The L Programming Language Reference Manual

CS 345H
UT Austin

(2015/08/24 at 13:45:37)

1 Introduction

This manual describes the L programming language, which is a simple, functional language small enough that a complete interpreter for L can be implemented in a semester-long course. The programming language is similar in nature to the untyped λ-calculus, but extends the λ-calculus with constructs such as `let` bindings and named functions to make it more convenient to program in L. The language is also very similar to real-world functional programming languages, such as Lisp.

This manual gives an informal overview of the language, describes its syntax, and gives precise semantics to the language. At the beginning of the semester, students should only focus on the informal discussion of L (Sections 2 and 3). Section 5 formally describing L's syntax should be studied as we discuss lexing, context-free grammars, and parsing in lecture. Similarly, students should only read Section 6 after we discuss operational semantics in lecture.

2 Informal Description of L

An L program is simply an expression, and executing the program is equivalent to evaluating the expression. For example, the simple expression

```
3
```

is a valid L program, and the value of this program is the integer 3.

The most basic expressions in L are integer constants and string constants, such as "cs345". More complicated integer expressions can be formed by using binary arithmetic operators, such as `+`, `*`, `-`, and `/`. For example,

```
(3+6-1)*2
```

is a valid L expression with value 16.

Strings in the L language can also be concatenated using the `+` operator. For example, the L expression

```
"cs" + "345"
```

evaluates to the string constant ``` ``cs345'' ```.

Conditional expressions in L are of the form:

```
if e1 then e2 else e3
```

If the expression `e1` is non-zero, the value of the whole `if` expression is the value of expression `e2`, otherwise it is the value of `e3`. For example,

```
if 5-3-2 then "yes" else "no"
```

evaluates to the string constant ``` ``no'' ``` since the expression `5-3-2` evaluates to 0. To allow more interesting `if` expressions, L also allows comparing two expressions using the logical operators `=, <> <, <=, >, >=, &, |`. The equals (`=`) and not-equals (`<>`) operator can be applied to both string and integer expressions. The expression `e1 = e2` evaluates to 1 if `e1` and `e2` have the same value, and to 0 otherwise. Similarly, the expression `e1 <> e2` evaluates to 1 if `e1` and `e2` evaluate to different values and to 0 otherwise. As an example, the expression

```
1+3 = 2+2
```

evaluates to 1, since the result of evaluating `1+3` and `2+2` yield the same value. Similarly, the expression

```
"cs" + "345" <> "cs345"
```

evaluates to 0, since ``` ``cs'' + ``345'' ``` evaluates to ``` ``cs345'' ```.

The comparison operators `<, <=, >` and `>=` only apply to integer expressions. The expression `e1 op e2` where `op` is `<, <=, >` or `>=` evaluates to 1 if `e1` is less than, less than or equal to, greater than, or greater than or equal to `e2` respectively, and to 0 otherwise. For example,

```
3+5 < 7
```

evaluates to 0, and the expression

```
"cs243" < "cs345"
```

yields the run-time error

```
Run-time error in expression ("cs243" < "cs345")
Binop < cannot be applied to strings
```

since the operator `<` cannot be applied to string expressions.

The only other logical operators in L are `&` and `|`, which also only apply to integer expressions. The expression `e1 & e2` evaluates to 1 if `e1` and `e2` both evaluate to non-zero integers, and to 0 otherwise. Similarly, `e1 | e2` evaluates to 1 if either `e1` or `e2` (or both) evaluate to a non-zero value, and to 0 otherwise. For example, the expression

```
if (3+4 < 10)  & ("cs"+"345" <> "cs310") then "yes" else 0
```

evaluates to ``‘‘yes’’`` since the expression `3+4 < 10` and `"cs"+"345" <> "cs310"` both evaluate to 1.

2.1 Let Bindings

Let bindings in L allow us to name and reuse expressions. Specifically, an expression of the form

```
let x = e1 in e2
```

binds the value of `e1` to identifier `x` and evaluates `e2` under this binding. The expression `e2` is referred to as the *body* of the let expression, and `e1` is called the *initializer*. The value of the `let` expression is the result of evaluating `e2`. For example,

```
let x = 3+5 in x-2
```

evaluates to 6, while the expression

```
let x = 3+5 in x+y
```

yields the run-time error:

```
Run-time error in expression y
Identifier y is not bound in current context
```

since the identifier `y` is not bound in the body of the let expression.

