix operator definition is shown on Fig. 3.1; a custom infix definition must specify exactly two arguments. An associativity and precedence level has to be assigned to each custom infix operator. A precedence level is assigned by specifying at which position, relative to other known infix operators, the operator being defined is inserted. Three kinds of specifications are allowed: at given level, immediately before or immediately after. For example, "**`at`** +" means that the operator is assigned exactly the same level of precedence as "+"; "**`after`** +" creates a new precedence level immediately *after* that for "+" (but *before* that for "∗"), and "**`before`** ∗" has exactly the same effect (provided there were no insertions of precedence levels between those for "+" and "∗").

When begin inserted at existing precedence level, an infix operator inherits the associativity from that level; hence, only "**`infix`**" keyword can be used for such definitions. When a new level is created, an associativity for this level

infixDefinition : *infixHead* (*functionArguments*) *functionBody*
infixHead : [public] *infixity* INFIX *level*
infixity : infix | infixl | infixr
level : [at | before | after] INFIX

Figure 3.1: The Syntax for Infix Operator Definition

has to be additionally specified by using corresponding keyword ("**infix**" for non-associative levels, "**infixr**" — for levels with right associativity, and "**infixl**" — for levels with left associativity).

When public infix operators are exported, their relative precedence levels and associativity are exported as well; since not all custom infix definitions may be made public some levels may disappear from the export. For example, let us have the following definitions:

```
infixl ** before * (x, y) {...}
public infixr *** before ** (x, y) {...}
```

Here in the top scope for the compilation unit we have two additional precedence levels: one for the "`**`" and another for the "`***`". However, as "`**`" is not exported its precedence level will be forgotten during the import. Thus, only the precedence level for "`***`" will be created during the import as if is was defined at the level "**before** `*`".

Respectively, multiple imports of units with custom infix operators will modify the precedence level in the order of their import. For example, if there are two units "`A`" and "`B`" with declarations "**infixl** `++` **before** `+`" and "**infixl** `+++` **before** `+`" correspondingly, then importing "`B`" after "`B`" will result in "`++`" having a *lower* precedence, then "`+++`".

3.2 Lazy Values and Eta-expansion

An expression

lazy *e*

where *e* — a *basicExpression* — is converted into

`makeLazy (fun () {`*e*`})`

where "`makeLazy`" — a function from standard unit "`Lazy`" (see Section 6.8). An import for "`Lazy`" is added implicitly.

An expression

eta *e*

where *e* — a *basicExpression* — is converted into

fun (*x*) {*e* (*x*)}

where "*x*" — a fresh variable which does not occur free in "*e*".

3.3 Dot Notation

A function call

$f\ (e_1, \ldots, e_k)$

where f — an identifier — can be rewritten as

$e_1.f\ (e_2, \ldots, e_k)$

In particular, a call to a one-argument function $f(e)$ can be rewritten as $e.f$.

3.4 Patterns in Function Arguments

Patterns can be used in function argument specification: a declaration

```
fun f (p1, ..., pk) { e }
```

is equivalent to

```
fun f (x1, ..., xk) {
  case x1 of
    p1 → case x2 of
           ... → e
         esac
  esac
}
```

where x_i — fresh variables, not free in e.

3.5 Syntax Definitions

Syntax definition extension represents an alternative simplified syntax for parsers written using standard unit `Ostap` (see Section 6.12). The syntax for syntax definition expressions is shown on Fig. 3.2.

syntaxExpression	:	syntax (*syntaxSeq* (| *syntaxSeq*)*)
syntaxSeq	:	*syntaxBinding*$^+$ [{ *expression* }]
syntaxBinding	:	[-] [*pattern* =] *syntaxPostfix*
syntaxPostfix	:	*syntaxPrimary* [* | + | ?]
syntaxPrimary	:	LIDENT ([[*expression* (, *expression*)*]])* |
		(*syntaxExpression*) |
		$(*expression*)

Figure 3.2: Syntax definition expressions

Syntax expressions can be used wherever regular expressions are allowed. Each syntax expressions is expanded in a certain combination of `Ostap` primitives. For example,

```
fun sum (str) {
  parseString (
    syntax (l=DECIMAL token["+"] r=DECIMAL eof {
              stringInt (l) + stringInt (r)
            }),
    str
  )
}
```

defines a function which parses its arguments into an expression "`l + r`", where `l` and `r` are decimal literals, and evaluates its value.

