Anatomy of Programming Languages

William R. Cook

Contents

Chapter 1

Preliminaries

1.1 Preface

1.1.1 What?

This document is a series of notes about programming languages, originally written for students of the undergraduate programming languages course at UT.

1.1.2 Why?

I'm writing these notes because I want to teach the theory of programming languages with a practical focus, but I don't want to use Scheme (or ML) as the host language. Thus many excellent books do not fit my needs, including *Programming Languages: Application and Interpretation* (Krishnamurthi 2012), *Essentials of Programming Languages* (Friedman and Wand 2008) or *Concepts in Programming Languages* (Mitchell and Apt 2001).

This book uses Haskell, a pure functional language. Phil Wadler (Wadler 1987) gives some good reasons why to prefer Haskell over Scheme in his review of Structure and Interpretation of Computer Programs (Abelson and Sussman 1996). I agree with most but not all of his points. For example, I do not care much for the fact that Haskell is lazy. None of the examples in this book rely upon this feature.

I believe Haskell is particularly well suited to writing interpreters. But one must be careful to read Haskell code as one would read poetry, not the way one would read a romance novel. Ponder each line and extract its deep meaning. Don't skim unless you are pretty sure what you are doing.

The title of this book is derived from one of my favorite books, *The Anatomy of Lisp* (Allen 1978).

1.1.3 Who?

These notes assume knowledge of programming, and in particular assume basic knowledge of programming in Haskell. When I teach the course I give a few hours of lectures to introduce Haskell. I teach the built-in data types including lists, the basic syntax for conditionals, function definitions, function calls, list comprehensions, and how to print out strings. I also spend a day on **data** definitions (algebraic data types) and pattern matching. Finally, I give a quick introduction to type classes so student will understand how *Eq* and *Show* work. During the course I teach more advanced topics, including first-class functions and monads. As background resources, I point students to the many excellent tutorials on Haskell. Search Google for "Haskell Tutorial" to find one. I recommend Learn You a Haskell for Great Good! or the Gentle Introduction To Haskell.

Acknowledgments

I thank the students in the spring 2013 semester of CS 345 *Programming Languages* at the University of Texas at Austin, who helped out while I was writing the book. Special thanks to Jeremy Siek, Chris Roberts and Guy Hawkins for corrections and Aiden Song and Monty Zukowski for careful proofreading. Tyler Allman Young captured notes in class. Chris Cotter improved the makefile and wrote the initial text for some sections.

1.2 Introduction

In order to understand programming languages, it is useful to spend some time thinking about *languages* in general. Usually we treat language like the air we breathe: it is everywhere but it is invisible. I say that language is invisible because we are usually more focused on the message, or the content, that is being conveyed than on the structure and mechanisms of the language itself. Even when we focus on our use of language, for example in writing a paper or a poem, we are still mostly focused on the message we want to convey, while working with (or struggling with) the rules and vocabulary of the language as a given set of constraints. The goal is to work around and avoid problems. A good language is invisible, allowing us to speak and write our intent clearly and creatively.

The same is true for programming. Usually we have some important goal in mind when writing a program, and the programming language is a vehicle to achieve the goal. In some cases the language may fail us, by acting as an impediment or obstacle rather than an enabler. The normal reaction in such situations is to work around the problem and move on.

The study of language, including the study of programming languages, requires a different focus. We must examine the language itself, as an artifact. What are its rules?

What is the vocabulary? How do different parts of the language work together to convey meaning? A user of a language has an implicit understanding of answers to these questions. But to really study language we must create an explicit description of the answers to these questions.

The concepts of structure and meaning have technical names. The structure of a language is called its *syntax*. The rules that defined the meaning of a language are called *semantics*. Syntax is a particular way to structure information, while semantics can be viewed as a mapping from syntax to its meaning, or interpretation. The meaning of a program is usually some form of behavior, because programs *do* things. Fortunately, as programmers we are adept at describing the structure of information, and at creating mappings between different kinds of information and behaviors. This is what data structures and functions/procedures are for.

Thus the primary technique in these notes is to use programming to study programming languages. In other words, we will write programs to represent and manipulate programs. One general term for this activity is *metaprogramming*. A metaprogram is any program whose input or output is a program. Familiar examples of metaprograms include compilers, interpreters, virtual machines. In this course we will read, write and discuss many metaprograms.

Chapter 2

Expressions and Variables

2.1 Simple Language of Arithmetic

A good place to start is analyzing the language of arithmetic, which is familiar to every grade-school child:

```
4
-5+6
3--2--7
3*(8+5)
3+(8*2)
3+8*2
```

These are examples of arithmetic *expressions*. The rules for understanding such expressions are surprisingly complex. For example, in the third expression the first and third minus signs $(-)$ mean subtraction, while the second and fourth mean that the following number is negative. The last two examples mean the same thing, because of the rule that multiplication must be performed before addition. The third expression is potentially confusing, even given knowledge of the rules for operations. It means $(3-(-2))-(-7)$ not $3-((-2)-(-7))$ because subtraction associates to the left.

Part of the problem here is that there is a big difference between our conceptual view of what is going on in arithmetic and our particular conventions for expressing arithmetic expressions in written form. In other words, there isn't any confusion about what negative number are or what subtraction or exponentiation do, but there is room for confusion about how to write them down.

The conceptual structure of a given expression can be defined much more clearly using pictures. For example, the following pictures make a clear description of the underlying arithmetic operations specified in the expressions given above:

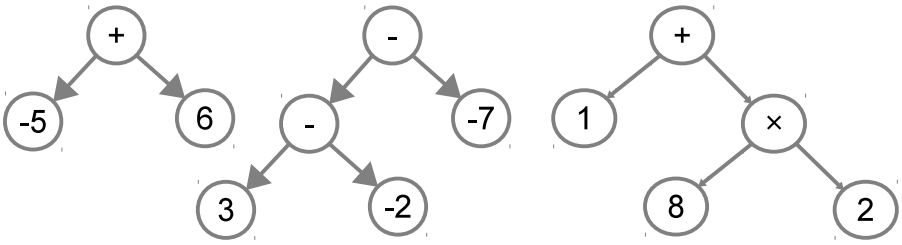

Figure 2.1: Graphical illustration of abstract structure

The last picture represents both the last expressions in the previous example. This is because the pictures do not need parentheses, since the grouping structure is explicit.

The conceptual structure (illustrated by the pictures) is called the *abstract syntax* of the language. The particular details and rules for writing expressions as strings of characters is called the *concrete syntax*. The abstract syntax for arithmetic expressions is very simple, while the concrete syntax is quite complex. As a result, for now we will focus on the abstract syntax, and deal with concrete syntax later.

2.1.1 Abstract Syntax in Haskell

This section describes how to represent abstract syntax using Haskell. The code for this section is found in the Simple file. Arithmetic expressions can be represented in Haskell with the following data type:

```
data Exp = Number Int
         | Add      Exp Exp
         | Subtract Exp Exp
         | Multiply Exp Exp
         | Divide   Exp Exp
```

This data type defines five representational variants, one for numbers, and four for the the binary operators of addition, subtraction, multiplication, and division. A number that appears in a program is called a *literal*.

The five examples given above can be written as values of type *Exp* to create five test cases:

```
--4
t1 = Number 4

---5+6
t2 = Add (Number (-5)) (Number 6)

--3-(-2)-(-7)
t3 = Subtract (Subtract (Number 3) (Number (-2))) (Number (-7))
```

$--\ 3 * (8 + 5)$
$t4 = Multiply\ (Number\ 3)\ (Add\ (Number\ 8)\ (Number\ 5))$

$--\ 3 + 8 * 2$
$t5 = Add\ (Number\ 3)\ (Multiply\ (Number\ 8)\ (Number\ 2))$

Each test case is preceded by a comment giving the concise notation for the corresponding expression. NOTE: It is not legal to write $Add\ (-4)\ 6$ because -4 and 6 are of type *Int* not *Exp*. Also, Haskell requires parentheses around negative numbers, for some reason.

Writing abstract syntax directly in Haskell is certainly very ugly. There is approximately a 10-fold expansion in the number of characters needed to represent a concept: a 5-character mathematical expression $3 + 8 * 2$ uses 47 characters to create the corresponding Haskell data structure. This is not a defect of Haskell, it is merely a side effect of the lack of a proper parser, which we haven't developed yet. Writing these data constructors explicitly is not something that we enjoy doing, but for now it is useful to be very explicit about the representation of our programs.

For now expressions will be written using the concise and familiar concrete syntax $3+7$, adding parentheses where necessary. But keep in mind that this concise syntax is just a short-hand for the real value $Add\ (Number\ 3)\ (Number\ 7)$. As new features are added to the language, both the familiar concrete syntax and the abstract syntax will be extended.

TODO: talk about meta language: language of study versus language of implementation. Better words?

2.1.2 Evaluating Arithmetic Expressions

The normal meaning assigned to arithmetic expressions is the evaluation of the arithmetic operators to compute a final answer. This evaluation process is defined by cases in Haskell:

$evaluate :: Exp \to Int$
$evaluate\ (Number\ i) = i$
$evaluate\ (Add\ a\ b) = evaluate\ a + evaluate\ b$
$evaluate\ (Subtract\ a\ b) = evaluate\ a - evaluate\ b$
$evaluate\ (Multiply\ a\ b) = evaluate\ a * evaluate\ b$
$evaluate\ (Divide\ a\ b) = evaluate\ a\ `div`\ evaluate\ b$

In Haskell, the two-argument function *div* can be used as an infix operator by surrounding it in back-quotes. Here is a main program that tests evaluation:

```
main = do
  putStrLn "Evaluating the following expression:"
```

```
  putStr "   "
  print t3
  putStrLn "Produces the following result:"
  putStr "   "
  print (evaluate t3)
```

The output is

```
Evaluating the following expression :
  Subtract (Number 3) (Subtract (Number (−2)) (Number (−7)))
Produces the following result :
  − 2
```

This looks pretty good, except that the default *Show* format for expressions is quite ugly.

2.1.3 Formatting Expressions

TODO: Question: is it useful to go into this much detail about formatting at this point?

Another way to interpret abstract *Exp* values is as a string that corresponds to our normal way of writing arithmetic expressions, with binary operators for +, ∗, − and /.

```
instance Show Exp where
  show (Number i)     = show i
  show (Add a b)      = showBinary a "+" b
  show (Subtract a b) = showBinary a "-" b
  show (Multiply a b) = showBinary a "*" b
  show (Divide a b) = showBinary a "/" b
showBinary a op b = show a ++ op ++ show b
```

If you don't know about *instance* declarations in Haskell, please go and read about *type classes*. (TODO: need citation here)

Note that the *show* function for expressions is fairly similar to the *evaluate* function, but it performs string concatenation instead of numeric operations. To test many different kinds of functions, it is useful to define a generalized test function.

```
test fun input = do
  putStr "      "
  putStr (show input)
  putStr " ==> "
  putStrLn (show (fun input))
```

The *test* function takes a function and an input as arguments. It prints the input and then prints the result of applying the function to the input. The following main program invokes *test* to evaluate each of the five sample expressions defined above:

```
main = do
  test evaluate t1
  test evaluate t2
  test evaluate t3
  test evaluate t4
  test evaluate t5
```

Running this main program produces less than satisfactory results:

```
4 ==> 4
-4+6 ==> 2
3--2--7 ==> -2
1*8+5 ==> 13
1+8*2 ==> 17
```

We are back to the ambiguous expressions that we started with. The ambiguity can be resolved by adding parentheses around every expression:

```
showBinary a op b = paren (show a) ++ op ++ paren (show b)

paren str = "(" ++ str ++ ")"
```

But the results are still not very satisfying:

```
4 ==> 4
(-4)+(6) ==> 2
(3)-((-2)-(-7)) ==> -2
```

There are either too many or too few parentheses. The right thing to do is to check whether parentheses are needed, by comparing the *precedence* of an operator with the *precedence* of the operators nested within it. Multiplication $*$ has higher precedence than addition $+$ because $1 + 2 * 3$ means $1 + (2 * 3)$ not $(1 + 2) * 3$. In what follows, addition has precedence 1 and multiplication has precedence 2.

```
instance Show Exp where
  show e = showExp 0 e

showExp level (Number i) = if i < 0 then paren (show i) else show i
showExp level (Add a b)      = showBinary level 1 a " + " b
showExp level (Subtract a b) = showBinary level 1 a " - " b
showExp level (Multiply a b) = showBinary level 2 a "*" b
showExp level (Divide a b)   = showBinary level 2 a "/" b
showBinary outer inner a op b =
```

if *inner* < *outer* **then** *paren result* **else** *result*
  where *result* = *showExp inner a* ++ *op* ++ *showExp inner b*

The add and subtract operators are also modified to include a little more space. This definition produces an appealing result:

```
4 ==> 4
(-4) + 6 ==> 2
3 - (-2) - (-7) ==> -2
1*(8 + 5) ==> 13
1 + 8*2 ==> 17
```

The example of formatting expressions is a concrete illustration of the complexity of dealing with concrete syntax. The formatter converts abstract syntax into readable text. A later chapter presents a *parser* for expressions, which converts text into abstract syntax.

2.1.4 Errors

There are many things that can go wrong when evaluating an expression. In our current, very simple language, the only error that can arise is attempting to divide by zero. These examples are given in the Simple Examples file. For example, consider this small expression:

evaluate (*Divide* (*Number* 8) (*Number* 0))

In this case, the *div* operator in Haskell throws a low-level error, which terminates execution of the program and prints an error message:

∗ ∗ ∗ *Exception* : *divide by zero*

As our language becomes more complex, there will be many more kinds of errors that can arise. For now, we will rely on Haskell to terminate the program when these situations arise, but in Chapter 5 we will investigate how to manage errors within our evaluator.

2.2 Variables

Arithmetic expressions often contain variables in addition to constants. In grade school the first introduction to variables is usually to evaluate an expression where a variable has a specific value. For example, young students learn to evaluate $x + 2$ where $x = 5$.

The rule is to substitute every occurrence of x with the value 5 and the perform the required arithmetic computations.

To program this in Haskell, the first thing needed is to extend the abstract syntax of expressions to include variables. Since the name of a variable "x" can be represented as a string of characters, it is easy to represent variables as an additional kind of expression. The code for the section is given in the Substitute file. The following data definition modifies *Exp* to include a *Variable* case.

```
data Exp = Number Int
  | Add              Exp Exp
  | Subtract Exp Exp
  | Multiply Exp Exp
  | Divide           Exp Exp
  | Variable String -- added
```

An association of a variable x with a value v is called a *binding*, which can be written $x \mapsto v$. Bindings can be represented in Haskell as a pair. For example, the binding of $x \mapsto 5$ can be represented as `("x", 5)`.

2.2.1 Variable Discussion

We are used to calling x and y "variables" without really thinking much about what it means to be a "variable". Intuitively a variable is something that varies. But what does it mean for a name like x to vary? My view on this is that we call them variables because they can have different values in different contexts. For example, the equation πr^2 defines a relationship between several variables, but in the context of a particular word problem, the radius r has a particular value. In any particular context, a variable does *not* vary or change. It has exactly one value that is fixed and constant within that context. A variable can be bound to different values in different contexts, but in a given context the binding of a variable is fixed. In the examples above, the context is indicated by the phrase "where $x = 5$". The same expression, $x + 2$ can be evaluated in different contexts, for example, where $x = 7$.

This interplay between being constant and being variable can be quite confusing, especially since variables in most programming languages *can change* over time. The process of actually changing a variable's value over time, within a single context, is called *mutation*. This seems like a major difference between programming language variables and mathematical variables. However, if you think about things in a slightly different way then it is possible to unify these two apparently conflicting meanings for "variable". As a preview, we will keep the idea of variables having a fixed binding, but introduce the concept of a *mutable container* that can change over time. The variable will then be bound to the container. The variable's binding will not change (it will remain bound to the same container), but the contents of the container will change.

Mutatable variables are discussed in detail later (TODO: reference). For now, just remember that a variable has a fixed binding to a value in a given context.

Note that another common use for variables is to define *equations* or *constraints*. In this case, it is normal to use algebraic operations to simplify or *solve* the equation to find a value for the variable that satisfies the equation. While equation solving and constraints are fascinating topics, we will not discuss them directly in these notes. For our purposes, we will assume that we already know the value of the variable, and that the problem is to compute a result using that value.

2.3 Substitution

Substitution replaces a variable with a value in an expression. Here are some examples of substitution:

- substitute $x \mapsto 5$ in $x + 2 \longrightarrow 5 + 2$
- substitute $x \mapsto 5$ in $2 \longrightarrow 2$
- substitute $x \mapsto 5$ in $x \longrightarrow 5$
- substitute $x \mapsto 5$ in $x * x + x \longrightarrow 5 * 5 + 5$
- substitute $x \mapsto 5$ in $x + y \longrightarrow 5 + y$

Note that if the variable names don't match, they are left alone. Given these data types, the process of *substitution* can be defined by cases. The following Haskell function implements this behavior:

```
substitute1 :: (String, Int) → Exp → Exp
substitute1 (var, val) exp = subst exp where
  subst (Number i)     = Number i
  subst (Add a b)      = Add (subst a) (subst b)
  subst (Subtract a b) = Subtract (subst a) (subst b)
  subst (Multiply a b) = Multiply (subst a) (subst b)
  subst (Divide a b)   = Divide (subst a) (subst b)
  subst (Variable name) = if var ≡ name
    then Number val
    else Variable name
```

The *subst* helper function is introduced avoid repeating the *var* and *val* parameters for each of the specific cases of substitution. The *var* and *val* parameters are the same for all substitutions within an expression.

The first case says that substituting a variable for a value in a literal expression leaves the literal unchanged. The next three cases define substitution on binary operators as recursively substituting into the sub-expressions of the operator. The final case is

the only interesting one. It defines substitution into a *Variable* expression as a choice: if the variable in the expression (*name*) is the *same* as the variable being substituted (*var*) then the value is

```
x = Variable "x"
y = Variable "y"
main = do
  test (substitute1 ("x", 5)) (Add x (Number 2))
  test (substitute1 ("x", 5)) (Number 2)
  test (substitute1 ("x", 5)) x
  test (substitute1 ("x", 5)) (Add (Multiply x x) x)
  test (substitute1 ("x", 5)) (Add x y)
```

Running these tests produces the following results:

```
x + 2 ==> 5 + 2
2 ==> 2
x ==> 5
x*x + x ==> 5*5 + 5
x + y ==> 5 + y
```

It is important to keep in mind that there are now two stages for evaluating an expression containing a variable. The first stage is to *substitute* the variable for its value, then the second stage is to *evaluate* the resulting arithmetic expression.

TODO: talk about *renaming* variables, or substituting one variable for another

2.3.1 Multiple Substitution using Environments

There can be multiple variables in a single expression. For example, evaluating $2 * x + y$ where $x = 3$ and $y = -2$. A collection of bindings is called an *environment*.

Since a binding is represented as a pair, an environment can be represented as a list of pairs. The environment mentioned above would be

```
e1 = [("x", 3), ("y", -1)]
```

The corresponding type is

```
type Env = [(String, Int)]
```

The substitution function is easily modified to work with environments rather than single bindings:

```
substitute :: Env -> Exp -> Exp
substitute env exp = subst exp where
```

```
subst (Number i)    = Number i
subst (Add a b)     = Add (subst a) (subst b)
subst (Subtract a b) = Subtract (subst a) (subst b)
subst (Multiply a b) = Multiply (subst a) (subst b)
subst (Divide a b)  = Divide (subst a) (subst b)
subst (Variable name) =
  case lookup name env of
    Just val → Number val
    Nothing → Variable name
```

The last case is the only one that is different from the previous definition of substitution for a single binding. It uses the *lookup* function to search the list of bindings to find the corresponding value (*Just val*) or *Nothing* if the variable is not found. For the *Nothing* case, the substitute function leaves the variable alone.

```
z = Variable "z"
main = do
  test (substitute e1) (Add x y)
  test (substitute e1) (Number 2)
  test (substitute e1) x
  test (substitute e1) (Add (Multiply x x) x)
  test (substitute e1) (Add x (Add (Multiply (Number 2) y) z))
```

The test results show that multiple variables are substituted with values, but that unknown variables are left intact:

```
x + y ==> 3 + (-1)
2 ==> 2
x ==> 3
x*x + x ==> 3*3 + 3
x + 2*y + z ==> 3 + 2*(-1) + z
```

Note that it is also possible to substitute multiple variables one at a time:

```
substitute1R env exp = foldr substitute1 exp env
```

The *foldr fun init list* function applies a given function to each item in a list, starting with a given initial value.