Let expressions in L can be arbitrarily nested. For example, consider the nested let expressions:

```
let x = 3+5 in
let y = 2*x in
y+x
```

This is a valid L expression and evaluates to 24. Observe that the body of the first let expression is `let y = 2*x in y+x` while the body of the second (nested) let expression is `y+x`. As another example, consider the nested let expression:

```
let x = let x = 3 in
x+1 in x
```

evaluates to 4. The initializer for the first (outer-level) let expression is `let x=3 in x+1`, which evaluates to 4. Thus, the value of `x` in the body of the outer let expression is 4. As a final example of let expressions, consider:

```
let x = 2 in
let x = 3 in
x
```

evaluates to 3, since each identifier refers to the most recently bound value.

2.2 Lambda Expressions and Applications

As in λ-calculus, L also provides lambda expressions of the form:

```
lambda x1, ..., xn. e
```

For example, the L expression

```
lambda x, y. x+y
```

corresponds to an unnamed function that takes two arguments `x` and `y` and evaluates their sum. The above L expression is equivalent to the L expression:

```
lambda x. lambda y. x+y
```

The transformation from the first lambda expression `lambda x, y.  x+y` to the second expression `lambda x.  lambda y.  x+y` is known as *currying.*

Of course, for lambda expressions to be useful, we also need to be able to apply arguments to lambda abstractions. Application in L corresponds to expressions of the form `(e1 e2 ...en)`. Specifically, if `e1` is an expression of the form `lambda x2, ...xn.  e3`, the expression `e1 e2 ...en` evaluates `e3` with `e2` bound to `x2`, `e3` bound to `x3` and so on. For example, the application expression

```
(lambda x, y. x+y 6 7)
```

evaluates to 13.

As a more interesting example, consider the application expression

```
(lambda x, y. x+y 6)
```

which evaluates to the lambda expression

```
lambda y. (6 + y)
```

This example illustrates an interesting feature of L: Expressions in L do not have to evaluate to constants; they can be *partially evaluated* functions, such as `lambda y.  (6 + y)` in this example.

Here, we highlight two possible mistakes one can make using application expressions in L. First of all, observe that the expression

```
(lambda x. x) 4
```

is not a syntactically valid L expression, since it is not correctly parenthesized, and will yield a *parse error*. The correct way of writing this expression is

```
(lambda x. x 4)
```

As a second caveat, the application expression

```
( (let x =2 in x) 3)
```

is a syntactically valid L expression but will yield the run-time error:

```
Run-time error in expression (let x = 2 in x 3)
Only lambda expressions can be applied to other expressions
```

The problem here is that the first expression `e1` in the application `(e1 e2)` must evaluate to a lambda expression. On the other hand, the following expression

```
let x = lambda y. y in
(x 3)
```

is both syntactically and semantically valid and evaluates to 3.

Interesting aspects of L arise from the interaction between let expressions and lambda expressions, the combination of which allows us to conveniently define recursive functions. As an example, consider the following valid L program:

```
let f = lambda n. if n=0 then 1 else n* (f (n-1)) in
f 4
```

The let expression binds variable `f` to a recursive function for computing factorial, and thus, the expression `f 4` in the body evaluates to 4!, i.e., 24.

Observe that, unlike λ-calculus, L allows writing "naturally recursive" expressions such as the factorial function above. Specifically, it is legal in L to refer to the name of a `let` bound function in the body of the let expression.

2.3 Function Definitions

In addition to `lambda` expressions, which correspond to anonymous function definitions, the L language also makes it possible to define named functions using the syntax:

```
fun f with x1, ... xn = e in e'
```

Here `f` is the name of the function being defined, `x1, ...xn` are the arguments of function `f`, and `e` is the body of function `f`. The value of the expression is the result of evaluating `e'` where `e'` may refer to function `f`.

Named function definitions of this form are in fact only "syntactic sugar" in L. That is, they merely provide a more convenient way to write expressions that can already be expressed using other constructs in the language. Specifically, the function definition

```
fun f with x1, ... xn = e in e'
```

is in fact identical to the following let expression:

```
let f = lambda x1, ... xn. e in e'
```

To illustrate how to use named function definitions, here is an alternative way of writing a program that computes the factorial of 4:

```
fun fact with n = if n=0 then 1 else n* (fact (n-1)) in
fact 4
```

Here is another example that illustrates the use of named functions in L:

```
fun even with x = if x=0 then 1 else (odd x-1) in
fun odd with x = if x=0 then 0 else (even x-1) in
(odd 7)
```

This L program evaluates to 1, since 7 is an odd number. Observe that functions `even` and `odd` are mutually recursive; that is, these two functions are defined in terms of each other.