A syntax expression itself is a sequence of alternatives, and each alternative is a sequential composition (*syntaxSeq*) of primitive parsers equipped with optional semantic action (a *general* expression in curly brackets).

A primitive parser is either an l-indentfier (possibly supplied with arguments), or a *general* expression, surrounded by brackets `$( .. )`, or a *syntax* expression, surrounded by round brackets. Note, the arguments for primitive parsers in syntax expressions are surrounded by `[ .. ]` unlike general expressions; thus

```
x ("a")
```

means a sequential composition of `x` and "`a`", not a combinator `x` applied to "`a`".

A primitive parser can be followed by one of postfix operators ("`*`", "`+`", or "`?`"), corresponding to "`rep0`", "`rep`", or "`opt`" combinators of `Ostap` respectively, for example

```
token["a"]+
identifier?
```

A value recognized by a primitive parser can be matched against a pattern, for example

```
value=(identifier | constant)
h:tl=item+
```

The bindings provided by pattern-matching can be used in semantic actions.

Finally, if no semantic action is given, a sequential syntax expression returns a tuple of its components. However, if a parser in a sequential composition is preceded by "`-`" then its value is not included into the default result. Thus,

```
parse —eof
```

returns what "`parse`" recognized; the input stream is parsed against "`eof`", but the result of "`eof`" is omitted.

Chapter 4

Driver Options and Separate Compilation

Driver is a command-line utility "`lamac`" which controls the invocation of the compiler. The general format of invocation is

`lamac` *options filename*

Only one file name can be processed at once, the file name and the options can be specified in arbitrary order.

The driver operates in a few modes:

- Interpreter mode. Performs an interpretation of a source program using the reference source-level interpreter ("`-i`") or compiles and runs a source on the stack machine ("`-s`"). In this mode separate compilation is not supported, thus no external units can be accessed (including "`Std`"), only the standard set of builtins is available.

- Native mode, compilation ("`-c`"). Compiles a source file into native code and writes an object file. All referenced external unit interfaces must have to be accessible; however no linking is performed and no executable is built.

- Native mode, build (default). Same as for the native compilation, but additionally performs linking with the runtime library and all external units object files, generating executable.

In the native modes, the driver also creates import files ("`.i`") which are required for external units import to work properly. These files has to reside in the same directory as object files for corresponding units.

Each natively compiled object file implicitly references all imported units; the top-level expression of each unit is compiled into *unit initialization procedure*, which calls unit initialization procedures of all imported units in the same

order these unites were imported. It is guaranteed that unit initialization procedure for each unit will be called only once (regardless of the imports' shape for the whole application).

Additionally, the following options can be given to the driver:

- "`-o` *filename*" — specifies an alternative file name for the executable.
- "`-I` *path*" — specifies a path to look for external units. Multiples instances of this option can be given in driver's invocation, and the paths are looked up in that order.
- "`-dp`" — forces the driver to dump the AST of compiled unit in HTML representation. The dump is written in the file with the same basename as the source one, with the extension replaced with "`.html`".
- "`-ds`" — forces the driver to sump stack machine code. The option is only in effect in stack interpreter on native mode. The dump is written in the file "`.sm`".
- "`-g`" — compile with debug information (see Section 5).
- "`-v`" — makes the driver to print the version of the compiler.
- "`-h`" — makes the driver to print the help on the options.

Apart from the paths specified by the "`-I`" option the driver uses the environment variable "`LAMA`" to locate the runtime and standard libraries (see Section 6). Thus, the units from standard libraries are accessible without any "`-I`" option given.