2.3.2 Local Variables

So far all variables have been defined *outside* the expression itself. It is also useful to allow variables to be defined *within* an expression. Most programming languages support this capability by allowing definition of *local variables*.

In C or Java one can define local variables in a declaration:

```
int x = 3;
return 2*x + 5;
```

JavaScript is similar but does not specify the type of the variable:

```
var x = 3;
return 2*x + 5;
```

Haskell defines local variables with a **let** expression:

$$\textbf{let } x = 3 \textbf{ in } 2 * x + 5$$

In Haskell **let** is an expression, because it can be used inside other expressions:

$$2 * (\textbf{let } x = 3 \textbf{ in } x + 5)$$

TODO: note that **where** in Haskell is similar to **let**.

It is also possible to define multiple local variables in Java or C:

```
int x = 3;
int y = x*2;
return x + y;
```

and in Haskell

$$\textbf{let } x = 3 \textbf{ in let } y = x * 2 \textbf{ in } x + y$$

which is equivalent to

$$\textbf{let } x = 3 \textbf{ in } (\textbf{let } y = x * 2 \textbf{ in } x + y)$$

In general a **let** expression has the following concrete syntax:

let *variable* = *bound-expression* **in** *body*

The meaning of a **let** expression is to evaluate the bound expression, then bind the local variable to the resulting value, and then evaluate the body of the expression

In Haskell, a **let** expression can be represented by adding another case to the definition of expressions:

data *Exp* = ...
| *Let String Exp Exp*

where the string is the variable name, the first Exp is the bound expression and the second expression is the body.

2.3.3 Scope

The *scope* of a variable is the portion of the text of a program in which a variable is defined. Normally the scope of a local variable is all of the body of the let in which the variable is defined. However, it is possible for a variable to be redefined, which creates a hole in the scope of the outer variable:

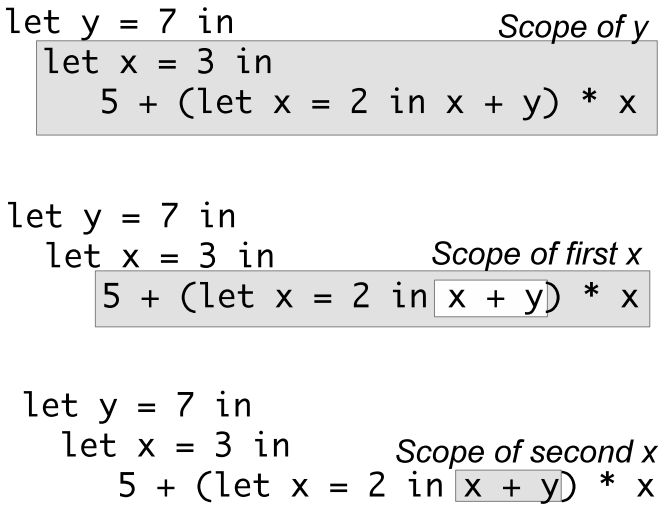

Figure 2.2: Variable Scope

In this example (Figure 2.2) there are two variables named x. Even though two variables have the same name, they are not the same variable.

TODO: talk about *free* versus *bound* variables

TODO: talk about renaming

2.3.4 Substituting into *Let* Expressions

When substituting a variable into an expression, care must be taken to correctly deal with holes in the variable's scope. In particular, when substituting for x in an expression, if the expression is of the form **let** $x = e$ **in** *body* then x should be substituted within e but not in *body*. Because x is redefined, the *body* is a hole in the scope of x.

$$
\begin{array}{l}
\mathit{substitute1}\ (\mathit{var}, \mathit{val})\ \mathit{exp} = \mathit{subst}\ \mathit{exp} \\
\quad \ldots \\
\quad \mathit{subst}\ (\mathit{Let}\ x\ \mathit{exp}\ \mathit{body}) = \mathit{Let}\ x\ (\mathit{subst}\ \mathit{exp})\ \mathit{body}' \\
\quad\quad \textbf{where}\ \mathit{body}' = \textbf{if}\ x \equiv \mathit{var} \\
\quad\quad\quad \textbf{then}\ \mathit{body} \\
\quad\quad\quad \textbf{else}\ \mathit{subst}\ \mathit{body}
\end{array}
$$

In the *Let* case for *subst*, the variable is always substituted into the bound expression e. But the substitution is only performed on the body b if the variable *var* being substituted is *not* the same as the variable x defined in the let expression.

TODO: need some test cases here

2.3.5 Evaluating *Let* Expressions using Substitution

The evaluation of a let expression is based on substitution. To evaluate **let** $x = e$ **in** b, first evaluate the bound expression e, then substitute its value for variable x in the body b. Finally, the result of substitution is evaluated.

$$\begin{aligned}&\mathit{evaluate} :: \mathit{Exp} \to \mathit{Int}\\ &\ldots\\ &\mathit{evaluate}\ (\mathit{Let}\ x\ \mathit{exp}\ \mathit{body}) = \mathit{evaluate}\ (\mathit{substitute1}\ (x, \mathit{evaluate}\ \mathit{exp})\ \mathit{body})\end{aligned}$$

There is no rule for evaluating a variable because all variables are substituted away before evaluation begins.

TODO: need some test cases here

2.3.6 Undefined Variable Errors

With the introduction of variables into our language, a new kind of error can arise: attempting to evaluate an expression containing a variable that does not have a value. For example, these expressions all contain undefined variables:

$$\begin{aligned}&x + 3\\ &\textbf{let}\ x = 2\ \textbf{in}\ x * y\\ &(\textbf{let}\ x = 3\ \textbf{in}\ x) * x\end{aligned}$$

What will happen when these expressions are evaluated? The definition of *evaluate* does not include a case for evaluating a variable. This is because all variables should be substituted for values before evaluation takes place. If a variable is not substituted then it is undefined. Since no case is defined for *evaluate* of a *Variable*, Haskell terminates the program and prints this error message:

$$***\mathit{Exception} : \mathit{anatomy} \circ \mathit{lhs} : \mathit{Non} - \mathit{exhaustive\ patterns}\ \textbf{in function}\ \mathit{evaluate}$$

The fact that a variable is undefined is a *static* property of the program: whether a variable is undefined depends only on the text of the program, not upon the particular data that the program is manipulating. (TODO: accurate example of static versus dynamic?) This is different from the divide by zero error, which depends upon the particular data that the program is manipulating. As a result, divide by zero is a *dynamic* error. Of course, it might be possible to identify, just from examining the

text of a program, that it will always divide by zero. Alternatively, it may be the case that the code containing an undefined variable is never executed at runtime. Thus the boundary between static and dynamic errors is not absolute. The issue of static versus dynamic properties of programs is discussed in more detail later (TODO: reference to chapter on Types).

2.3.7 Summary

Here is the full code evaluation using substitution of a language with local variables.

```
data Exp = Number Int
   | Add          Exp Exp
   | Subtract     Exp Exp
   | Multiply     Exp Exp
   | Divide       Exp Exp
   | Variable     String
   | Let          String Exp Exp


substitute1 (var, val) exp = subst exp where
  subst (Number i)    = Number i
  subst (Add a b)     = Add (subst a) (subst b)
  subst (Subtract a b) = Subtract (subst a) (subst b)
  subst (Multiply a b) = Multiply (subst a) (subst b)
  subst (Divide a b)  = Divide (subst a) (subst b)
  subst (Variable name) = if var ≡ name
                          then Number val
                          else Variable name
  subst (Let x exp body) = Let x (subst exp) body'
    where body' = if x ≡ var
      then body
      else subst body


evaluate :: Exp → Int
evaluate (Number i)    = i
evaluate (Add a b)     = evaluate a + evaluate b
evaluate (Subtract a b) = evaluate a − evaluate b
evaluate (Multiply a b) = evaluate a ∗ evaluate b
evaluate (Divide a b)  = evaluate a `div` evaluate b
evaluate (Let x exp body) = evaluate (substitute1 (x, evaluate exp) body)
```

2.4 Evaluation using Environments

For the basic evaluator substitution and evaluation were completely separate, but the evaluation rule for **let** expressions involves substitution.

One consequence of this rule is that the body of every let expression is copied, because substitution creates a copy of the expression with variables substituted. When let expressions are *nested*, the body of the inner let expression is copied multiple times. In the following example, the expression $x * y * z$ is copied three times:

$$\begin{array}{l}\textbf{let } x = 2 \textbf{ in} \\ \quad \textbf{let } y = x + 1 \textbf{ in} \\ \quad\quad \textbf{let } z = y + 2 \textbf{ in} \\ \quad\quad\quad x * y * z\end{array}$$

The steps are as follows:

Step	Result
initial expression	$\textbf{let } x = 2 \textbf{ in}$ $\quad \textbf{let } y = x + 1 \textbf{ in}$ $\quad\quad \textbf{let } z = y + 2 \textbf{ in } x * y * z$
evaluate bound expression	$2 \Rightarrow 2$
substitute x $\mapsto$ 2 in body	$\textbf{let } y = 2 + 1 \textbf{ in } (\textbf{let } z = y + 2 \textbf{ in } 2 * y * z)$
evaluate bound expression	$2 + 1 \Rightarrow 3$
substitute y $\mapsto$ 3 in body	$\textbf{let } z = 3 + 2 \textbf{ in } 2 * 3 * z$
evaluate bound expression	$3 + 2 \Rightarrow 5$
substitute z $\mapsto$ 5 in body	$2 * 3 * 5$
evaluate body	$2 * 3 * 5 \Rightarrow 30$

While this is a reasonable approach it is not efficient. We have already seen that multiple variables can be substituted at the same time. Rather than performing the substitution fully for each **let** expression, instead the **let** expression can add another binding to the list of substitutions being performed.

```
-- Evaluate an expression in an environment
evaluate :: Exp -> Env -> Int
evaluate exp env = eval exp where
  eval (Number i)   = i
  eval (Add a b)    = eval a + eval b
```

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Subtract}\ a\ b) = \mathit{eval}\ a - \mathit{eval}\ b \\
&\mathit{eval}\ (\mathit{Multiply}\ a\ b) = \mathit{eval}\ a * \mathit{eval}\ b \\
&\mathit{eval}\ (\mathit{Divide}\ a\ b) \quad = \mathit{eval}\ a\ `\mathit{div}`\ \mathit{eval}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Variable}\ x) = \mathit{fromJust}\ (\mathit{lookup}\ x\ \mathit{env}) \\
&\mathit{eval}\ (\mathit{Let}\ x\ \mathit{exp}\ \mathit{body}) = \mathit{evaluate}\ \mathit{body}\ \mathit{newEnv} \\
&\quad \textbf{where}\ \mathit{newEnv} = (x, \mathit{eval}\ \mathit{exp}) : \mathit{env}
\end{aligned}
$$

The helper function *eval* is defined in the scope of the *env* argument of the main *evaluate* function. Since the environment *env* does not change in most cases, it is convenient to not have to pass it around on every call to *eval*. Note that the final case, for *Let*, *does* change the environment so it calls *evaluate* rather than *eval*.

The case for *Let* first evaluates the bound expression in the current environment *env*, then it creates a new environment *newEnv* that binds x to the value of the bound expressions. It then evaluates the body b in the new environment *newEnv*.

The steps in evaluation with environments do not copy the expression:

Environment	Evaluation
$\emptyset$	**let** $x = 2$ **in**
	let $y = x + 1$ **in**
	let $z = y + 2$ **in** $x * y * z$
	{ evaluate bound expression 2}
$\emptyset$	$2 \Rightarrow 2$
	{ add new binding for x and evaluate body of let }
$x \mapsto 2$	**let** $y = x + 1$ **in** (**let** $z = y + 2$ **in** $x * y * z$)
	{ evaluate bound expression $x + 1$ }
$x \mapsto 2$	$x + 1 \Rightarrow 3$
	{ add new binding for y and evaluate body of let }
$y \mapsto 3$, $x \mapsto 2$	**let** $z = y + 2$ **in** $x * y * z$
	{ evaluate bound expression $y + 2$ }
$y \mapsto 3$, $x \mapsto 2$	$y + 2 \Rightarrow 5$
	{ add new binding for z and evaluate body of let }
$z \mapsto 5$, $y \mapsto 3$, $x \mapsto 2$	$x * y * z \Rightarrow 70$

In the *Let* case of *eval*, a new environment *newEnv* is created and used as the environ-

ment for evaluation of the body b.

The new environments add the additional bindings to the *front* of the list of environments. Since *lookup* searches an environment list from left to right, it will find the most recent enclosing binding for a variable, and ignore any additional bindings. For example, consider the evaluation of this expression:

$\textbf{let } x = 9 \textbf{ in } (\textbf{let } x = x * x \textbf{ in } x + x)$

Environment	Evaluation
$\emptyset$	$\textbf{let } x = 9 \textbf{ in } (\textbf{let } x = x * x \textbf{ in } x + x)$
	{ evaluate bound expression 9 }
$\emptyset$	$9 \Rightarrow 9$
	{ add new binding for x and evaluate body of let }
$x \mapsto 9$	$\textbf{let } x = x * x \textbf{ in } x + x$
	{ evaluate bound expression $x * x$ }
$x \mapsto 9$	$x * x \Rightarrow 81$
	{ add new binding for x and evaluate body of let }
$x \mapsto 81, x \mapsto 9$	$x + x \Rightarrow 162$

Note that the environment contains two bindings for x, but only the first one is used. Having multiple bindings for the same name implements the concept of 'holes' in the scope of a variable: when a new binding for the same variable is added to the environment, the original binding is no longer accessible.

The old environment is not changed, so there is no need to reset or restore the previous environment. For example, evaluating the following expression creates to extensions of the base environment

$$\begin{array}{l}\textbf{let } x = 3 \textbf{ in} \\ \quad (\textbf{let } y = 3 * x \textbf{ in } 2 + y) + (\textbf{let } z = 7 * x \textbf{ in } 1 + z)\end{array}$$

The first **let** expressions creates an environment $x \mapsto 3$ with a single binding. The next two let expressions create environments

$y \mapsto 9, x \mapsto 3$

$z \mapsto 21, x \mapsto 3$

Internally Haskell allows these two environments to share the definition of the original environment $x \mapsto 3$.

The Haskell function *fromJust* raises an exception if its argument is *Nothing*, which occurs when the variable named by x is not found in the environment *env*. This is where undefined variable errors arise in this evaluator.

TODO: define *exception*?

Exercise 2.1: Multi-variable let expressions

Modify the **let** expression to take a list of bindings, rather than a single one. Modify the *evaluate* function to handle evaluation of multi-variable **let** expressions.

2.5 More Kinds of Data: Booleans and Conditionals

In addition to arithmetic computations, it is useful for expressions to include conditions and also return different kinds of values. Until now our expressions have always returned *Int* results, because they have only performed arithmetic computations. The code for this section is given in the Int Bool file. The type *Value* is defined to support multiple different kinds of values:

```
data Value = Int Int
   | Bool Bool
deriving (Eq, Show)
```

Some example values are *Bool True* and *Int* 3. We will define additional kinds of values, including functions and lists, later. Keep in mind that the first uses of *Int* and *Bool* in this type definition are the *labels* for data variants, while the second uses are *types* that define what kind of data are associated with that data variant.

The abstract syntax of expressions can now be expanded to include operations involving booleans. Some examples are $4 < 10$ and $3 * 10 = 7$. Once booleans are included in the language, it is possible to define a *conditional* expression, with the following concrete syntax:

if *test* **then** *true-part* **else** *false-part*

A conditional expression allows selection of one of two different values based on whether a boolean is true or false. Note that a conditional *expression* is expected to produce a value. This is different from the conditional *statement* found in many languages (most notably C and Java), which executes one of two blocks but does not produce a value. In these languages, conditional expressions are written *test* ? *true-part* : *false-part*. Haskell, however, only has conditional expressions of the kind discussed here.

Given a full set of arithmetic operators, some comparison operators (equality EQ, less than LT, greater than GT, less than or equal LE), plus *and*, *or* and $\neg$ for booleans, it is useful to generalize the abstract syntax to support a general notation for binary

and unary operators. When an expression includes a value it is called a *literal* value. Literals generalize the case of *Number* used above to include constants in an arithmetic expression. The conditional expression is sometimes called a *ternary* operator because it has three arguments. But since there is only one ternary operator, and also because a conditional expression is fairly special, it is included directly as *If* expression. These changes are implemented in the following definition for the abstract syntax *Exp*:

```
data BinaryOp = Add | Sub | Mul | Div | And | Or
  | GT | LT | LE | GE | EQ
  deriving Eq
data UnaryOp = Neg | Not
  deriving Eq
data Exp = Literal Value
  | Unary      UnaryOp Exp
  | Binary     BinaryOp Exp Exp
  | If         Exp Exp Exp
  | Variable   String
  | Let        String Exp Exp
  deriving Eq
```

Evaluation is then defined by cases as before. Two helper functions, *binary* and *unary* (defined below), perform the actual computations for binary and unary operations, respectively.

```
type Env = [(String, Value)]
-- Evaluateanexpressioninanenvironment
evaluate :: Exp → Env → Value
evaluate exp env = eval exp where
  eval (Literal v)       = v
  eval (Unary op a)      = unary op (eval a)
  eval (Binary op a b) = binary op (eval a) (eval b)
  eval (Variable x)      = fromJust (lookup x env)
  eval (Let x exp body) = evaluate body newEnv
    where newEnv = (x, eval exp) : env
```

The conditional expression first evaluates the condition, forces it to be a boolean, and then evaluates either the *then* or *else* expression.

```
eval (If a b c) = if fromBool (eval a)
                  then eval b
                  else eval c
fromBool (Bool b) = b
```

The binary and unary helper functions perform case analysis on the operator and the arguments to compute the result of basic operations.

$$
\begin{aligned}
&\mathit{unary\ Not\ (Bool\ b)} = \mathit{Bool\ (\neg\ b)}\\
&\mathit{unary\ Neg\ (Int\ i)} = \mathit{Int\ (-i)}\\
&\mathit{binary\ Add\ (Int\ a)\ (Int\ b)} = \mathit{Int\ (a + b)}\\
&\mathit{binary\ Sub\ (Int\ a)\ (Int\ b)} = \mathit{Int\ (a - b)}\\
&\mathit{binary\ Mul\ (Int\ a)\ (Int\ b)} = \mathit{Int\ (a * b)}\\
&\mathit{binary\ Div\ (Int\ a)\ (Int\ b)} = \mathit{Int\ (a\ `div`\ b)}\\
&\mathit{binary\ And\ (Bool\ a)\ (Bool\ b)} = \mathit{Bool\ (a \wedge b)}\\
&\mathit{binary\ Or\ \ (Bool\ a)\ (Bool\ b)} = \mathit{Bool\ (a \vee b)}\\
&\mathit{binary\ LT\ \ (Int\ a)\ (Int\ b)} = \mathit{Bool\ (a < b)}\\
&\mathit{binary\ LE\ \ (Int\ a)\ (Int\ b)} = \mathit{Bool\ (a \leqslant b)}\\
&\mathit{binary\ GE\ (Int\ a)\ (Int\ b)} = \mathit{Bool\ (a \geqslant b)}\\
&\mathit{binary\ GT\ (Int\ a)\ (Int\ b)} = \mathit{Bool\ (a > b)}\\
&\mathit{binary\ EQ\ a\ \ \ \ b\ \ \ \ } = \mathit{Bool\ (a \equiv b)}
\end{aligned}
$$

TODO: talk about strictness!