2.4 Lists

In addition to integers and strings, the L language also supports lists. A list in L is a general data structure consisting of a *head* and *tail*. Head refers to the first element in the list, while the tail refers to the remaining elements in the list. For example, the head of the list `[1, 2, 3]` is the integer 1, and its tail is another list `[2, 3]`.

L supports the following list operations:

- `isNil`: The expression `isNil list` evaluates to 1 if `list` is an empty list, and 0 otherwise.
- `e1@e2`: The expression `e1@e2` evaluates to a new list with head corresponding to the value of `e1` and tail corresponding to the value of `e2`.
- `!e`: The expression `!e` yields the head of the list if `e` evaluates to a list, and the value `e` otherwise. For example, `!(2@3)` evaluates to 2, and `!''abc''` evaluates to `''abc''`.
- `#e`: The expression `#e` yields the tail of the list if `e` evaluates to a list, and `Nil` otherwise. For example, `#(2@3)` evaluates to 3, `#(1@2@3)` evaluates to `[2, 3]` and `#2` evaluates to `Nil`.

To further illustrate lists in L, we consider some examples:

```
fun length with list  =
if isNil list then 0 else (length #list)+1
in
(length 1@2@2@1)
```

Here, we define a function `length` to compute the number of elements in a list and use this function to determine the size of the list `[1, 2, 2, 1]`, which evaluates to 4.

Here is another list function to concatenate two lists:

```
fun cat with l1, l2 =
if isNil l1 then l2 else
!l1@(cat #l1 l2)
in
(cat 1@2@3@4 5@6)
```

This L program evaluates to the list `[1,2,3,4,5,6]`.

As a final example, here is a program that adds `n` to every element in a list:

```
fun add with l, n =
if isNil #l then l+n else
let hd = !l in
let tl = #l in
(hd+n)@(add tl n)
in
(add 1@2@3 2)
```

This program evaluates to the list `[3,4,5]`.

2.5 Input Output in L

L also provides basic operators for performing input/output:

- `print:` The expression `print e` prints the value of expression `e` on the console and evaluates to 0. For example,

  ```
  print ‘‘a’’+‘‘bc’’
  ```

  prints `‘‘abc’’` on the console and evaluates to 0; the expression

  ```
  print (lambda x,y. x+y 2)
  ```

  prints

  ```
   lambda y. (2 + y)
  ```

  and evaluates to 0. As a final example, the expression

  ```
  let x = print (lambda x,y. x+y 2) in x+1
  ```

prints `lambda y.(2 + y)` and evaluates to 1.

- `readInt`: The expression `readInt` reads the integer entered by the user and evaluates to this integer value. If the value entered by the user is not an integer, then the expression evaluates to 0.
- `readString`: The expression `readString` reads the string entered by the user and evaluates to this string constant. If the value entered by the user is not a string constant, then the expression evaluates to the empty string.

2.6 Comments in L

The L language also supports comments. Anything written inside `(* ...*)` corresponds to a comments. Comments in L may also be nested. For example, the following is a syntactically well-formed comment in L:

```
(*
Comment 1 (* Here is comment 2 *)
)*
```

3 Running L Programs

To run L programs, call the L interpreter with the file containing the L program. A binary version of the L interpreter is available at

```
/projects/cs345.tdillig/l-interpreter
```

on all UT Austin computer science machines. By convention, L programs end with the extension `.L`. For example, assume you have a `test.L` file with the following content:

```
(* A simple example in L *)
let x = 1 in
let y = 3 in
x+y
```

To interpret this file, you type:

```
/projects/cs345.tdillig/l-interpreter test.L
```

The output is as follows:

```
4
```

This means that the above program evaluates to value 4. You can also pass the option `-ast` to `l-interpreter` to print the abstract syntax tree of any expression. For this you simply run

```
/projects/cs345.tdillig/l-interpreter -ast test.L
```

This will also output the abstract syntax tree in addition to the result of the program. You may find this option useful when implementing your own lexer and parser. In this case, the output will be:

```
***************** AST *****************
Let x
VAL
    INT: 1
BODY
    Let y
    VAL
        INT: 3
    BODY
        BINOP: +
            x
            y

****************************************
4
```

4 Lexical structure

The lexical units of L are integers, identifiers, strings, keywords, operators and comments. Lexical units that require clarification are discussed below.

4.1 Integers

Integers are non-empty strings of digits 0-9. Observe that integers may contain leading 0s, but no leading `-`.

4.2 Identifiers

Identifiers always start with an alpha character and can be followed by any number of alpha-numeric characters. In L, alpha characters are the characters `a-z`, `A-Z` and the character `_`. For example `_12AbC` is a legal identifier, but `001D` is not. Numeric characters in L are the digits `0-9`.