Chapter 5

Debugging Support

Current implementation supports a minimalistic debugging with GDB [1]. In order to include the debug information into object files/executable these files have to be compiled with the command-line option "`-g`" (see Section 4).

The following debugging features are supported:

- setting breakpoints on lines; note, the line number information is generated only for identifiers, so, if a line does not contain even a single identifier, it will not be visible for the debugger;
- setting breakpoints on functions:
  - by source name for top-level function;
  - by internal name for nested functions or lambdas; an internal name can be found in stack machine program dump (option "`-ds`", see Section 4);
- stepping over/into;
- inspecting the values of global variables by their source names;
- inspecting the values of function arguments and local variables (include those in nested scopes) by their source names;
- inspecting the values in closures by their indices; the indices for closure elements can be found in stack machine program dump (option "`-ds`", see Section 4).

In addition a number of customized debugging command definitions is provided to make the debugging easier. These definitions reside in the "`gdb/.gdbinit`" file of the distribution; to make effect either the whole file has to be put in a proper place (usually the root of the home directory), or its content has to be imported into an existing GDB profile; consult GDB documentation for details.

The following customized commands are available:

- "`pp` *e*", where "*e*" is a GDB expression. The commands prints in a human-readable form the value of the expression. For example, "`pp x`" prints a value of a variable/parameter "`x`".
- "`pc` *i*", where "*i*" is an integer number. The commands prints a value of *i*-component of current closure.

Chapter 6

Standard Library

The standard library is comprised of the runtime for the language and a set of pre-shipped units written in $\lambda^a\mathcal{M}^a$ itself.

6.1 Unit `Std`

The unit "`Std`" provides the interface for the runtime of the language. The implementation of entities, defined in "`Std`", resides in the runtime itself. The import of "`Std`" is added implicitly by the compiler and can not be specified by an end user.

The following declarations are accessible:

fun `stringInt (s)`
: Converts a string representation of a signed decimal number into integer.

fun `read ()`
: Reads an integer value from the standard input, printing a prompt "`>`".

fun `write (int)`
: Writes an integer value to the standard output.

`sysargs`
: A variable which holds an array of command-line arguments of the application (including the name of the executable itself).

fun `makeArray (size)`
: Creates a fresh array of a given length. The elements of the array are left uninitialized.

fun `makeString (size)`
: Creates a fresh string of a given length. The elements of the string are left uninitialized.

fun `stringcat (list)`
: Takes a list of strings and returns the concatenates all its elements.

fun `matchSubString (subj, patt, pos)`

Takes two strings "subj" and "patt" and integer position "pos" and checks if a substring of "subj" starting at position "pos" is equal to "patt"; returns integer value, treated as a boolean.

fun sprintf (fmt, ...)
Takes a format string (as per GNU C Library [5]) and a variable number of arguments and returns a string, acquired via processing these arguments according to the format string. Note: indexed arguments are not supported.

fun substring (str, pos, len)
Takes a string, an integer position and length, and returns a substring of requested length of given string starting from given position. Raises an error if the original string is shorter then pos+len−1.

infix ++ **at** + (str1, str2)
String concatenation infix operator.

fun clone (value)
Performs a shallow cloning of the argument value.

fun hash (value)
Returns integer hash for the argument value; also works for cyclic data structures.

fun tagHash (s)
Returns an integer value for a hash of tag, represented by string s.

fun compare (value1, value2)
Performs a structural deep comparison of two values. Determines a linear order relation for every pairs of values. Returns 0 if the values are structurally equal, negative or positive integers otherwise. May not work for cyclic data structures.

fun flatCompare (x, y)
Performs a shallow comparison of two values. The result is similar to that for compare.

fun fst (value)
Returns the first subvalue for a given boxed value.

fun snd (value)
Returns the second subvalue for a given boxed value.

fun hd (value)
Returns the head of a given list.

fun tl (value)
Return the tail of a given list.

fun readLine ()
Reads a line from the standard input and returns it as a string. Return "0" if end of standard input was encountered.

fun printf (fmt, ...)