Using the new format, here are the expressions for the test cases given above:

$$
\begin{aligned}
&\mathit{--4}\\
&\mathit{t1} = \mathit{Literal\ (Int\ 4)}\\
&\mathit{---4-6}\\
&\mathit{t2} = \mathit{Binary\ Sub\ (Literal\ (Int\ (-4)))\ (Literal\ (Int\ 6))}\\
&\mathit{--3-(-2)-(-7)}\\
&\mathit{t3} = \mathit{Binary\ Sub\ (Literal\ (Int\ 3))}\\
&\quad \mathit{(Binary\ Sub\ (Literal\ (Int\ (-2)))\ (Literal\ (Int\ (-7))))}\\
&\mathit{--3*(8+5)}\\
&\mathit{t4} = \mathit{Binary\ Mul\ (Literal\ (Int\ 3))}\\
&\quad \mathit{(Binary\ Add\ (Literal\ (Int\ 8))\ (Literal\ (Int\ 5)))}\\
&\mathit{--3+8*2}\\
&\mathit{t5} = \mathit{Binary\ Add\ (Literal\ (Int\ 3))}\\
&\quad \mathit{(Binary\ Mul\ (Literal\ (Int\ 8))\ (Literal\ (Int\ 2)))}
\end{aligned}
$$

In addition, new expressions can be defined to represent conditional expressions:

$$
\begin{aligned}
&\mathit{--if3 > 3*(8+5)then1else0}\\
&\mathit{t6} = \mathit{If\ (Binary\ GT\ (Literal\ (Int\ 3))\ t4)}\\
&\quad \mathit{(Literal\ (Int\ 1))}\\
&\quad \mathit{(Literal\ (Int\ 0))}\\
&\mathit{--2+(if3 <= 0then9else-5)}\\
&\mathit{t7} = \mathit{Binary\ Add\ (Literal\ (Int\ 2))}\\
&\quad \mathit{(If\ (Binary\ LE\ (Literal\ (Int\ 3))}\\
&\qquad \mathit{(Literal\ (Int\ 0)))}\\
&\qquad \mathit{(Literal\ (Int\ 9))}\\
&\qquad \mathit{(Literal\ (Int\ (-5))))}
\end{aligned}
$$

Running these test cases with the *test* function defined above yields these results:

```
4 ==> 4
(-4) - 6 ==> (-10)
3 - (-2) - (-7) ==> (-2)
1*(8 + 5) ==> 13
3 + 8*2 ==> 17
if 3 > 1*(8 + 5) then 1 else 0 ==> 0
2 + (if 3 <= 0 then 9 else (-5)) ==> (-3)
```

2.5.1 Type Errors

Now that our language supports two kinds of values, it is possible for an expression to get *type errors*. A type error occurs when evaluation of an expression attempts to perform an operation but one or more of the values involved are not of the right type. For example, attempting to add an integer and a boolean value, as in $3 + True$, leads to a type error.

In our Haskell program, type errors exhibit themselves in the *binary* and *unary* functions, which match certain legal patterns of operations, but leave illegal combinations of operations and arguments undefined. Attempting to evaluate $3 + True$ results in a call to *binary Add* (*Int* 3) (*Bool True*), which is not one of the patterns handled by the *binary* function. As a result, Haskell generates a *Non-exhaustive pattern* error:

$$Main > evaluate\ [\,]\ (Binary\ Add\ (Literal\ (Int\ 3))\ (Literal\ (Bool\ True)))$$
$$***Exception: Non-exhaustive\ patterns\ \textbf{in function}\ binary$$

Here are some examples of expression that generate type errors:

$$--if3then5else8$$
$$err1 = If\ (Literal\ (Int\ 3))\ (Literal\ (Int\ 5))\ (Literal\ (Int\ 8))$$
$$--3 + True$$
$$err2 = Binary\ Add\ (Literal\ (Int\ 3))\ (Literal\ (Bool\ True))$$
$$--3|True$$
$$err3 = Binary\ Or\ (Literal\ (Int\ 3))\ (Literal\ (Bool\ True))$$
$$---True$$
$$err4 = Unary\ Neg\ (Literal\ (Bool\ True))$$

We will discuss techniques for preventing type errors later, but for now it is important to realize that programs may fail at runtime.

Chapter 3

Functions

Functions are familiar to any student of mathematics. The first hint of a function in grade school may be some of the standard operators that are introduced early in the curriculum. Examples include absolute value $\mid x \mid$ and square root $\sqrt{x}$. The concept of a function is also implicit in the standard equation for a line $y = mx + b$. Trigonometry introduces the standard functions *sin(a)* and *cos(a)* to support computation on angles. While these operators use more traditional function syntax, they are still considered predefined computations, much like absolute value or square root. However, the concept of a function as an explicit object of study is not usually introduced until calculus.

Programming languages all support some form of function definition. A function allows a computation to be written down once and reused many times.

TODO: explain why this is about "first-order" and "top-level" functions.

3.1 Top-Level Function Definitions

Some programming languages, including C and ACL2, allow functions to be defined only at the top level of the program. The "top level" means outside of any expression. In this case, the program itself is a list of function definitions followed by a main expression. The main expression in a C program is an implicit call to a function named *main*. Even if a programming language does support more flexible definition of functions, top-level functions are quite common. The code for this section is given in the Top Level Functions file. Here is an example of some top-level functions, written in JavaScript:

```
// compute n raised to the m-th power
function power(n, m) {
  if (m == 0)
    return 1;
```

```
  else
    return n * power(n, m - 1);
}

function main() {
  return power(3, 4);
}
```

This code resembles C or Java, but without types. Our expression language does not need *return* statements, because every expression automatically returns a value. A similar program can be written in Haskell, also without return statements:

$$
\begin{array}{l}
power\ (n, m) = \\
\quad \textbf{if}\ (m \equiv 0)\ \textbf{then} \\
\quad\quad 1 \\
\quad \textbf{else} \\
\quad\quad n * power\ (n, m - 1) \\
main = -- notreallyavalidHaskellmainfunction \\
\quad power\ (3, 4)
\end{array}
$$

These examples provides an outline for the basic concrete syntax of a function:

function *function-name* (*parameter-name*, ..., *parameter-name*) *body-expression*

The exact syntax varies from language to language. Some languages begin with a keyword **function** or *def*. Other languages require brackets $\lambda\{$ ... $\lambda\}$ around the body of the function. These functions are less powerful than Haskell, because they take a simple parameter list rather than a full pattern. But this simple form of function defined above captures the essence of function definition in many languages.

A call to a function is an expression that has the following concrete syntax:

function-name (*expression*, ..., *expression*)

Again, there are some variations on this theme. For example, in Haskell the parentheses are optional. The program has a series of named functions, each of which has a list of parameter names and a body expression. The following data type definitions provide a means to represent such programs:

$$
\begin{array}{l}
\textbf{type}\ FunEnv = [(String, Function)] \\
\textbf{data}\ Function = Function\ [String]\ Exp
\end{array}
$$

A list of function definitions is a *function environment*. This list represents a list of bindings of function names to function definitions.

A program is then a function environment together with a main expression:

$$\textbf{data}\ Program = Program\ FunEnv\ Exp$$

Any of the expressions can contain calls to the top-level functions. A call has a function name and a list of actual argument expressions:

```
data Exp = ...
  | Call String [Exp]
```

As an example, here is an encoding of the example program:

```
f1 = Function ["n", "m"]
  (If (Binary EQ (Variable "m") (Literal (Int 0)))
    (Literal (Int 1))
    (Binary Mul
      (Variable "n")
      (Call "power" [Variable "n",
        Binary Sub (Variable "m")
                   (Literal (Int 1))])))
p1 = Program [("power", f1)]
  (Call "power" [Literal (Int 3),
    Literal (Int 4)])
```

3.1.1 Evaluating Top-Level Functions

A new function, *execute*, runs a program. It does so by evaluating the main expression in the context of the programs' function environment and an empty variable environment:

```
execute :: Program → Value
execute (Program funEnv main) = evaluate main [] funEnv
```

The evaluator is extended to take a function environment *funEnv* as a additional argument.

```
evaluate :: Exp → Env → FunEnv → Value
evaluate exp env funEnv = eval exp where
  ...
    eval (Call fun args) = evaluate body newEnv funEnv
      where Function xs body = fromJust (lookup fun funEnv)
        newEnv = zip xs [eval a | a ← args]
```

Evaluation of a call expression performs the following steps:

1. Look up the function definition by name *lookup fun funEnv*, to get the function's parameter list *xs* and *body*.
2. Evaluate the actual arguments [*eval a* | *a* ← *args*] to get a list of values
3. Create a new environment *newEnv* by zipping together the parameter names with the actual argument values.

4. Evaluate the function *body* in the new environment *newEnv*

TODO: work out an example to illustrate evaluation of functions?

The only variables that can be used in a function body are the parameters of the function. As a result, the only environment needed to evaluate the function body is the new environment created by zipping together the parameters and the actual arguments.

The evaluator now takes two environments as input: one for functions and one for normal variables. A given name is always looked up in one or the other of these two environments, and there is never any confusion about which place to look. The certainty about where to look up a name comes from the the fact that the names appear in completely different places in the abstract syntax:

data *Exp* = ...
  | *Variable String* −− *variablename*
  | *Call String* [*Exp*] −− *functionname*

A variable name is tagged as a *Variable* and a function name appears in a *Call* expression.

Because the names of function and the names of variables are completely distinct, they are said to be in different *namespaces*. The separation of the variable and function namespace is clear in the following (silly) example:

```
function pow(pow)
  if pow <= 0 then
    2
  else
    let pow = pow(pow - 1) in
      pow * pow(pow - 2)
```

This is the same as the following function, in which variables are renamed to be less confusing:

```
function pow(a)
  if a <= 0 then
    2
  else
    let b = pow(a - 1) in
      b * pow(b - 2)
```

When renaming variables, the *functions* are *not* renamed. This is because functions and variables are in separate namespaces.

Another consequence of the separation between variable and function namespaces is that functions can not be passed as arguments to other functions, or returned as values from functions. In the expression *pow* (*pow*) the two uses of *pow* are completely distinct. This is analogous to the concept of a *homonym* in natural languages like English. The exact same word has two completely different meanings, which are distinguished only by context. English has many homonyms, including 'stalk' and 'left'. In our expression language, the first *pow* must mean the function because it appears in front of a parenthesis where a function name is expected, while the second *pow* must be a variable because it appears where an expression is expected.

In this language functions are *not* values. When something is treated specially in a programming language, so that it cannot be used where a any value is allowed, it is called *second class*.

It is worth noting that many of the example functions presented above, including *power* and *pow*, are *recursive*. Recursion is possible because the function definitions can be used in any expression, including in the body of the functions themselves. This means that all functions have *global scope*.

3.1.2 Stack Diagrams

TODO: illustrate how stacks work in languages that don't have first-class functions

3.1.3 Summary

Here is the full code for the evaluator supporting top-level functions definitions.

```
data Exp = Literal Value
   | Unary       UnaryOp Exp
   | Binary      BinaryOp Exp Exp
   | If          Exp Exp Exp
   | Variable    String
   | Let         String Exp Exp
   | Call        String [Exp]
type Env = [(String, Value)]
evaluate :: Exp → Env → FunEnv → Value
evaluate exp env funEnv = eval exp where
  eval (Literal v)       = v
  eval (Unary op a)      = unary op (eval a)
  eval (Binary op a b) = binary op (eval a) (eval b)
  eval (If a b c)        = if fromBool (eval a)
                           then eval b
                           else eval c
```

```
eval (Variable x)      = fromJust (lookup x env)
eval (Let x exp body) = evaluate body newEnv funEnv
  where newEnv = (x, eval exp) : env
eval (Call fun args) = evaluate body newEnv funEnv
  where Function xs body = fromJust (lookup fun funEnv)
    newEnv = zip xs [eval a | a ← args]
```

Exercise 3.1: Stack-based evaluation

Modify the evaluator for top-level functions to use a stack with a list of values, rather than an environment. Use a function to look up the position of a variable in an argument list, then access the corresponding position from the top of the stack.

3.2 First-Class Functions

In the Section on Top-Level Functions, function definitions were defined using special syntax and only at the top of a program. The function names and the variable names are in different namespaces. One consequence of this is that all the expressive power we have built into our language, for local variables, conditionals and even functions, does not work for creating function themselves. If you believe that functions are useful for writing reusable computations, as suggested above, then it should be useful to use functions to create and operate on functions. In this section we rework the concept of functions presented above to integrate them into the language, so that functions are *first-class* values.

Consider the following function definition:

$$f\ (x) = x * 2$$

The intent here is to define f, but it doesn't really say what f is, it only says what f does when applied to an argument. A true definition for f would have the form $f = ...$

Finding a value for f is related the idea of solving equations in basic algebra. For example, consider this equation:

$x^2 = 5$

This means that x is value that when squared equals 5. We can solve this equation to compute the value of x:

$x = \sqrt{5}$

But this involved creating a new concept, the *square root* of a number. We know we have a solution for a variable when the variable appears by itself on the left side of an equation.

The function definition $f\ (x) = x * 2$ is similar. It means that f is a function that when applied to an argument x computes the value $x * 2$. *But we don't have a solution for* f, because f does not appear on the left side of an equation by itself. To 'solve for f' we need some new notation, just the way that the square root symbol $\sqrt{x}$ was introduced to represent a new operation.

3.3 Lambda Notation

The standard solution is to use a *lambda expression*, or *function expression*, which is a special notation for representing a function. Here is a solution for f using a lambda:

$f = \lambda x.\ x * 2$

The symbol λ is the greek letter *lambda*. Just like the symbol $\sqrt{x}$, λ has no inherent meaning, but is assigned a meaning for our purposes. The general form of a function expression is:

$\lambda var.\ body$

This represents a function with parameter *var* that computes a result defined by the *body* expression. The *var* may of course be used within the *body*. In other words, *var* may be free in *body*, but *var* is bound (not free) in $\lambda var.\ body$. A function expression is sometimes called an *abstraction* or a *function abstraction* (TODO: discuss this more later).

Thus $f = \lambda x.\ x * 2$ means that f is defined to be a function of one parameter x that computes the result $x * 2$ when applied to an argument. One benefit of function expressions is that we don't need special syntax to name functions, which was needed in dealing with top-level functions. Instead, we can use the existing **let** expression to name functions, because functions are just another kind of value.

Lambda notation was invented in 1930s by Alonzo Church, who was investigating the foundations of functions. Lambda notation is just one part of the *lambda calculus*, which is an extremely elegant analysis of functions. Lambda calculus has had huge influence on programming languages. We will study the lambda calculus in more detail in a later section, but the basic concepts are introduced here.

3.3.1 Using Lambdas in Haskell

Haskell is based directly on the lambda calculus. In fact, the example illustrating how to "solve" for the function f can be written in Haskell. The Examples file contains the code for the simples examples in this section, while the Function Examples file contains the more complex examples given in the subsections below. The following definitions are all equivalent in Haskell:

$$f\ (x) = x * 2$$
$$f\ x = x * 2$$
$$f = \lambda x \rightarrow x * 2$$

The last example uses Haskell's notation for writing a lambda expression. Because λ is not a standard character on most keyboards (and it is not part of ASCII), Haskell uses an *ASCII art* rendition of λ as a backslash λ. The dot used in a traditional lambda expression is replaced by ASCII art $\rightarrow$ for an arrow. The idea is that the function maps from x to its result, so an arrow makes some sense.

The concept illustrated above is an important general rule, which we will call the *Rule of Function Arguments*:

$$name\ var = body \qquad \equiv \qquad name = \lambda var \rightarrow body$$

A parameter can always be moved from the left of an equality sign to the right. Haskell programmers prefer to write them on the left of the equals if possible, thus avoiding explicit use (and somewhat ugly ASCII encoding) of lambdas. Technically in Haskell the *var* can be any pattern, but for now we will focus on the case where the pattern is just a single variable. (TODO: see later chapter?) Since every function definition in Haskell is implicitly a lambda expression, you have already been using lambdas without realizing it. As the old dishwashing soap commercial said "You are soaking in it."

3.3.2 Function Calls

A function call in Haskell is represented by placing one expression next to another expression. Placing two expressions next to each other is sometimes called *juxtaposition*. It is useful to think of juxtaposition as an operator much like + and $*$. The only difference is that juxtaposition is the *invisible* operator. In other words, just as $n + m$ means addition, $f\ n$ means function call. This is not to say that the space character is an operator, because the space is only needed to separate the two characters, which otherwise would be a single symbol *fn*. It is legal to add parenthesis, yielding the more traditional function call syntax, $f\ (n)$, just as it is legal (but useless) to add parentheses to $n + (m)$. A function call in Haskell can also be written as $(f)\ n$ or $(f)\ (n)$. There are no spaces in these examples, but they do exhibit juxtaposition of two expressions.[1]

Haskell has the property that definitions really are equations, so that it is legal to substitute f for $\lambda x \rightarrow x * 2$ anywhere that f occurs. For example, we normally perform a function call $f\ (3)$ by looking up the definition of f and then evaluating the body of the function in the normal way. However, it is also legal to substitute f for its definition.

[1] Church's original presentation of the lambda calculus followed the mathematical convention that all variables were single characters. Thus xy means a function call, $x\ y$, just as xy is taken to mean $x * y$ in arithmetic expressions. Normally in computer science we like to allow variables to have long names, so xy would be the name of a single variable. We don't like it when *foo* means $f\ o\ o$, which in Haskell means $(f\ (o))\ (o)$.

$-- \ versionA$
$f\ (3)$

In this form, the function f is *applied* to the argument 3. The expression $f\ (3)$ is called a function *application*. In this book I use "function call" and "function application" interchangeably.

$-- \ versionB$
$(\lambda x \rightarrow x * 2)\ (3)$

The A and B versions of this expression are equivalent. The latter is a juxtaposition of a function expression $\lambda x \rightarrow x * 2$ with its argument, 3. When a function expression is used on its own, without giving it a name, it is called an *anonymous function*.

The *Rule of Function Invocation* says that applying a function expression to an argument is evaluated by substituting the argument in place of the function's bound variable everywhere it occurs in the body of the function expression.

Rule of Function Invocation (informal):

$(\lambda var.\ body)$arg **evaluates to** *body* with *arg* substituted for *var*

For now this is an informal definition. We will make it more precise when we write an evaluator that handles function expressions correctly.

3.4 Examples of First-Class Functions

Before we begin a full analysis of the semantics of first-class functions, and subsequently implementing them in Haskell, it is useful to explore some examples of first-class functions. Even if you have used first-class functions before, you might find these examples interesting.

3.4.1 Function Composition

One of the simplest examples of a using functions as values is defining a general operator for *function composition*. The composition $f \circ g$ of two functions f and g is a new function that first performs g on an input, then performs f on the result. Composition can be defined in Haskell as:

$compose\ f\ g = \lambda x \rightarrow f\ (g\ x)$

The two arguments are both functions, and the result of composition is also a function. The type of *compose* is

$compose :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$

As an example of function composition, consider two functions that operate on numbers:

$$square\ n = n * n$$
$$mulPi\ m = pi * m$$

Now using composition we can define a function for computing the area of a circle, given the radius:

$$areaR = compose\ mulPi\ square$$

To compute the area given the diameter, we can compose this function with a function that divides by two:

$$areaD = compose\ areaR\ (\lambda x \rightarrow x\ /\ 2)$$

3.4.2 Mapping

One of the earliest and widely cited examples of first class functions is in the definition of a *map* function, which applies a function to every element of a list, creating a new list with the results.

For example, given the standard Haskell function *negate* that inverts the sign of a number, it is easy to quickly negate a list of numbers:

$$map\ negate\ [1, 3, -7, 0, 12]$$
$$--\ returns [-1, -3, 7, 0, -12]$$

The *map* function takes a function as an argument. You can see that *map* takes a function argument by looking at its type:

$$map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

The first argument $a \rightarrow b$ is a function from a to b where a and b are arbitrary types.

Personally, I tend to use list comprehensions rather than *map*, because list comprehensions give a nice name to the items of the list. Here is an equivalent example using comprehensions:

$$[negate\ n \mid n \leftarrow [1, 3, -7, 0, 12]]$$
$$--\ returns [-1, -3, 7, 0, -12]$$

A function that takes another function as an input is called a *higher-order function*. Higher-order functions are quite useful, but what I find even more interesting are functions that *return* functions as results.