4.3 Strings

Strings in L start and end with `"`. Strings may contain newlines and L does not expand any escape characters. For example, the following is a legal string in L:

```
"this is just
 a test"
```

Strings may be of any length in L.

4.4 Keywords

L has the following keywords:

`let, in, fun, with, lambda, if, then, else, print, readInt, readString, isNil, Nil`

All keywords in L are case-insensitive.

4.5 Comments

As discussed earlier, L supports nested comments. For example, the following is a valid comment in L:

```
(* this (* is (* a test*)
*)*)
```

5 L Syntax

The complete syntax of L is specified by the context-free grammar presented in Figure 1.

Observe that this grammar in Figure 1 is ambiguous, and we discuss the intended meaning of the ambiguous constructs. The first source of ambiguity in the grammar is binary operators. For example, the L expression `2*3+4` can be parsed in two ways: either as `(2*3)+4` or as `2*(3+4)`. To disambiguate the grammar, we therefore need to declare the precedence and associativity of operators. Figure 2 shows the precedence of operators, where operators higher up in the figure have higher precedence than those lower down in the figure. Operators shown on the same line have the same precedence.

To illustrate how precedence declarations allow us to resolve ambiguities, consider again the expression `2*3+4`. Since `*` has higher precedence than `+`, this means the expression should be parsed as `(2*3)+4`, instead of `2*(3+4)`. Similarly, since `!` has precedence than `@`, this means the expression `!x@y` should be understood as `(!x)@y` rather than `!(x@y)`.

Observe that precedence declarations are not sufficient to resolve all ambiguities concerning these operators; we also need associativity declarations. For example, precedence declarations alone are not sufficient to decide whether the expression `1+2+3` should be parsed as `(1+2)+3` or as `1+(2+3)`. To resolve this issue, we also need to specify the associativity of the binary operators.

In the L language, the binary operators `+, -, *, /, &, |, =, <>, <, >, <=, >=` are all left-associative; the only right-associative operator is `@`. This indicates that the expression `1+2+3` should be parsed as `(1+2)+3`, while the expression `1@2@3` should be parsed as `1@(2@3)`.

In addition to the ambiguities arising from operators, there are additional ambiguities in the grammar arising from `let`, `fun`, and `lambda` expressions. For example, consider the expression:

$$
\begin{array}{lcl}
\mathit{Program} & = & \mathit{Expr} \\
\mathit{Expr} & = & \mathit{let\ ID = Exp\ in\ Expr} \\
 & & \mid \mathit{fun\ ID\ with\ Id\_list = Expr\ in\ Expr} \\
 & & \mid \mathit{lambda\ Id\_list.\ Expr} \\
 & & \mid \mathit{if\ Expr\ then\ Expr_1\ else\ Expr_2} \\
 & & \mid \mathit{Expr_1} + \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1}\ \&\ \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} \mid \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} - \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} * \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} / \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} = \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} <> \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} < \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} <= \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} > \mathit{Expr_2} \\
 & & \mid \mathit{Expr_1} >= \mathit{Expr_2} \\
 & & \mid \mathit{INT\_CONST} \\
 & & \mid \mathit{STRING\_CONST} \\
 & & \mid (\mathit{expr\_list}) \\
 & & \mid \mathit{print\ Expr} \\
 & & \mid \mathit{readInt} \\
 & & \mid \mathit{readString} \\
 & & \mid !\mathit{Expr} \\
 & & \mid \#\mathit{Expr} \\
 & & \mid \mathit{Expr_1} @ \mathit{Expr_2} \\
 & & \mid \mathit{Nil} \\
 & & \mid \mathit{isNil\ Expr} \\
 & & \mid \mathit{ID} \\
\mathit{Id\_list} & = & \mathit{ID} \mid \mathit{ID},\ \mathit{Id\_list} \\
\mathit{expr\_list} & = & \mathit{Expr} \mid \mathit{Expr_1\ expr\_list}
\end{array}
$$

Figure 1: Syntax of the L Programming Language

```
#  !
@
isNil
*  /
+  -
&  |
=  <>  <  >  <=  >=
print
```

Figure 2: Precedence of Operators in L

```
let x = 2 in let x = 3 in x+x
```

This expression can be parsed either as:

```
let x = 2 in (let x = 3 in x)+x
```

which has the value 5, or as the expression

```
let x = 2 in (let x = 3 in x+x)
```

which has the value 6.

