Anatomy of Programming Languages

William R. Cook

January 20, 2013

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*, *Essentials of Programming Languages* or *Concepts in Programming Languages*.

This book uses Haskell, a pure functional language. Phil Wadler gives some good reasons why to prefer Haskell over Scheme in his review of *Structure and Interpretation of Computer Programs*. I agree with most but not all of his points. For example, I do not care much for the fact that Haskell is lazy. Only small portions of 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*.

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". I recommend Learn You a Haskell for Great Good! or the Gentle Introduction To Haskell.

1.1.3.1 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.

language A *language* is a means to communicate information.

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.

syntax The structure of a language is called its *syntax*.
semantics The rules that define 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*.

metaprogram 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.

1.3 Introduction to Haskell Programming

The goal of this tutorial is to get the students familiar with Haskell Programming. Students are encouraged to bring their own laptops to go through the installation process of Haskell and corresponding editors, especially if they haven't tried to install Haskell before or if they had problems with the installation. In any case the lab machines will have Haskell installed and students can also use these machines for the tutorial.

1.3.1 Installing Haskell and related tools

If you have your laptop and have not installed Haskell yet, you can try to install it now. The Haskell platform is the easiest way to install Haskell in Windows or Mac OS.

In Ubuntu Linux you can use:

sudo apt − get install haskell − platform

1.3.2 Installing Emacs

We recommend using emacs as the editor for Haskell, since it is quite simple to use and it has a nice Haskell mode. In Ubuntu you can do the following to install emacs and the corresponding Haskell mode:

sudo apt − get install emacs
sudo apt − get install haskell − mode

In Mac OS you can try to use Aquamacs. Look here for a version of emacs for Windows.

However students are welcome to use whatever editor they prefer. If students are more comfortable using Vim, for example, they are welcome to. The choice of the editor is not important.

1.3.3 Basic steps in Haskell

In this tutorial we are going to implement our first Haskell code. To begin with, Haskell has normal data as in other programming languages. When writing Haskell code, lines that begin *Prelude*> are input to the Haskell interpreter, *ghci*, and the next line is the output.

Prelude $> 3 + 8 * 8$
67
Prelude $>$ *True* $\wedge$ *False*
False
Prelude $>$ `"this is a "` $+\!\!+$ `"test"`
`"this is a test"`

As illustrated above, Haskell has standard functions for manipulating numbers, booleans, and strings. Haskell also supports tuples and lists, as illustrated below:

Prelude $> (3 * 8,$ `"test"` $+\!\!+$ `"1"`$, \neg$ *True*$)$
$(24,$ `"test1"`$,$ *False*$)$
Prelude $> ()$
$()$
Prelude $> [1, 1 + 1, 1 + 1 + 1]$
$[1, 2, 3]$
Prelude $> 1 : [2, 3]$
$[1, 2, 3]$
Prelude $> 1 : 2 : 3 : [\,]$
$[1, 2, 3]$

Prelude > *length* $[1, 2, 3]$
3

Tuples are fixed length but can contain different types of values. Lists are variable length but must contain only one type of value. The colon function : adds a new item to the front of a list.

1.3.3.1 Functions

First, create a file called "Tutorial1.hs". After the creation of the file define the module header as follows:

module *Tutorial1* **where**

It is possible to define simple functions in Haskell. For example, consider a function that computes the absolute value of a number:

absolute :: *Int* → *Int*
absolute x = **if** $x < 0$ **then** $-x$ **else** x

This first line declares a function *absolute* with type *Int* → *Int*, which means that it takes a single integer as an input, and produces a single integer result. The second line defines *absolute* by naming the input value x and then providing an expression to compute the result. Everything is an expression in Haskell. This means that everything, including **if** expressions, have a value. Generally speaking Haskell definitions have the following basic form:

name arg_1 ... arg_n = *expression*

Note that in the right side of = (the body of the definition) is an expression. This is different from a conventional imperative language, where the body of a definition is usually a *statement*, which does not have a value.

expression An *expression* is a syntactic category for syntax that produces a value, and possibly has other side effects.

statement A *statement* is a syntactic category for syntax that has a side effect rather than producing a value.

side effect A *side effect* is an effect on the program state. Examples of side effects include allocating memory, assigning to memory, input/output, or exceptions.

In Haskell there are no statements, only expressions. As mentioned above, the **if** construct is also an expression.

Question 1: Are both of the following Haskell programs valid?

nested_if1 x = **if** (*absolute* $x \leqslant 10$)
  then x
  else *error* `"Only numbers between [-10, 10] allowed"`

```
nested_if2 x = if ((if x < 0 then - x else x) <= 10)
  then x
  else error "Only numbers between [-10, 10] allowed"
```

Once you have thought about it you can try these definitions on your Haskell file and see if they are accepted or not.

Question 2: Can you have nested **if** statements in Java, C or C++? For example, would this be valid in Java?

```
int m(int x) {
    if ((if (x < 0) -x else x) > 10)
      return x;
    else
      return 0;
}
```

1.3.3.2 Data Types

Data types in Haskell are defined by variants and components. In other words, a data type has a set of variants each with their own constructor, or tag, and each variant has a set of components or fields. For example, here is a data type for simple geometry:

```
data Geometry = Point Int Int          -- x and y
  | Circle            Int Int Int      -- x, y, and radius
  | Rectangle Int Int Int Int          -- top, left, right, bottom
```

A data type definition always begins with **data** and is followed by the name of the data type, in this case *Geometry*. There then follows a list of variants with unique tag names, in this case *Point*, *Circle*, and *Rectangle*, which are separated by a vertical bar |. Following each constructor tag is a list of data types specifying the types of components of that variant. A *Point* has two components, both integers. A *Circle* has three components, and a rectangle has 4 integer components. One issue with this notation is that it is not clear what the components *mean*. The meaning of each component is specified in a comment.

The tags are called *constructors* because they are defined to construct values of the data type. For example, here are three expressions that construct three geometric objects:

```
Point 3 10
Circle 10 10 10
Rectangle 0 0 100 10
```

Data types can also be recursive, allowing the definition of complex data types.

```
data Geometry = Point Float Float          -- x and y
  | Circle          Float Float Float      -- x, y, and radius
  | Rectangle Float Float Float Float       -- top, left, right, bottom
  | Composite [Geometry]                    -- list of geometry objects
```

Here is a composite geometric value:

```
Composite [Point 3 10, Circle 10 10 10, Rectangle 0 0 100 10]
```

Two special cases of data types are *enumerations*, which only have variants and no components, and *structures* which only have a single variant, with multiple components. An example enumeration is the data type of days of the week:

```
data Days = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

In this case the tags are constants. One well known enumeration is the data type *Boolean*:

```
data Boolean = True | False
```

An example of a structure is a person data type:

```
data Person = Person String Int Int   -- name, age, shoe size
```

Note that the data type and the constructor tag are the same. This is common for structures, and doesn't cause any confusion because data types and constructor functions are always distinguished syntactically in Haskell. Here is an example person:

```
Person "William" 42 10
```

A Haskell program almost always includes more than one data type definition.

1.3.3.3 Parametric Polymorphism and Type-Inference

We have seen that Haskell supports definitions with a type-signature or without. When a definition does not have a signature, Haskell infers one and it is still able to check whether some type-errors exist or not. For example, for the definition

```
newline s = s ++ "\n"
```

Haskell is able to infer the type: $String \rightarrow String$.

In Haskell strings are represented as lists of characters, whose type is written $[Char]$. The operator $++$ is a built-in function in Haskell that allows concatenating two lists. However for certain definitions it appears as if there is not enough information to infer a type. For example, consider the definition:

```
identity x = x
```

This is the definition of the identity function: the function that given some argument returns that argument unmodified. This is a perfectly valid definition, but what type should it have? The answer is:

$identity :: a \to a$

The function *identity* is a (parametrically) polymorphic function. Polymorphism means that the definition works for multiple types; and this type of polymorphism is called parametric because it results from abstracting/parameterizing over a type. In the type signature the a is a type variable (or parameter). In other words it is a variable that can be replaced by some valid type (for example *Int*, *Char* or *String*). Indeed the *identity* function can be applied to any type of values. Try the following in *ghci*, and see what is the resulting type:

```
identity 3
identity 'c'
identity identity   -- you may try instead :t identity
```

Question 3: Have you seen this form of polymorphism in other languages? Perhaps under a different name?

1.3.3.4 Pattern matching

One feature that many functional languages support is pattern matching. Pattern matching plays a role similar to conditional expressions, allowing us to create definitions depending on whether the input matches a certain pattern. For example, the function *hd* (to be read as "head") given a list returns the first element of the list:

```
hd :: [a] -> a
hd []     = error "cannot take the head of an empty list!"
hd (x:xs) = x
```

In this definition, the pattern [] denotes the empty list, whereas the pattern $(x : xs)$ denotes a list where the first element (or the head) is x and the remainder of the list is xs. The parentheses are required because $hd\ x : xs$ would group as $(hd\ x) : xs$ which is meaningless. Note that instead of a single clause in the definition there are now two clauses for each case.

Question 4: Define a *tl* function that given a list, drops the first element and returns the remainder of the list. That is, the function should behave as follows for the sample inputs:

```
Prelude> tl [1,2,3]
[2,3]

