A Short Introduction to Programming Languages: Application and Interpretation

Rodrigo Bonifácio, Luisa Fantin, Gabriel Lobão, and João Sousa

September 20, 2017

Preface

This book presents a short introduction to Programming Languages: Application and Interpretation (the PLAI book from Shriram Krishnamurthi). Actually, it is a derivative of the aforementioned work, licensed under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 United States License. We selected several chapters from the original book, condensed them a bit, and migrated all Scheme source code to Haskell. Here our goal is to make an introduction to programming language operational semantics, which might help us to present related concepts to our Programming Languages students at University of Brasília. The original version of this work can be found at `www.plai.org/`.

Chapter 1

Interpreting Arithmetic and Substitution

This chapter summarizes some relevants aspects of the first three chapters of the original version of this book, presents a small discussion about parsers and abstract syntax trees, evaluating simple arithmetic expressions, and one of the most fundamental aspects of the first part of this book: substitution. The reader should spend special dedication to this aspect, once it will be further explored in other chapters of this book.

1.1 A simple Arithmetic Expression Language

Having established a handle on parsing, which addresses syntax, we now begin to study semantics. We will study a language with only numbers, addition, and subtraction, and futher assume both these operations are binary. This is indeed a very rudimentary exercise, but that's the point. By picking something you know well, we can focus on the mechanics. Once you have a feel for the mechanics, we can use the same methods to explore languages you have nevever seen before.

The interpreter has the following contract and purpose:

```haskell
module AE where
import Test.HUnit
  -- consumes an AE and computes the corresponding number
calc :: AE -> Integer
  -- some HUnit test cases to better understand the calc semantics
exp1, exp2 :: String
exp1 = "Num 3"
exp2 = "Add (Num 3) (Sub (Num 10) (Num 5))"
tc1 = TestCase (assertEqual "tc01" (calc (parse exp1)) 3)
```

```haskell
tc2 = TestCase (assertEqual "tc02" (calc (parse exp2)) 8)
```

An arithmetic expression AE might be represented using a notation named *Backus-Naur Form* (BNF), after two early programming languages pioneers. A BNF description of rudimentary arithmetic looks like:

```
<AE> ::= Num <int>
       | Add <AE> <AE>
       | Sub <AE> <AE>
```

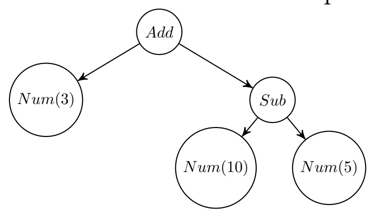
The <AE> in the BNF is calleed a non-terminal, which means we can rewrite it as one of the things on the right-hand side. Read ::= as "can be rewritten as". Each line presents one more choice, called a *production*. Everything in a production that isn't enclosed in the symbos < ...> is literal syntax. In Haskell, as well as in other programming languages, it is quite easy to represent an abstract representation for arithmetic expressions based on a BNF specification. Abstract representations are independent of our choices to concretely represent arithmetic expressions (or other more advanced programming language constructs) as strings of characters. For instance, we might express an expression 3 + (10 - 5) in many different ways, for instance:

```
+ 3 (- 10 5)
+ (3, -(10, 5))
add(3, sub(10, 5))
```

All these forms of expressing the same arithmetic expression could be *parsed* to generate the same abstract syntax tree (a possible alternative is shown in Figure 1.1). That is, after choosing an interesting concrete syntax for a language, a parser reads a program written according to the concrete syntax and ouputs an instance of an abstract syntax tree (AST). We will not give to much attention to concrete syntaxes and parsers in this book. Here, we are mostly intereted in the semantics of a programming language (in this case, expressed in terms of the calc function). We also have to define a *data type* for representing the AE AST using Haskell (see Listing **??**). Note how similar with the BNF such a data representation is (Integer is a primitive data type in Haskell). In this particular case, we define a new data type (named AE) with three data constructors: Num, Add and Sub. The first constructor expects an Integer as argument, while the other two constructors expect two sub-expressions of type AE. Let's ignore some details about the deriving directive right now, though it explains to the Haskell compiler / interpreter to automatically implement support for reading an AE from a string, to show an instance of an AE as a string, and to infer the semantics of the == operator for the AE data type.

```haskell
data AE = Num Integer
  | Add AE AE
  | Sub AE AE
deriving (Read, Show, Eq)
```

Figure 1.1: An illustration of the AST for the simple expression 3 + (10 - 5).