The comprehensions used earlier in this document could be replace by invocations of *map*:

$$[eval\ a \mid a \leftarrow args] \quad \equiv \quad map\ eval\ args$$

TODO: make a comment about point-free style?

TODO: is a function that returns a function also called higher order?

3.4.3 Representing Environments as Functions

In Chapter 1, an environment was defined as a list of bindings. However, it is often useful to consider the *behavior* of a concept rather than its concrete *representation*. The purpose of a environment is to map variable names to values. A map is just another name for a function. Thus it is very reasonable to think of an environment as a *function* from names to values. Consider the environment

$$\textbf{type } \mathit{EnvL} = [(\mathit{String}, \mathit{Value})]$$
$$\mathit{envL1} = [(\texttt{"x"}, \mathit{Int}\ 3), (\texttt{"y"}, \mathit{Int}\ 4), (\texttt{"size"}, \mathit{Int}\ 10)]$$

Since environments always have a finite number of bindings, it is more precise to say that an environment is a *partial function* from names to values. A partial function is one that produces a result for only some of its inputs. One common way to implement partial functions in Haskell is by using the *Maybe* type, which allows a function to return a value (tagged by *Just*) or *Nothing*. Here is an implementation of the same environment as a function:

$$\textbf{type } \mathit{EnvF} = \mathit{String} \to \mathit{Maybe}\ \mathit{Value}$$

$$\mathit{envF1}\ \texttt{"x"} = \mathit{Just}\ (\mathit{Int}\ 3)$$
$$\mathit{envF1}\ \texttt{"y"} = \mathit{Just}\ (\mathit{Int}\ 4)$$
$$\mathit{envF1}\ \texttt{"size"} = \mathit{Just}\ (\mathit{Int}\ 10)$$
$$\mathit{envF1}\ \_ \quad = \mathit{Nothing}$$

Looking up the value of a variable in either of these environments is quite different:

$$x1 = \mathit{lookup}\ \texttt{"x"}\ \mathit{envL1}$$
$$x2 = \mathit{envF1}\ \texttt{"x"}$$

The *lookup* function searches a list environment *envL1* for an appropriate binding. An functional environment *envF1* is applied to the name to get the result. One benefit of the function environment is that we don't need to know how the bindings are represented. All we need to do is call it to get the desired answer.[2] There is no need to use a *lookup* function, because the functional environment *is* the lookup function.

The only other thing that is done with an environment is to extend it with additional bindings. Let's define bind functions that add a binding to an environment, represented as lists or functions. For lists, the *bindL* function creates a binding (val, val) and then prepends it to the front of the list:

[2]This kind of behavioral representation will come again when we discuss object-oriented programming.

$bindL :: String \to Value \to EnvL \to EnvL$
$bindL\ var\ val\ env = (var, val) : env$

Since *lookup* searches lists from the front, this new binding can shadow existing bindings.

$envL2 = bindL$ `"z"` $(Int\ 5)\ envL1$
$\quad -- [("z", Int5), ("x", Int3), ("y", Int4), ("size", Int10)]$
$envL3 = bindL$ `"x"` $(Int\ 9)\ envL1$
$\quad -- [("x", Int9), ("x", Int3), ("y", Int4), ("size", Int10)]$

To extend an environment expressed as a partial function, we need to write a *higher-order* function. A higher-order function is one that takes a function as input or returns a function as an result. The function *bindF* takes an *EnvF* as an input and returns a new *EnvF*.

$bindF :: String \to Value \to EnvF \to EnvF$

Expanding the definition of *EnvF* makes the higher-order nature of *bindF* clear:

$bindF :: String \to Value \to (String \to Maybe\ Int) \to (String \to Maybe\ Int)$

The definition of *bindF* is quite different from *bindL*:

$bindF\ var\ val\ env = \lambda testVar \to$ **if** $testVar \equiv var$
then $Just\ val$
else $env\ testVar$

Understanding how this function works takes a little time. The first thing to keep in mind is that *env* is a function. It is a function representing an environment, thus it has type $EnvF = String \to Maybe\ Int$. The other arguments, *var* and *val* are the same as for *bindL*: a string and an integer.

The second thing to notice is that the return value (the expression on the right side of the = sign) is a function expression $\lambda testVar \to ....$ That means the return value is a function. The argument of this function is named *testVar* and the body of the function is a conditional expression. The conditional expression checks if *testVar* is equal to *var*. It returns *val* if they are equal, and otherwise it calls the function *env* with *testVar* as an argument.

The key to understanding how this works is to keep in mind that there are two very different *times* or *contexts* involved in *bindF*. The first time is when an environment is being extended with a new binding. At this time the arguments *var*, *val*, and *env* are determined. The second important time is when the newly extended environment is searched for a particular variable. This is when *testVar* is bound. Since the environment can be searched many times, *testVar* will be bound many times. Consider a specific example:

$-- versionA$
$envF2 = bindF$ `"z"` $(Int\ 5)\ envF1$

Let's execute this program manually. The call to *bindF* has three arguments, creating these bindings: $var \mapsto$ `"z"`, $val \mapsto 5$, $env \mapsto envF1$. Substituting these bindings into the definition of *bindF* gives

```
-- versionB
envF2 = λtestVar → if testVar ≡ "z"
  then Just (Int 5)
  else envF1 testVar
```

This makes more sense! It says that *envF2* is a function that takes a variable name as an argument. It first tests if the variable is named z and if so it returns 5. Otherwise it returns what *envF1* returns for that variable. Another way to write this function is

```
-- versionC
envF2 "z" = Just (Int 5)
envF2 testVar = envF1 testVar
```

These two versions are the same because of the way Haskell deals with functions defined by cases: it tries the first case (argument == `"z"`), else it tries the second case. Since *bindF* tests for the most recently bound variable first, before calling the base environment, variables are properly shadowed when redefined.

It is also useful to consider the *empty* environment for both list and function environments.

```
emptyEnvL :: EnvL
emptyEnvL = []

emptyEnvF :: EnvF
emptyEnvF = λvar → Nothing
```

The empty function environment *emptyEnvF* is interesting: it maps every variable name to *Nothing*.

In conclusion, functions can be used to represent environments. This example illustrates passing a function as an argument as well as returning a function as a value. The environment-based evaluators for expressions and top-level functions could be easily modified to use functional environments rather than lists of bindings. For example, the environment-based evaluation function becomes:

```
-- Evaluateanexpressionina(functional)environment
evaluateF :: Exp → EnvF → Value
evaluateF exp env = eval exp where
  eval (Literal v)      = v
  eval (Unary op a)     = unary op (eval a)
  eval (Binary op a b) = binary op (eval a) (eval b)
  eval (Variable x)     = fromJust (env x)    -- changed
```

$$eval\ (Let\ x\ exp\ body) = evaluateF\ body\ newEnv$$
$$\quad \textbf{where}\ newEnv = bindF\ x\ (eval\ exp)\ env -- changed$$

The result looks better than the previous version, because it does not have spurious references to list functions *lookup* and :, which are a distraction from the fundamental nature of environments as maps from names to values. It is still OK to think of environments as 'data', because functions are data and this function is being used to represent an environment. In this case it is a functional representation of data. In the end, the line between data and behavior and data is quite blurry.

TODO: define "shadow" and use it in the right places.

3.4.4 Multiple Arguments and Currying

Functions in the lambda calculus always have exactly *one* argument. If Haskell is based on Lambda calculus, how should we understand all the functions we've defined with multiple arguments? The answer is surprisingly subtle. Let's consider a very simple Haskell function that appears to have two arguments:

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

The Rule of Function Arguments for Haskell says that arguments on the left of a definition are short-hand for lambdas. The b argument can be moved to the right hand side to get an equivalent definition:

$$add\ a = \lambda b \rightarrow b + a$$

Now the a argument can also be moved. We have now "solved" for *add*:

$$add = \lambda a \rightarrow \lambda b \rightarrow b + a$$

It's useful to add parentheses to make the grouping explicit:

$$add = \lambda a \rightarrow (\lambda b \rightarrow b + a)$$

What this means is that *add* is a function of one argument a whose return value is the function $\lambda b \rightarrow b + a$. The function that is returned also takes one argument, named b, and finally returns the value of $b + a$. In other words, a function of two arguments is actually a function that takes the first argument and returns a new function that takes the second argument. Even for this simplest case Haskell uses a function returning a function!

One consequence of this arrangement is that it is possible to apply the *add* function to the arguments one at a time. For example applying *add* to just one argument returns a new function:

$$inc = add\ 1 \qquad -- \lambda b + 1$$
$$dec = add\ (-1) -- \lambda b + (-1)$$

These two functions each take a single argument. The first adds one to its argument. The second subtracts one. Here are two examples that use the resulting functions:

$$eleven = inc\ 10$$
$$nine = dec\ 10$$

To see how the definition of *inc* works, we can analyze the function call $add\ 1$ in more detail. Replacing *add* by its definition yields:

$$inc = (\lambda a \rightarrow (\lambda b \rightarrow b + a))\ 1$$

The Rule of Function Invocation says that in this situation, a is substituted for 1 in the body $\lambda b \rightarrow b + a$ to yield:

$$inc = \lambda b \rightarrow b + 1$$

Which is the same (by the Rule of Function Arguments) as:

$$inc\ b = b + 1$$

One way to look at what is going on here is that the two arguments are split into stages. Normally both arguments are supplied at the same time, so the two stages happen simultaneously. However, it is legal to perform the stages at different times. After completing the first stage to create an increment/decrement function, the new increment/decrement function can be used many times.

$$inc\ 5 + inc\ 10 + dec\ 20 + dec\ 100$$

(remember that this means $(inc\ 5) + (inc\ 10) + (dec\ 20) + (dec\ 100)$)

Separation of arguments into different stages is exactly the same technique used in the section on representing environments as functions. The *bindF* function takes three arguments in the first stage, and then returns a function of one argument that is invoked in a second stage. To make it look nice, the first three arguments were listed to the left of the = sign, while the last argument was placed to the right as an explicit lambda. However, this choice of staging is just the intended use of the function. The function could also have been defined as follows:

$$bindF\ var\ val\ env\ testVar = \textbf{if}\ testVar \equiv var$$
$$\quad \textbf{then}\ Just\ val$$
$$\quad \textbf{else}\ env\ testVar$$

The ability to selectively stage functions suggests a design principle for Haskell that is not found in most other languages: *place arguments that change most frequently at the end of the argument list.* Conversely, arguments that change rarely should be placed early in the argument list.

TODO: talk about pairs and define curry/uncurry

3.4.5 Church Encodings

Other kinds of data besides environments can be represented as functions. These examples are known as Church encodings.

Booleans

Booleans represent a choice between two alternatives. Viewing the boolean itself as a behavior leads to a view of a boolean as a function that chooses between two options. One way to represent a choice is by a function with two arguments that returns one or the other of the inputs:

$$true\ x\ y = x$$
$$false\ x\ y = y$$

The *true* function returns its first argument. The *false* function returns its second argument. For example *true* 0 1 returns 0 while *false* `"yes"` `"no"` returns `"no"`. One way to write the type for booleans is a generic type:

$$\textbf{type}\ BooleanF = forall\ a \circ a \to a \to a$$
$$true :: BooleanF$$
$$false :: BooleanF$$

Things get more interesting when performing operations on booleans. Negation of a boolean b returns the result of applying b to *false* and *true*. If b is true then it will return the first argument, *false*. If b is false then it will return the second argument, *true*.

$$notF :: BooleanF \to BooleanF$$
$$notF\ b = b\ false\ true$$

The unary function $\neg$ is a higher-order function: it takes a functional boolean as an input and returns a functional boolean as a result. We can also define binary operations on booleans:

$$orF :: BooleanF \to BooleanF \to BooleanF$$
$$orF\ a\ b = a\ true\ b$$

The behavior of "or" is to return true if a is true, and return b if a is false. It works by calling a as a function, passing true and b as arguments.

$$andF :: BooleanF \to BooleanF \to BooleanF$$
$$andF\ a\ b = a\ b\ false$$

You get the idea. Calling a with b and false as arguments will return b if a is true and false otherwise.

To use a Church boolean, the normal syntax for **if** expressions is completely unnecessary. For example,

if $\neg$ *True* **then** 1 **else** 2

is replaced by

$(notF\ true)\ 1\ 2$

This code is not necessarily more readable, but it is concise. In effect a Church boolean *is* an **if** expression: it is a function that chooses one of two alternatives.

Natural Numbers

Natural numbers can also be represented functionally. The Church encoding of natural numbers is known as *Church Numerals*. The idea behind Church Numerals is related to the Peano axioms of arithmetic. The Peano axioms define a constant 0 as the *first* natural number and a *successor* function, *succ*. *succ* takes a natural number and returns the *next* natural number. For example,

$1 = succ(0)$

$2 = succ(1) = succ(succ(0))$

$3 = succ(2) = succ(succ(succ(0)))$

$n = succ^n(0)$

The last equation uses the notation $succ^n$, which means to apply the successor function n times. Basic arithmetic can be carried out by applying the following relations.

$f^{n+m}(x) = f^n(f^m(x))$

$f^{n*m}(x) = (f^n)^m(x)$

Functionally, we can represent the Church numerals as functions of two arguments, f and x. Thus, a Church numeral is a lambda, not a concrete value like 0 or 1. The Church numeral 0 applies f zero times to x. Similarly, 1 applies f once to x.

$zero = \lambda f \rightarrow \lambda x \rightarrow x$
$one = \lambda f \rightarrow \lambda x \rightarrow f\ x$
$two = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ x)$
$three = \lambda f \rightarrow \lambda x \rightarrow f\ (f\ (f\ x))$

Note that f and x have no restrictions. To demonstrate Church numerals, let us evaluate *three* by setting f to the successor function $(+1)$ and x to 0.

$three\ (+1)\ 0 - - evaluatesto3$

To further demonstrate the flexibility, suppose we want our Church numerals to start with [] as the base value, and our successor function to append the character 'A' to the beginning of the list.

three (`'A'`:) []−− *evaluatesto*"*AAA*"

In Haskell we can write the generic type for Church numerals as

type *ChurchN* = *forall* $a \circ (a \to a) \to a \to a$

If we are given a Haskell *Integer*, we can represent the equivalent Church numeral with the following Haskell definition.

church :: *Integer* → *ChurchN*
church 0 = $\lambda f \to \lambda x \to x$
church n = $\lambda f \to \lambda x \to f$ (*church* $(n-1)$ f x)

To retrieve the *Integer* value of a Church numeral, we can evaluate the lambda using the usual successor and base value.

unchurch :: *ChurchN* → *Integer*
unchurch n = n (+1) 0
−− 5 == (*unchurch*(*church5*)) −−*thisevaluatestoTrue*

We define addition and multiplication in Haskell by using the above arithmetic relations.

plus :: *ChurchN* → *ChurchN* → *ChurchN*
plus n m = $\lambda f \to \lambda x \to n$ f (m f x)
mul :: *ChurchN* → *ChurchN* → *ChurchN*
mul n m = $\lambda f \to n$ (m f)

We can use these functions to produce simple arithmetic equations.

x = *church* 10
y = *church* 5
z = *church* 2
a = *plus* x (*mul* y z)−− *isequivalenttochurch20*

3.4.6 Relationship between Let and Functions

TODO: prove that **let** $x = e$ **in** b is equivalent to $(\lambda x \circ b)e$

Others

There are many other uses of first-class functions, including callbacks, event handlers, thunks, continuations, etc.

3.5 Evaluating First-Class Functions using Environments

Its now time to define the syntax and semantics of a language with first-class functions. Based on the examples in the previous section, some features are no longer needed. For example, `let` expressions are not needed because they can be expressed using functions. Functions only need one argument, because multi-argument functions can be expressed by returning functions from functions.

Evaluation of first-class functions (lambdas) is complicated by the need to properly enforce *lexical scoping* rules. Lexical scope means that a variable refers to the closest enclosing definition of that variable. TODO: move this discussion earlier!

3.5.1 A Non-Solution: Function Expressions as Values

The first idea for achieving "functions are values" is to make function expressions be values. It turns out that this "solution" does not really work. The reason I spend so much time discussing an incorrect solution is that understanding why the obvious and simple solution is wrong helps to motivate and explain the correct solution. This section is colored red to remind you that the solution it presents is *incorrect*. The correct solution is given in the next section, on closures. The code for the incorrect soluiton mentioned here is in the Incorrect Functions file.

To try this approach, function expressions are included in the *Value* data type, which allows functions appears a literal values in a program:

```
data Value = ...
  | Function String Exp-- new
  deriving Eq
```

The two components of a function expression *Function* are the *bound variable String* and the *body expression Exp*. This new kind of value for functions looks a little strange. Its not like the others.

We normally think of values as things that a simple data, like integers, strings, booleans, and dates. Up until now, this is what values have been. Up until now, values have not contained *expressions* in them. On the other hand, we are committed to making functions be values, and the body of a function is necessarily an expression, so one way or the other values are going to contain expressions.

TODO: the call expression discussion is really not part of this *incorrect* solution, so it could be moved out? The only problem is that the code assumes that functions are literals, which is not the code in the correct version. Sigh.

The call expression changes slightly from the version with top-level functions. Instead of the *name* of the function to be called, the *Call* expression now contains an expression *Exp* for both the function and the argument:

```
data Value = Int Int
  | Bool Bool
  | Function String Exp-- new
 deriving (Eq, Show)
```

To create the new value type, the simpler type of basic *Value* is included with the tag *Scalar*. A *scalar* value is a simple basic primitive value.

To clarify the effect of this change, consider these two versions of a simple program, written using top-level functions or first-class functions:

Top-Level Functions (A)	First-Class Functions (B)
function $f\ (x)\ x * x$	**let** $f = \lambda x \circ x * x$ **in**
$f\ (10)$	$f\ (10)$

The explicit abstract syntax for example (A) is:

```
Program
  [("f", Function ["x"]
    (Binary Mul (Variable "x")
      (Variable "x")))]
  (Call "f" [Literal (Int 10)])
```

The explicit abstract syntax for example (B) is:

```
Let "f" (Literal (Function "x"
  (Binary Mul (Variable "x")
    (Variable "x"))))
  (Call (Variable "f") (Literal (Int 10)))
```

Note that the function in the *Call* is string `"f"` in the first version, but is an expression *Variable*"f" in the second version.

In many cases the first expression (the function) will be *variable* that names the function to be called. Since there is no longer any special function environment, the names of functions are looked up in the normal variable environment. (TODO: should this come earlier?) TODO: example where function to be called is not a variable.

The first few cases for evaluation are exactly the same as before. In particular, evaluating a literal value is the same, although now the literal value might be a function.

```
evaluate :: Exp → Env → Value
evaluate exp env = eval exp where
  eval (Literal v) = v
  ...
```

Calling a function works almost the same as the case for function calls in the language with top-level functions. Here is the code:

```
eval (Call fun arg) = evaluate body newEnv
  where Function x body = eval fun
    newEnv = bindF x (eval arg) env
```

To evaluate a function call *Call fun arg*,

1. First evaluate *eval fun* the function *fun* of the call.
2. Use pattern matching to ensure that the result of step 1 is a *Function* value, binding x and *body* to the argument name and body of the function.
3. Evaluate the actual argument (*eval arg*) and then extend the environment *env* with a binding between the function parameter x and the argument value:

   bindF x (eval arg) env
4. Evaluate the *body* of the function in the extended environment *newEnv*:

   evaluate newEnv body

Note that this explanation jumps around in the source code. The explanation follows the sequence of data dependencies in the code: what logically needs to be evaluated first, rather than the order in which expressions are written. Since Haskell is a lazy language, it will actually evaluate the expressions in a completely different order!

The main difference from the case of top-level functions is that the function is computed by calling *eval fun* rather than *lookup fun funEnv*. The other difference is that functions now only have one argument, while we allowed multiple arguments in the previous case.

The key question is: **why doesn't the code given above work?** There are two problems. One has to do with returning functions as values, and the other with passing functions as arguments. They both involve the handling of free variables in the function expression.