Takes a format string (as per GNU C Library [5] and a variable number of arguments and prints these arguments on the standard output, according to the format string.

fun fopen (fname, mode)
Opens a file of given name in a given mode. Both arguments are strings, the return value is an external pointer to file structure.

fun fclose (file)
Closes a file. The file argument should be that acquired by "fopen" function.

fun fread (fname)
Reads a file content and returns it as a string. The argument is a file name as a string, the file is automatically open and closed within the call.

fun fwrite (fname, contents)
Writes a file. The arguments are file name and the contents to write as strings. The file is automatically created and closed within the call.

fun fprintf (file, fmt, ...)
Same as "printf", but outputs to a given file. The file argument should be that acquired by fopen function.

fun regexp (str)
Compiles a string representation of a regular expression (as per GNULib's regexp [6]) into an internal representation. The return value is a external pointer to the internal representation.

fun regexpMatch (pattern, subj, pos)
Matches a string "subj", starting from the position "pos", against a pattern "pattern". The pattern is an external pointer to a compiled representation, returned by the function "regexp". The return value is the number of matched characters.

fun failure (fmt, ...)
Takes a format string (as per GNU C Library [5], and a variable number of parameters, prints these parameters according to the format string on the standard error and exits. Note: indexed arguments are not supported.)

fun system (cmd)
Executes a command in a shell. The argument is a string representing a command.

fun getEnv (name)
Returns a value for an environment variable "name". The argument is a string, the return value is either "0" (if not environment variable with given name is set), or a string value.

fun random (n)
Returns a pseudo-random number in the interval $0..n-1$. The seed is auto-initialized by current time at program start time.

fun time ()
Returns the elapsed time from program start in microseconds.

6.2 Unit `Data`

Generic data manupulation.

infix =?= **at** < (x, y)
: A generic comparison operator similar to `compare`, but capable of handling cyclic/shared data structures.

infix === **at** == (x, y)
: A generic equality operator capable of handling cyclic/shared data structures.

6.3 Unit `Timer`

A simple timer.

fun timer ()
: Creates a timer. Creates a zero-argument function which, being called, returns the elapsed time in microseconds since its creation.

fun toSeconds (n)
: Converts an integer value, interpreted as microseconds, into a floating-point string.

6.4 Unit `Random`

Random data structures generation functions.

fun randomInt ()
: Generates a random representable integer value.

fun randomString (len)
: Generates a random string of printable ASCII characters of given length.

fun randomArray (f, n)
: Generates a random array of *deep* size `n`. The length of the array is chosen randomly, and `f` is intended to be an element-generating function which takes the size of the element as an argument.

fun split (n, k)
: Splits a non-negative integer `n` in `k` random summands. Returns an array if length `k`. `k` has to be non-negative.

fun structure (n, nodeSpec, leaf)
: Generates a random tree-shaped data structure of size `n`. `nodeSpec` is an array of pairs $[k,f_k]$, where k is a non-negative integer and f_k is a function which takes an array of length k as its argument. Each pair describes a generator of a certain kind of interior node with degree k. `leaf` is a zero-argument function which generates the leaves of the tree. For example, the following code

```
structure (100,
           [[2, fun ([x, y]) {Add (x, y)}],
            [2, fun ([x, y]) {Sub (x, y)}]],
           fun () {Const (randomInt ())}
          )
```

can be used to generate a random arithmetic expression of size 100.

6.5 Unit `Array`

Array processing functions:

fun `initArray (n, f)`
Takes an integer value "`n`" and a function "`f`" and creates an array

`[f (0), f (1), ..., f (n−1)]`

fun `mapArray (f, a)`
Maps a function "`f`" over an array "`a`" and returns a new array.

fun `arrayList (a)`
Converts an array to list (preserving the order of elements).

fun `listArray (l)`
Converts a list to array (preserving the order of elements).

fun `foldlArray (f, acc, a)`
Folds an array "`a`" with a function "`f`" and initial value "`acc`" in a left-to-right manner. The function "`f`" takes two arguments — an accumulator and an array element.

fun `foldrArray (f, acc, a)`
Folds an array "`a`" with a function "`f`" and initial value "`acc`" in a right-to-left manner. The function "`f`" takes two arguments — an accumulator and an array element.

fun `iterArray (f, a)`
Applies a function "`f`" to each element of an array "`a`"; does not return a value.

fun `iteriArray (f, a)`
Applies a function "`f`" to each element of an array "`a`" and its index (index first); does not return a value.