The `calc` function we previously specified must be defined to each `AE` construct (or *term*, in some references). Note that this is an indutive definition on the constructs of `AE`. The first definition is the base case, and states that `calc` for a given `Num n` is `n`. In the other situations, we have to first evaluate both `lhs` (left-hand side) and `rhs` (right-hand side) before calculating the corresponding addition or subtraction.

$$
\begin{aligned}
&\mathit{calc}\ (\mathit{Num}\ n) = n \\
&\mathit{calc}\ (\mathit{Add}\ \mathit{lhs}\ \mathit{rhs}) = \mathit{calc}\ \mathit{lhs} + \mathit{calc}\ \mathit{rhs} \\
&\mathit{calc}\ (\mathit{Sub}\ \mathit{lhs}\ \mathit{rhs}) = \mathit{calc}\ \mathit{lhs} - \mathit{calc}\ \mathit{rhs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{parse} :: \mathit{String} \to \mathit{AE} \\
&\mathit{parse}\ s = \mathit{read}\ s
\end{aligned}
$$

Running the test suite helps validate our interpreter (the `calc` function). For instance, if you open the GHCi on the terminal, you can open this module and run the test suites, executing in the prompt `runTestTT tc1` and `runTestTT tc2`. The results must be as follows.

$$
\begin{aligned}
&\mathit{Counts}\ \{\mathit{cases} = 1, \mathit{tried} = 1, \mathit{errors} = 0, \mathit{failures} = 0\} \\
&\mathit{Counts}\ \{\mathit{cases} = 1, \mathit{tried} = 1, \mathit{errors} = 0, \mathit{failures} = 0\}
\end{aligned}
$$

What we have seen is actually quite remarkable, though its full power may not yet be apparent. We have shown that a programming language with just the ability to represent structured data can represent one of the most interesting forms of data, namely programs themselves. That is, we have just written a program that consumes programs (in the `AE` language); perhaps we can even write programs that generate programs. The former is the foundation for an interpreter semantics, while the later is the foundation for a compiler. This same idea—but with a much more primitive language, namely arithmetic, and a much poorer collection of data, namely just numbers—is at the heart of the proof of Gödel's Theorem.

1.2 Substitution

Even in a simple arithmetic language, we sometimes encounter repeated expressions. For instance, the Newtonian formula for the gravitational force between two objects has a squared term in the denominator. We would like to avoid redundant expressions: they are annoying to repeat, we might make a mistake while repeating them, and evaluating them wastes computational cycles.

The normal way to avoid redundancy is to introduce an identifier.[1] As its name suggests, an identifier names, or identifies, (**the value of**) an expression. We can then use its name in place of the larger computation. Identifiers may sound exotic, but you are use to them in every programming language you have used so far: they are called *variables*. We choose not to call them that because the term "variable" is semantically charged: it implies that the value associated with the identifier can change (*vary*). Since our language initially won't offer any way of changing the associated value, we use the more conservative term "identifier". For now, they are therefore just names for computed constants.

Let's first write a few sample programs that use identifiers, inventing notation as we go along:

```
Let x = 5 + 5 in x + x
```

We want this to evaluate to 20. Here is more elaborate example:

```
Let x = 5 + 5
 in Let y = x - 3
  in y + y

= Let x = 10 in Let y = x - 3 in y + y    [+ operation]
= Let y = 10 - 3 in y + y                 [substitution]
= Let y = 7 in y + y                      [- operation]
= 7 + 7                                   [substitution]
= 14                                      [+ operation]
```

En passant, notice that the act of reducing an expression to a value requires more than just substitution; indeed, it is an interleaving of substitution and calculation steps. Furthermore, when we have completed substitution we implicitly "descend"into the inner expression to continue calculating. Now, let's define the language more formally. To honor the addition of identifiers, we will give our language a new name: LAE, short for "'Let with arithmetic expressions". Its BNF is:

```
<LAE> ::= Int Num
        | Add <LAE> <LAE>
        | Sub <LAE> <LAE>
        | Let <Id> <LAE> <LAE>
        | Ref <Id>
```

Notice that we have had to add two rules to the BNF: one for associating values with identifiers and another for actually using the identifiers. The nonterminal `<Id>` stands for some suitable syntax for identifiers (usually a sequence of alphanumeric characters).

[1] As the authors of Concrete Mathematics say: "Name and conquer".