Problems with Returning Functions as Values

Let's look at the problem of returning functions as values first. The section on Multiple Arguments showed how a two-argument function could be implemented by writing a

function that takes one argument, but then returns a function that takes the second argument. Here is a small program that illustrates this technique:

let $add = \lambda a \rightarrow (\lambda b \rightarrow b + a)$ **in** $add\ 3\ 2$

This program is encoded in our language as follows:

```
Let "add" (Literal (Function "a"
  (Literal (Function "b"
    (Binary Add (Variable "b")
      (Variable "a"))))))
  (Call (Call (Variable "add")
    (Literal (Int 3)))
    (Literal (Int 2)))
```

Rather than work with the ugly constructor syntax in Haskell, we will continue to use the convention of writing $b + a$ to mean (*Binary Add* (*Variable*"b") (*Variable*"a")).

Here is how evaluation of this sample program proceeds:

1. Evaluate **let** $add = \lambda a \rightarrow (\lambda b \rightarrow b + a)$ **in** $add\ 3\ 2$
2. Bind $add \mapsto \lambda a \rightarrow (\lambda b \rightarrow b + a)$
3. Call $(add\ 3)\ 2$
    a. Call $add\ 3$
    b. Evaluate the variable add, which looks it up in the environment to get $\lambda a \rightarrow (\lambda b \rightarrow b + a)$
    c. Bind $a \mapsto 3$
    d. Return $\lambda b \rightarrow b + a$ as result of $add\ 3$
4. Call $\lambda b \rightarrow b + a$ on argument 2
    a. Bind $b \mapsto 2$
    b. Evaluate $b + a$
    c. Look up b to get 2
    d. Look up a to get... **unbound variable!**

To put this more concisely, the problem arises because the call to $add\ 3$ returns $\lambda b \rightarrow b + a$. But this function expression is not well defined because it has a free variable a. What happened to the binding for a? It had a value in Steps 12 through 14 of the explanation above. But this binding is lost when returning the literal $\lambda b \rightarrow b + a$. The problem doesn't exhibit itself until the function is called.

The problems with returning literal function expressions as values is that bindings for free variables that occur in the function are lost, leading to later unbound variable errors. Again, this problem arises because we are trying to treat function expressions

as *literals*, as if they were number or booleans. But function expressions are different because they contain variables, so care must be taken to avoid losing the bindings for the variables.

Problems with Rebinding Variables

A second problem can arise when passing functions as values. This problem can occur, for example, when composing two functions, mapping a function over a list, or many other situations. Here is a program that illustrates the problem.

$$\begin{array}{l}\textbf{let } k = 2 \textbf{ in} \\ \quad \textbf{let } \mathit{double} = \lambda n \to k * n \textbf{ in} \\ \quad\quad \textbf{let } k = 9 \textbf{ in} \\ \quad\quad\quad \mathit{double}\ k\end{array}$$

The correct answer, which is produced if you run this program in Haskell, is 18. The key point is that k is equal to 2 in the body of *double*, because that occurrence of k is within the scope of the first **let**. Evaluating this function with the evaluator given above produces 81, which is not correct. In summary, the evaluation of this expression proceeds as follows:

1. Bind $k \mapsto 2$
2. Bind $\mathit{double} \mapsto \lambda n \to k * n$
3. Bind $k \mapsto 9$
4. Call $\mathit{double}\ k$
    a. Bind $n \mapsto 9$
    b. Evaluate body $k * n$
    c. Result is 81 given $k = 9$ and $n = 9$

The problem is that when k is looked up in step 4b, the most recent binding for k is 9. This binding is based on the *control flow* of the program, not on the *lexical* structure. Looking up variables based on control flow is called *dynamic binding*.

3.5.2 A Correct Solution: Closures

As we saw in the previous section, the problem with using a function expression as a value is that the bindings of the free variables in the function expression are either lost or may be overwritten. The solution is to *preserve the bindings that existed at the point when the function was defined*. The mechanism for doing this is called a *closure*. A closure is a combination of a function expression and an environment. Rather than think of a function expression as a function value, instead think of it as a part of the

program that *creates* a function. The actual function value is represented by a closure, which captures the current environment at the point when the function expression is executed. The code for this section is given in the First-Class Functions file.

To implement this idea, we revise the definition of *Exp* and *Value*. First we add function expressions as a new kind of expression:

```
data Exp = ....
  | Function String Exp-- new
```

As before, the two components of a function expression are the *bound variable String* and the *body expression Exp*. Function expressions resemble **let** expressions, so they fit in well with the other kinds of expressions.

The next step is to introduce *closures* as a new kind of value. Closures have all the same information as a function expressions (which we previously tried to add as values), but they have one important difference: closures also contain an environment.

```
data Value = Int Int
  | Bool Bool
  | Closure String Exp Env-- new
  deriving (Eq, Show)
```

The three parts of a closure are the *bound variable String*, the *function body Exp*, and *the closure environment Env*. The bound variable and function body are the same as the components of a function expression.

With these data types, we can now define a correct evaluator for first-class functions using environments. The first step is to *create a closure* when evaluating a function expression.

```
--Evaluateanexpressioninanenvironment
evaluate :: Exp → Env → Value
evaluate exp env = eval exp where
  ...
  eval (Function x body) = Closure x body env
```

The resulting closure is the value that represents a function. The function expression *Function x body* is not actually a function itself, it is an expression that *creates* a function when executed. Once a closure value has been created, it can be bound to a variable just like any other value, or passed to a function or returned as the result of a function. Closures are values.

Since closures represent functions, the only thing you can *do* with a closure is *call* it. The case for evaluating a function call starts by analyzing the function call expression, *eval* (*Call fun arg*). This pattern says that call expression has two components: a function *fun* and an argument *arg*. Here is the code for this case:

$$
\begin{aligned}
&\mathit{eval}\ (\mathit{Call}\ \mathit{fun}\ \mathit{arg}) = \mathit{evaluate}\ \mathit{body}\ \mathit{newEnv}\\
&\quad \textbf{where}\ \mathit{Closure}\ x\ \mathit{body}\ \mathit{closeEnv} = \mathit{eval}\ \mathit{fun}\\
&\qquad \mathit{newEnv} = (x, \mathit{eval}\ \mathit{arg}) : \mathit{closeEnv}
\end{aligned}
$$

The code starts by evaluating both the function part *fun* to produce a value. The **where** clause $\mathit{Closure}\ x\ \mathit{body}\ \mathit{newEnv} = \mathit{eval}\ \mathit{fun}$ says that the result of evaluating *fun* must be a closure, and the variables x, *body*, and *newEnv* are bound to the parts of the closure. If the result is not a closure, Haskell throws a runtime error.

Next the environment from the closure *newEnv* is extended to include a new binding $(x, \mathit{eval}\ \mathit{arg})$ of the function parameter to the value of the argument expression. The new environment is called *newEnv*. At a high level, the environment is the same environment that existed when the function was created, together with a binding for the function parameter.

Finally, the *body* of the function is evaluated in this new environment, $\mathit{evaluate}\ \mathit{body}\ \mathit{newEnv}$.

TODO: give an example of how this runs?

Exercise 3.2: Multiple Arguments

Modify the definition of *Function* and *Call* to allow multiple arguments. Modify the *evaluate* function to correctly handle the extra arguments.

3.6 Environment/Closure Diagrams

The behavior of this evaluator is quite complex, but its operation on specific programs can be illustrated by showing all the environments and closures created during its execution, together with the relationships between these structures.

An Environment/Closure Diagram is a picture that shows the closures and environments created during execution of an expression.

- Start State

  Set current environment to empty environment $\emptyset$

- Case $\mathit{Let}\ x\ e\ \mathit{body}$

  1. Draw binding box for x with unknown value
     Set parent of new binding to be the current environment
  2. Create the diagram for bound expression e
     Put the value of e into the binding as the value of x
  3. Set current environment to be the new binding

4. Draw diagram for *body* and remember value
5. Set current environment back to what it was before

- Case *Call fun arg*

1. Draw diagram for *fun*
   Result must be a closure with variable x, *body* and *env*
2. Make binding for argument using name x from closure
   Set parent of new binding to be the environment of the closure *env*
3. Draw diagram for *arg*
   Put the value into the new binding as the value of x
4. Set current environment to be the new binding
5. Draw diagram for *body* and remember value
6. Set current environment back to what it was before

- Case *Var x*

1. Look up the variable in the current environment

- Case *Function x e*

1. Make a closure with variable x, body e
   Set the environment of the closure to be the current environment

3.6.1 Example 1

let $k = 2$ **in**
  let $double = \lambda n \rightarrow k * n$ **in**
    let $k = 9$ **in**
      $double\ k$

3.6.2 Example 2

let $add = \lambda a \rightarrow (\lambda b \rightarrow b + a)$ **in** $(add\ 3)\ 2$

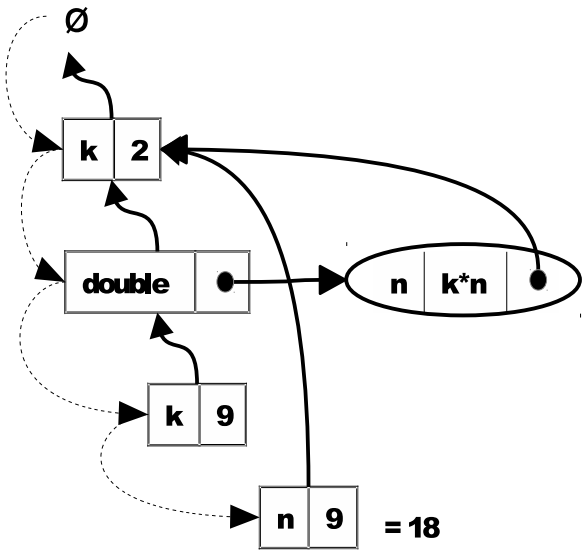

Figure 3.1: Environment Diagram 1

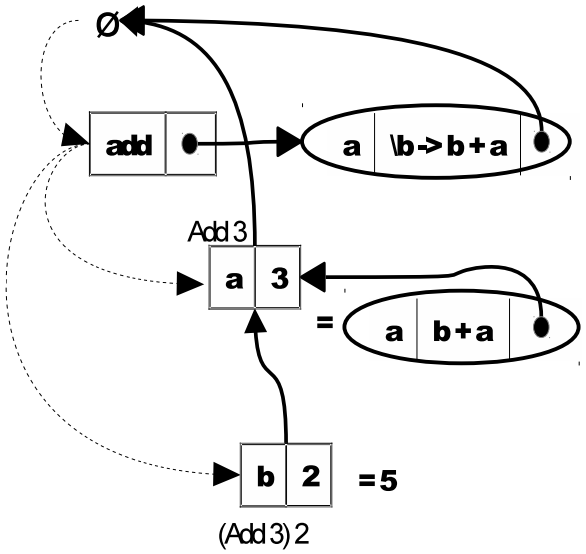

Figure 3.2: Environment Diagram 2

3.6.3 Example 3

let $m = 2$ **in**
  let $proc = \lambda n \rightarrow m + n$
    $part = \lambda(g, n) \rightarrow \lambda m \rightarrow n * g\ (m)$
  in let $inc = part\ (proc, 3)$ **in**
    $inc\ 7$

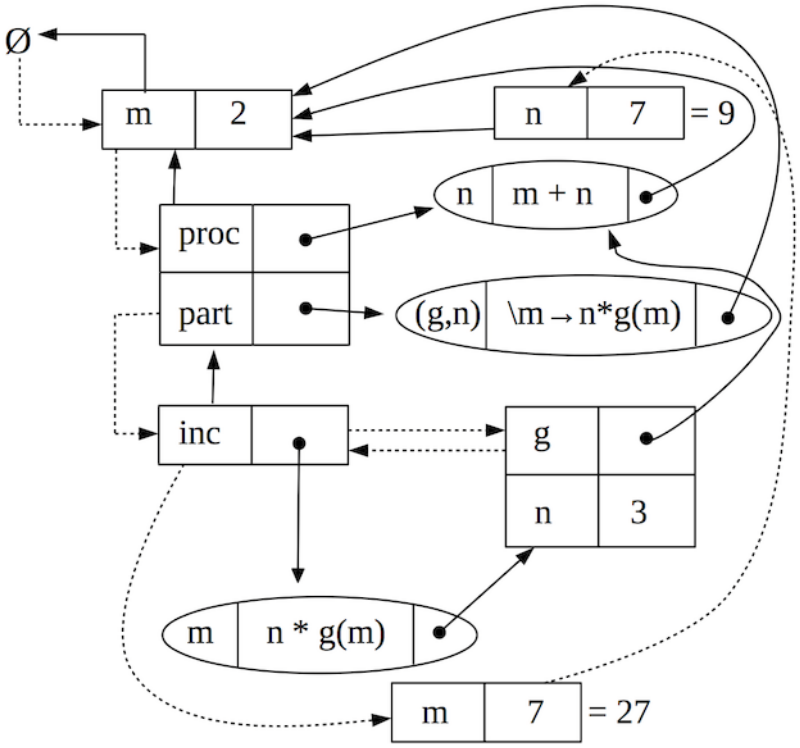

Figure 3.3: Environment Diagram 3

3.7 Summary of First-Class Functions

Here is the full code for first-class functions with non-recursive definitions:

```
data Exp = Literal Value
  | Unary      UnaryOp Exp
  | Binary     BinaryOp Exp Exp
  | If         Exp Exp Exp
  | Variable   String
  | Let        String Exp Exp
  | Function   String Exp-- new
  | Call       Exp Exp   -- changed
  deriving (Eq, Show)

type Env = [(String, Value)]

evaluate :: Exp -> Env -> Value
evaluate exp env = eval exp where
  eval (Literal v)      = v
  eval (Unary op a)     = unary op (eval a)
  eval (Binary op a b) = binary op (eval a) (eval b)
  eval (If a b c)       = if fromBool (eval a)
                          then eval b
                          else eval c
  eval (Variable x)     = fromJust (lookup x env)
  eval (Let x exp body) = evaluate body newEnv
    where newEnv = (x, eval exp) : env
  eval (Function x body) = Closure x body env -- new
  eval (Call fun arg)   = evaluate body newEnv-- changed
    where Closure x body closeEnv = eval fun
      newEnv = (x, eval arg) : closeEnv

fromBool ((Bool b)) = b
```

Chapter 4

Recursive Definitions

One consequence of using a simple **let** expression to define functions is that it is no longer possible to define *recursive functions*, which were supported in the Section on Top-Level Functions. A recursive function is a function that calls itself within its own definition. For example, consider this definition of the factorial function:

$$\textbf{let } \mathit{fact} = \lambda n \rightarrow \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}\ (n-1)$$
$$\textbf{in } \mathit{fact}\ (10)$$

The *fact* function is recursive because it calls *fact* within its definition.

The problem with our existing language implementation is that the scope of the variable *fact* is the body of the **let** expression, which is *fact* (10), so while the use of *fact* in *fact* (10) is in scope, the other use in *fact* $(n-1)$ is *not* in scope. (TODO: wordy)

To solve this problem, we need to change how we understand the **let** expression: the scope of the bound variable must be both the body of the let, and the bound expression that provides a definition for the variable. This means that the variable can be defined in terms of itself. This is exactly what we want for recursive functions, but it can cause problems. For example,

$$\textbf{let } x = x + 1 \textbf{ in } x$$

This is now syntactically correct, as the bound variable x is in scope for the expression $x+1$. However, such a program is either meaningless, or it can be understood to mean "infinite loop". There are similar cases that are meaningful. For example, this program is meaningful:

$$\begin{aligned}&\textbf{let } x = y + 1\\ &\quad\ \ y = 99\\ &\textbf{in } x * y\end{aligned}$$

This example includes two bindings at the same time (which we do not currently support. TODO: see homework?). In this case the result is 9900 because $x = 100$ and

$y = 99$. It works because the binding expression for x, namely $y + 1$, is in the scope of y.

4.1 Semantics of Recursion

A more fundamental question is *what does a recursive definition* **mean**? In grade school we get used to dealing with equations that have the same variable on both sides of an equal sign. For example, consider this simple equation:

$a = 1 + 3a$

Our instinct, honed over many years of practice, is to "solve for a".

- $a = 1 + 3a$
- *{ subtract* $3a$ *from both sides }*
- $-2a = 1$
- *{ divide both sides by* -2 *}*
- $a = -1/2$

I feel a little silly going through this in detail (although I have spent a lot of time recently practicing algebra with my son, so I know how hard it is to master). The point is that the definition of *fact* has exactly the same form:

$$fact = \lambda n \rightarrow \textbf{if } n \equiv 0 \textbf{ then } 1 \textbf{ else } n * fact\ (n - 1)$$

This is an equation where *fact* appears on both sides, just as a appears on both sides in $a = 1 + 3a$. The question is: *how do we solve for fact*? It's not so easy, because we don't have algebraic rules to divide by lambda and subtract conditionals, to get both occurrences of *fact* onto the same side of the equation. We are going to have to take another approach.

The first thing to notice is that *fact* is a function, and like most functions it is an *infinite* structure. This makes sense in several ways. It is infinite in the sense that it defines the factorial for every natural number, and there is an infinity of natural numbers. If you consider the grade-school definition of a function as a set of pairs, then the set of pairs in the factorial function is infinite.

Finally, and most importantly for us, if you consider *fact* as a computational method or rule, then the computational rule has an unbounded number of steps that it can perform. We can count the steps: first it performs an equality comparison $n \equiv 0$, then it either stops or it performs a subtraction $n-1$ and then *performs the steps recursively*, then when it is done with that it performs a multiplication $n*$.... In other words, given a natural number n the computation will perform $3n+1$ steps. Since it will handle any natural number, there is no bound on the number of steps it performs. If you tried to write out the steps that might be performed, then the list of steps would be infinite.

4.1.1 Three Analyses of Recursion

In what follows we will explore three ways to understand recursion. The first explanation just allows us to define recursive **let** expression by using the capabilities for recursion that are built into Haskell. This explanation is elegant and concise, but not very satisfying (like pure sugar!). The problem is that we have just relied on recursion in Haskell, so we don't really have an explanation of recursion. The second explanation is a practical introduction to the concept of fixed points. This solution can also be implemented elegantly in Haskell, and has the benefit of providing a mathematically sound explanation of recursive definitions. While fixed points can be implemented directly, they are not the most efficient approach, especially in conventional languages. As a result, we will consider a third implementation, based on self application. This explanation is messy but practical. In fact, it is the basis for real-world implementations of C++ and Java. A forth explanation A fourth explanation, languages in traditional imperative languages.

4.2 Understanding Recursion using Haskell Recursion

Haskell makes it easy to create infinite structures and functions. Understanding how this works can help us in implementing our language. We've already seen many examples of recursive functions in Haskell: for example, every version of *evaluate* has been recursive. However, Haskell also allows creation of recursive data structures. For example, this line creates an infinite list of 2's:

$$twos = 2 : twos$$

Remember that the : operator adds an item to the front of a list. This means that *twos* is a list with 2 concatenated onto the front of the list *twos*. In other words, *twos* is an infinite list of 2's:

$$twos = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]$$

It's also possible to make infinite lists that change:

$$numbers = 0 : [n + 1 \mid n \leftarrow numbers]$$

This creates an infinite list of the natural numbers:

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, ...]$$

All these definitions work in Haskell because of *laziness*. Haskell creates an internal representation of a potentially infinite value, but it only creates as much of the value as the program actually needs. If you try to use all of *two* or *numbers* then the result will be an infinite loop that never stops. However, if the program only needs the first

10 items of *twos* or *numbers* then only the first 10 elements of the infinite value will be created.

Interestingly, Haskell also accepts the algebraic expression discussed earlier:

$$a = 1 + 3 * a$$

Haskell considers this a valid program, but it does *not* solve for a. Instead it treats the definition as a computational rule: to evaluate a, add one to three times the value of a, which requires evaluating a, and so on, again, and again, and again. The result is an infinite loop. The quickest way to write an infinite loop is:

$$inf = inf$$

TODO: make pictures to illustrate the cyclic values in this section.

Attempting to use this value leads to an immediate infinite loop[1]. If the value is not used, then it has no effect on the program results.

It is not always easy to determine if a value will loop infinitely or not. One rule of thumb is that if the recursive variable is used *within* a data constructor (e.g. :) or inside a function (in the body of a lambda), then it will probably not loop infinitely. This is because both data constructors and functions are lazy in Haskell.