Prelude> tl ['a','b']
['b']
```

More Pattern Matching: Pattern matching can be used with different types. For example, here are two definitions with pattern matching on tuples and integers:

$$
\begin{aligned}
&\mathit{first} :: (a, b) \to a \\
&\mathit{first}\ (x, y) = x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isZero} :: \mathit{Int} \to \mathit{Bool} \\
&\mathit{isZero}\ 0 = \mathit{True} \\
&\mathit{isZero}\ n = \mathit{False}
\end{aligned}
$$

1.3.3.5 Pattern Matching Data Types

Functions are defined over data types by *pattern matching*. For example, to compute the area of a geometric figure, one would define:

$$
\begin{aligned}
&\mathit{area} :: \mathit{Geometry} \to \mathit{Float} \\
&\mathit{area}\ (\mathit{Point}\ x\ y) \quad = 0 \\
&\mathit{area}\ (\mathit{Circle}\ x\ y\ r) \quad = \mathit{pi} * r \uparrow 2 \\
&\mathit{area}\ (\mathit{Rectangle}\ t\ l\ r\ b) = (b - t) * (r - l) \\
&\mathit{area}\ (\mathit{Composite}\ cs) = \mathit{sum}\ [\mathit{area}\ c \mid c \leftarrow cs]
\end{aligned}
$$

1.3.3.6 Recursion

In functional languages mutable state is generally avoided and in the case of Haskell (which is purely functional) it is actually forbidden. So how can we write many of the programs we are used to? In particular how can we write programs that in a language like C would normally be written with some mutable state and some type of loop? For example:

```
int sum_array(int a[], int num_elements) {
   int i, sum = 0;
   for (i = 0; i < num_elements; i++) {
     sum = sum + a[i];
   }
   return sum;
}
```

The answer is to use recursive functions. For example here is how to write a function that sums a list of integers:

$$
\begin{aligned}
&\mathit{sumList} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sumList}\ [\,] = 0 \\
&\mathit{sumList}\ (x : xs) = x + \mathit{sumList}\ xs
\end{aligned}
$$

Question 5: The factorial function can be defined as follows:

n! =	1	if n = 0
	n * (n-1)!	if n > 0

Translate this definition into Haskell using recursion and pattern matching.

Question 6: The Fibonacci sequence is:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$$

write a function:

$$\mathit{fib} :: \mathit{Int} \rightarrow \mathit{Int}$$

that given a number returns the corresponding number in the sequence. (If you don't know Fibonacci numbers you may enjoy finding the recurrence pattern; alternatively you can look it up in WikiPedia).

Question 7: Write a function:

$$\mathit{mapList} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$$

that applies the function of type $a \rightarrow b$ to every element of a list. For example:

$\mathit{Prelude} > \mathit{mapList}\ \mathit{absolute}\ [4, -5, 9, -7]$
$[4, 5, 9, 7]$

Question 7: Write a function that given a list of characters returns a list with the corresponding ASCII number of the character. Note that in Haskell, the function *ord*:

$$\mathit{ord} :: \mathit{Char} \rightarrow \mathit{Int}$$

gives you the ASCII number of a character. To use it add the following just after the module declaration:

import *Data.Char*

to import the character handling library.

Question 8: Write a function *filterList* that given a predicate and a list returns another list with only the elements that satisfy the predicate.

$$\mathit{filterList} :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow [a]$$

For example, the following filters all the even numbers in a list (even is a built-in Haskell function):

$\mathit{Prelude} > \mathit{filterList}\ \mathit{even}\ [1, 2, 3, 4, 5]$
$[2, 4]$

Question 9: Haskell has a function *zip*:

$$zip :: [a] \to [b] \to [(a, b)]$$

that given two lists pairs together the elements in the same positions. For example:

$$Prelude > zip\ [1,2,3]\ [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}]$$
$$[(1, \texttt{'a'}), (2, \texttt{'b'}), (3, \texttt{'c'})]$$

For lists of different lengths it truncates the larger list:

$$Prelude > zip\ [1,2,3]\ [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \texttt{'d'}]$$
$$[(1, \texttt{'a'}), (2, \texttt{'b'}), (3, \texttt{'c'})]$$

Write a definition *zipList* that implements the *zip* function.

Question 10: Define a function *zipSum*:

$$zipSum :: [Int] \to [Int] \to [Int]$$

that sums the elements of two lists at the same positions. Suggestion: You can define this function recursively, but a simpler solution can be found by combining some of the previous functions.

Assignment 0 (Optional and not graded):

Your first assignment is to try to complete as many questions in the tutorial as you can.

Chapter 2

Expressions, Syntax, and Evaluation

This chapter introduces three fundamental concepts in programming languages: *expressions*, *syntax* and *evaluation*. These concepts are illustrated by a simple language of arithmetic expressions.

expression An *expression* is a combination of variables, values and operations over these values. For example, the arithmetic expression $2 + 3$ uses two numeric values 2 and 3 and an operation + that operates on numeric values.

syntax The *syntax* of an expression prescribes how the various components of the expressions can be combined. In general it is not the case that the components of expressions can be combined arbitrarily: they must obey certain rules. For example 2 3 or ++ are not valid arithmetic expressions.

evaluation Each expression has a meaning (or value), which is defined by the *evaluation* of that expression. Evaluation is a process where expressions composed of various components get simplified until eventually we get a value. For example evaluating $2 + 3$ results in 5.

2.1 Simple Language of Arithmetic

Lets have a closer look at the language of arithmetic, which is familiar to every grade-school child.

```
4
-5+6
3--2--7
```

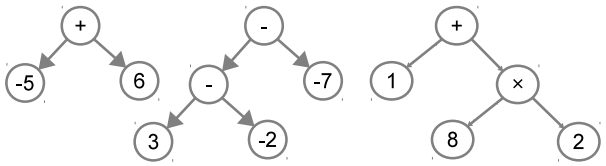

Figure 2.1: Graphical illustration of abstract structure

```
3*(8+5)
1+(8*2)
1+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 operations are performed left to right.

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 conventions for expressing arithmetic expressions in written form. In other words, there isn't any confusion about what negative numbers 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:

These pictures are similar to *sentence diagramming* that is taught in grade school to explain the structure of English.

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

2.2 Syntax

Syntax comes in two forms: abstract and concrete.

abstract syntax The conceptual structure (illustrated by the pictures) is called the *abstract syntax* of the language.

concrete syntax 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. To make these concepts more precise, we show how to represent abstract syntax as a data structure, and how to define a *parser*, which converts from the concrete written form to the abstract syntax.

parser A parser is a program that converts concrete syntax into abstract syntax. A parser typically inputs text and outputs the abstract syntax structure.

2.2.1 Abstract Syntax in Haskell

This section describes how to represent abstract syntax using Haskell. The code for this section is found in the Simple.zip 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. The symbols *Number*, *Add*, *Subtract* etc are the *constructors* of the data type. The types that follow the constructors are the components of the data type. Thus a *Number* expression has an integer component, while the other constructors all have two expression components. A number that appears in a program is called a *literal*.

literal A *literal* is a constant value that appears in a program. A literal can be a number, string, boolean, or other constant.

As an example of abstract syntax, consider this expression:

```
-- 3 - -2 - -7
t1 = Subtract (Subtract (Number 3) (Number (-2))) (Number (-7))
```

NOTE: It is not legal to write *Subtract* 3 (-2) because 3 and -2 are of type *Int* not *Exp*. Also, Haskell requires parentheses around negative numbers when used with other infix operators to prevent parsing ambiguity.

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 $1 + 8 * 2$ uses 47 characters to create the corresponding Haskell data structure. This is not a defect of Haskell, it is merely because we haven't developed a way to convert concrete syntax into abstract syntax.

2.2.2 Concrete Syntax and Grammars

The concrete syntax of a language describes how the abstract concepts in the language are represented as text. For example, lets consider how to convert the string "3 + 81 ∗ 2 " into the abstract syntax *Add* (*Number* 3) (*Multiply* (*Number* 81) (*Number* 2)). The first step is to break a text up into *tokens*.

2.2.2.1 Tokens

Tokens are the basic units of a language. In English, for example, words are tokens. But English also uses many symbol tokens, including ".", "!", "?", "(" and ")". In the example "3 + 81 ∗ 2 " the tokens are 3, " + ", 81, " ∗ ", and 2. It is also important to classify tokens by their kind. The tokens 3, 81 and 2 are sequences of digits. The tokens " + " and " ∗ " are *symbol* tokens.

token A token is the basic syntactic unit of a language. Tokens can be individual characters, or groups of characters. Token are often classified into kinds, for example *integers*, *strings*, *identifiers*.

identifier An identifier is a string of characters that represents a name. Identifiers usually begin with a alphabetic character, then continue with one or more numeric digits or special symbols. Special symbols that may be used include underscore "_" and "$", but others may be included.

Tokens are typically as simple as possible, and they must be recognizable without considering any context. This means that the integer " − 23 " might not be a good token, because it contains the symbol " − ", which is also used in other contexts.

More complex languages may have other kinds of tokens (other common kinds of token are *keyword* and *identifier* tokens, which are discussed later in the book). Token kinds are similar to the kinds of words in English, where some words are *verbs* and other words are *nouns*.