6.6 Unit `Collection`

Collections, implemented as AVL-trees. Four types of collections are provided: sets of ordered elements, maps of ordered keys to other values, memo tables and hash tables. For sets and maps the generic "`compare`" function from the unit "`Std`" is used as ordering relation. For memo table and hash tables the comparison of generic hash values, delivered by function "`hash`" of unit "`Std`" is used.

6.6.1 Maps

Maps are immutable structures with the following interface:

fun `emptyMap (f)`
Creates an empty map. An argument is a comparison function, which returns zero, positive or negative integer values depending on the order of its arguments.

fun `compareOf (m)`
Returns a comparison function, associated with the map given as an argument.

fun `addMap (m, k, v)`
Adds a binding of a key "`k`" to a value "`v`" into a map "`m`". As a result, a new map is returned.

fun `findMap (m, k)`
Finds a binding for a key "`k`" in the map "`m`". Returns the value "`None`" if no binding is found, and "`Some (v)`" if "`k`" is bound to "`v`".

fun `removeMap (m, k)`
Removes the binding for "`k`" from the map "`m`" and returns a new map. This function restores a value which was previously bound to "`k`".

fun `bindings (m)`
Returns all bindings for the map "`m`" as a list of key-value pairs, in key-ascending order.

fun `listMap (l)`
Converts a list of key-value pairs into a map.

fun `iterMap (f, m)`
Iterates a function "`f`" over the bindings of map "`m`". The function takes two arguments (key and value). The bindings are enumerated in an ascending order.

fun `mapMap (f, m)`
Maps a function "`f`" over all values of "`m`" and returns a new map of results.

fun `foldMap (f, acc, m)`
Folds a map "`m`" using a function "`f`" and initial value "`acc`". The function takes an accumulator and a pair key-value. The bindings are enumerated in an ascending order.

6.6.2 Sets

Sets are immutable structures with the following interface:

fun `emptySet (f)`
Creates an empty set. An argument is a comparison function, which returns zero, positive or negative integer values depending on the order of its arguments.

fun `compareOf (m)`
Returns a comparison function, associated with the set given as an argument.

fun `addSet (s, v)`
Adds an element "`v`" into a set "`s`" and returns a new set.

fun `memSet (s, v)`
Tests if an element "`v`" is contained in the set "`s`". Returns zero if there is no such element and non-zero otherwise.

fun `removeSet (s, v)`
Removes an element "`v`" from the set "`s`" and returns a new set.

fun `elements (s)`
Returns a list of elements of a given set in ascending order.

fun `union (a, b)`
Returns a union of given sets as a new set.

fun `diff (a, b)`
Returns a difference between sets "`a`" and "`b`" (a set of those elements of "`a`" which are not in "`b`") as a new set.

fun `listSet (l)`
Converts a list into a set.

fun `iterSet (f, s)`
Applied a function "`f`" to each element of the set "`s`". The elements are enumerated in ascending order.

fun `mapSet (f, s)`
Applies a function "`f`" to each element of the set "`s`" and returns a new set of images. The elements are enumerated in an ascending order.

fun `foldSet (f, acc, s)`
Folds a set "`s`" using the function "`f`" and initial value "`acc`". The function "`f`" takes two arguments — an accumulator and an element of the set. The elements of set are enumerated in an ascending order.

6.6.3 Memoization Tables

Memoization tables can be used for *hash-consing* [4] — a data transformation which converts structurally equal data structures into physically equal. Memoization tables are mutable; they do not work for cyclic data structures.

fun `emptyMemo ()`
Creates an empty memo table.

fun `lookupMemo (m, v)`
Lookups a value "`v`" in a memo table "`m`", performing hash-consing and returning a hash-consed value.