To write programs that process LAE terms, we need a data definition to represent those terms. Most of LAE carries over unchanged from AE, but we must pick some concrete representation for identifiers. Fortunately, Haskell has a primitive type called String, which server this role admirably. Nevertheless, it is also interesting to introduce a new name to the String type, to make clear the purpose of identifying expressions. We choose the name `Id` as synonymous to the `String` data type. Therefore, the data definition in Haskell is

```haskell
module LAE where

import Test.HUnit

type Id = String
type Value = Integer

data LAE = Num Integer
   | Add LAE LAE
   | Sub LAE LAE
   | Let Id LAE LAE
   | Ref Id
 deriving (Read, Show, Eq)
```

The `Let` data constructor expects three arguments: the name of the identifier, the named expression associated to the identifier, and the `Let` expression body. The `Ref` data constructor expects only one argument: the name of the identifier.

1.2.1 Defining Substitution

Without ceremony, we use the concept of *substitution* to explain how the `Let` construct works. We are able to do this because substitution is not unique to `Let`: we have studied it for years in algebra courses, because that is what happens when we pass arguments to functions. For instance, let $f(x, y) = x^3 + y^3$. Then

$$\begin{aligned} f(12,1) &= 12^3 + 1^3 = 1728 + 1 = 1729 \\ f(10,9) &= 10^3 + 9^3 = 1000 + 729 = 1729 \end{aligned}$$

Nevertheless, it is a good idea to pin down this operation precisely.

Let's make sure we understand what we are trying to define. We want a crisp description of the process of substitution, namely what happens when we replace an identifier (such as x or `x`) with a value (such as 12 or `5`) in an expression (such as $x^3 + y^3$ or `x + x`).

Recall from the sequence of reductions above that substitution is a part of, but not the same as, calculating an answer for an expression that has identifiers. Looking back at the sequence of steps in the evaluation example above, some of them invoke substitution while the rest are calculation as defined for AE. For now, we are first going to pin down substitution. Once we have done that, we will revisit the related question of calculation. But it will take us a few tries to get substitution right!

Definition 1.2.1: Substitution

Given an expression like `Let` x `=` exp_1 `in` exp_2, the components of the `Let` expression are the identifier x, the named expression exp_1, and expression body exp_2. To substitute the identifier x in the expression body exp_1 with the named expression exp_2, replace all identifiers in the expression body that have the name x with the named expression (in this case exp_1).

Beginning with the program

```
Let x = 5 in x + x
```

we will use substitution to replace the identifier `x` with the named expression it is bound to (`5`). The above definition of substitution certainly does the trick: after substitution, we get

```
Let x = 5 in 5 + 5
```

as we would want. Likewise, it correctly substitutes when there are no instances of the identifier. For instance,

```
Let x = 5 in 10 + 4
```

the definition of substitution leads to `Let x = 5 in 10 + 4`, since there are no instances of x in the expression body. Now consider

```
Let x = 5 in x + Let x = 3 in 10
```

The rules reduce this to `Let x = 5 in 5 + Let 5 = 3 in 10`. Huh? Our substitution rule converted a perfectly reasonable program (whose value is 15) into one that isn't even syntactically legal, i.e., it would be rejected by a parser because the program contains a `5` where the the `BNF`tells us to expect an identifier. We definitely don't want substitution to have such an effect! It's obvious that the substitution algorithm is too naive. To state the problem with the algorithm precisely, though, we need to introduce a little terminology.

Definition 1.2.2: Binding Instance

A binding instance of an identifier is the occurrence of the identifier that gives it its value. In LAE, the `Id` position of a `Let` expression is the only binding instance.

Definition 1.2.3: Scope

The scope of a binding instance is the region of a program text in which instances of the identifier **refer to the value** bound by the binding instance.

Definition 1.2.4: Bound Instance

An identifier is bound if it is contained within the scope of a binding instance of its name.

Definition 1.2.5: Free Instance

An identifier not contained in the scope of any binding instance of its name is said to be free.

With this terminology in hand, we can now state the problem with the first definition of substitution more precisely: it failed to distinguish between bound instances (which should be substituted) and binding instances (which should not). This leads to a refined notion of substitution.

Definition 1.2.6: Substitution, take 2

Given an expression like $Let\ x =\ exp_1\ in\ exp_2$, the components of the `Let` expression are the identifier x, the named expression exp_1, and expression body exp_2. To substitute the identifier x in the expression body exp_1 with the named expression exp_2, replace all identifiers in the expression body which are not binding instances and that have the name x with the named expression (in this case exp_1).

A quick check reveals that this does not affect the outcome of the examples that the previous definition substituted correctly. In addition, this definition of substitution reduces `Let x = 5 in x + Let x = 3 in 10` to `Let x = 5 in 5 + Let x = 3 in 10`.