To resolve this ambiguity, we stipulate that `let`, `fun`, and `lambda` bindings *extend as far to the right as possible.* This means that the expression

```
let x = 2 in let x = 3 in x+x
```

should be parsed as:

```
let x = 2 in (let x = 3 in x+x)
```

Similarly, the ambiguous expression

```
lambda x. lambda y. y+x
```

should be parsed as

```
lambda x. (lambda y. y+x)
```

rather than as

```
lambda x. ((lambda y. y)+x)
```

6 Operational Semantics

Operational semantics is a formal specification for a programming language. The operational semantics for L describes the meaning of every expression in the L language. More specifically, it describes how, in a given context, every expression e should be reduced to a simpler expression e'.

To define the operational semantics for L, we will first need an *environment* E which maps each identifier to a value. The environment

$$E : [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$$

indicates that the value of identifier x_i is v_i. For example, when analyzing the body of the let expression

```
let x = 2 in x+8
```

the environment E contains the mapping $x \mapsto 2$ to indicate that x is bound to the value 2 in the body of the let expression. We use the notation $E(x) = v$ to denote that the result of looking up the identifier x in E is v. For our example, we have $E(x) = 2$. We also use the notation

$$E[x \leftarrow c]$$

to describe the environment that maps x to c and agrees with E for values of all identifiers other than x. For example, if E is the environment

$$[x \mapsto 2, y \mapsto 4, z \mapsto \text{“}abc''\,]$$

then $E[x \leftarrow 99]$ is the environment:

$$[x \mapsto 99, y \mapsto 4, z \mapsto \text{“}abc''\,]$$

In this manual, we will give the specification of L using *big step operational semantics* which are described using inference rules of the general form:

$$\frac{\begin{array}{c} E_1 \vdash e_1 : v_1 \\ \vdots \\ E_k \vdash e_k : v_k \end{array}}{E \vdash e(e_1, \ldots, e_k) : v}$$

This inference rule says that if e is an expression consisting of sub-expressions $e_1, \ldots e_k$, expression e is evaluated in environment E by recursively reducing the expressions $e_1, \ldots e_k$ to $v_1, \ldots v_k$ in contexts $E_1, \ldots E_k$, and that the result of evaluating e in context E is the value v.

Let us start with the specification for the simplest of L expressions, namely integer and string constants:

$$\frac{\textit{Integer } i}{E \vdash i : i}\ (\text{Int}) \quad \frac{\textit{String } s}{E \vdash s : s}\ (\text{String})$$

The first rule (Int) says that if i is an integer constant, then the result of evaluating expression i is simply the integer value i. Similarly, the second rule (String) says that if s is a string constant, the result of evaluating s is s itself.

A slightly more interesting rule is that for identifiers:

$$\frac{\begin{array}{c} \textit{Identifier id} \\ E(id) = e \end{array}}{E \vdash id : e}\ (\text{Id})$$

This rule says that if *id* is an identifier, then the expression *id* evaluates to the result of looking up *id* in environment E. For instance, evaluating identifier x in context $E : [x \mapsto 2, \ldots]$ yields the integer value 2.

Next, we present the semantics for `let` expressions:

$$\frac{\begin{array}{rcl} E & \vdash & e1 : e1' \\ E[x \leftarrow e_1'] & \vdash & e_2 : e_2' \end{array}}{E \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 : e_2'} \text{ (Let)}$$

To evaluate a let expression $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$ in context E, we first evaluate the initializer expression e_1 in environment E, which yields value e_1'. Then, to evaluate the body e_2, we first obtain a new environment E' in which to evaluate e_2 by binding identifier x to value e_1', i.e., $E' = E[x \leftarrow e_1']$. Next, we evaluate the body e_2 in this new environment E', which yields value e_2', which is also the result of evaluating the entire let expression.

We now present the semantics for if-then-else expressions. The operational semantics for this expression is presented using two inference rules, one in which the conditional evaluates to a non-zero value, and another in which it evaluates to zero:

$$\frac{\begin{array}{c} E \vdash p : \mathit{non\text{-}zero\ integer} \\ E \vdash e_1 : e' \end{array}}{E \vdash \mathit{if}\ p\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : e'} \text{ (If, true)}$$

$$\frac{\begin{array}{c} E \vdash p : 0 \\ E \vdash e_2 : e' \end{array}}{E \vdash \mathit{if}\ p\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : e'} \text{ (If, false)}$$

In the first rule (If, true), the expression p evaluates to a non-zero value in Environment E; thus we evaluate the expression e_1 in the then branch, which yields the value e'. In this case, the result of the whole if expression is also e'. Observe that, in the case where p evaluates to a non-zero value, the expression e_2 is never evaluated.

The second rule (If, false) is similar to first one, except that the expression p in environment E now evaluates to 0. Thus, we only evaluate expression e_2 in environment E, which yields the value e' as the value of the whole if expression.