4.2.1 Using Results of Functions as Arguments

TODO: consider discussing this example. Here is an outline:

A type for trees:

```
data Tree = Leaf Int | Branch Tree Tree
  deriving Show
```

An example tree:

```
Branch (Branch (Leaf 5) (Leaf 3))
  (Leaf (-99))
```

Computing the minimum and maximum of a tree:

```
minTree (Leaf n) = n
minTree (Branch a b) = min (minTree a) (minTree b)

maxTree (Leaf n) = n
maxTree (Branch a b) = max (maxTree a) (maxTree b)
```

Point out that computing both requires two traversals.

[1] Oddly enough, this kind of *inf* value is not useless! It has some legitimate uses in debugging Haskell programs (more on this later).

Computing minimum and maximum at the same time.

$$
\begin{aligned}
&minMax\ (Leaf\ n) = (n, n)\\
&minMax\ (Branch\ a\ b) = (min\ min1\ min2, max\ max1\ max2)\\
&\quad \textbf{where}\ (min1, max1) = minMax\ a\\
&\qquad (min2, max2) = minMax\ b
\end{aligned}
$$

minMax is an example of *fusing* two functions together.

Another operation: copying a tree and replacing all the leaves with a specific integer value:

$$
\begin{aligned}
&repTree\ x\ (Leaf\ n) = Leaf\ x\\
&repTree\ x\ (Branch\ a\ b) = Branch\ (repTree\ x\ a)\ (repTree\ x\ b)
\end{aligned}
$$

Now for our key puzzle: replacing every leaf in a tree with the minimum value of the tree:

$$
repMinA\ tree = repTree\ (minTree\ tree)\ tree
$$

This requires two traversals. It seems to truly *require* two traversals the minimum must be identified before the process of replacement can begin.

But lets fuze them anyway: TODO: need to develop this in a few more steps! Here is a helper function:

$$
\begin{aligned}
&repMin'\ (Leaf\ n, r) = (n, Leaf\ r)\\
&repMin'\ (Branch\ a\ b, r) = (min\ min1\ min2, Branch\ newTree1\ newTree2)\\
&\quad \textbf{where}\ (min1, newTree1) = repMin'\ (a, r)\\
&\qquad (min2, newTree2) = repMin'\ (b, r)
\end{aligned}
$$

Finally to do the replacement with the minimum:

$$
\begin{aligned}
&repMin\ tree = newTree\\
&\quad \textbf{where}\ (min, newTree) = repMin'\ (tree, min)
\end{aligned}
$$

Note how one of the results of the function call, the *min* value, is passed as an argument to the function call itself!

TODO: Explain how this works, and give a picture.

4.2.2 Implementing Recursive *Let* with Haskell

The powerful techniques for recursive definition illustrated in the previous section are sufficient to implement recursive **let** expressions. In the Section on Evaluation using Environments, **let** was defined as follows:

$$
\begin{aligned}
&eval\ (Let\ x\ exp\ body) = evaluate\ body\ newEnv\\
&\quad \textbf{where}\ newEnv = (x, eval\ exp) : env
\end{aligned}
$$

The problem here is that the bound expression *exp* is evaluated in the parent environment *env*. To allow the bound variable x to be used within the expression *exp*, the expression must be evaluated in the new environment. Fortunately this is easy to implement in Haskell:

$$eval\ (Let\ x\ exp\ body) = evaluate\ body\ newEnv$$
$$\quad \textbf{where}\ newEnv = (x, evaluate\ exp\ newEnv) : env$$

The new environment being created is passed as an argument to the evaluation function that is used during the creation of the new environment! It may see odd to use the result of a function as one of its arguments. However, as we have seen, Haskell allows such definitions.

The explanation of recursion in Haskell is almost too simple. In fact, it is too simple: it involved changing 6 characters in the code for the non-recursive program. The problem is that we haven't really explained recursion in a detailed way, because we have simply used Haskell's recursion mechanism to implement recursive **let** expressions in our language. The question remains: how does recursion work?

TODO: come up with a *name* for the little language we are defining and exploring. PLAI uses names like ArithC and ExprC.

Recursive Definitions in Environment/Closure Diagrams

For the case of recursive bindings, the only difference is that the order of step 2 and 3 is swapped:

- Case **Recursive** *Let x e body*
  1. Draw binding box for x with unknown value
     Set parent of new binding to be the current environment
  2. Set current environment to be the new binding
  3. Create the diagram for bound expression e
     Put the value of e into the binding as the value of x
  4. Draw diagram for *body* and remember value
  5. Set current environment back to what it was before

Note that in this case the binding in Step 3 becomes the current environment *before* it is fully defined.

TODO: examples here

4.3 Understanding Recursion with Fixed Points

Another way to explain recursion is by using the mathematical concept of a fixed point. A *fixed point* of a function f if a value x where $x = f(x)$. If you think of a function as a transformation on values, then fixed points are values that are unchanged by the function. For example, if the function represents a rotation (imagine simple rotation of a book on a table) then the fixed point is the center of the rotation... that is the point on the book that is unchanged by rotating it. If you really did rotate a book, you'd probably push your finger down in the middle, then rotate the book around your finger. The spot under your finger is the fixed point of the rotation function.

There is a large body of theory about fixed points, including applications in mathematics and fundamental theorems (see the Knaster Tarski theorem), but I'm going to avoid the math and give a practical discussion of fixed-points with examples. TODO: give citations to appropriate books.

TODO: nice picture of the book and the fixed point? Use a fun book, like "Theory of Lambda Conversion".

4.3.1 Fixed Points of Numeric Functions

Fixed-points can also be identified for simple mathematical functions:

function	*fixed point(s)*
$i_{10}(x) = 10 - x$	5
$square(x) = x^2$	$0, 1$
$g_\phi(x) = 1 + \frac{1}{x}$	1.6180339887...
$k_4(x) = 4$	4
$id(x) = x$	all values are fixed points
$inc(x) = x + 1$	no fixed points

As you can see, some functions have one fixed point. Some functions have multiple fixed points. Others have an infinite number of fixed points, while some don't have any at all. The fixed point of g_ϕ is the *golden ratio*, also known as ϕ.

Fixed points are useful because they can provide a general approach to solving equations where a variable appears on both sides of an equation. Consider this simple equation:

$x = 10 - x$

Rather than performing the normal algebraic manipulation to solve it, consider expressing the right side of the equation using a new helper function, g:

$g(x) = 10 - x$

Functions created in this way are called *generators* for recursive equations. Given the generator g, the original equation can be rewritten as:

$x = g(x)$

Any value x that satisfies $x = g(x)$ is a fixed point of g. Conversely, any fixed point of g is a solution to the original equation. This means that finding a solution to the original equation is equivalent to finding a fixed point for g. Imagine that there was a magic function *fix* that could automatically find a fixed point for any function[2]. Then one way to find a fixed point of g would be to use *fix*, by calling $fix(g)$. Then the solution to the equation above could be rewritten using *fix*:

$x = fix(g)$

This result looks like a *solution* for x, in the sense that it is an equation where x appears only by itself on the left of the equation. Any equation where a variable appears by itself on the left and anywhere on the right side of the equation, can be rewritten as a fixed point equation.

Note that *fix* is a higher-order function: it takes a function as an input, and returns a value as a result.

The problem is that the solution relies on *fix*, a function that hasn't been defined yet, and maybe cannot be defined. Is it possible to automatically find a fixed point of any function? Does the function *fix* exist? Can it be defined?

4.3.2 Fixed Points by Iterative Application

It turns out that there is no way to find fixed points for *any* arbitrary function f, but for a certain class of well behaved functions, it *is* possible to compute fixed points automatically. In this case, "well behaved" means that the function converges on the solution when applied repeatedly. For example, consider function g_ϕ defined above:

$$g_\phi(x) = 1 + \frac{1}{x}$$

Consider multiple invocations of g_ϕ starting with $g_\phi(1)$. The following table summarizes this process. The first column represents the iteration number, which starts at one and increases with each iteration. The second column is a representation of the computation as an explicit *power* of a function. The power of a function $f^n(x)$ means to apply f repeatedly until it has been performed n times, passing the result of one call as the

[2]The function *fix* is often called Y. For further reading, see Scott (1975), Gunter (1993), Gabriel (1988) and Thomas (2006).

input of the next call. For example, $f^3(x)$ means $f(f(f(x)))$. The next column shows just the application of g_ϕ to the previous result. The final column gives the result for that iteration.

#	power	previous		result
1	$g_\phi^1(1)$	$g_\phi(1)$	=	2
2	$g_\phi^2(1)$	$g_\phi(2)$	=	1.5
3	$g_\phi^3(1)$	$g_\phi(1.5)$	=	1.6666666667
4	$g_\phi^4(1)$	$g_\phi(1.66666666666667)$	=	1.6
5	$g_\phi^5(1)$	$g_\phi(1.6)$	=	1.625
6	$g_\phi^6(1)$	$g_\phi(1.625)$	=	1.6153846154
7	$g_\phi^7(1)$	$g_\phi(1.61538461538462)$	=	1.619047619
8	$g_\phi^8(1)$	$g_\phi(1.61904761904762)$	=	1.6176470588
9	$g_\phi^9(1)$	$g_\phi(1.61764705882353)$	=	1.6181818182
10	$g_\phi^{10}(1)$	$g_\phi(1.61818181818182)$	=	1.6179775281
11	$g_\phi^{11}(1)$	$g_\phi(1.61797752808989)$	=	1.6180555556
12	$g_\phi^{12}(1)$	$g_\phi(1.61805555555556)$	=	1.6180257511
13	$g_\phi^{13}(1)$	$g_\phi(1.61802575107296)$	=	1.6180371353
14	$g_\phi^{14}(1)$	$g_\phi(1.61803713527851)$	=	1.6180327869
15	$g_\phi^{15}(1)$	$g_\phi(1.61803278688525)$	=	1.6180344478
16	$g_\phi^{16}(1)$	$g_\phi(1.61803444782168)$	=	1.6180338134
17	$g_\phi^{17}(1)$	$g_\phi(1.61803381340013)$	=	1.6180340557

TODO: create a little plot of this function convergence.

The result converges on 1.6180339887... which is the value of ϕ. It turns out that iterating g_ϕ converges on ϕ for any starting number. The fixed point is the *limit* of applying the transformation function g_ϕ infinitely many times. One way to express the fixed point is

$fix(f) = f^\infty(start)$

This means the application of f an infinite number of times to some starting value. Finding the right starting value can be difficult. In some cases any starting value will work, but in other cases it's important to use a particular value. In the theory of fixed points, (TODO: discuss the theory somewhere), the initial value is the bottom of an

appropriate lattice.

The fixed point of some, but not all, functions can be computed by repeated function application. Here are the results for this technique, when applied to the examples given above:

function	result for repeated invocation
$inv_{10}(x) = 10 - x$	infinite loop
$square(x) = x^2$	infinite loop
$g_\phi(x) = 1 + \frac{1}{x}$	1.6180339887...
$const_4(x) = 4$	4
$id(x) = x$	infinite loop
$inc(x) = x + 1$	infinite loop

Only two of the six examples worked. Fixed points are not a general method for solving numeric equations.

4.3.3 Fixed Points for Recursive Structures

The infinite recursive structures discussed in Section on Haskell Recursion can also be defined using fixed points:

$$g\_twos\ l = 2 : l$$

The function g_twos is a non-recursive function that adds a 2 to the front of a list. Here are some test cases for applying g_twos to various lists:

input	output	input = output
$[\,]$	$[2]$	no
$[1]$	$[2, 1]$	no
$[3, 4, 5]$	$[2, 3, 4, 5]$	no
$[2, 2, 2, 2, 2]$	$[2, 2, 2, 2, 2, 2]$	no
$[2, 2, 2, ...]$	$[2, 2, 2, ...]$	*yes*

The function g_twos can be applied to any list. If it is applied to any finite list, then

the input and output lists cannot be the same because the output is one element longer then the input. This is not a problem for infinite lists, because adding an item to the front of an infinite list is still an infinite list. Adding a 2 onto the front of an infinite list of 2s will return an infinite list of 2s. Thus an infinite list of 2s is a fixed point of g_twos.

$$fix\ (g\_twos) = [2, 2, 2, ...]$$

Functions used in this way are called generators because they generate recursive structures. One way to think about them is that the function performs *one step* in the creation of a infinite structure, and then the *fix* function repeats that step over and over until the full infinite structure is created. Consider what happens when the output of the function is applied to the input of the previous iteration. The results are [ ], [2], [2, 2], [2, 2, 2], [2, 2, 2, 2], ... At each step the result is a better approximation of the final solution.

The second example, a recursive definition that creates a list containing the natural numbers, is more interesting:

$$g\_numbers\ ns = 0 : [n + 1 \mid n \leftarrow ns]$$

This function takes a list as an input, it adds one to each item in the list and then puts a 0 on the front of the list.

Here are the result when applied to the same test cases listed above:

input	output	input = output
[ ]	[0]	no
[1]	[0, 2]	no
[3, 4, 5]	[0, 4, 5, 6]	no
[2, 2, 2, 2, 2]	[0, 3, 3, 3, 3, 3]	no
[2, 2, 2, ...]	[0, 3, 3, 3, ...]	no

A more interesting set of test cases involves starting with the empty list, then using each function result as the next test case:

input	output	input = output
[ ]	[0]	no
[0]	[0, 1]	no
[0, 1]	[0, 1, 2]	no
[0, 1, 2]	[0, 1, 2, 3]	no

$[0,1,2,3]$	$[0,1,2,3,4]$	no
$[0,1,2,3,4]$	$[0,1,2,3,4,5]$	no
$[0,1,2,3,4,5,...]$	$[0,1,2,3,4,5,6,...]$	*yes*

The only list that is unchanged after applying $g\_numbers$ is the list of natural numbers:

$$fix\ (g\_numbers) = [0,1,2,3,4,5,...]$$

By staring with the empty list and then applying $g\_numbers$ repeatedly, the result eventually converges on the fixed point. Each step is a better approximation of the final answer.

4.3.4 Fixed Points of Higher-Order Functions

TODO: text explaining how to implement *fact* using fix.

$$g\_fact = \lambda f \to \lambda n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * f\ (n-1)$$

$$fact = fix\ g\_fact$$

more...

4.3.5 A Recursive Definition of *fix*

Haskell allows an elegant definition of *fix* using recursion, which avoids the issue of selecting a starting value for the iteration.

$$fix\ g = g\ (fix\ g)$$

This definition is beautiful because it is a direct translation of the original mathematic definition of a fixed point: $fix(f)$ is a value x such that $x = f(x)$. Substituting $fix(f)$ for x gives the definition above.

From an algorithmic viewpoint, the definition of only works because of lazy evaluation in Haskell. To compute $fix\ g$ Haskell evaluates $g\ (fix\ g)$ but does not immediately evaluate the argument $fix\ g$. Remember that arguments in Haskell are only evaluated if they are *needed*. Instead it begins evaluating the body of g, which may or may not use its argument.

4.3.6 A Non-Recursive Definition of *fix*

It is also possible to define *fix* non-recursively, by using *self application*. Self application is when a function is applied to itself. This works because functions are values, so a function can be passed as an argument to itself. For example, consider the identity function, which simply returns its argument:

$$id\ x = x$$

The identity function can be applied to *any* value, because it doesn't do anything with the argument other than return it. Since it can be applied to any value, it can be applied to itself:

$$id\ (id)$$
$$--\ returnsid$$

Self application is not a very common technique, but it is certainly interesting. Here is a higher-order function that takes a function as an argument and immediately applies the function to itself:

$$stamp\ f = f\ (f)$$

Unfortunately, the *stamp* function cannot be coded in Haskell, because it is rejected by Haskell's type system. When a function of type $a \to b$ is applied to itself, the argument type a must be equivalent to $a \to b$. There are no types in the Haskell type system that can express a solution to type equation $a = a \to b$. Attempting to define *stamp* results in a Haskell compile-time error:

$$Occurs\ check : cannot\ construct\ the\ infinite\ \mathbf{type} : t1 = t1 \to t0$$

Many other languages allow *stamp* to be defined, either using more complex or weaker type systems. Dynamic languages do not have any problem defining *stamp*. For example, here is a definition of *stamp* in JavaScript:

```
stamp = function (f) { return f(f); }
```

The interesting question is what happens when *stamp* is applied to itself: *stamp* (*stamp*). This call binds f to *stamp* and then executes $f\ (f)$ which is *stamp* (*stamp*). The effect is an immediate infinite loop, where stamp is applied to itself over and over again. What is interesting is that *stamp* is not recursive, and it does not have a while loop. But it manages to generate an infinite loop anyway.

Given the ability to loop infinitely, it is also possible to execute a function infinitely many times.

$$fix\ g = stamp\ (g \circ stamp)$$

TODO: explain composition ($\circ$) operator

Here are the steps in executing *fix* for a function g:

- *fix* g
  { definition of *fix*}

- $=$ *stamp* $(g \circ$ *stamp*$)$
  { definition of *stamp*}

- $= (g \circ$ *stamp*$)$ $(g \circ$ *stamp*$)$
  { definition of $\circ$}

- $= g$ (*stamp* $(g \circ$ *stamp*$)$)
  { definition of *fix*}

- $= g$ (*fix* g)

This version of *fix* uses self-application to create a self-replicating program, which is then harnessed as an engine to invoke a function infinitely many times. This version of *fix* is traditionally written as $\lambda g.(\lambda x.g(xx))(\lambda x.g(xx))$, but this is the same as the version given above with the definition of *stamp* expanded.

A second problem with this definition of *fix* is that it *diverges*, or creates an infinite loop, when executed in non-lazy languages. Thus it cannot be used in Haskell because of self-application, and it cannot be used in most other languages because of strict evaluation. A non-strict version can be defined:

Y = *stamp*(λf.(λx.f(λv.(*stamp* x v))))

Finally, explicit fixed points involve creation of many closures.

4.4 Understanding Recursion with Self-Application

Another way to implement recursion is by writing self-application directly into a function. For example, here is a non-recursive version of fact based on integrated self-application, defined in JavaScript.

fact_s = **function** (f, n) {
  if $(n \equiv 0)$

```
    return 1;
  else
    return n * f (f, n - 1);
}
```

To call this function, it is necessary to pass itself as an argument to itself:

$$fact\_s\ (fact\_s, 10);$$

This definition builds the self-application into the $fact\_s$ function, rather than separating it into a generator and a fixed point function. One way to derive $fact\_s$ is from the self-applicative *fix* function. Remember that

$$fact = stamp\ (g\_fact \circ stamp)$$

$fact\_s$ is created by *merging* g_fact with *stamp*. The other use of *stamp* indicates that $fact\_s$ must be applied to itself to compute a factorial.

One interesting thing about this final implementation strategy is that it is *exactly* the strategy used in the actual implementation of languages like C++ and Java.

Chapter 5

Computational Strategies

In previous sections the Exp language was extended with specific kinds of expressions and values, for example the **let** and *functions*. In addition to augmenting the language with new expression types, it is also possible to consider extensions that have a general impact on every part of the language. Some examples are error handling, tracing of code, and mutable state.

5.1 Error Checking

Errors are an important aspect of computation. They are typically a pervasive feature of a language, beause they affect the way that every expression is evaluated. For example, the expression $a + b$ may not cause any errors, but if evaluating a or b can cause an error, then the evaluation of $a + b$ will have to deal with the possibility that a or b is an error. The full code is given in the Error Checking file.

Error checking is a notorious problem in programming languages. When coding in C, everyone agrees that the return codes of all system calls should be checked to make sure that an error did not occur. However, most C programs don't check the return codes, leading to serious problems when things start to go wrong.

Errors are pervasive because any expression can either return a value or it can signal an error. One way to represent this possibility is by defining a new data type that has two possibilities: either a *good* value or an error.

> **data** *Checked a* = *Good a* | *Error String*
> **deriving** *Show*

The declaration defines a generic *Checked* type that has a parameter a representing the type of the good value. The *Checked* type has two constructors, *Good* and *Error*. The *Good* constructor takes a value of type a and labels it as good. The *Error* constructor

has an error message. The following figure is an abstraction illustration of a *Checked* value, which represents a computation that may either be a good value or an error.