6.6.4 Hash Tables

Hash table is an immutable map which uses hashes as keys and lists of key-value pairs as values. For hashing a generic hash function is used, the search within the same hash class is linear with physical equality "==" used for comparison.

fun `emptyHashTab (n, h, c)`
Creates an empty hash table. Argument are: a number of classes, hash and comparison functions.

fun `compareOf (m)`
Returns a comparison function, associated with the hash table given as an argument.

fun `hashOf (m)`
Returns a hash function, associated with the hash table given as an argument.

fun `addHashTab (t, k, v)`
Adds a binding of "`k`" to "`v`" to the hash table "`t`" and returns a new hash table.

fun `findHashTab (t, k)`
Searches for a binding for a key "`k`" in the table "`t`". Returns "`None`" if no binding is found and "`Some (v)`" otherwise, where "`v`" is a bound value.

fun `removeHashTab (t, k)`
Removes a binding for the key "`k`" from hash table "`t`" and returns a new hash table. The previous binding for "`k`" (if any) is restored.

6.7 Unit `Fun`

The unit defines some generic functional stuff:

fun `id (x)`
The identify function.

infixl `$` **after** `:= (f, x)`
Left-associative infix for function application.

infix `#` **after** `∗ (f, g)`
Non-associative infix for functional composition.

fun `fix (f)`
Fixpoint combinator. The argument is a knot-accepting function, thus a factorial can be defined as

```
fix (fun (f) {
       fun (n) {
         if n == 1 then 1 else n * f (n−1) fi
       }
     })
```

6.8 Unit Lazy

The unit provides primitives for lazy evaluation.

fun makeLazy (f)
Creates a lazy value from a function "f". The function must not require any arguments.

fun force (f)
Returns a suspended value, forcing its evaluation if needed.

6.9 Unit List

The unit provides some list-manipulation functions. None of the functions mutate their arguments.

fun singleton (x)
Returns a one-element list with the value "x" as its head.

fun size (l)
Returns the length of the list.

fun foldl (f, acc, l)
Folds a list "l" with a function "f" and initial value "acc" is the left-to-right manner. The function "f" takes two arguments — an accumulator and a list element.

fun foldr (f, acc, l)
Folds a list "l" with a function "f" and initial value "acc" is the right-to-left manner. The function "f" takes two arguments — an accumulator and a list element.

fun iter (f, l)
Applies a function "f" to the elements of the list "l" in the giver order.

fun map (f, l)
Maps a function "f" to the elements of the list "l" and returns a fresh list if images in the same order.

infix +++ **at** + (x, y)
Returns the concatenation of lists "x" and "y".

fun reverse (l)
Reverses a list.

fun assoc (l, x)
Finds a value for a key "x" in an associative list "l". Returns "None" if no value is found and "Some (v)" otherwise, where "v" — the first value whise key equals "x". Uses generic comparison to compare keys.

fun find (f, l)
Finds a value in a list "l" which satisfies the predicate "f". The predicate must return integer value, treated as boolean. Returns "None" if no element satisfies "f" and "Some (v)" otherwise, where "v" — the first value to satisfy "f".

fun `flatten (l)`
Flattens an arbitrary nesting of lists into a regular list. The order of elements is preserved in both senses.

fun `zip (a, b)`
Zips a pair of lists into the list of pairs. Does not work for lists of different lengths.

fun `unzip (a)`
Splits a list of pairs into pairs of lists.

fun `remove (f, l)`
Removes the first value, satisfying the predicate "`f`", from the list "`l`". The function "`f`" should return integers, treated as booleans.

fun `filter (f, l)`
Removes all values, not satisfying the predicate "`f`", from the list "`l`". The function "`f`" should return integers, treated as booleans.