Next, we give semantics for lambda abstractions. Since lambda abstractions are essentially function definitions, we cannot really evaluate them until this function is "called", i.e., they are applied to some value. Thus, lambda expressions just evaluate to themselves:

$$\frac{}{E \vdash \mathit{lambda}\ x_1, \ldots, x_n.e : \mathit{lambda}\ x_1, \ldots, x_n.e} \text{ (Lambda)}$$

We now discuss the formal semantics for application expressions, which is perhaps the most involved one. We consider two cases, one where the application is of the form $(e_1\ e_2)$ and one where it is of the form $(e_1\ e_2\ e_3 \ldots\ e_k)$ where

$k \geq 3$.

$$\frac{\begin{array}{ccc} E \vdash e_1 & : & \mathit{lambda}\ x.e_1' \\ E \vdash e_1'[e_2/x] & : & e \end{array}}{E \vdash (e_1\ e_2) : e}\ \text{(Application, single)}$$

$$\frac{\begin{array}{ccc} E \vdash e_1 & : & \mathit{lambda}\ x.e_1' \\ E \vdash e_1'[e_2/x] & : & e \\ E \vdash (e\ e_3 \ldots e_k) & : & e' \end{array}}{E \vdash (e_1\ e_2\ e_3 \ldots e_k) : e'}\ \text{(Application, multi)}$$

Let's first focus on the simpler rule, called (Application, single). To evaluate the application $(e_1\ e_2)$, we first evaluate the expression e_1. Note that application is semantically nonsensical if the expression e_1 is not a lambda abstraction; thus, the operational semantics "get stuck" if e_1 is not a lambda abstraction of the form $\mathit{lambda}\ x.e_1'$. This notion of "getting stuck" in the operational semantics corresponds to having a run-time error. Assuming the expression e_1 evaluates to a lambda expression $\mathit{lambda}\ x.e_1'$, we evaluate the application expression by binding e_2 to x and then evaluating the expression$e_1'[e_2/x]$ as in β-reduction in lambda calculus. Observe that we do not evaluate e_2 first before binding it to x; thus, L is a *call-by-name* language. (In a *call-by-value* language, the argument e_2 would be evaluated first and the result bound to formal x.)

We now explain the rule (Application, multi), which gives the semantics for applications with more than one argument. As before, expression e_1 should evaluate to a lambda expression of the form $\mathit{lambda}\ x.e_1'$. Steps one and two of the rule (Application, multi) are identical to the (Application, single) rule. However, after we compute the expression e as the result of applying e_1 to e_2, we still need to apply the new expression e to the remaining expressions $e_3, \ldots, e_k$. For this, we evaluate the new expression $(e\ e_3 \ldots e_k)$ at step three of the (Application, multi) rule. Observe that this rules makes progress since the application evaluated at step three always contains one fewer element than the original application, eventually ending with an application of rule (Application, single).

In addition to lambda bindings, the L language also contains named functions. Fortunately, we can define the meaning of function definitions by rewritten them as let expressions and lambda bindings. Specifically, we use the following equivalence in L:

```
fun f with x = e1 in e2    <=>     let f = lambda x.e1 in e2
```

The only small complication is that function definitions in L may have more than one argument, in which case we have to introduce more than one lambda expression. Specifically, we can define the meaning of function definitions as follows:

$$\frac{E \vdash \mathit{let}\ f = \mathit{lambda}\ x_1 \ldots .\ \mathit{lambda}\ x_n.e_1\ \mathit{in}\ e_2 : e_f}{E \vdash \mathit{fun}\ f\ \mathit{with}\ x_1, \ldots x_n = e_1\ \mathit{in}\ e_2 : e_f}\ \text{(funDef)}$$

Observe that this rule is structurally different form all previous rules in that it defines the meaning of function definitions by "translating" them into a different language construct.

6.1 Binary Operators

In this section we give operation semantics for all the binary operators in L. We start with the arithmetic operators, which have the expected semantics:

$$\frac{\begin{array}{c} E \vdash e_1 : i'_1\,\textit{(integer)} \\ E \vdash e_2 : i'_2\,\textit{(integer)} \end{array}}{E \vdash e_1 + e_2 : i'_1 + i'_2}\ \text{(PlusInt)} \qquad \frac{\begin{array}{c} E \vdash e_1 : i'_1\,\textit{(integer)} \\ E \vdash e_2 : i'_2\,\textit{(integer)} \end{array}}{E \vdash e_1 - e_2 : i'_1 - i'_2}\ \text{(Minus)}$$

$$\frac{\begin{array}{c} E \vdash e_1 : i'_1\,\textit{(integer)} \\ E \vdash e_2 : i'_2\,\textit{(integer)} \end{array}}{E \vdash e_1 * e_2 : i'_1 * i'_2}\ \text{(Times)} \qquad \frac{\begin{array}{c} E \vdash e_1 : i'_1\,\textit{(integer)} \\ E \vdash e_2 : i'_2\,\textit{(integer)} \end{array}}{E \vdash e_1 / e_2 : i'_1 / i'_2}\ \text{(Divide)}$$