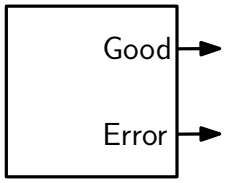

Figure 5.1: A computation that may produce an error.

5.1.1 Error Checking in Basic Expressions

To keep things simple and focused on errors, this section will only consider expressions with literals, variables, binary operators. This smaller language is similar to the one that was introduced at the beginning of the book. More features will be added later. Although the syntax of expressions does not have to change, but the type of the *evaluate* function must be changed to return an *Error* value:

```
evaluate :: Exp → Env → Checked Value
evaluate exp env = eval exp where
  eval (Literal v) = Good v
```

Evaluation of a literal can never cause an error. The value is marked as a *Good* value and returned.

A variable can be undefined, so it evaluating a variable may return an error:

```
eval (Variable x) =
  case lookup x env of
    Nothing → Error ("Variable " ++ x ++ " undefined")
    Just v → Good v
```

5.1.2 Error Checking in Multiple Sub-expressions

The case for binary operations is more interesting. Here is the original rule for evaluating binary expressions:

```
eval (Binary op a b) = binary op (eval a) (eval b)
```

The problem is that either *eval a* or *eval b* could return an *Error* value. The actual binary operation is only performed if they both return *Good* values. Finally, the binary operation itself might cause a new error. Thus there are three places where errors can arise: in *eval a*, in *eval b*, or in *binary*. This definition for *eval* of a binary operator handles the first two situations:

```
eval (Binary op a b) =
  case eval a of
    Error msg → Error msg
    Good av →
      case eval b of
        Error msg → Error msg
        Good bv →
          checked_binary op av bv
```

Now it should be clear why error return codes are not always checked. What was originally a one-line program is now 8 lines and uses additional temporary variables. When multiple sub-expressions can generate errors, it is necessary to *compose* multiple error checks together. The situation in the case of *Binary* operations is illustrated in the following figure:

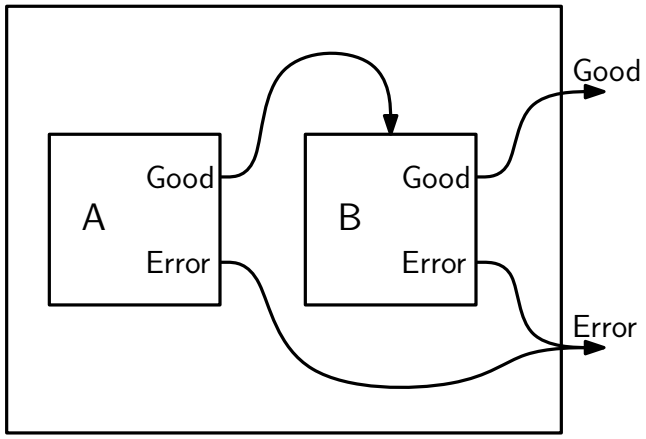

Figure 5.2: Composing computations that may produce errors.

This figure illustrates the composition of two sub-expressions A and B which represent computations of checked values. The composition of the two computations is a new computation that also has the shape of a checked value. If either A or B outputs an error, then the resulting computation signals an errors. The arrow from A to the top of B represents passing the good value from A into B as an extra input. This means that B can depend upon the good value of A. But B is never invoked if A signals an error.

The *binary* helper function must be updated to signal divide by zero:

```
checked_unary :: UnaryOp → Value → Checked Value
checked_unary Not (Bool b) = Good (Bool (¬ b))
checked_unary Neg (Int i) = Good (Int (−i))
checked_binary :: BinaryOp → Value → Value → Checked Value
checked_binary Add (Int a) (Int b) = Good (Int (a + b))
```

$$
\begin{aligned}
&\mathit{checked\_binary}\ \mathit{Sub}\ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Int}\ (a - b))\\
&\mathit{checked\_binary}\ \mathit{Mul}\ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Int}\ (a * b))\\
&\mathit{checked\_binary}\ \mathit{Div}\ (\mathit{Int}\ a)\ (\mathit{Int}\ 0) = \mathit{Error}\ \texttt{"Divide by zero"}\\
&\mathit{checked\_binary}\ \mathit{Div}\ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Int}\ (a\ `\mathit{div}`\ b))\\
&\mathit{checked\_binary}\ \mathit{And}\ (\mathit{Bool}\ a)\ (\mathit{Bool}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a \wedge b))\\
&\mathit{checked\_binary}\ \mathit{Or}\ \ (\mathit{Bool}\ a)\ (\mathit{Bool}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a \vee b))\\
&\mathit{checked\_binary}\ \mathit{LT}\ \ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a < b))\\
&\mathit{checked\_binary}\ \mathit{LE}\ \ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a \leqslant b))\\
&\mathit{checked\_binary}\ \mathit{GE}\ \ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a \geqslant b))\\
&\mathit{checked\_binary}\ \mathit{GT}\ \ (\mathit{Int}\ a)\ (\mathit{Int}\ b) = \mathit{Good}\ (\mathit{Bool}\ (a > b))\\
&\mathit{checked\_binary}\ \mathit{EQ}\ \ a\ \ \ \ \ b\ \ \ \ \ = \mathit{Good}\ (\mathit{Bool}\ (a \equiv b))\\
&\mathit{checked\_binary}\ \_\ \ \ \_\ \ \ \ \ \_\ \ \ \ \ = \mathit{Error}\ \texttt{"Type error"}
\end{aligned}
$$

All the other cases are the same as before, so *checked_binary* calls *binary* and then tags the resulting value as *Good*.

5.1.3 Examples of Errors

Evaluating an expression may now return an error for unbound variables:

$$\mathit{evaluate}\ (\mathit{Variable}\ \texttt{"x"})\ [\,]$$

The result of evaluation is:

$$\mathit{Error}\ \texttt{"Variable x undefined"}$$

Or for divide by zero:

$$\mathit{evaluate}\ (\mathit{Binary}\ \mathit{Div}\ (\mathit{Literal}\ (\mathit{Int}\ 3))\ (\mathit{Literal}\ (\mathit{Int}\ 0)))\ [\,]$$

The result of evaluation is:

$$\mathit{Error}\ \texttt{"Divide by zero"}$$

Your take-away from this section should be that checking error everywhere is messy and tedious. The code for binary operators has to deal with errors, even though most binary operators don't have anything to do with error handling.

Exercise 5.1: Complete Error Checking (was 5.1.1)

Extend the evaluator with error checking for the remaining expression cases, including **if**, non-recursive **let**, and function definition/calls. Ensure that all errors, including pattern match failures, are captured by your code and converted to *Error* values, rather than causing Haskell execution errors.

As a bonus, implement error checking for recursive **let** expressions.

Exercise 5.2: Error Handling (was 5.1.2)

In the code given above, all errors cause the program to terminate execution. Extend the language with a *try/catch* expression that allows errors to be caught and handled within a program.

Exercise 5.3: Multiple Bindings and Arguments

If you really want to experience how messy it is to explicitly program error handling, implement error checking where **let** expressions can have multiple bindings, and functions can have multiple arguments.

5.2 Mutable State

A second common pervasive computational strategy, besides error handling, is the use of *mutable state*. Mutable state means means that the state of a program changes or mutates: that a variable can be assigned a new value or a part of a data structure can be modified. Mutable state is a pervasive feature because it is something that happens in addition to the normal computation of a value or result from a function.

Here is one typical example of a program that uses mutable variables. The code is valid in C, Java or JavaScript:

```
x = 1;
for (i = 2; i <= 5; i = i + 1) {
  x = x * i;
}
```

It declares a local variable named x with initial value 1 and then performs an iteration where the variable i changes from 1 to 10. On each iteration of the loop the variable x is multiplied by i. The result of x a the end is the factorial of 5, namely 120.

Another typical example of mutable state is modification of data structures. The following code, written in JavaScript, creates a circular data structure

```
record = { first: 2, next: null };
record.next = record;
```

Roughly equivalent code could be implemented in C or Java (or any other imperative language), although the resulting code is usually somewhat longer.

It would be easy to recode the factorial example above as a pure functional program. With more work it may be possible to encoding the circular data structure as well. But the point of this book is not to teach you how to do functional programming. The point is to explain programming languages, and to code the explanation explicitly as an evaluator. Since many programming languages allow mutable values, it is important to be able to explain mutation. But we cannot *use* mutation to provide the explanation, because we have chosen to write the evaluator in Haskell, a pure functional language. The hope is that detailed and explicit analysis of how mutation works in programming languages will lead to insights about the costs and benefits of using mutation. The code for this section is in the Mutable State file.

5.2.1 Addresses

Imperative languages typically allow everything to be mutable by default: all variables and mutable and all data structures are mutable. While this is often convenient, it has the disadvantage that there is no way to turn off mutation. Many variables and data structures, even in imperative languages, are logically immutable. Even when the programmer *intends* for the variables or data structure to be constant and unchanging, there is no way in most imperative languages for the programmer to make this intention explicit.

To rectify this situation, at the cost of being somewhat unconventional, this book takes a different approach to mutable state, where mutability must be explicitly declared. Variables are not mutable by default. Instead a new kind of value, an *address*, is introduced to support mutation. An address identifies a mutable container that stores a single value, but whose contents can change over time. The storage identified by an address is sometimes called a *cell*. You can think of it as a *box* that contains a value. Addresses are sometimes called *locations*. (Note that the concept of an address of a mutable container is also used in ML and BLISS for mutable values, where they are known as *ref* values. This is also closely related to the concept of an address of a memory cell, as it appears in assembly language or C).

There are three fundamental operations involving addresses: creating a new cell with an initial value and a new address, accessing the current value at a address, and changing the value stored at an address. The following table gives the concrete syntax of these operations.

Operation	Meaning
Mutable (e)	Creates a mutable cell with initial value given by e
$!a$	Accesses the contents stored at address a
$a := e$	Updates the contents at address a to be value of expression e

Using these operations, the factorial program given above can be expressed as follows, using mutable cells:

```
x = Mutable(1);
for (i = Mutable(2); !i <= 5; i := !i + 1) {
  x := !x * !i;
}
```

In this model a variable always denotes the address to which it is bound. If the variable x appears on the right side of an assignment, it must be *dereferenced* as $!x$. If the variable appears on the left side of an assignment, it denotes an address that is updated.

It should be clear that the *variables* don't actually change in this model. The variables are bound to an address, and this binding does not change. What changes is the value stored at an address. This interpretation resembles the computational model underlying C, where address identify memory cells. (TODO: make more careful comparison to C, with attention to *l-values* and *r-values*)

An address is a new kind of value. Although addresses can be represented by any unique set of labels, one convenient representation for addresses is as integers. Using integers as addresses is also similar to the use of integers for addresses in a computer memory.

data *Value* = *Int Int*
| *Bool Bool*
| *Closure String Exp Env*
| *Address Int*−− *new*
deriving (*Eq*, *Show*)

When writing programs and values, it is useful to distinguish addresses from ordinary integer values. As a convention, addresses will be tagged with an "at sign", so that *Address* 3 will be written @3.

Another advantage of explicit cells for mutability is that the treatment of local variables given in previous chapters is still valid. Variables are still immutably bound to values. By introducing a new kind of value, namely addresses, it is possible to bind a variable to an address. It is the content stored at an address that changes, not the variable. (reminds me of the line of The Matrix: "it is not the spoon that bends...") Introducing cells and addresses does not fundamentally change the nature or capabilities of imperative languages, it just modifies how the imperative features are expressed.

Memory

The current value of all mutable cells used in a program is called *memory*. Logically, a memory is a map or association of addresses to values. The same techniques used

for environments could be used for memories, as a list of pairs or a function. Memory can also be represented as a function mapping integers to values, similar to the representation of environments as functions. Note that a memory is also sometimes called a *store*, based on the idea that is provides a form of *storage*.

Since addresses are integers, one natural representation is as a list or array of values, where the address is the position or index of the value. Such an array is directly analogous to the memory of a computer system, which can be thought of as an array of 8 bit values. In this chapter memory will be implemented as a list of values, although many other representations are certainly possible.

type *Memory* = [*Value*]

One complication is that the memory must be able to *grow* by adding new addresses. The initial empty memory is the empty list [ ]. The first address added is zero [@0]. The next address is one to create a memory [@0, @1]. In general a memory with n cells will have addresses [@0, @1, ..., @$n-1$]. Here is an example memory, with two addresses:

[*Value* (*Scalar* (*Int* 120)), *Value* (*Scalar* (*Int* 6))]

This memory has value 120 at address 0 and value 6 at address 1. More concisely, this memory can be written as

[120, 11]

This memory could be the result of executing the factorial program given above, under the assumption that x is bound to address 0 and i is bound to address 1. An appropriate environment is:

$[x \mapsto @0,\ i \mapsto @1]$

During the execution of the program that computes the factorial of 5, there are 10 different memory configurations that are created:

Step	Memory
start	[]
$x =$ *Mutable* (1);	[1]
$i =$ *Mutable* (2);	[1, 2]
$x = !x * !i;$	[2, 2]
$i = !i + 1;$	[2, 3]
$x = !x * !i;$	[6, 3]
$i = !i + 1;$	[6, 4]
$x = !x * !i;$	[24, 4]

$i = !i + 1;$	$[24, 5]$
$x = !x * !i;$	$[120, 5]$
$i = !i + 1;$	$[120, 6]$

5.2.2 Pure Functional Operations on Memory

The two fundamental operations on memory are memory *access*, which looks up the contents of a memory cell, and *update*, which modifies the contents of a memory cell.

Access

The memory *access* function takes a memory address i and a memory (list) and returns the item of the list at position i counting from the left and starting at 0. The Haskell function !! returns the nth item of a list, so it is exactly what we need:

$$access\ i\ mem = mem\ !!\ i$$

TODO: rename "access" to be "contents"?

Update

It is not possible to actually *change* memory in pure functional languages, including Haskell, because there is no way to modify a data structure after is has been constructed. But it is possible to compute a new data structure that is based on an existing one. This is the notion of *functional update* or *functional change*: a function can act as a transformation of a value into a new value. A functional update to memory is a function of type $Memory \to Memory$ Such functions take a memory as input and create a *new* memory as an output. The new memory is typically nearly identical to the input memory, but with a small change.

For example, the *update* operator on memory replaces the contents of a single address with a new value.

$$\begin{aligned}
&update :: Int \to Value \to Memory \to Memory \\
&update\ addr\ val\ mem = \\
&\quad \textbf{let}\ (before, _ : after) = splitAt\ addr\ mem\ \textbf{in} \\
&\qquad before +\!\!+ [val] +\!\!+ after
\end{aligned}$$

The *update* function works by splitting the memory into the part before the address and the part starting with the address $addr$. The pattern $\_after$ binds $after$ to be the memory after the address. The *update* function then recreates a new memory containing the before part, the updated memory cell, and the after part. The function is inefficient

because it has to copy all the memory cells it has scanned up to that point! We are not worried about efficiency, however, so just relax. It is fast enough.

Using *access* and *update* it is possible to define interesting *transformations* on memory. For example, the function *mul10* multiplies the contents of a memory address by 10:

$$
\begin{array}{l}
mul10\ addr\ mem = \\
\quad \textbf{let}\ n = fromInt\ (access\ addr\ mem)\ \textbf{in} \\
\quad\quad update\ addr\ (toValue\ (10 * n))\ mem \\
fromInt\ (Int\ n) = n \\
toValue\ n = (Int\ n)
\end{array}
$$

Here is an example calling *mul10* on a memory with 4 cells:

$$mul10\ 1\ [toValue\ 3, toValue\ 4, toValue\ 5, toValue\ 6]$$

The result is

$$[Scalar\ (Int\ 3), Scalar\ (Int\ 4), Scalar\ (Int\ 50), Scalar\ (Int\ 6)]$$

The fact that *mul10* is a transformation on memory is evident from its type:

$$mul10 :: Int \to Memory \to Memory$$

This means that *mul10* takes an memory address as an input and returns a function that transforms an input memory into an output memory.

5.2.3 Stateful Computations

A stateful computation is one that produces a value and *also* accesses and potentially updates memory. In changing *evaluate* to be a stateful computation, the type must change. Currently *evaluate* takes an expression and an environment and returns a value:

$$evaluate :: Exp \to Env \to Value$$

Now that an expression can access memory, the current memory must be an input to the evaluation process:

$$evaluate :: Exp \to Env \to Memory \to ...$$

The evaluator still produces a value, but it may also return a new modified memory. These two requirements, to return a value and a memory, can be achieved by returning a *pair* of a value and a new memory:

$$evaluate :: Exp \to Env \to Memory \to (Value, Memory)$$

This final type is the type of a *stateful* computation. Since it is useful to talk about, we will give it a name:

type *Stateful* t = *Memory* $\rightarrow$ (*Value*, *Memory*)

This is a *generic* type for a memory-based computation which returns a value of type t. Just as in the case of errors, it is useful to give a visual form to the shape of a stateful computation:

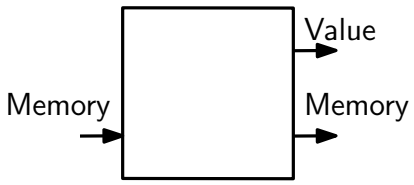

Figure 5.3: Shape of a stateful computation.

Thus the final type for *evaluate* is written concisely as:

evaluate :: *Exp* $\rightarrow$ *Env* $\rightarrow$ *Stateful Value*

This type is very similar to the type given for *evaluate* in the error section, where *Checked* was used in place of *Stateful*. This similarity is not an accident, as we will see in a later chapter.

5.2.4 Semantics of a Language with Mutation

The first step in creating a function with mutable cells is to add abstract syntax for the three operations on mutable cells. The following table defines the abstract syntax:

Operation	Abstract Syntax	Meaning
Mutable (e)	*Mutable* e	Allocate memory
$!a$	*Access* a	Accesses memory
$a := e$	*Assign* a e	Updates memory

The abstract syntax is added to the data type representing expressions in our language:

data *Exp* = ...
| *Mutable Exp*$--$*new*
| *Access Exp*$--$*new*
| *Assign Exp Exp*$--$*new*

The *Mutable* (e) expression creates a new memory cell and returns its address. First the expression e is evaluated to get the initial value of the new memory cell. Evaluating e may modify memory, so care must be taken to allocate the new cell in the new memory.

The address of the new memory cell is just the length of the memory.

$$
\begin{array}{l}
eval\ (Mutable\ e)\ mem = \\
\quad \textbf{let}\ (ev, mem') = eval\ e\ mem\ \textbf{in} \\
\quad\quad (Address\ (length\ mem'), mem' +\!\!+ [ev])
\end{array}
$$

The access expression $!a$ expression evaluates the address expression a to get an address, then returns the contents of the memory at that address. Note that if the $Address\ i$ pattern fails, Haskell raises an error. This is another case where error handling, as in the previous section, could be used.

$$
\begin{array}{l}
eval\ (Access\ a)\ mem = \\
\quad \textbf{let}\ (Address\ i, mem') = eval\ a\ mem\ \textbf{in} \\
\quad\quad (access\ i\ mem', mem')
\end{array}
$$

An assignment statement $a := e$ first evaluates the target expression a to get an address. It is an error if a does not evaluate to an address. Then the source expression e is evaluated. Evaluating a and e may update the memory, so

$$
\begin{array}{l}
eval\ (Assign\ a\ e)\ mem = \\
\quad \textbf{let}\ (Address\ i, mem') = eval\ a\ mem\ \textbf{in} \\
\quad\quad \textbf{let}\ (ev, mem'') = eval\ e\ mem'\ \textbf{in} \\
\quad\quad\quad (ev, update\ i\ ev\ mem'')
\end{array}
$$

Mutable State with Multiple Sub-expressions

The interesting thing is that even parts of the evaluator that have nothing to do with mutable cells have to be completely rewritten:

$$
\begin{array}{l}
eval\ (Binary\ op\ a\ b)\ mem = \\
\quad \textbf{let}\ (av, mem') = eval\ a\ mem\ \textbf{in} \\
\quad\quad \textbf{let}\ (bv, mem'') = eval\ b\ mem'\ \textbf{in} \\
\quad\quad\quad (Binary\ op\ av\ bv, mem'')
\end{array}
$$

This form of composition is illustrated in the following diagram:

The memory input of the combined expression is passed to A. The value out and the memory output of A are given as inputs to B. The final result of the composition is the value of B and the memory that results from B. Note that the shape of the overall composition (the thick box) is the same as the shape of the basic stateful computations.