The following data structure is useful for representing basic tokens.

```
data Token = Digits Int
  | Symbol String
```

A *Token* is either an integer token or a symbol token with a string. For example, the tokens from the string "3 + 81 ∗ 2 " are:

Digits 3
Symbol `"+"`
Digits 81
Symbol `"*"`
Digits 2

The Lexer.hs file contains the code for a simple lexer that creates tokens in this form. It defines a function *lexer* that transforms a string (i.e. a list of characters) into a list of tokens. The *lexer* function takes as input a list of symbols and a list of keywords.

2.2.2.2 Grammars

Grammars are familiar from studying natural languages, but they are especialy important when studying computer languages.

grammar A *grammar* is a set of rules that specify how tokens can be placed together to form valid expressions of a language.

To create a grammar, it is essential to identify and *name* the different parts of the language.

syntactic category The parts of a language are called *syntactic categories*. For example, in English there are many different parts, including *verb*, *noun*, *gerund*, *prepositional phrase*, *declarative sentence*, etc. In software langauges, example syntactic categories include *expressions*, *terms,* functions, types*, or* classes*.

It is certainly possible to be a fluent English speaker without any explicit awareness of the rules of English or the names of the syntactic categories. How many people can identify a gerund? But understanding syntactic categories is useful for studying a language. Creating a complete syntax of English is quite difficult, and irrelevant to the purpose of this book. But defining a grammar for a (very) small fragment of English is useful to illustrate how grammars work. Here is a simple grammar:

Sentence : *Noun Verb* | *Sentence PrepositionalPhase*
PrepositionalPhase : *Preposition Noun*
Noun : `'Dick'` | `'Jane'` | `'Spot'`
Verb : `'runs'` | `'talks'`
Preposition : `'to'` | `'with'`

The names *Sentence*, *PrepositionalPhase*, *Noun*, *Verb*, and *Preposition* are the syntactic categories of this grammar. Each line of the grammar is a *rule* that specifies a syntactic category, followed by a colon (:) and then sequence of alternative forms for that syntactic category. The words in quotes, including *Dick*, *Jane*, and *Runs* are the tokens of the language.

Here is a translation of the grammar into English:

- a *sentence* is either:

  - a *noun* followed by a *verb*, or
  - a *sentence* followed by a *prepositional phase*

- a *prepositional phase* is a *preposition* followed by a *noun*
- a *noun* is one of "Dick", "Jane", or "Spot"
- a *verb* is either "runs" or "talks"
- a *preposition* is either "to" or "with"

Some sentences in the language defined by this grammar are:

Dick talks
Jane runs
Dick runs with Spot
Dick runs to Jane with Spot
Spot runs to Jane to Dick to Jane to Dick

There are also some sentences that don't make much sense:

Dick talks with Dick
Jane runs to Jane to Jane to Jane to Jane

These sentences are *syntactically correct* because they follow the pattern specified by the grammar, but that doesn't ensure that they are meaningful.

To summarize, here is a formal description of grammars.

production rule A *production rule* defines how a non-terminal can be translated into a sequence of tokens and other syntactic categories.
terminal A *terminal* is a token used in a grammar rule.
non-terminal The *non-terminals* are the names of syntactic categories used in a grammar.

The intuition behind the use of the terms *non-terminal* and *terminal* is that the grammar rules *produce* sequences of tokens. Starting with a start symbol, a grammar can be viewed as generating sequences of non-terminal/terminals by replacing the left side with the right side. As long as the resulting sentence still has non-terminals that haven't been replaced by real words, the process is not done (not terminated).

2.2.2.3 Grammar Actions and Construction of Abstract Syntax

In addition to specifying the set of legal sentences, a grammar can also specify the meaning of those sentences. Rather than try to specify a meaning for English, here is a simple grammar for arithmetic expressions, which has been annotated to specify the meaning that should be associated with each pattern in the grammar:

```
Exp : digits          { Number $ 1 }
    | '-' digits      { Number (- $ 2) }
    | Exp '+' Exp     { Add $ 1 $ 3 }
    | Exp '-' Exp     { Subtract $ 1 $ 3 }
    | Exp '*' Exp     { Multiply $ 1 $ 3 }
    | Exp '/' Exp     { Divide $ 1 $ 3 }
    | '(' Exp ')'     {$2}
```

This grammar is similar to the one given above for English, but each rule includes an *action* enclosed in curly braces {...}. The action says what should happen when that rule is recognized. In this case, the action is some Haskell code with calls to *constructors* to create the abstract syntax that corresponds to the concrete syntax of the rule. The special syntax \$*n* in an action means that the value of the *nth* item in the grammar rule should be used in the action. For example, in the last rule the \$2 refers to the second item in the parenthesis rule, which is *Exp*.

Written out explicitly, this grammar means:

- An *expression* Exp is either
  - a digit token
    - which creates a *Number* with the integer value of the digits
  - a minus sign followed by a digits token
    - which creates a *Number* with the negative of the integer value of the digits
  - an expression followed by a + followed by an expression
    - which creates an *Add* node containing the value of the expressions
  - an expression followed by a − followed by an expression
    - which creates a *Subtract* node containing the value of the expressions
  - an expression followed by a ∗ followed by an expression
    - which creates a *Multiply* node containing the value of the expressions
  - an expression followed by a / followed by an expression
    - which creates a *Divide* node containing the value of the expressions
  - a open parenthesis '(' followed by an expression followed by a close parenthesis ')'
    - which returns the expression an throws away the parentheses

Given this lengthy and verbose explanation, I hope you can see the value of using a more concise notation!

Just like other kinds of software, there are many design decisions that must be made in creating a grammar. Some grammars work better than others, depending on the situation.

2.2.2.4 Ambiguity, Precedence and Associativity

One problem with the straightforward grammar is allows for *ambiguity*.

ambiguity A sentence is ambiguous if there is more than one way that it can be derived by a grammar.

For example, the expression $1 - 2 - 3$ is ambiguous because it can be parsed in two ways to create two different abstract syntax trees [TODO: define "parse"]:

Subtract (*Number* 1) (*Subtract* (*Number* 2) (*Number* 3))
Subtract (*Subtract* (*Number* 1) (*Number* 2)) (*Number* 3)

TODO: show the parse trees? define "parse tree"

The same abstract syntax can be generated by parsing $1 - (2 - 3)$ and $(1 - 2) - 3$. We know from our training that the second one is the "correct" version, because subtraction operations are performed left to right. The technical term for this is that subtraction is *left associative*. (note that this use of the associative is not the same as the mathematical concept of associativity.) But the grammar as it's written doesn't contain any information associativity, so it is ambiguous.

Similarly, the expression $1 - 2 * 3$ can be parsed in two ways:

Subtract (*Number* 1) (*Multiply* (*Number* 2) (*Number* 2))
Multiply (*Subtract* (*Number* 1) (*Number* 2)) (*Number* 2)

The same abstract syntax can be generated by parsing $1-$ and $(1 - 2)\}$ 3. Again we know that the first version is the correct one, because multiplication should be performed before subtraction. Technically, we say that multiplication has higher *precedence* than subtraction.

precedence *Precedence* is an order on grammar rules that defines which rule should apply first in cases of ambiguity. Precedence rules are applied before associativity rules.

associativity Associativity specifies whether binary operators are grouped from the *left* or the *right* in order to resolve ambiguity.

The grammar can be adjusted to express the precedence and associativity of the operators. Here is an example:

Exp : *Term* {$1}
Term : *Term* '+' *Factor* { *Add* $ 1 $ 3}
| *Term* '-' *Factor* { *Subtract* $ 1 $ 3}
| *Factor* {$1}
Factor : *Factor* '*' *Primary* { *Multiply* $ 1 $ 3}
| *Factor* '/' *Primary* { *Divide* $ 1 $ 3}

```
    | Primary                    {$1}
Primary : digits {Number $ 1}
    | '-' digits    {Number (- $ 2)}
    | '(' Exp ')' {$2}
```

This grammar works by splitting the *Exp* non-terminal of the original grammar into multiple non-terminals, each of which represents a precedence level. The low-precedence operators + and − are grouped into a *Term* non-terminal, which allows addition and subtraction of factors. A *Factor* non-terminal allows multiplication and division, but does not allow addition. A *Primary* non-terminal allows primitive constructs, including a parenthesized expression, which allows addition and subtraction to be used under multiplication.

2.2.2.5 Parser Generators

The grammar notation used above is also a language. It is a language of grammars.

How to create simple grammars using the Happy Parser Generator.