In addition to being applicable to integers, the operator `+` also concatenates strings:

$$\frac{\begin{array}{c} E \vdash e_1 : s'_1\,\textit{(string)} \\ E \vdash e_2 : s'_2\,\textit{(string)} \end{array}}{E \vdash e_1 + e_2 : s'_1 s'_2}\ \text{(PlusString)}$$

The predicate operators `=, <>, <, >, <=, >=` evaluate to 0 if the predicate does not hold and to 1 otherwise, as stated in the following operational semantic rules. The operators `=, <>` can be applied to integers and strings, the remaining predicate operators can only be applied to integers:

$$\frac{\begin{array}{l} E \vdash e_1 : e_1' \\ E \vdash e_2 : e_2' \\ e_1', e_2' \textit{ both integer or both string} \\ e_1' = e_2' \end{array}}{E \vdash e_1 = e_2 : 1} \quad \text{(EqualTrue)}$$

$$\frac{\begin{array}{l} E \vdash e_1 : e_1' \\ E \vdash e_2 : e_2' \\ e_1', e_2' \textit{ both integer or both string} \\ e_1' \neq e_2' \end{array}}{E \vdash e_1 = e_2 : 0} \quad \text{(EqualFalse)}$$

$$\frac{\begin{array}{l} E \vdash e_1 : e_1' \\ E \vdash e_2 : e_2' \\ e_1', e_2' \textit{ both integer or both string} \\ e_1' \neq e_2' \end{array}}{E \vdash e_1 <> e_2 : 1} \quad \text{(NotEqualTrue)}$$

$$\frac{\begin{array}{l} E \vdash e_1 : e_1' \\ E \vdash e_2 : e_2' \\ e_1', e_2' \textit{ both integer or both string} \\ e_1' = e_2' \end{array}}{E \vdash e_1 <> e_2 : 0} \quad \text{(NotEqualFalse)}$$

Here are the rules for the remaining predicates:

$$\frac{\begin{array}{l} E \vdash e_1 : i_1' \textit{ (integer)} \\ E \vdash e_2 : i_2' \textit{ (integer)} \\ i_1' \odot i_2' \end{array}}{E \vdash e_1 \odot e_2 : 1} \quad \text{(PredTrue)} \qquad \frac{\begin{array}{l} E \vdash e_1 : i_1' \textit{ (integer)} \\ E \vdash e_2 : i_2' \textit{ (integer)} \\ i_1' \neg \odot i_2' \end{array}}{E \vdash e_1 \odot e_2 : 0} \quad \text{(PredFalse)}$$

where $\odot = \{ <, <=, >, >= \}$.

The last two operators are binary and and or. The binary and operator `&` evaluates to 1 if both arguments evaluate to-non-zero integers and to 0 if at least one argument evaluates to the integer 0. Again this operator is only defined for integers:

$$\frac{\begin{array}{l} E \vdash e_1 : i_1' \textit{ (integer)} \\ E \vdash e_2 : i_2' \textit{ (integer)} \\ i_1' \neq 0 \textit{ and } i_2' \neq 0 \end{array}}{E \vdash e_1 \& e_2 : 1} \quad \text{(AndTrue)} \qquad \frac{\begin{array}{l} E \vdash e_1 : i_1' \textit{ (integer)} \\ E \vdash e_2 : i_2' \textit{ (integer)} \\ i_1' = 0 \textit{ or } i_2' = 0 \end{array}}{E \vdash e_1 \& e_2 : 0} \quad \text{(AndFalse)}$$

The binary or operator evaluates to 1 if at least one of its arguments evaluates to 1 and to zero if both of its arguments evaluate to 0:

$$\frac{\begin{array}{l} E \vdash e_1 : i'_1 \ \textit{(integer)} \\ E \vdash e_2 : i'_2 \ \textit{(integer)} \\ i'_1 \neq 0 \ \textit{or} \ i'_2 \neq 0 \end{array}}{E \vdash e_1 | e_2 : 1} \ \text{(OrTrue)} \qquad \frac{\begin{array}{l} E \vdash e_1 : i'_1 \ \textit{(integer)} \\ E \vdash e_2 : i'_2 \ \textit{(integer)} \\ i'_1 = 0 \ \textit{and} \ i'_2 = 0 \end{array}}{E \vdash e_1 | e_2 : 0} \ \text{(OrFalse)}$$