Let's consider a closely related expression `Let x = 5 in x + Let x = 3 in x`. Think a little bit. What should the value of this expression? Hopefully, we can agree that the value of this program is 8)the left `x` in the addition evaluates to `5`, the right `x` is given the value `3`, by the inner `Let`, so the sum is `8`). The refined substitution algorithm, however, converts this expression into `Let x = 5 in 5 + Let x = 3 in 5`, which, when evaluated, yields `10`.

What went wrong here? Our substitution algorithm respected binding instances, but not their scope. In the sample expression, the `Let` introduces a new scope for the inner `x`. The scope of the outer `x` is *shadowed* or *masked* by the inner binding. Because substitution doesn't recognize this possibility, it incorrectly substitutes the inner `x`.

Definition 1.2.7: Substitution, take 2

Given an expression like $Let\ x =\ exp_1\ in\ exp_2$, the components of the `Let` expression are the identifier x, the named expression exp_1, and expression body exp_2. To substitute the identifier x in the expression body exp_2 with the named expression exp_1, replace all identifiers in the expression body which are not binding instances and that have the name x with the named expression (in this case exp_1), unless the identifier is in a scope different from that introduced by `x`.

While this rule avoids the faulty substitution we have discussed earlier, it has the following effect: after substitution, the expression `Let x = 5 in x + Let y = 3 in x` becomes `Let x = 5 in 5 + Let y = 3 in x`. The inner expression should result in an error, because `x` has no value. Once again, substitution has changed a correct program into an incorrect one!

Let's understand what went wrong. Why didn't we substitute the inner `x`? Substitution halts at the `Let` because, by definition, every `Let` introduces a new scope, which we said should delimit substitution. But this `Let` contains an instance of `x`, which we very much want substituted! So which is it—substitute within nested scopes or not? Actually, the two examples above should reveal that our latest definition for substitution, which might have seemed sensible at first blush, is too draconian: it rules out substitution within *any* nested scopes.

Definition 1.2.8: Substitution, take 2

Given an expression like *Let* $x =$ exp_1 *in* exp_2, the components of the `Let` expression are the identifier x, the named expression exp_1, and expression body exp_2. To substitute the identifier x in the expression body exp_2 with the named expression exp_1, replace all identifiers in the expression body which are not binding instances and that have the name x with the named expression (in this case exp_1), except within *nested scopes of* `x`.

Finally, we have a version of substitution that works. A different, more succinct way of phrasing this definition is

Definition 1.2.9: Substitution, take 5

Given an expression like *Let* $x =$ exp_1 *in* exp_2, the components of the `Let` expression are the identifier x, the named expression exp_1, and expression body exp_2. To substitute the identifier x in the expression body exp_2 with the named expression exp_1, replace all free instances of `x` in the expression body with the named expression (in this case, exp_1).

Recall that we are still defining substitution, not evaluation. Substitution is just an algorithm defined over expressions, independent of any use in an evaluator. It is the calculator's job to invoke substitution as many times as necessary to reduce a program down to an answer. That is, substitution simply converts `Let x = 5 in x + Let y = 3 in x` into `Let x = 5 in 5 + Let y = 3 in 5`. Reducing this to an actual value is the task of the rest of the calculator. Phew! Just to be sure we understand this, let's express it in the form of a function.

-- substitutes the first argument (x) by the second argument (v)
-- in the free occurrences of the let expression body (the third
-- argument of the function). the resulting expression must not have
-- any free occurrence of the first argument.

$subst :: Id \to LAE \to LAE \to LAE$
$subst\ \_\ \_\ (Num\ n) = Num\ n$
$subst\ x\ v\ (Add\ lhs\ rhs) = Add\ (subst\ x\ v\ lhs)\ (subst\ x\ v\ rhs)$
$subst\ x\ v\ (Sub\ lhs\ rhs) = Sub\ (subst\ x\ v\ lhs)\ (subst\ x\ v\ rhs)$
$subst\ x\ v\ (Let\ i\ e1\ e2) = \bot$
$subst\ x\ v\ (Ref\ i) = \bot$

The `subst` function is defined in terms of *pattern matching*. In the first case, a substitution of any identifier by any named expression within a `Num n` expression body actually returns the expression body. When the message body is an expression like `Add e1 e2` or `Sub e1 e2` we return either and `Add` or a `Sub` expression, respectively, though having as sub expressions recursive calls to the `subst` function on their respective sub expressions `e1` and `e2`. Based on the previous definitions, you should implement the case for `subst` on `Let` expressions. This is the most interesting case. Finally, substituting a `Ref i` expression have to deal with two new situations. The first, we are trying to substitute the identifier `x` by the named expression `v` within a `Ref x`. In this case, we just return `v`. In the second, we are trying to substitute within a `Ref i`, where `x != i`, and thus we return `Ref i`—there is no substitution to perform in this case.