2.3 Evaluating Arithmetic Expressions

The normal meaning assigned to arithmetic expressions is the evaluation of the arithmetic operators to compute a final answer. This section describes how to define a simple evaluator as defined in the Simple.zip file. This evaluation process is defined by cases in Haskell:

```
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
```

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 t1
  putStrLn "Produces the following result:"
  putStr "   "
  print (evaluate t1)
```

The output is

Evaluating the following expression :
  Subtract (*Subtract* (*Number* 3) (*Number* (−2))) (*Number* (−7))
Produces the following result :
  12

TODO: Explain Show

2.3.1 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 zip file. For example, consider this small expression:

evaluate (*parseExp* `"8 / 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.

2.4 Object Language and Meta-Language

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

To implement our interpreter we have to deal with two different languages. On the one hand Haskell is the language being used to implement the interpreter, and on the other hand a simple language of arithmetic is being defined and interpreted. The use of two languages can lead to some ambiguity and confusion. For example when referring to an expression $2 + 3$ do we mean an expression of the implementation language (in this case Haskell), or do we mean an expression of the language being defined (in this case arithmetic)?

In general it is useful to have different terminology to distinguish the roles of the two different languages. We say that Haskell is the *meta language*, whereas the language of arithmetic is the *object language* being defined. The term meta-language is used to denote the language used for the implementation. The term object language (or just language) is used to denote the language that is the

object of our study, or in other words the language being defined. Using this terminology allows us to eliminate sources of potential ambiguity. For example when referring to the meta-language expression $2 + 3$, it becomes clear that what is meant is an expression of the implementation language. Similarly when referring to the object language expression $2+3$ what is meant is an expression of the language being defined. When there is space for potential confusion we will use this terminology to disambiguate meaning. However, we also want to avoid being repetitive and overly explicit, especially when it is clear by the context which of the two meanings is intended. By default it is safe to assume that when we are not explicit about which of the two languages are we talking about what we mean is the object language. In other words, when referring to the expression $2 + 3$ the default meaning is the object language expression $2 + 3$.

Chapter 3

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.

variable A *variable* is a symbol that refers to a value.

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 zip 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
  deriving (Eq, Show)
```

An association of a variable x with a value v is called a *binding*, which can be written $x \mapsto v$.

binding A *binding* $x \mapsto v$ is an association of a variable with its value.

Bindings can be represented in Haskell as a pair. For example, the binding of $x \mapsto 5$ can be represented as $(\text{“}x'', 5)$.

3.0.1 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 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*.

mutation Mutation refers to the ability of a variable or data structure to change over time.

This seems to be 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. Mutable variables are discussed in the Section on Mutable State later in this book. 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 here. 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.

3.1 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 the substitution *val*.

Running a few tests produces the following results:

```
substitute1 ("x", 5) Add (Variable "x") (Number 2)
==> Add (Number 5) (Number 2)

substitute1 ("x", 5) Number 32
==> Number 32

substitute1 ("x", 5) Variable "x"
==> Number 5

substitute1 ("x", 5) Add (Multiply (Variable "x") (Variable "x")) (Variable "x")
==> Add (Multiply (Number 5) (Number 5)) (Number 5)

substitute1 ("x", 5) Add (Add (Variable "x") (Multiply (Number 2) (Variable "y"))) (Variable "z")
==> Add (Add (Number 5) (Multiply (Number 2) (Variable "y"))) (Variable "z")
```

Note that the test case prints the concrete syntax of the expression in square brackets, as $[x + 2]$. The print format represents the more longwinded abstract

syntax representation of the expression $x + 2$: *Add* (*Variable*"x'') (*Number* 2). So the first expression corresponds to the following piece of real Haskell code:

substitute1 (`"x"`, 5) (*Add* (*Variable* `"x"`) (*Number* 2))
$==>$ $[5 + 2]$

However it will be useful to us to use the pseudo-code $[x + 2]$ instead, since it can be quite difficult to read abstract syntax directly.

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.

3.2 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*.

environment An *environment* is a mapping from variables to values.

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

$e1 = [(\texttt{"x"}, 3), (\texttt{"y"}, -2)]$

The corresponding type is

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

An important operation on environments is *variable lookup*. Variable lookup is an operation that given a variable name and an environment looks up that variable in the environment. For example:

- lookup "x" in $e1 \longrightarrow 3$
- lookup "y" in $e1 \longrightarrow -2$

In each case the corresponding value of the variable being looked up in the environment is returned. However what happens when a variable that is not in the environment is looked up?

- lookup "z" in $e1 \longrightarrow$???

In this case variable lookup fails, and it is necessary to deal with this possibility by signaling an error or triggering an exception.

unbound variable An *unbound variable* is a variable that does not have a binding. Unbound variables are a common form of errors in programs.

Haskell already provides a function, called *lookup*, that implements the functionality that is needed for variable lookup. The type of *lookup* is as follows:

$$lookup :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$$

This type is more general than what we need for variable lookup, but we can see that if a is instantiated to *String* and b is instantiated to *Int*, then the type is almost what we expect: $String \rightarrow Env \rightarrow Maybe\ Int$. The return type of lookup ($Maybe\ b$) deserves a little bit more of explanation. The type *Maybe* is part of the Haskell libraries and it is widely used. The definition is as follows:

$$\textbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

The basic intuition is that *Maybe* is a container that may either contain a value of type a ($Just\ a$) or no value at all ($Nothing$). This is exactly what we need for *lookup*: when *lookup* succeeds at finding a variable in the environment it can return the looked-up value v using $Just\ v$; otherwise if variable lookup fails *lookup* returns *Nothing*. The *Maybe* datatype provides us with a way to deal with lookup-up errors gracefully and later to detect such errors using pattern-matching to check whether the result was $Just\ v$ or *Nothing*.

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.

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

$$e1 = [(\texttt{"x"}, 3), (\texttt{"y"}, -2)]$$

substitute e1 Add (*Variable* `"x"`) (*Number* 2)
$==>$ *Add* (*Number* 3) (*Number* 2)

substitute e1 Number 32
$==>$ *Number* 32

substitute e1 Variable `"x"`
$==>$ *Number* 3

substitute e1 Add (*Multiply* (*Variable* `"x"`) (*Variable* `"x"`)) (*Variable* `"x"`)
$==>$ *Add* (*Multiply* (*Number* 3) (*Number* 3)) (*Number* 3)

substitute e1 Add (*Add* (*Variable* `"x"`) (*Multiply* (*Number* 2) (*Variable* `"y"`))) (*Variable* `"z"`)
$==>$ *Add* (*Add* (*Number* 3) (*Multiply* (*Number* 2) (*Number* (-2)))) (*Variable* `"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.

3.2.1 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:

let $x = 3$ **in** $2 * x + 5$

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

$2 * ($**let** $x = 3$ **in** $x + 5)$

Haskell's **where** allows a declarative style (vs **let**'s expressive style) that states an algorithm in a manner that that assumes equations shall be eventually satisfied.

There are nuanced performance and binding differences between the two, but most uses are relatively straightforward:

isOdd n = **if** *predicate* **then** *t* **else** *f*
  where
    predicate = *odd n*
    t = *True*
    f = *False*

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

let $x = 3$ **in let** $y = x * 2$ **in** $x + y$

which is equivalent to

let $x = 3$ **in** (**let** $y = x * 2$ **in** $x + y$)

Since the language we are defining is a subset of JavaScript, we will use its syntax. In general a *variable declaration* expression has the following concrete syntax:

var variable = *bound-expression*; *body*

The meaning of a *variable declaration* 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 our Haskell code, a *variable declaration* expression can be represented by adding another case to the definition of expressions:

data *Exp* = ...
  | *Declare String Exp Exp*

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

3.2.2 Scope

Variables have a range of text in which they are defined.

scope The *scope* of a variable is the portion of the text of a program in which a variable is defined.

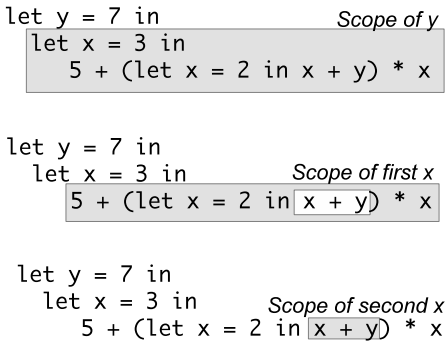

Figure 3.1: Variable Scope

Normally the scope of a local variable is all of the body of the declaration 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:

In this example Figure [Variable Scope] 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

3.2.3 Substituting into Variable Declarations

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 *var* $x = e$; *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.

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

In the *Declare* 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 variable declaration.

TODO: need some test cases here

3.2.4 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:

$x + 3$
$var\ x = 2; x * y$
$(var\ x = 3; x) * x$

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:

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

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. 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.

static A *static* property of a program can be determined by examining the text of the program but without executing or evaluating it.

dynamic A *dynamic* property of a program can only be determined by evaluating the program.

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).

3.2.5 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
  | Declare         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 (Declare x exp body) = Declare 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 (Declare x exp body) =
  evaluate (substitute1 (x, evaluate exp) body)
```

3.3 Evaluation using Environments

For the basic evaluator substitution and evaluation were completely separate, but the evaluation rule for variable declarations involves substitution. One consequence of this rule is that the body of every variable declaration is copied, because substitution creates a copy of the expression with variables substituted. When variable declarations are *nested*, the body of the inner variable declaration is copied multiple times. In the following example, the expression $x * y * z$ is copied three times:

```
var x = 2;
var y = x + 1;
var z = y + 2;
x * y * z
```

The steps are as follows:

Step	Result
initial expression	$var\ x = 2;$ $var\ y = x + 1;$ $var\ z = y + 2;$ $x * y * z$
evaluate bound expression	$2 \Rightarrow 2$
substitute x $\mapsto$ 2 in body	$var\ y = 2 + 1;$ $var\ z = y + 2;$ $2 * y * z$
evaluate bound expression	$2 + 1 \Rightarrow 3$
substitute y $\mapsto$ 3 in body	$var\ z = 3 + 2;$ $2 * 3 * z$
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 variable declaration, instead the variable declaration can add another binding to the list of substitutions being performed.