6.2 List operations

In this section, we give precise meaning to the list-related operations defined in L. Let us start with the constant `Nil`, that indicates the empty list. This rule is a straightforward analogy of the Int or String rule presented earlier:

$$\frac{}{E \vdash \textit{Nil} : \textit{Nil}} \ \text{(Nil)}$$

Now that we have defined the meaning of `Nil`, we can now state the semantics of the `isNil` operator. Since any value can either be `Nil` or non-nil, we have two cases:

$$\frac{E \vdash e : \textit{Nil}}{E \vdash \textit{isNil} \ e : 1} \ (\text{isNil}, \text{true})$$

$$\frac{E \vdash e : c \neq \textit{Nil}}{E \vdash \textit{isNil} \ e : 0} \ (\text{isNil}, \text{false})$$

Here, which rule is triggered will depend on whether `e` evaluates to `Nil` or not.

Next, let us discuss the semantics of the `!` operator. If the expression `e` this operator is applied to is a list, `!e` returns the head of this list. We state this as follows:

$$\frac{E \vdash e : [e_1, e_2]}{E \vdash !e : e_1} \ (\text{Car}, \text{list})$$

If expression `e` does not evaluate to a list, `!e` simply evaluates to `e`:

$$\frac{E \vdash e : e_1 (e_1 \ \textit{not list})}{E \vdash !e : e_1} \ (\text{Car}, \text{not list})$$

The rules for the `#` operator are similar. First if `#e` is applied to a list, it returns the tail of this list:

$$\frac{E \vdash e : [e_1, e_2]}{E \vdash \#e : e_2} \ (\text{Cdr}, \text{list})$$

Second, if `#e` is applied to en expression that is not a list, it returns `Nil`:

$$\frac{E \vdash e : e_1 (e_1 \ \textit{not list})}{E \vdash \#e : \textit{Nil}} \ (\text{Cdr}, \text{not list})$$

The last list operation `@` concatenates two elements into the head and tail respectively of a new list, if the tail concatenated is non-`nil`. More specifically:

$$\frac{\begin{array}{l} E \vdash e_1 : e'_1 \\ E \vdash e_2 : e'_2 \ (e'_2 \ \textit{not Nil}) \end{array}}{E \vdash e_1 @ e_2 : [e'_1, e'_2]} \ (\text{Cons}, \text{not Nil})$$

If the tail element is `Nil`, the operator `@` simply returns the head.

$$\frac{\begin{array}{c}E \vdash e_1 : e_1' \\ E \vdash e_2 : Nil\end{array}}{E \vdash e_1 @ e_2 : e_1'} \ (\text{Cons}, \text{Nil})$$

7 Run-time Errors

As we have discussed before, in the formal specification of the L language it is possible that the operational semantics "get stuck", i.e., that no operational semantics rule applies at a step in the computation. For example, consider the simple expression:

```
"cs345" - 77
```

Here, the operator `-` is only defined on two integers (rule Minus), and there is no rule in the operational semantics that applies in this case. Since this corresponds to a run-time error in the execution, we report a run-time error to the user. Specifically, we want to report that the operator `-` can only be applied to two integers. Similarly, the following code also exhibits a run-time error:

```
let x = y in x
```

Here, the identifier `y` is not bound in E, therefore the rule Id does not apply. The following is a comprehensive list of all run-time errors that can occur in L and the message that should be reported in this case. If multiple error descriptions fit, you should report the one listed first:

- Any binary operator other than `@` is applied to at least one expression that is a list:

  ```
  "Binpo @ is the only legal binop for lists"
  ```

- A binary operator other than `@` is applied to two expressions of different type:

  ```
  "Binop can only be applied to expressions of same type"
  ```

- A binary operator `X` other than `+, =, <>, @` is applied to two strings:

  ```
  "Binop X cannot be applied to strings"
  ```

- The first argument of an application expression is not a lambda-expression[1]:

[1] recall that function definitions are rewritten as let-bindings and lambda expressions

```
"Only lambda expressions can be applied to other expressions"
```

- Identifier ID in an expression that is being evaluated is not bound:

```
"Identifier ID is not bound in current context"
```

- The predicate in a conditional does not evaluate to an integer:

```
"Predicate in conditional must be an integer"
```

- Nil is used with a binop other than `@`:

```
"Nil can only be used with binop @"
```