1.2.2 Calculating with `Let`

We have finally defined substitution, but we still have not specified how we will use it to reduce expressions to answers. To do this, we must modify our calculator. Specifically, we must add rules for our two new source language syntactic constructs: `Let` and `Ref`.

- To evaluate `Let` expressions, we **first calculate** the named expression and then substitutes identifier by its value in the body of the `Let` expression.

- How about identifiers? Well any identifier that is in the scope of a `Let` expression must be replaced with a value when the calculator encounters that identifiers binding instance. Consequently, the purpose statement of *subst* said there would be no free instances of the identifier given as an argument left in the result. In other words, *subst* replaces identifiers with values before the calculator ever finds them. As a result, any *as-yet-unsubstituted* identifier must be free in the whole program. The calculator can't assign a value to a free identifier, so it halts with an error.

Please, considering the implementation of the `calc` function for AE, implement a new function (also named `calc`) for LAE. Consider the following test cases.

```
calc :: LAE -> Integer
calc = ⊥
  -- some HUnit test cases to better understand the calc semantics
exp1, exp2, exp3, exp4 :: String
exp1 = "Num 5"
exp2 = "Add (Num 5) (Num 5)"
exp3 = "Let \"x\" (Add (Num 5) (Num 5)) (Add (Ref \"x\") (Ref \"x\"))"
exp4 = "Let \"x\" (Num 5) (Let \"y\" (Ref \"x\") (Ref \"y\"))"
exp5 = "Let \"x\" (Num 5) (Let \"x\" (Ref \"x\") (Ref \"x\"))"
tc01 = TestCase (assertEqual "tc01" (calc (parse exp1)) 5)
tc02 = TestCase (assertEqual "tc02" (calc (parse exp2)) 10)
tc03 = TestCase (assertEqual "tc03" (calc (parse exp3)) 20)
tc04 = TestCase (assertEqual "tc04" (calc (parse exp4)) 5)
tc05 = TestCase (assertEqual "tc05" (calc (parse exp5)) 5)
parse :: String -> LAE
parse = read
```

Chapter 2

Functions and Functions as Values

This chapter introduces several concepts related to function declaration, the scope of functions, and a classification of functions.

2.1 An Introduction to Functions

In the previous chapter, we have added identifiers and and the ability to name expressions to the language. Much of the time, though, simply being able to name an expression isn't enough: the expression's value is going to depend on the context of its use. That means the expression needs to be parameterized and, thus, it must be a *function*.

Dissecting a `Let` expression is a useful exercise in helping us design functions. Consider the program

```
Let x = 5 in x + 3
```

In this program, the expression `x + 3` is parameterized over the value of `x`. In that sense, it is just like a function definition: in mathematical notation we might write:

$$f(x) \quad = \quad x + 3$$

Having named and def ind f, what do we do with it? The `LAE` program introduces `x` and than immediately binds it to `5`. The way we bind a function's argument to a value is to apply it. Thus, it is as if we wrote:

$$f(x) \quad = \quad x + 3; \; f(5)$$

In general, functions are useful entities to have in programming languages, and it would be instructive to model them.

2.1.1 Enriching the Languages with Functions

To add functions to LAE, we must define their abstract syntax. In particular, we must both describe a *function definition* (declaration) and provide a means for its *application* or *invocation*. To do the latter, we must add a new kind of expression, resulting in the language F1LAE. We will presume, as a simplification, that functions consume only one argument. This expression language has the following BNF.

```
<F1LAE> ::= Int Num
          | Add <F1LAE> <F1LAE>
          | Sub <F1LAE> <F1LAE>
          | Let <Id> <F1LAE> <F1LAE>
          | Ref <Id>
          | App <Id> <F1LAE>
```

The expression representing the argument supplied to the function is known as the actual parameter. To capture this new language, we again have to declare a Haskell data type.

```haskell
module F1LAE where

import Test.HUnit

type Id = String
type Name = String
type FormalArg = String
type Value = Integer

data Exp = Num Integer
  | Add Exp Exp
  | Sub Exp Exp
  | Let Id Exp Exp
  | Ref Id
  | App Name Exp
 deriving (Read, Show, Eq)
```

Now, let's study function declaration. A function declaration has three components: the name of the function, the names of its arguments (known as the formal parameters), and the function's body. (The function's parameters might have types, which we will discuss later in this book). For now, we will presume that functions consume only one argument. A simple data definition captures this.

```haskell
data FunDec = FunDec Name FormalArg Exp
 deriving (Read, Show, Eq)
```