```
  -- Evaluate an expression in an environment
evaluate :: Exp → Env → Int
evaluate (Number i) env = i
evaluate (Add a b) env      = evaluate a env + evaluate b env
evaluate (Subtract a b) env = evaluate a env − evaluate b env
evaluate (Multiply a b) env = evaluate a env ∗ evaluate b env
evaluate (Divide a b) env   = evaluate a env `div` evaluate b env
evaluate (Variable x) env   = fromJust (lookup x env)
evaluate (Declare x exp body) env = evaluate body newEnv
  where newEnv = (x, evaluate exp env) : env
```

In most cases the environment argument is simply passed unchanged to all recursive calls to *evaluate*. But in the final case, for *Declare*, the environment *does* change.

The case for *Declare* 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 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*?

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

Environment	Evaluation
$\emptyset$	$var\ x = 2;$
	$var\ y = x + 1;$
	$var\ z = y + 2;$
	$x * y * z$
	evaluate bound expression 2
$\emptyset$	$2 \Rightarrow 2$
	add new binding for x and evaluate body of variable declaration
$x \mapsto 2$	$var\ y = x + 1;$
	$var\ z = y + 2;$
	$x * y * z$
	evaluate bound expression $x + 1$
$x \mapsto 2$	$x + 1 \Rightarrow 3$
	add new binding for y and evaluate body of variable declaration
$y \mapsto 3,\ x \mapsto 2$	$var\ z = y + 2;$
	$x * y * z$
	evaluate bound expression $y + 2$
$y \mapsto 3,\ x \mapsto 2$	$y + 2 \Rightarrow 5$
	add new binding for z and evaluate body of variable declaration
$z \mapsto 5,\ y \mapsto 3,\ x \mapsto 2$	$x * y * z \Rightarrow 30$

In the *Declare* case of *evaluate*, a new environment *newEnv* is created and used as the environment 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:

$var\ x = 9;$
$var\ x = x * x;$
$x + x$

Environment	Evaluation
$\emptyset$	$var\ x = 9; var\ x = x * x; x + x$
	evaluate bound expression 9
$\emptyset$	$9 \Rightarrow 9$
	add new binding for x and evaluate body of variable declaration
$x \mapsto 9$	$var\ x = x * x; x + x$
	evaluate bound expression $x * x$
$x \mapsto 9$	$x * x \Rightarrow 81$
	add new binding for x and evaluate body of variable declaration

Environment	Evaluation
$x \mapsto 81,\ x \mapsto 9$	$x + x \Rightarrow 162$

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

$$\begin{array}{l} \mathit{var}\ x = 3; \\ (\mathit{var}\ y = 3 * x; 2 + y) + (\mathit{var}\ z = 7 * x; 1 + z) \end{array}$$

The first variable declaration creates an environment $x \mapsto 3$ with a single binding. The next two variable declarations 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 code for this section is given in the Declare zip file.

3.4 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 zip file. The type *Value* is defined to support multiple different kinds of values:

$$\begin{array}{l} \textbf{data}\ \mathit{Value} = \mathit{IntV}\ \mathit{Int} \\ \qquad\qquad\quad \mid\ \mathit{BoolV}\ \mathit{Bool} \\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \end{array}$$

Some example values are $\mathit{BoolV}\ \mathit{True}$ and $\mathit{IntV}\ 3$. We will define additional kinds of values, including functions and lists, later. The names IntV and BoolV in this type definition are the *tags* for data variants, while *Int* and *Bool* uses are *types* that specify 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*) *true-exp* **else** *false-exp*

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-exp* : *false-exp*. 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 ¬ 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 (Show, Eq)
data UnaryOp = Neg | Not
  deriving (Show, Eq)

data Exp = Literal  Value
         | Unary    UnaryOp Exp
         | Binary   BinaryOp Exp Exp
         | If       Exp Exp Exp
         | Variable String
         | Declare  String Exp Exp
```

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)]

  -- Evaluate an expression in an environment
evaluate :: Exp → Env → Value
evaluate (Literal v) env    = v
evaluate (Unary op a) env = unary op (evaluate a env)
evaluate (Binary op a b) env =
```

```
    binary op (evaluate a env) (evaluate b env)
evaluate (Variable x) env = fromJust (lookup x env)
evaluate (Declare x exp body) env = evaluate body newEnv
  where newEnv = (x, evaluate exp env) : env
```

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

```
evaluate (If a b c) env =
  let BoolV test = evaluate a env in
    if test then evaluate b env
      else evaluate c env
```

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

```
unary Not (BoolV b) = BoolV (¬ b)
unary Neg (IntV i) = IntV (−i)
binary Add (IntV a) (IntV b) = IntV (a + b)
binary Sub (IntV a) (IntV b) = IntV (a − b)
binary Mul (IntV a) (IntV b) = IntV (a ∗ b)
binary Div (IntV a) (IntV b) = IntV (a `div` b)
binary And (BoolV a) (BoolV b) = BoolV (a ∧ b)
binary Or  (BoolV a) (BoolV b) = BoolV (a ∨ b)
binary LT  (IntV a) (IntV b) = BoolV (a < b)
binary LE  (IntV a) (IntV b) = BoolV (a ⩽ b)
binary GE  (IntV a) (IntV b) = BoolV (a ⩾ b)
binary GT  (IntV a) (IntV b) = BoolV (a > b)
binary EQ  a        b        = BoolV (a ≡ b)
```

TODO: talk about strictness!

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

```
4
 # IntV 4
 − 4 − 6
 # IntV (−10)
if (3 ≡ 6) − 2 else − 7
 # IntV (−7)
3 ∗ (8 + 5)
 # IntV 39
3 + 8 ∗ 2
 # IntV 19
```

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

$\textbf{if}\ (3 > 3 * (8 + 5))\ 1\ \textbf{else}\ 0$
$\#\ \mathit{IntV}\ 0$
$2 + (\textbf{if}\ (3 \leqslant 0)\ 9\ \textbf{else} - 5)$
$\#\ \mathit{IntV}\ (-3)$

3.4.1 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 $3 + \mathit{True}$, 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 $3 + \mathit{True}$ results in a call to $\mathit{binary}\ \mathit{Add}\ (\mathit{IntV}\ 3)\ (\mathit{BoolV}\ \mathit{True})$, which is not one of the patterns handled by the *binary* function. As a result, Haskell generates a *Non-exhaustive pattern* error:

$\mathit{Main} > \mathit{evaluate}\ [\,]\ (\mathit{parseExp}\ \texttt{"3 + true"})$
$***\mathit{Exception} : \mathit{Non} - \mathit{exhaustive\ patterns}\ \textbf{in}\ \mathit{function\ binary}$

Here are some examples of expression that generate type errors:

$\textbf{if}\ (\mathit{true})\ 5\ \textbf{else}\ 8$
$\#\ \mathit{IntV}\ 5$
$3 + \mathit{true}$
$\#\ \mathit{Error} : \mathit{Value.hs} : (19, 1) - (29, 47) : \mathit{Non} - \mathit{exhaustive\ patterns}\ \textbf{in}\ \mathit{function\ binary}$
$3 \vee \mathit{true}$
$\#\ \mathit{Error} : \mathit{Value.hs} : (19, 1) - (29, 47) : \mathit{Non} - \mathit{exhaustive\ patterns}\ \textbf{in}\ \mathit{function\ binary}$
$-\ \mathit{true}$
$\#\ \mathit{Error} : \mathit{Value.hs} : (16, 1) - (17, 31) : \mathit{Non} - \mathit{exhaustive\ patterns}\ \textbf{in}\ \mathit{function\ unary}$

Running these tests produce error messages, but the errors are not very descriptive of the problem that actually took place.

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

Assignment 1: Basic Interpreter

Extend the *parser* and *interpreter* of Section on Evaluating Using Environments to allow multiple bindings in a variable binding expression. For example,

$var\ x = 3, y = 9;$
$x * y$

The abstract syntax of the *Exp* language with multiple bindings can be expressed by changing the *Declare* rule to support a list of pairs of strings and expressions:

data *Exp* = ...
  | *Declare* [(*String*, *Exp*)] *Exp*

If a *Declare* expression has duplicate identifiers, your program must signal an error. It is legal for a nested *Declare* to reuse the same name. Two examples:

$var\ x = 3, x = x + 2; x * 2$ -- illegal
$var\ x = 3; var\ x = x + 2; x * 2$ -- legal

The meaning of a *var* declaration is that all of the expressions associated with variables are evaluated first, in the environment before the *var* is entered. Then all the variables are bound to the values that result from those evaluations. Then these bindings are added to the outer environment, creating a new environment that is used to evaluate the body of the *var* declaration. This means that the *scope* of all variables is the body of the *var* in which they are defined.

Note that a multiple declare is not the same as multiple nested declares. For examle,

$var\ a = 3; var\ b = 8; var\ a = b, b = a; a + b$ -- evaluates to 11
$var\ a = 3; var\ b = 8; var\ a = b; var\ b = a; a + b$ -- evaluates to 16

You must write and include test cases that amply exercise all of the code you've written.

You can assume that the inputs are valid programs and that your program may raise arbitrary errors when given invalid input.

Here is an example test case:

$var\ a = 2, b = 7; (var\ m = 5 * a, n = b - 1; a * n + b\ /\ m) + a$

The previous version of this example contained an unbound use of the variable m:

$var\ a = 2, b = 7; (var\ m = 5 * a, n = m - 1; a * n + b\ /\ m) + a$

The code that you must modify is given in the Declare.zip file.

Chapter 4

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 $| x |$ 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.

So while you might already have a good grasp of what functions do, there's a good chance that the more abstract question of what functions *are* remains unanswered. In order to help you answer this question, first we will implement the ability to evaluate a restricted subset of functions called Top-Level Functions in our developing language. Then, to help you actually answer the question of what a function is, we will explore the idea of functions as first-class values in a programming language.

4.1 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 zip 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 is written in JavaScript. It 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:

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

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

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

The exact syntax varies from language to language. Some languages begin with a keyword *function* or *def*. Other languages require brackets { ... } 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:

name(*expression*, ..., *expression*)

Again, there are some variations on this theme. For example, in Haskell the parentheses are optional.

A program is a sequence of function definitions, followed by a main expression. Each function definition has a list of parameter names and a body expression. The following data type definitions provide a means to represent such programs:

