Last updated: 2025-07-04 19:18:00-07:00

1 Introduction

This manual describes the LAMP programming language, which is a simple language we'll be designing and implementing in CS 162. The name "lamp" is short for "lambda calculus plus a bunch of other stuff".

2 Abstract Syntax

The abstract syntax of LAMP expressions is described by the following AST:

$$
\begin{array}{rcll}
e \in expr & ::= & n & \text{natural numbers} \\
& \mid & expr + expr & \text{addition} \\
& \mid & expr - expr & \text{subtraction} \\
& \mid & expr * expr & \text{multiplication} \\
& \mid & expr == expr & \text{equality (between nats only)} \\
& \mid & b & \text{booleans} \\
& \mid & \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 & \text{if-then-else} \\
& \mid & x & \text{variables} \\
& \mid & \lambda x.\ e & \text{lambda function} \\
& \mid & e_1\ e_2 & \text{application} \\
& \mid & \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 & \text{let-binding} \\
& \mid & X & \text{abbreviations (global variables)} \\
n & \in & \mathsf{Nat} & \\
b & \in & \mathsf{Bool} = \{\mathsf{True}, \mathsf{False}\} &
\end{array}
$$

3 Concrete Syntax

3.1 Associativity and Precedence

The operator associativity for LAMP is given by the following table. Higher up = higher precedence. For example, the

Operation	Associativity
Application	Left
$*$	Left
$+, -$	Left
$==$	Non-associative

Table 1: Caption

expression $f\ x\ y$ will be parsed as $(f\ x)\ y$, and $f\ 1 + 2 * 3$ will be parsed as $(f\ 1) + (2 * 3)$. Just remember that function applications always bind more strongly than *anything else*.

3.2 Identifiers

As shown in the abstract syntax, LAMP has two kinds of identifiers:

1. A *variable* is like your usual, lexically-scoped variable. In the concrete syntax, the name of the variable must start with a lower case letter, followed by any number of digits, lower or upper case letters, or underscores "_". A variable can also be a single underscore.

2. An *abbreviation* must start with a capital letter, followed by any number of digits, lower- or upper-case letters, or underscores "_". The meaning of abbreviations is explained in the next subsection.

3.3 Lambdas

Since the greek letter λ is difficult to type in a text editor, we'll use the backslack symbol "\" in place of "λ" when writing LAMP program in concrete syntax. For example, the lambda function $\lambda x.\ \lambda y.\ x + y$ will be written as:

```
\x. \y. x + y
```

3.4 Lamp Programs

A LAMP program is a (possibly empty) list of abbreviations followed by a main expression. An abbreviation has the form

```
def X = <e>
```

where X is the name of the abbreviation and e is an expression. Abbreviations are implicitly unfolded during evaluation, and can be (mutually) recursive.

The main expression should appear after all abbreviations and must have the form:

```
main = <e>
```

where `main` is a reserved keyword and e is an expression.

For example, the following is a valid LAMP program:

```
def Fib = \n.
    if n == 0 then 0
    else if n == 1 then 1
    else Fib (n-1) + Fib (n-2)

main = Fib 10
```

4 Operational Semantics

We define *values* using the following grammar:

$$
\begin{array}{rcll}
v \in \mathit{value} & ::= & n \in \mathsf{Nat} & \text{natural numbers} \\
& | & b \in \mathsf{Bool} & \text{booleans} \\
& | & \lambda x.\ e & \text{lambda function}
\end{array}
$$

We assume that Δ holds a list of all abbrevations defined in a LAMP program. We use $\Delta(X) = e$ denote that looking up the abbrevation X returns the expression e.

The operational semantics is given by the judgment "$e \Rightarrow v$", defined by the following rules:

$$
\frac{}{n \Rightarrow n}\ \textsc{Nat} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (\oplus \in \{+, -, *\}) \quad (n_1 \oplus n_2 = n_3)}{e_1 \oplus e_2 \Rightarrow n_3}\ \textsc{Arith}
$$

$$
\frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 = n_2)}{e_1 == e_2 \Rightarrow \mathsf{True}}\ \textsc{EqTrue} \qquad \frac{e_1 \Rightarrow n_1 \quad e_2 \Rightarrow n_2 \quad (n_1 \neq n_2)}{e_1 == e_2 \Rightarrow \mathsf{False}}\ \textsc{EqFalse}
$$

$$
\frac{}{b \Rightarrow b}\ \textsc{Bool} \qquad \frac{e_1 \Rightarrow \mathsf{True} \quad e_2 \Rightarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v}\ \textsc{IfTrue} \qquad \frac{e_1 \Rightarrow \mathsf{False} \quad e_3 \Rightarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v}\ \textsc{IfFalse}
$$

$$
\text{(No rule for variables)} \qquad \frac{(\Delta(X) = e) \quad e \Rightarrow v}{X \Rightarrow v}\ \textsc{Abbrev}
$$

$$
\frac{}{\lambda x.\ e \Rightarrow \lambda x.\ e}\ \text{Lambda} \qquad \frac{e_1 \Rightarrow \lambda x.\ e \quad e_2 \Rightarrow v \quad ([v/x]e = e') \quad e' \Rightarrow v'}{e_1\ e_2 \Rightarrow}\ \text{App}
$$

Several rules deserve special attention:

- In the Arith rule, if the operator is $-$, then we perform *truncated subtraction*: given two natural numbers n_1 and n_2, $n_1 - n_2$ should return the usual difference if $n_1 \geq n_2$, and 0 otherwise. This is to ensure that we always stay within the range of natural numbers, which cannot be negative. For example, $3 - 4$ should evaluate to 0.
- The Abbrev rule evaluates an abbreviation by looking up its definition in Δ. Note that if Δ did not contain a definition for X, then the side condition $\Delta(X) = e$ would be violated, so the rule would not apply, making the evaluation stuck.