Using this definition, one might declare a standard function for doubling its argument as:

$$\begin{aligned}&\mathit{double} :: \mathit{FunDec}\\ &\mathit{double} = \mathit{FunDec}\ \texttt{"double"}\ \texttt{"x"}\ (\mathit{Add}\ (\mathit{Ref}\ \texttt{"x"})\ (\mathit{Ref}\ \texttt{"x"}))\end{aligned}$$

Now we are ready to write the calculator, which we will call *interp*—short for interpreter-rather than *calc* to reflect the fact that our language has grown beyond arithmetic. The interpreter must consume two arguments: the expression to evaluate and the set of known function declarations. Most of the rules of `LAE` remain the same, so we can focus on the new rule.

$$\begin{aligned}&\mathit{interp} :: \mathit{Exp} \to [\mathit{FunDec}] \to \mathit{Value}\\ &\mathit{interp} = \bot\end{aligned}$$

The rule for an application first looks up the named function. If this access succeeds, then interpretation proceeds in the body of the function after first substituting its formal parameter with the (interpreted) value of the actual parameter. We can see the result using GHCi.

2.1.2 The scope of substitution

Suppose we ask our interpreter to evaluate the expression

$$\begin{aligned}&\mathit{app1} :: \mathit{Exp}\\ &\mathit{app1} = \mathit{App}\ \texttt{"f"}\ (\mathit{Num}\ 10)\end{aligned}$$

In the presence of the solitary function definition

$$\begin{aligned}&f :: \mathit{FunDec}\\ &f = \mathit{FunDec}\ \texttt{"f"}\ \texttt{"n"}\ (\mathit{App}\ \texttt{"n"}\ (\mathit{Ref}\ \texttt{"n"}))\end{aligned}$$

What should happen? Should the interpreter try to substitute the n in the function position of the application with the number 10, than complains that no such function can be found (or even that function lookup fundamentally fails because the names of the functions must be identifiers, not numbers)? Or should the interpreter decide that function names and function arguments live in two different "spaces", and let the context determines in which space to lookup a name? Languages like Scheme take the former approach: the name of a function can be bound to a value in a local scope, thereby rendering the function inaccessible through that name. This later strategy is known as employing namespaces and languages like Common Lisp adopt it.

2.1.3 The Scope of Function Definitions

Suppose our *definition list* contains multiple function declarations. How do these interact with one another? For instance, suppose we evaluate the following input `eval app2 [g, h]`, where

$$\begin{aligned}&\mathit{app2} :: \mathit{Exp}\\ &\mathit{app2} = \mathit{App}\ \texttt{"f"}\ (\mathit{Num}\ 5)\end{aligned}$$

$g :: FunDec$
$g = FunDec$ `"g"` `"n"` $(App$ `"h"` $(Add\ (Ref$ `"n"`$)\ (Num\ 5)))$
$h :: FunDec$
$h = FunDec$ `"h"` `"m"` $(Sub\ (Ref$ `"m"`$)\ (Num\ 1))$

What does the mentioned evaluation do? The main expression applies g to 5. The definition of g, in turn, invokes function h. Should g be able to invoke h? Should the invocation fail because h is defined after g in the list of definitions? What if there are multiple bindings for a given function's name? We will expect this evaluation to reduce to 9. That is, we employ the more natural interpretation that each function can "see" every function's definition, and the natural assumption that each name is bound at most once so we don't need to disambiguate between definitions. It is, however, possible to define more sophisticated scopes.

Exercise 1

Implement the `interp` function as specified above.

Exercise 2

If a function can invoke every defined function, that means it can also invoke itself. This is currently of limited value because our `F1LAE` language lacks a harmonious way of terminating recursion. Implement a simple conditional construct (`if0`) which succeeds if the term in the first position evaluates to zero, and write interesting recursive functions in this language.

2.2 First Class Functions

There is a similarity between a `Let` expression and a function definition applied immediately to a value. For instance, not that:

```
Let x = 5 in x + 3
```

is essentially the same as $f(x) = x + 3; f(5)$. Actually, that is not quite right: in the math equation, we give the function a name (f), whereas there is no identifier named f anywhere in the `Let` expression above. We can, however, rewrite the mathematical formulation as $f = \lambda x.x + 3; f(5)$, which can then be rewritten as $(\lambda x.x+3)(5)$ to get rid of the unnecessary name f. Notice, however, that our language `F1LAE` does not permit anonymous functions (a concept that currently is also present in imperative languages like Java, Scala, and Python, for instance) of the style we have used above. Because such a functions are useful in their own right, we now extend our study of functions.