```
type FunEnv = [(String, Function)]
data Function = Function [String] Exp
```

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:

```
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:

```
function power (n, m) {
if (m ≡ 0)
  1
else
  n * power (n, m − 1)
}
power (3, 4)
 =>> IntV 81
```

4.1.1 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 → String
execute (Program funEnv main) = show (evaluate main [ ] funEnv)
```

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

```
evaluate :: Exp → Env → FunEnv → Value
```

All the cases of evaluation are the same as before, except for the new case for calling a function:

```
evaluate (Call fun args) env funEnv = evaluate body newEnv funEnv
  where Function xs body = fromJust (lookup fun funEnv)
    newEnv = zip xs [evaluate a env funEnv | 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 [*evaluate a env funEnv* | *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          -- variable name
   | Call     String [Exp]    -- function name
```

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 (
    var pow = pow(pow - 1);
    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 (
```

```
    var b = pow(a - 1);
    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. 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*.

4.1.2 Summary

Here is the full code for the evaluator supporting top-level functions definitions, taken from the Top Level Functions zip file.

```
data Exp = Literal Value
         | Unary   UnaryOp Exp
         | Binary  BinaryOp Exp Exp
         | If      Exp Exp Exp
         | Variable String
         | Declare String Exp Exp
         | Call    String [Exp]
  deriving Show

evaluate :: Exp -> Env -> FunEnv -> Value
evaluate (Literal v) env funEnv = v
evaluate (Unary op a) env funEnv =
  unary op (evaluate a env funEnv)
evaluate (Binary op a b) env funEnv =
  binary op (evaluate a env funEnv) (evaluate b env funEnv)
```

```
evaluate (If a b c) env funEnv =
  let BoolV test = evaluate a env funEnv in
    if test then evaluate b env funEnv
      else evaluate c env funEnv
evaluate (Variable x) env funEnv = fromJust (lookup x env)
evaluate (Declare x exp body) env funEnv =
  evaluate body newEnv funEnv
    where newEnv = (x, evaluate exp env funEnv) : env
evaluate (Call fun args) env funEnv = evaluate body newEnv funEnv
  where Function xs body = fromJust (lookup fun funEnv)
    newEnv = zip xs [evaluate a env funEnv | a ← args]
```

4.2 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.

first-class values A *first-class value* is a value that can be used like any other value. A value is first class if it can be passed to functions, returned from functions, and stored in a variable binding.

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 $f = \lambda ldots$.

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.

4.2.1 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.

lambda or function expression A *lambda expression* is an expression that creates a function. The general form of a function expression is:

λ *variable* . *body*

This represents a function with parameter *variable* that computes a result defined by the *body* expression. The *variable* may of course be used within the *body*. In other words, *variable* may be free in *body*, but *variable* is bound (not free) in λ *variable* . *body*. A function expression is sometimes called an *abstraction* or a *function abstraction* (we).

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 variable declarations 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.

4.2.1.1 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 zip file contains the code for the simples examples in this section, and 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 \. 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*:

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

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."

4.2.2 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 +. 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]

[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

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.

```
   -- version A
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.

```
   -- version B
(λx → 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):

(λ *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.

4.3 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.

4.3.1 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

to mean $x * y$ in arithmetic expressions. Normally in computer science we 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*).

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:

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

4.3.2 Mapping and List Comprehensions

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 replaced by invocations of *map*:

$[evaluate\ a\ env \mid a \leftarrow args] \quad \equiv \quad map\ (\lambda a \rightarrow evaluate\ a\ env)\ args$

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

4.3.3 Representing Environments as Functions

In Chapter 1, an environment was represented 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

type *EnvL* = [(*String*, *Value*)]
envL1 = [(`"x"`, *IntV* 3), (`"y"`, *IntV* 4), (`"size"`, *IntV* 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:

type *EnvF* = *String* → *Maybe Value*

envF1 `"x"` = *Just* (*IntV* 3)
envF1 `"y"` = *Just* (*IntV* 4)
envF1 `"size"` = *Just* (*IntV* 10)
envF1 _ = *Nothing*

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

x1 = *lookup* `"x"` *envL1*
x2 = *envF1* `"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 (var, val) and then prepends it to the front of the list:

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

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

```
envL2 = bindL "z" (IntV 5) envL1
  -- [("z", IntV 5), ("x", IntV 3), ("y", IntV 4), ("size", IntV 10)]
envL3 = bindL "x" (IntV 9) envL1
  -- [("x", IntV 9), ("x", IntV 3), ("y", IntV 4), ("size", IntV 10)]
```

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 -> Value -> (String -> Maybe Int) -> (String ->
Maybe Int)
```

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

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

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 $\lambda testVar \to \lambda ldots$. That means the return value is a function. The argument of this function is named *testVar* and

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

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:

```
  -- version A
envF2 = bindF "z" (IntV 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

```
  -- version B
envF2 = λtestVar → if testVar ≡ "z"
  then Just (IntV 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

```
  -- version C
envF2 "z" = Just (IntV 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:

```
  -- Evaluate an expression in a (functional) environment
evaluateF :: Exp → EnvF → Value
evaluateF (Literal v) env = v
evaluateF (Unary op a) env =
  unary op (evaluateF a env)
evaluateF (Binary op a b) env =
  binary op (evaluateF a env) (evaluateF b env)
evaluateF (Variable x) env =
  fromJust (env x)   -- changed
evaluateF (Declare x exp body) env =
  evaluateF body newEnv
    where newEnv = bindF x (evaluateF exp env) 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 is quite blurry.

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

4.3.4 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:

$$inc = add\ 1 \qquad \text{-- . b + 1}$$
$$dec = add\ (-1) \quad \text{-- . b + (-1)}$$

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:

$$\begin{aligned}&\mathit{bindF}\ \mathit{var}\ \mathit{val}\ \mathit{env}\ \mathit{testVar} = \textbf{if}\ \mathit{testVar} \equiv \mathit{var}\\&\quad\textbf{then}\ \mathit{Just}\ \mathit{val}\\&\quad\textbf{else}\ \mathit{env}\ \mathit{testVar}\end{aligned}$$

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.

It is also possible to convert between functions that take multiple arguments and chains of functions taking one argument. The standard way to represent multiple arguments in Haskell is with a *tuple*, or a collection of values. One simple case of a tuple is a *pair*. For example, here are two pairs:

$$(3, 5)$$
$$(\texttt{"test"}, 99)$$

A Haskell function that takes a tuple as an argument resembles functions in most other programming languages:

$$\mathit{max}\ (a, b) = \textbf{if}\ a > b\ \textbf{then}\ a\ \textbf{else}\ b$$

But in Haskell this is really just a function taking one argument, which is pattern matched to be a tuple, in this case a pair that binds a to the first component and b to the second component. This function performs that same computation as the standard *max* function, which is a function that takes a single argument and returns a function that takes the second argument,

$$\mathit{max}\ a\ b = \textbf{if}\ a > b\ \textbf{then}\ a\ \textbf{else}\ b$$

which is equivalent to

$$\mathit{max} = \lambda a \rightarrow \lambda b \rightarrow \textbf{if}\ a > b\ \textbf{then}\ a\ \textbf{else}\ b$$

Note that multiple functions can be represented in a lambda expression as well:

$$\mathit{max} = \lambda a\ b \rightarrow \textbf{if}\ a > b\ \textbf{then}\ a\ \textbf{else}\ b \quad \text{-- equivalent to above definition}$$

currying The process of converting from a function taking a tuple to a chain of functions that take one argument at a time is called *currying*.

A curry function is a higher-order function that performs this operation:

$$curry2\ f = \lambda a\ b \to f\ (a, b)$$

The function *curry2* converts a function f that takes a pair to a new function that takes one argument and returns a function that takes the second argument. A function that takes a 3-tuple and converts it to curryied form can also be defined:

$$curry3\ f = \lambda a\ b\ c \to f\ (a, b, c)$$

In Haskell it is not possible to take a arbitrary length tuple and curry it. It is possible to write *uncurry* functions that perform that opposite transformation, from a function with multiple arguments to a function with a single tuple argument:

$$uncurry2\ f = \lambda(a, b) \to f\ a\ b$$
$$uncurry3\ f = \lambda(a, b, c) \to f\ a\ b\ c$$

uncurrying The process of converting from a chain of functions that take one argument at a time into a function that takes a single tuple argument is called *uncurrying*.

4.3.5 Church Encodings

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

4.3.5.1 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:

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.

Church Boolean A *Church Boolean* is a encoding of a boolean value as a function of two arguments, which returns the first argument for true, and the second argument for false.