Similar transformations are needed for *Unary* operations and function definitions/calls.

Most languages with mutable state also have *sequences* of expressions, of the form $e1; e2; ...; eN$. It would be relatively easy to add a semicolon operator to the binary operators. In fact, C has such an operator: the expression $e1, e2$ evaluates $e1$ and then evaluates $e2$. The result of the expression is the value of $e2$. The value of $e1$ is

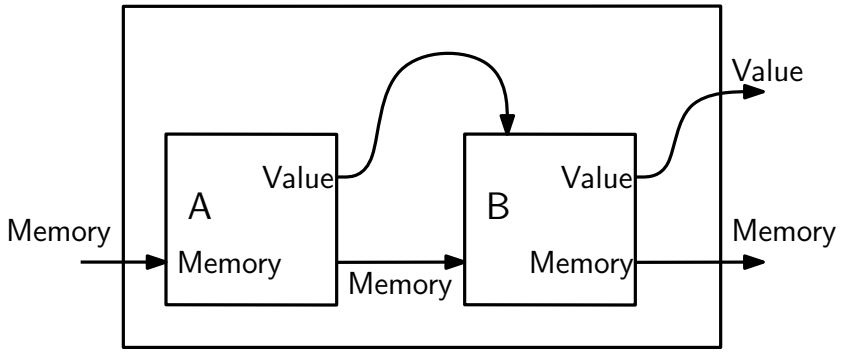

Figure 5.4: Composing stateful computations.

discarded. Note that **let** can also be used to implement sequences of operations: *e1*; *e2* can be represented as **let** *dummy* = *e1* **in** *e2* where *dummy* is a variable that is not used anywhere in the program.

5.2.5 Summary of Mutable State

Again, the take-away should be that mutation is messy when programmed in this way. Mutation affects every part of the evaluation process, even for parts that are not involved with creating or manipulating mutable cells.

Here is the complete code for mutable cells.

```
data Exp = Literal Value
   | Unary      UnaryOp Exp
   | Binary     BinaryOp Exp Exp
   | If         Exp Exp Exp
   | Variable   String
   | Let        String Exp Exp
   | Function   String Exp
   | Call       Exp Exp
   | Mutable    Exp-- new
   | Access     Exp-- new
   | Assign     Exp Exp-- new
  deriving (Eq, Show)
type Env = [(String, Value)]
```

All the existing cases of the evaluator are modified:

```
evaluate :: Exp → Env → Stateful Value
evaluate exp env mem = eval exp mem where
  eval (Literal v) mem     = (v, mem)
  eval (Unary op a) mem =
```

```
    let (av, mem') = eval a mem in
      (unary op av, mem')
  eval (Binary op a b) mem =
    let (av, mem') = eval a mem in
      let (bv, mem'') = eval b mem' in
        (binary op av bv, mem'')
  eval (If a b c) mem =
    let (av, mem') = eval a mem in
      eval (if fromBool av then b else c) mem'
  eval (Variable x) mem = (fromJust (lookup x env), mem)
  eval (Let x e body) mem =
    let (ev, mem') = eval e mem
        newEnv = (x, ev) : env
    in
      evaluate body newEnv mem'
  eval (Function x body) mem = (Closure x body env, mem)
  eval (Call f a) mem =
    let (Closure x body closeEnv, mem') = eval f mem
        (av, mem'') = eval a mem'
        newEnv = (x, av) : closeEnv
    in
        evaluate body newEnv mem''
```

Here are the mutation-specific parts of the evaluator:

```
eval (Mutable e) mem =
  let (ev, mem') = eval e mem in
    (Address (length mem'), mem' ++ [ev])
eval (Access a) mem =
  let (Address i, mem') = eval a mem in
      (access i mem', mem')
eval (Assign a e) mem =
  let (Address i, mem') = eval a mem in
    let (ev, mem'') = eval e mem' in
      (ev, update i ev mem'')
fromBool (Bool b) = b
```

Exercise 5.6: Errors and Mutable State

Write a version of *evaluate* that supports both error checking and mutable state.

5.3 Monads: Abstract Computational Strategies

At first glance it does not seem there is anything that can be done about the messy coding involved in implementing errors and mutable state. These features are *aspects* of the evaluation process, because they effect all the code of the evaluator, not just the part that directly involves the new feature.

What is worse is that combining the code for errors and mutable state is not possible without writing yet another completely different implementation. The *features* of our evaluator are not implemented in a modular way.

The concept of a *monad* provides a framework that allows different computational strategies to be invoked in a uniform way. The rest of this section shows how to derive the monad structure from the examples of error handing and mutable state given above. The basic strategy is to compare the two examples and do whatever is necessary to force them into a common structure, by moving details into helper functions. By defining appropriate helper functions that have the same interface, the two examples can be expressed in a uniform format.

5.3.1 Abstracting Simple Computations

The first step is to examine how the two evaluators deal with simple computations that return values. Here are the cases for Consider the way that the *Literal* expression is evaluated for both the Checked and the Stateful evaluators.

Checked	Stateful
$eval :: Exp \to Checked\ Value$	$eval :: Exp \to Stateful\ Value$
$eval\ (Literal\ v) = Good\ v$	$eval\ (Literal\ v)\ mem = (v, mem)$

One important point is that literal values never cause errors and they do not modify memory. They represent the simple good base case for a computation. In monad terminology, this operation is called *return* because it describes how to return a value from the computation. The return functions for checked and stateful computations are different, but they both have same interface: they take a value as input and output an appropriate checked or stateful value.

Checked	Stateful
$return_C :: Value \to Checked\ Value$	$return_S :: Value \to Stateful\ Value$
$return_C\ v = Good\ v$	$return_S\ v = \lambda\ mem \circ (v, mem)$

Figure 5.5: Modified Dilbert Cartoon

Using these return functions, the original *eval* code can be written so that the two cases are nearly identical. The details of how to deal with the checked or stateful values are hidden in the *return* helper functions.

Checked	Stateful
$eval :: Exp \to Checked\ Value$	$eval :: Exp \to Stateful\ Value$
$eval\ (Literal\ v) = return_C\ v$	$eval\ (Literal\ v) = return_S\ v$

5.3.2 Abstracting Computation Composition

The next step is to unify the case when there are multiple sub-expressions that must be evaluated. The binary operator provides a good example of multiple sub-expressions.

Checked	Stateful
$eval :: Exp \to Checked\ Value$	$eval :: Exp \to Stateful\ Value$
$eval\ (Binary\ op\ a\ b) =$	$eval\ (Binary\ op\ a\ b) =$
case $eval\ a$ **of**	$\lambda mem \circ$ **let** $(av, mem') = eval\ a\ mem$ **in**
$Error\ msg \to Error\ msg$	**let** $(bv, mem'') = eval\ b\ mem'$ **in**
$Good\ av \to$	$(Binary\ op\ av\ bv, mem'')$
case $eval\ b$ **of**	
$Error\ msg \to Error\ msg$	
$Good\ bv \to$	
$checked\_binary\ op\ av\ bv$	

TODO: note that the *mem* argument has become a lambda!

In this case computation proceeds in steps: first evaluate one expression (checking errors and updating memory) and then evaluating the second expression (checking errors and updating memory as appropriate). They both have a similar pattern of code for dealing with the evaluation of a and b. Factoring out the common parts as A and F, the core of the pattern is:

Checked	Stateful
case *first-part* **of**	$\lambda mem \circ$ **let** $(v, mem') =$ *first-part* mem **in**
$Error\ msg \to Error\ msg$	*next-part* $v\ mem'$

Good $v \rightarrow$ *next-part* v

This *first-part* corresponds to *eval* a or *eval* b in both the original versions. The *second-part* represents the remainder... just everything that appears after the main pattern, but with all the free variables made explicit. For the Checked case, the only variable needed in the *second-part* is the variable v that comes form the *Good* case. For the Stateful case, in addition to v the *second-part* also requires access to mem'

These patterns can be made explicit as a special operator that combines the two parts, where the second part is a function with the appropriate arguments. To be more concrete, these parts are converted into explicit variables. The *first-part* is named A and the *second-part*, which is a function, is named F:

Checked	Stateful
$A \rhd_C F =$	$A \rhd_S F =$
case A **of**	$\lambda mem \circ$ **let** $(v, mem') = A\ mem$ **in**
Error msg $\rightarrow$ *Error msg*	$F\ v\ mem'$
Good $v \rightarrow F\ v$	

These generic operators for Checked $\rhd_C$ and Stateful $\rhd_S$ computations abstract away the core pattern composing two Checked or Stateful computations. The family of operators $\rhd$ are called *bind* operators, because they bind together computations.

Using these operators, the *original* code can be written in simpler form:

Checked: $(eval\ a) \rhd_C (\lambda va \circ (eval\ b) \rhd_C (\lambda vb \circ checked\_binary\ op\ av\ bv))$

Stateful: $(eval\ a) \rhd_S (\lambda va \circ (eval\ b) \rhd_S (\lambda vb \circ \lambda mem \circ (Binary\ op\ av\ bv, mem)))$

All mention of *Error* and *Good* have been removed from the Checked version! The error 'plumbing' has been hidden. Most of the memory plumbing has been removed from the Stateful version, but there is still a little at the end. But the pattern that has emerged is the same one that was identified in the previous section, where the $return_S$ function converts a value (the result of *Binary op av bv*) into a default stateful computation. To see how this works, consider that

$return_S\ (Binary\ op\ av\ bv) \quad \equiv \quad \lambda mem \circ (Binary\ op\ av\ bv, mem)$

Using $return_S$ the result is:

Checked: $(eval\ a) \rhd_C (\lambda va \circ (eval\ b) \rhd_C (\lambda vb \circ checked\_binary\ op\ av\ bv))$

Stateful: $(eval\ a) \rhd_S (\lambda va \circ (eval\ b) \rhd_S (\lambda vb \circ return_S\ (Binary\ op\ av\ bv)))$

Now all references to memory have been removed in these cases. Of course, in the evaluation rules for *Mutable*, assignment, and access there will be explicit references to memory. Similarly, in the cases where errors are generated, for example for undefined variables, the code will still have to create *Error* values. What we have done here is examine the parts of the program that *don't* involve errors or memory, namely literals and binary operators, and figured out way to hide the complexity of error checking and mutable memory. This complexity has been hidden in two new operators, *return* and bind $\rhd$. The type of the bind operators is also interesting:

Checked: $\rhd_C :: Checked\ Value \to (Value \to Checked\ Value) \to Checked\ Value$

Stateful: $\rhd_S :: Stateful\ Value \to (Value \to Stateful\ Value) \to Stateful\ Value$

It should be clear that an consistent pattern has emerged. This is a *very* abstract pattern, which has to do with the structure of the underlying computation: is it a checked computation or a stateful computation? Other forms of computation are also possible.

5.3.3 Monads Defined

A *monad* m is a computational structure that involves three parts:

- A generic data type m
- A *return* function $return_m :: t \to m\ t$
- A *bind* function $\rhd_m :: m\ t \to (t \to m\ s) \to m\ s$

The symbol m gives the name of the monad and also defines the *shape* of the computation. A program that uses the monad m is called an m-computation. Examples of m in the previous section are *Checked* and *Stateful*. The instantiation of the generic type $m\ t$ at a particular type t represents an m-computation that produces a value of type t. For example, the type *Checked Int* represents an error-checked computation that produces an *Int*. Saying that it is a "checked computation" implies that it might produce an error rather than an integer. As another example, the type *Stateful String* represents a stateful computation that produces a value of type *String*. The fact that it is a "stateful computation" implies that there is a memory which is required as input to the computation, and that it produces an updated memory in addition to the string result.

The $return_m$ function specifies how values are converted into m-computations. The $return_m$ function has type $t \to m\ t$ for any type t. What this means is that it converts

a value of type t into an m-computation that just returns the value. It is important that the computation *just* returns the value, so, for example, it is not legal for the stateful return function to modify memory. Examples of return were given in the previous section.

The *bind* function $\rhd_m$ specifies how m-computations are combined together. In general the behavior of $A \rhd_m F$ is to perform the m-computation A and then pass the value it produces to the function F to create a second m-computation, which is returned as the result of the bind operation. Note that the A may not produce a value, in which case F is not called. This happens, for example, in the *Checked* monad, if A produces an error. At a high level, bind combines the computation A with the (parameterized) computation F to form a composite computation, which performs the effect of both A and F.

The type of bind given here is slightly more general than the type of bind used in the previous examples. In the previous examples, the type was $m\ t \to (t \to m\ t) \to m\ t$. However, it is possible for the return types of the two computations to differ. As long as the output of the first computation A can be passed to F, there is not problem.

TODO: mention the monad laws.

5.4 Monads in Haskell

The concept of a monad allows pervasive computational features, e.g. error checking and mutable state, to be defined in using a high-level interface that allows hides the plumbing involved in managing errors or state. Unfortunately, the resulting programs are still somewhat cumbersome to work with. Haskell provides special support for working with monads that makes them easy to use.

5.4.1 The Monad Type Class

Haskell allow monads to be defined very cleanly using *type classes*. The *Monad* class has the following definition:

class *Monad* m **where**
$(\gg\!=) :: m\ t \to (t \to m\ s) \to m\ s$
$return :: t \to m\ t$

It say that for a generic type m to be a monad, it must have two functions, bind $(\gg\!=)$ and *return*, with the appropriate types. The only difference from the definition given above is that the bind operator is called $\gg\!=$ rather than $\rhd$.

The type *Checked* is an instance of the *Monad* class:

```
instance Monad Checked where
  a >>= f =
    case a of
      Error msg → Error msg
      Good v → f v
  return v = Good v
```

It turns out to be a little more complex to define the stateful monad instance, so this topic is delayed until the end of this section. The code for error checking using monads is give in the Checked Monad file.

5.4.2 Haskell do Notation

Haskell also supports special syntax for writing programs that use monads. One problem with the monadic version of the program is apparent in the code for evaluation of binary expressions. The code given above is ugly because of the nested use of lambda functions. Here an attempt to make the Checked case more readable:

```
eval (Binary op a b) =
 (eval a) ▷C (λva∘
  (eval b) ▷C (λvb∘
   checked_binary op av bv))
```

The effect here is for *av* to be bound to the value produced by the *eval a*, and for *bv* to be bound to the result of *eval b*. Unfortunately, the variables come *to the right* of the expression that produces the value, which is not the way we naturally think about binding. Also, the nested lambdas and parenthesis are distracting.

Haskell has a special notation, the **do** notations, for the bind operator that allows the variables to be written in the right order. Using **do** the program above can be written as follows:

```
eval (Binary op a b) = do
  av ← eval a
  bv ← eval b
  checked_binary op av bv
```

Another benefit of the **do** notation is that the bind operator is implicit. Haskell type inference and the type class system arrange for the right bind operator to be selected automatically.

5.5 Using Haskell Monads

The messy evaluators for error checking and mutable state can be rewritten much more cleanly using monads.

5.5.1 Monadic Error Checking

Here is a version of error checking using the *Checked* monad defined above:

```
evaluate :: Exp → Env → Checked Value
evaluate exp env = eval exp where
  eval (Literal v)   = return v
  eval (Variable x) =
    case lookup x env of
      Nothing → Error ("Variable " ++ x ++ " undefined")
      Just v → return v
  eval (Unary op a) = do
    av ← eval a
    checked_unary op av
  eval (Binary op a b) = do
    av ← eval a
    bv ← eval b
    checked_binary op av bv
  eval (If a b c) = do
    av ← eval a
    case av of
      (Bool cond) → eval (if cond then b else c)
      _ → Error ("Expected boolean but found " ++ show av)
  eval (Let x e body) = do -- non-recursivecase
    ev ← eval e
    let newEnv = (x, ev) : env
    evaluate body newEnv
  eval (Function x body) = return (Closure x body env)
  eval (Call fun arg) = do
    funv ← eval fun
    case funv of
      Closure x body closeEnv → do
        argv ← eval arg
        let newEnv = (x, argv) : closeEnv
        evaluate body newEnv
      _ → Error ("Expected function but found " ++ show funv)
```

5.5.2 Monadic Mutable State

The full code for the stateful evaluator using monads is give in the Stateful Monad file.

The main complexity in defining a stateful monad is that monads in Haskell can only be defined for **data** types, which have explicit constructor labels. It is not possible to define a monad instance for the stateful type given in the Section on Stateful Computations, since it is a pure function type:

```
type Stateful t = Memory → (Value, Memory)
```

Instead it requires a data type that labels the funtion:

```
data Stateful t = Stateful (Memory → (t, Memory))
```

The data type is isomorphic to the function type, because it is just a type with a label.

```
instance Monad Stateful where
  return val = Stateful (λm → (val, m))
  (Stateful c) >>= f =
    Stateful (λm →
      let (val, m') = c m
          Stateful f' = f val
      in f' m')
```

Here is a version of evaluator using the *Stateful* monad defined above:

```
evaluate :: Exp → Env → Stateful Value
evaluate exp env = eval exp where
  eval (Literal v) = return v
  eval (Unary op a) = do
    av ← eval a
    return (unary op av)
  eval (Binary op a b) = do
    av ← eval a
    bv ← eval b
    return (binary op av bv)
  eval (If a b c) = do
    Bool cond ← eval a
    eval (if cond then b else c)
  eval (Let x e body) = do -- non-recursivecase
    ev ← eval e
    let newEnv = (x, ev) : env
    evaluate body newEnv
  eval (Variable x) = return (fromJust (lookup x env))
  eval (Function x body) = return (Closure x body env)
  eval (Call fun arg) = do
```

```
    Closure x body closeEnv ← eval fun
    argv ← eval arg
    let newEnv = (x, argv) : closeEnv
    evaluate body newEnv
eval (Mutable e) = do
    ev ← eval e
    newMemory ev
eval (Access a) = do
    Address i ← eval a
    readMemory i
eval (Assign a e) = do
    Address i ← eval a
    ev ← eval e
    updateMemory ev i
    return ev
```

The evaluate function depends on three helper functions that provide basic stateful computations to create memory cells, read memory, and udpate memory.

```
newMemory val = Stateful (λmem → (Address (length mem), mem ++ [val]))

readMemory i = Stateful (λmem → (access i mem, mem))

updateMemory val i = Stateful (λmem → ((), update i val mem))
```

Chapter 6

More Chapters on the way...

6.1 Abstract Interpretation and Types

6.2 Data Abstraction: Objects and Abstract Data Types

6.3 Algebra and Coalgebra

6.4 Partial Evaluation

6.5 Memory Management

Chapter 7

References

Abelson, Harold, and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed.. Cambridge, MA, USA: MIT Press.

Allen, John. 1978. *Anatomy of LISP*. New York, NY, USA: McGraw-Hill, Inc.

Friedman, Daniel P., and Mitchell Wand. 2008. *Essentials of Programming Languages, 3rd Edition*. 3rd ed.. The MIT Press.

Gabriel, Richard P. 1988. "The why of Y." *SIGPLAN Lisp Pointers* 2 (2): 15–25.

Gunter, Carl A. 1993. *Semantics of programming languages - structures and techniques*. *Foundations of computing*. MIT Press.

Krishnamurthi, Shriram. 2012. *Programming Languages: Application and Interpretation*.

Mitchell, John C., and Krzysztof Apt. 2001. *Concepts in Programming Languages*. New York, NY, USA: Cambridge University Press.

Scott, Dana. 1975. "Data types as lattices." In *ISILC Logic Conference*, 499:579–651.

Thomas, Scarlett. 2006. *The End of Mr. Y*. Houghton Mifflin Harcourt.

Wadler, P. 1987. "A critique of Abelson and Sussman or why calculating is better than scheming." *SIGPLAN Not.* 22 (3) (mar): 83–94. doi:10.1145/24697.24706. http://doi.acm.org/10.1145/24697.24706.