2.2.1 A Taxonomy of Functions

The translation of `Let` into mathematical notation exploits two features of functions: the ability to create anonymous functions, and the ability to define functions anywhere in the program (in this case, in the function position of a Lambda application). Not every programming language offers one or both of these capabilities. There is, therefore, a taxonomy that governs these different features, which we can use when discussing what kind of functions a language provides. The taxonomy is as what follows.

first-order Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in `F1LAE` are of this nature, which explains the `1` in the name of the language.

higher-order Functions can return other functions as values.

first-class Functions are values with all the rights of other values. In particular, they can be supplied as the value arguments to functions, returned by functions as answers, and stored in data structures.

2.2.2 Enriching F1LAE with First-Class Functions

To add *first-class functions* to `F1LAE`, we must proceed as usual, by first defining its concrete and abstract syntaxe trees. First, let us examine some concrete programs:

```
(\x . x + 4) 5
```

This program (consisting of a sole expression) defines a function that adds 4 to its argument and immediately applies this function to 5, resulting in the value 9. This one

```
Let double = (\x . x + x)
 in (double 10) + (double 5)
```

evaluates to 30. The program defines a function, binds it to `double`, then uses that name twice in slightly different contexts (i.e., it instantiates the formal parameter with different actual parameters). From these examples, it should be clear that we must introduce two new kinds of expressions: anonymous functions and anonymous function applications. Here is the revised `BNF`corresponding to these examples.

$\langle Name \rangle$ ::= $\langle Id \rangle$

$\langle Arg \rangle$ :: = $\langle Id \rangle$

$\langle FunDec \rangle$ ::= '`def`' $\langle Name \rangle$ $\langle Arg \rangle$ '`=`' $\langle FLAE \rangle$

$\langle FLAE \rangle$ ::= $\langle Num \rangle$
  | Add $\langle FLAE \rangle$ $\langle FLAE \rangle$
  | Sub $\langle FLAE \rangle$ $\langle FLAE \rangle$

| Let $\langle Id \rangle$ $\langle FLAE \rangle$ $\langle FLAE \rangle$
| Ref $\langle Id \rangle$
| App $\langle Name \rangle$ $\langle FLAE \rangle$
| λ $\langle Arg \rangle$ '.' $\langle FLAE \rangle$
| AppLambda $\langle FLAE \rangle$ $\langle FLAE \rangle$

In this language, it is possible to declare both named functions (using the function declarations) and anonymous functions (using the lambda abstractions), which might appear anywhere we are expecting a `FLAE` expression (in particular, in the first component of a lambda application). Therefore, instead of just the name of a function, programmers can write an arbitrary expression that must be evaluated to obtain the function to apply. The corresponding abstract syntax is:

```
module F2LAE where

type Id = String
type Name = String
type FormalArg = String

type Value = Exp

data FunDec = FunDec Name FormalArg Exp

data Exp = Num Integer
  | Add Exp Exp
  | Sub Exp Exp
  | Let Id Exp Exp
  | Ref Id
  | App Name Exp
  | Lambda Id Exp
  | AppLambda Exp Exp
```

To define our interpreter, we must think a little about what kinds of values it consumes and produces. Naturally, the interpreter consumes values of `FLAE` expressions (and a list of function declarations). What does it produces? Clearly, a program that meets `FLAE` must yields numbers. As we have seen above, some programs that use functions and applications also evaluate to numbers. How about a program that consists solely of a function? That is, what is the value of the program $(\lambda x.x)$? It clearly does not represent a number. It might be a function that, when applied to a numeric argument, produces a number, but it is not itself a number. We instead realized from this that *anonymous functions are also values* that may be the result of a computation.

We could design an elaborate representation for function values, but for now, we will remain modest. We will let the function evaluate to its abstract syntax representation (i.e., a `Lambda` structure). For consistency, we will also let numbers evaluate to a `Num` structure. Thus, the result of evaluating $(\lambda x.x)$ would be the value `Lambda ''x'' (Ref ''x'')`.