4.3.5.2 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 function, not a simple value like 0 or 1. The Church numeral 0 applies f zero times to x. Similarly, 1 applies f once to x.

$$
\begin{aligned}
&\mathit{zero} = \lambda f \to \lambda x \to x \\
&\mathit{one} = \lambda f \to \lambda x \to f\ x \\
&\mathit{two} = \lambda f \to \lambda x \to f\ (f\ x) \\
&\mathit{three} = \lambda f \to \lambda x \to f\ (f\ (f\ x)) \\
\\
&\mathit{succ} = \lambda n \to (\lambda f \to \lambda x \to f\ (n\ f\ x))
\end{aligned}
$$

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.

$\mathit{three}\ (+1)\ 0 ==> 3$

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':)` $[\,] ===>$ `"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.

$$
\begin{aligned}
&\mathit{church} :: \mathit{Integer} \to \mathit{ChurchN} \\
&\mathit{church}\ 0 = \lambda f \to \lambda x \to x \\
&\mathit{church}\ n = \lambda f \to \lambda x \to f\ (\mathit{church}\ (n-1)\ f\ x)
\end{aligned}
$$

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

$\mathit{unchurch} :: \mathit{ChurchN} \to \mathit{Integer}$
$\mathit{unchurch}\ n = n\ (+1)\ 0$
-- 5 == (unchurch (church 5)) – this evaluates to True

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

$$
\begin{aligned}
&\mathit{plus} :: \mathit{ChurchN} \to \mathit{ChurchN} \to \mathit{ChurchN} \\
&\mathit{plus}\ n\ m = \lambda f \to \lambda x \to n\ f\ (m\ f\ x) \\
&\mathit{mul} :: \mathit{ChurchN} \to \mathit{ChurchN} \to \mathit{ChurchN} \\
&\mathit{mul}\ n\ m = \lambda f \to n\ (m\ f)
\end{aligned}
$$

We can use these functions to produce simple arithmetic equations.

$$
\begin{aligned}
&x = \mathit{church}\ 10 \\
&y = \mathit{church}\ 5 \\
&z = \mathit{church}\ 2 \\
&a = \mathit{plus}\ x\ (\mathit{mul}\ y\ z) \quad \text{-- is equivalent to church 20}
\end{aligned}
$$

4.3.6 Relationship between Declarations and Functions

TODO: prove that $\mathit{var}\ x = e;\ b$ is equivalent to $(\lambda x.b)e$

The variable declaration expression in our language is not necessary, because a *var* can be simulated using a function. In particular, any expression $\mathit{var}\ x = e;\ b$ is equivalent to $(\mathit{function}\ (x)\ \{\ b\ \})e$.

The expression $\mathit{var}\ x = e;\ b$ binds value of e to the variable x for use in the body, b. The creation of bindings in a *var* statement is equivalent to the bindings created from arguments provided to a function. So, if a function was defined as: $\mathit{foo} = \mathit{function}\ (x)\ \{\ b\ \}$ Calling $\mathit{foo}\ (e)$ is equivalent to $\mathit{var}\ x = e;\ b$. So, a *var* statement is another rewording of a lambda function that takes a certain argument binding before interpreting the body.

4.3.7 Others

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

4.4 Evaluating First-Class Functions using Environments

It's 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, variable declaration 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 *Lexical scope* means that a variable refers to the closest enclosing definition of that variable.

4.4.1 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 solution mentioned here is in the Incorrect Functions zip 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 = IntV Int
  | BoolV Bool
  | Function String Exp   -- new
  deriving (Eq, Show)
```

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 $f\ (x)\ \{x * x\}$ $f\ (10)$	*var* $f =$ *function* $(x)\ \{x * x\};$ $f\ (10)$

The explicit abstract syntax for the call in example (A) is:

Call `"f"` (*Literal* (*IntV* 10))

The explicit abstract syntax for the call in example (B) is:

Call (*Variable* `"f"`) (*Literal* (*IntV* 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 a *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.

There are many examples where the function to be called is not a variable. For example, one could call one of two different functions depending on a condition. The following example calls either f or g depending on whether $a > b$.

(**if** $a > b$ **then** f **else** g) (4)

This is similar to calling a function on a conditional argument:

f (**if** $a > b$ **then** 4 **else** 7)

Which is equivalent to the perhaps more familiar form:

if $a > b$ **then**
  f (4)
else
  f (7)

Another example of not using a variable to name a function is the use of *function literals*. The following example applies a function literal that squares a number to the argument 7.

(*function* $(x)\ \{x * x\}$) (7)

Lets now define the (incorrect) interpreter. The first few cases for evaluation are exactly the same as before. In particular, evaluating a literal value is the same, except that now the literal value might be a function.

```
evaluate :: Exp → Env → Value
evaluate (Literal v) env = 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:

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

To evaluate a function call *Call fun arg*,

1. First evaluate the function *fun* of the call: *evaluate fun env*
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 (*evaluate arg env*) and then extend the environment *env* with a binding between the function parameter *x* and the argument value:

   newEnv = bindF x (evaluate arg env) 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 *evaluate fun env* 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.

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.

4.4.1.1 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 Haskell 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:

$var\ add = function\ (a)\ \{function\ (b)\ \{b + a\}\};$
$add\ (3)\ (2)$

Here is how evaluation of this sample program proceeds:

1. Evaluate $var\ add = function\ (a)\ \{function\ (b)\ \{b + a\}\};\ add\ 3\ 2$
2. Bind $add \mapsto function\ (a)\ \{function\ (b)\ \{b + a\}\}$
3. Call $(add\ 3)\ 2$
    a. Call $add\ 3$
    b. Evaluate the variable add, which looks it up in the environment to get $function\ (a)\ \{function\ (b)\ \{b + a\}\}$
    c. Bind $a \mapsto 3$
    d. Return $function\ (b)\ \{b + a\}$ as result of $add\ 3$
4. Call $function\ (b)\ \{b + a\}$ 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 $function\ (b)\ \{b + a\}$. 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 3.c through 3.d of the explanation above. But this binding is lost when returning the literal $function\ (b)\ \{b + a\}$. 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.

4.4.1.2 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.

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

let $k = 9$ **in**
  double k

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 *double* $\mapsto \lambda n \rightarrow k * n$
3. Bind $k \mapsto 9$
4. Call *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*.

dynamic binding dynamic binding occurs when a symbol's value is found by scanning the dynamic calls for the most recent binding of the symbol.

4.4.2 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.*

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 zip 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 variable declarations, 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 = IntV Int
  | BoolV Bool
  | ClosureV 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.

```
evaluate (Function x body) env = ClosureV x body env   -- new
```

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, *evaluate* (*Call fun arg*) *env*. This pattern says that a call expression has two components: a function *fun* and an argument *arg*. Here is the code for this case:

```
evaluate (Call fun arg) env = evaluate body newEnv   -- changed
  where ClosureV x body closeEnv = evaluate fun env
    newEnv = (x, evaluate arg env) : closeEnv
```

The code starts by evaluating both the function part *fun* to produce a value. The **where** clause *ClosureV x body newEnv* = *evaluate fun env* 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 $(x, evaluate\ arg\ env)$ 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, *evaluate body newEnv*.

TODO: give an example of how this runs?

4.4.2.1 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.

4.5 Environment/Closure Tree

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 Tree is text file that concisely shows the environments and closures created during execution of an expression. This is a new representation that follows the structure of environments more explicitly.

4.5.1 Example 1

```
var k = 2;
var double = function (n) { k * n };
var k = 9;
double(k)
```

Here is the Environment/Closure Tree that results from executing this code.

$*\ k = 2$
$|$
$+--+\ CLOSURE\ C1 : n, k * n$
$|\ |$
$|\ |\ INVOKE\ I1 :$
$|\ +---*\ n = 9$
$|\quad RESULT : 18$
$|$
$*\ double = C1$

$|$
$* k = 9$
$|$
$|\ CALL\ I1 : C1\ (9)$
$|\ RESULT : 18$
$RESULT : 18$

The idea here is that the bindings created during execution have their parent environment implicitly represented by the text: A binding $x = v$ appears below the environment it extends. If it is a local binding or function argument binding, it is indented.

Closures have environments, which capture the environment in which they were created. This is represented by a *CLOSURE name*. They key point is that calls to this function create invocations below the closure. This means that the function call/invoke's environment is the same as the closure's environment.

Here is a step-by-step discussion of the first example.

1. A binding is created for $k = 2$.
2. The expression in the *var double* is evaluated. This creates a closure named *C1*, with argument n and body $k * n$
3. A binding is created for $double = C1$ which refers to the closure *C1*.
4. A binding is created for $k = 9$.
5. A call is created to *double*, which is the closure *C1*. The argument is 9. The call is named *I1*.
6. The call creates an *INVOKE I1*: *under* the closure *C1*.
7. A binding for the argument $n = 9$ is created under the *INVOKE*
8. The closure body $k * n$ is evaluated, creating a $RESULT : 18$.
    - Note that $k = 2$ and $n = 9$ because lookup starts within the closure, not at the *CALL* site.
9. The result is copied to the *CALL I1*.