6.10 Unit `Buffer`

Mutable buffers.

fun `emptyBuffer ()`
Creates an empty buffer.

fun `singletonBuffer (x)`
Creates a buffer from a single element.

fun `listBuffer (x)`
Creates a buffer from a list.

fun `getBuffer (buf)`
Gets the contents of a buffer as a list.

fun `addBuffer (buf, x)`
Adds an element `x` to the end of buffer `buf` and returns the updated buffer. The buffer `buf` can be updated in-place.

fun `concatBuffer (buf, x)`
Adds buffer `x` to the end of buffer `buf` and returns the updated buffer. The buffer `buf` can be updated in-place.

infixl <+> **before** + (`b1`, `b2`)
Infix synonym for `concatBuffer`.

infix <+ **at** <+> (`b`, `x`)
Infix synonym for `addBuffer`.

6.11 Unit `Matcher`

The unit provides some primitives for matching strings against regular patterns. Matchers are immutable structures which store string buffers with

current positions. Matchers are designed to be used as stream representation for parsers written using combinators of "Ostap"; in particular, return values for "endOf", "matchString" and "matchRegexp" respect the conventions for such parsers.

fun createRegexp (r, name)
Creates an internal representation of regular expression; argument "r" is a string representation of regular expression (as per function "regexp"), "name" — a string name for diagnostic purposes.

fun initMatcher (buf)
Takes a string argument and returns a fresh matcher.

fun showMatcher (m)
Returns a printable representation for a matcher "m" (for debugging purposes).

fun endOfMatcher (m)
Tests if the matcher "m" reached the end of string. Return value represents parsing result as per "Ostap".

fun matchString (m, s)
Tests if a matcher "m" at current position matches the string "s". Return value represents parsing result as per "Ostap".

fun matchRegexp (m, r)
Tests if a matcher "m" at current position matches the regular expression "r", which has to be constructed using the function "createRegexp". Return value represents parsing result as per "Ostap".

fun getLine (m)
Gets a line number for the current position of matcher "m".

fun getCol (m)
Gets a column number for the current position of matcher "m".

6.12 Unit Ostap

Unit "Ostap" implements monadic parser combinators in continuation-passing style with memoization [7, 9, 8]. A parser is a function of the shape

```
fun (k) {
  fun (s) {...}
}
```

where "k" — a *continuation*, "s" — an input stream. A parser returns either "Succ (v, s)", where "v" — some value, representing the result of parsing, "s" — residual input stream, or "Fail (err, line, col)", where "err" — a string, describing a parser error, "line", "col" — line and column at which the error was encountered.

The unit describes some primitive parsers and combinators which allow to construct new parsers from existing ones.

fun initOstap ()

Clears and initializes the internal memoization tables. Called implicitly at unit initiliation time.

fun `memo (f)`
Takes a parser "`a`" and returns its memoized version. Needed for some parsers (for expamle, left-recursive ones).

fun `token (x)`
Takes a string or a representation of regular expression, returned by "`createRegexp`" (see unit `Matcher`), and returns a parser which recognizes exactly this string/regular expression.

fun `eof (k)`
A parser which recognizes the end of stream.

fun `empty (k)`
A parser which recognizes empty string.

fun `loc (k)`
A parser which returns the current position (a pair "`[line, col]`") in a stream.

fun `alt (a, b)`
A parser combinator which constructs a parser alternating between "`a`" and "`b`".

fun `seq (a, b)`
A parser combinator which construct a sequential composition of "`a`" and "`b`". While "`a`" is a requlal parser, "`b`" is a *function* which takes the result of parsing by "`a`" and returns a parser (*monadicity*).

infixr | **before** !! `(a, b)`
Infix synonym for "`alt`".

infixr |> **after** | `(a, b)`
Infix synonym for "`seq`".

infix `@` **at** `*` `(a, f)`
An operation which attaches a semantics action "`f`" to a parser "`a`". Returns a parser which behaves exactly as "`a`", but additionally applies "`f`" to the result if the parsing is succesfull.

fun `lift (f)`
Lifts "`f`" into a function which ignores its argument.

fun `bypass (f)`
Convert "`f`" into a function which parser with "`f`" but returns its argument. Literally, "`bypass (f) =` **fun** `(x) f @ lift (x)`"

fun `opt (a)`
For a parser "`a`" returns a parser which parser either "`a`" of empty string.