Now we are ready to write the interpreter. We must pick a type for the value that *interp* returns. Since we have decided to represent function and number answers using the abstract syntax, it makes sense to use `FLAE` expressions, with the caveat that only two kinds of expressions can appear in the output: numbers and functions. Our first interpreter will use explicit substitution, to offer a direct comparison with the interpreters discussed before.

$$\begin{aligned}&\mathit{interp} :: \mathit{Exp} \to [\mathit{FunDec}] \to \mathit{Value}\\&\mathit{interp} = \bot\end{aligned}$$

2.3 Making Let Expressions Redundant

Now that we have functions as first class citizens, we can combine lambda abstractions and lambda applications to recover the behaviour of `Let` expressions as a special case. Every time we encounter an expression of the form `Let var = named in body` we can replace it with $(\lambda var\ .\ \mathit{body})\ \mathit{named}$ and obtain the same effect. The result of this translation reduces some boilerplate code that is necessary to interpret the application of lambda and let expressions.

Exercise 3

Implement a pre-processor that performs this translation.

2.4 Implementing Functions Using Deferred Substitutions

Let's examine the process of interpreting the following small program.

```
Let x = 3
 in Let y = 4
  in Let z = 5
   in x + y + z
```

$$\begin{aligned}&= \mathit{Let}\ y = 4\ \mathit{in}\ \mathit{Let}\ z = 5\ \mathit{in}\ 3 + y + z\ (\mathit{subst})\\&= \mathit{Let}\ z = 5\ \mathit{in}\ 3 + 4 + z\ (\mathit{subst})\\&= 3 + 4 + 5\ (\mathit{subst})\\&= 12\ (\mathit{arithmetic})\end{aligned}$$

On the right is the sequence of evaluation steps. To reduce it to an arithmetic problem, the interpreter had to apply substitution three times: once for each `Let` expression. This is slow! How slow? Well, if the program has size n (measured in abstract syntax tree nodes), than each substitution *traverses* the rest of the program once, making the complexity of this interpreterter at least $O(n^2)$. That seems rather wasteful, surely we can do better.

How will avoid computational redundancy? We should create and use a *repository of deferred substitutions*. Concretly, here is the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a `Let` or `Application`), we augment the repository with one more entry, recording the identifier's name and the value (if *eager*) or expression (if *lazy*) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.

This strategy breaks a key invariant we had established earlier, which is that any identifier the interpreter encounters is of necessity free—in the case it had been bound, it would have been replaced by substitution. Because we are no longer using substitution, we will encounter bound identifiers during interpretation. How do will handle them? We must consult the repository in order to substitute them. Our new language F3LAE is quite similar to the previous one

```haskell
module F3LAE where

type Name = String
type FormalArg = String
type Id = String

data FunDec = FunDec Name FormalArg Exp

data Exp = Num Integer
   | Add Exp Exp
   | Sub Exp Exp
   | Let Id Exp Exp
   | App Name Exp
   | Lambda FormalArg Exp
   | LambdaApp Exp Exp
  deriving (Show, Eq)
```

though we have to declare a new type for representing our repository of deferred substitutions.

```haskell
type DefrdSub = [(Id, Value)]
```

Several situations must be considered when implementing the `interp` function for F3LAE. For instance, consider the following test case

```
Let x = 3
 in Let f = (\y . y + x)
   in Let x = 5
     in f 4
```

Depending on the evaluation strategy and scope resolution, its evaluation must result either in 7 (static scope) or 9 (dynamic scope). In the later case, the value of x within the function definition depends on the context of application of f, not on the scope of its definition.

That is, to properly defer substitution, the value of a function should be not only its definition, but also the substitutions that were due to be performed on it. Therefore, we must define a new datatype for the interpreter's return value, which attaches the definition-time repository to every function value. Our `Value` datatype is either a *numeric value* or a *closure*, a kind of function definition that comes together with the list of deferred substitutions that appear until its definition. We call this constructed value a *closure* because it "closes" the function body of lambda expressions over the substitutions that are waiting to occur.

$$\textbf{data}\ \mathit{Value} = \mathit{NumValue}\ \mathit{Integer}$$
$$\mid \mathit{Closure}\ \mathit{FormalArg}\ \mathit{Exp}\ \mathit{DefrdSub}$$

When the interpreter encounters a function application, it must ensure that the function's pending substitutions are not forgotten. It must however, ignore the substitutions pending at the location of the invocation, for that is precisely what led us to dynamic instead of static scope. It must instead use the substitutions of the invocation location to convert the function and argument into values, hope that the function expression evaluated to a closure, then proceed with evaluating the body of the function employing the repository of deferred substitutions stored in the closure.

Exercise 4

Implement the interpreter function for F3LAE, considering the following specification.

$$\mathit{interp} :: \mathit{Exp} \to \mathit{DefrdSub} \to [\mathit{FunDec}] \to \mathit{Value}$$
$$\mathit{interp} = \bot$$