fun `rep0 (a)`
For a parser "`a`" returns a parser which parser a zero or more repetitions of "`a`"

fun `rep (a)`

For a parser "a" returns a parser which parser a one or more repetitions of "a"

fun listBy (item, sep)
Constructs a parser which parses a non-empty list of "item" delimited by "sep".

fun list0By (item, sep)
Constructs a parser which parses a possibly empty list of "item" delimited by "sep".

fun list (item)
Constructs a parser which parses a non-empty list of "item" delimited by ",".

fun list0 (item)
Constructs a parser which parses a possibly empty list of "item" delimited by ",".

fun parse (p, m)
Parsers a matcher "m" with a parser "p". Returns ether "Succ (v)" where "v" — a parsed value, or "Fail (err, line, col)", where "err" — a stirng describing parse error, "line", "col" — this error's coordinates. This function may fail if detects the ambiguity of parsing.

fun parseString (p, s)
Parsers a string "s" with a parser "p". Returns ether "Succ (v)" where "v" — a parsed value, or "Fail (err, line, col)", where "err" — a stirng describing parse error, "line", "col" — this error's coordinates. This function may fail if detects the ambiguity of parsing.

fun expr (ops, opnd)
A super-combinator to generate infix expression parsers. The argument "opnd" parses primary operand, "ops" is a list of infix operator descriptors. Each element of the list describes one *precedence level* with precedence increasing from head to tail. A descriptor on each level is a pair, where the first element describes the associativity at the given level ("Left", "Right" or "None") and the second element is a list of pairs — a parser for an infix operator and the semantics action (a three-argument function accepting the left parser operand, that that infix operator parser returns, and the right operand). For example,

```
{[Left, {[token ("+"), fun (l, op, r) {Add (l, r)}],
         [token ("-"), fun (l, op, r) {Sub (l, r)}]
        }],
 [Left, {[token ("*"), fun (l, op, r) {Mul (l, r)}],
         [token ("/"), fun (l, op, r) {Div (l, r)}]
        }]}
```

specifies two levels of precedence, both left-associative, with infix operators "+" and "−" at the first level and "∗" and "/" at the second. The semantics for these operators constructs abstract syntax trees (in this particular example the second argument of semantics functions is unused).

6.13 Unit `Ref`

The unit provides an emulation for first-class references.

fun `ref (x)`
Creates a mutable reference with the contents ″`x`″.

fun `deref (x)`
Dereferences a reference ″`x`″ and returns stored value.

infix ::= **before** := `(x, y)`
Assigns a value ″`y`″ to a cell designated by the ″`x`″. Returns ″`y`″.

Bibliography

[1] *Debugging with GDB*. URL: https://sourceware.org/gdb/current/onlinedocs/gdb/.

[2] *Haskell Language*. URL: https://www.haskell.org.

[3] *OCaml Language*. URL: https://www.ocaml.org.

[4] Jean-Christophe Filliundefinedtre and Sylvain Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, ML 06, page 1219, New York, NY, USA, 2006. Association for Computing Machinery. URL: https://doi.org/10.1145/1159876.1159880, doi:10.1145/1159876.1159880.

[5] Free Software Foundation. *The GNU C Library*. URL: https://www.gnu.org/software/libc/manual.

[6] Free Software Foundation. *The GNU Portability Library*. URL: https://www.gnu.org/software/gnulib/manual.

[7] Graham Hutton and Erik Meijer. Monadic parser combinators, 1996.

[8] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 16, page 112, New York, NY, USA, 2016. Association for Computing Machinery. URL: https://doi.org/10.1145/2847538.2847539, doi:10.1145/2847538.2847539.

[9] Mark Johnson. Memoization in top-down parsing. *Comput. Linguist.*, 21(3):405417, September 1995.

[10] A. van Wijngaarden. *Report on the Algorithmic Language ALGOL 68*. Printing by the Mathematisch Centrum, 1969.