Here are some rules for defining Environment/Closure Trees (ECTs):

- Case *var* $x =$ *exp*; *body*
    1. Create an ECT for evaluation of *exp*.
    2. Create a binding $x = \lambda emph\ \{result\}$.
    3. Create an ECT for *body*
- Case *function* $(x)\ \{e\}$
    1. Make a *CLOSURE* with a unique name C, variable x, body e

2. The *result* is C, which refers to the new closure

- Case *fun* (*arg*)
  1. Create an ECT for evaluation of *fun*.
     - The *result* must be a *CLOSURE* named C with x, *body*.
  2. Create an ECT for evaluation of *arg*.
     - Let the *result* be called *argval*
  3. Create a *CALL I* : C (*argval*)
  4. Make an *INVOKE I*: under the *CLOSURE* with a vertical bar to the left
  5. Make a binding $x = argval$ under the *INVOKE*
  6. Create an ECT for *body* under the *INVOKE*
  7. Under the *INVOKE I* and *CALL I* create a *RESULT*: *result*
- Case *exp*
  1. Evaluate *exp*, setting *result* to the value
  2. Find value of variable x by moving *up* and to the *left* on the ECT searching for $x = val$

The vertical bars below a closure are there to highlight the fact that these *INVOKE*s are created later, when the closure is *called*, not when it is created.

4.5.2 Example 2

```
var add = function(a) {
             function(b) {
                b + a
          } };
add(3)(2)
```

Here is the Environment/Closure Tree (ECT)

| *CLOSURE C1* : a, *function* − 1
|
| *INVOKE I1* :
+ − −∗ $a = 3$
| | *CLOSURE C2* : b, *function* − 2
| |
| | *INVOKE I2* :
| + − −∗ $b = 2$
| | *RESULT* : 5 -- b + a
| |
| *RESULT* : *C2*

|
$*\ add = C1$
|
$|\ CALL\ I1 : C1\ (3)$
$|\ RESULT : C2$
$|\ CALL\ I2 : C2\ (2)$
$|\ RESULT : 5$
$RESULT : 5$

4.5.3 Example 3

```
var m = 2;
var proc = function(n) {
              m + n
           };
var part = function(g, n) {
              function(m) {
                n * g(m)
           }};
var inc = part(proc, 3);
inc(7)
```

Here is the Environment/Closure Tree (ECT)

$*\ m = 2$
|
$|\ CLOSURE\ C1 : n, function - 1$
|
$|\ INVOKE\ I3$
$+ - -* n = 7$
$|\ RESULT : 9$
|
$*\ proc = C1$
|
$|\ CLOSURE\ C2 : g, n, function - 2$
|
$|\ INVOKE\ I1 :$
$+ - -* g = C1$
$|\ *\ n = 3$
$|\ |\ CLOSURE\ C3 : m, function - 3$ -- n * g(m)
$|\ |$
$|\ |\ INVOKE\ I2$
$|\ + - -* m = 7$
$|\ |\ |\ CALL\ I3 : C1\ (7)$
$|\ |\ |\ RESULT : 9$

```
| | RESULT : 27   -- n * g(m)
| |
| RESULT : C3
|
* part = C2
|
| CALL I1 : C2 (C1, 3)   -- part(proc, 3)
| RESULT : C3
* inc = C3
|
| CALL I2 : C3 (7)       -- inc(7)
| RESULT : 27
| RESULT : 27
```

4.6 Call-by-value and Call-by-name

In languages with declarations, functions or first-class functions there are a few different design options when it comes to evaluation.

4.6.1 Call-by-value

So far all our interpreters have explored one particular design option, called call-by-value.

call-by-value In *call-by-value* interpretation, expressions, such as parameters of functions arguments or variable initializers, are always evaluated before being added to the environment.

Here is the code of evaluation for declarations and function application:

```
evaluate (Declare x exp body) env =
  case evaluate exp env of
    VException → VException
    v → evaluate body ((x, v) : env)

evaluate (Call fun arg) env =
  case evaluate arg env of
    VException → VException
    v → case evaluate fun env of
      ClosureV name body denv → evaluate body ((name, v) : denv)
```

In the case of *Declare* expressions, expression *exp* is evaluated before evaluating the body of the declaration. In the case of *Call* expressions the argument *arg* is

evaluated before evaluating the function and function *body*. Note also that we need to be a bit more careful in the presence of exception values to ensure that exceptional values propagate.

Call-by-value is the most common design option for function calls in programming languages. All major programming languages (including C, Java or C#) use call-by-value. Note that some programming languages also support other function call mechanisms. For example, C also supports call-by-reference.

Drawbacks of call-by-value: Call-by-value can waste resources when evaluating expressions in some cases. For example, consider the following expressions:

$$var\ x = longcomputation; 3$$

$$(\lambda x \to 3)\ longcomputation$$

In the two expressions the idea is that *longcomputation* stands for an expression that takes a long time to compute. In both cases the returned value will be 3 and the final result does not depend on the value that is computed by *longcomputation*. So, in this case, it seems wasteful to spend time computing *longcomputation*.

4.6.2 Call-by-Name

A different design option is to use call-by-name.

call-by-name In call-by-name an expression is not evaluated until it is needed.

So, in the programs:

$$var\ x = longcomputation; 3$$

$$(\lambda x \to 3)\ longcomputation$$

the expression *longcomputation* is never evaluated and therefore evaluating such expressions is very fast.

Haskell is one of the few languages where (a variant of) call-by-name is the default mechanism for evaluating function applications.

Drawbacks of call-by-name: While for the program above there seems to benefit from using call-by-name, there are also programs where call-by-name is worse than call-by-value. For example:

$$var\ x = longcomputation; x + x$$

$$(\lambda x \to x + x)\ longcomputation$$

In both programs a call-by-name language will evaluate *longcomputation* twice. Since evaluation is delayed to the use-point of the expression, the expression bound to x is evaluated twice.

In languages like Haskell this drawback is avoided by using an optimization of call-by-name called call-by-need.

call-by-need In call-by-need, when the use of a variable forces the evaluation of an expression, the result of that evaluation is cached and next time the variable is needed the cached value is simply returned.

This means that in call-by-need an expression will be evaluated at most once.

4.6.3 Call-by-Value, Call-by-Name and Exceptions

In a language with exceptions, the same expression may evaluate to different results. under call-by-name or call-by-value. For example:

$var\ x = 3\ /\ 0; 7$

evaluates to:

1) an exception in a call-by-value language;
2) 7 in a call-by-name language.

The reason is that exceptions propagate. Since in call-by-value all expressions arguments (or initializers) are evaluated, the program will raise an exception which is then propagated throughout evaluation. In contrast in call-by-name the expression bound to x is not needed to compute the result. Therefore, because that expression is not evaluated, no exception is propagated.

4.7 Summary of First-Class Functions

Here is the full code for first-class functions with non-recursive definitions. The grammar changes are as follows, taken from the FirstClassFunctionsParse.y file:

$Exp : function\ \texttt{'('}\ id\ \texttt{')'}\ \texttt{'\{'}\ Exp\ \texttt{'\}'}\ \{Function\ \$\ 3\ \$\ 6\}$
$Primary : Primary\ \texttt{'('}\ Exp\ \texttt{')'}\ \{Call\ \$\ 1\ \$\ 3\}$

Here is the definition of the abstract syntax and the evaluator, taken from the First Class Functions zip file:

data $Exp = Literal\ Value$
$\quad \mid Unary\quad\quad\quad UnaryOp\ Exp$

```
  | Binary      BinaryOp Exp Exp
  | If          Exp Exp Exp
  | Variable    String
  | Declare     String Exp Exp
  | Function    String Exp    -- new
  | Call        Exp Exp       -- changed
  deriving (Eq, Show)
type Env = [(String, Value)]
evaluate :: Exp → Env → Value
evaluate (Literal v) env = v
evaluate (Unary op a) env =
  unary op (evaluate a env)
evaluate (Binary op a b) env =
  binary op (evaluate a env) (evaluate b env)
evaluate (If a b c) env =
  let BoolV test = evaluate a env in
    if test then evaluate b env
      else evaluate c env
evaluate (Variable x) env = fromJust (lookup x env)
evaluate (Declare x exp body) env = evaluate body newEnv
  where newEnv = (x, evaluate exp env) : env
evaluate (Function x body) env = ClosureV x body env   -- new
evaluate (Call fun arg) env = evaluate body newEnv   -- changed
  where ClosureV x body closeEnv = evaluate fun env
    newEnv = (x, evaluate arg env) : closeEnv
```

Test cases can be found in and the FirstClassFunctionsTest.js file.

Assignment 2: First-Class Functions

Extend the *parser* and *interpreter* of Section on First-Class Functions to allow passing multiple parameters, by adding a *tuple* data type and allowing *patterns* to be used in function and variable definitions. For example:

Here are two example test cases:

```
var f = function (a, b) {a + 2 * b};
f (3, 4) - f (5, 2)

var z = 12;
var tup = (3, z * 2);
var (x, y) = tup;
z / x + y
```

You must change the parser to allow creation of tuples, patterns in function and variable definitions, and functions called with a tuple. Here are required changes to your abstract syntax:

```
data Exp = ...
   | Declare Pattern Exp Exp   -- declarations bind patterns
   | Function Pattern Exp      -- functions have patterns
   | Tuple [Exp]               -- tuple expression

data Pattern = VarP String     -- variable patterns
   | TupleP [Pattern]          -- tuple patterns

data Value = ...
   | ClosureV Pattern Exp Env  -- functions have patterns
   | TupleV [Value]   -- tuple value
  deriving (Eq, Show)
```

This will cause you to make significant changes to the *evaluate* function, because of the change to the types. You'll need to define a function to match a *Pattern* against a *Value* to get a environment *Env*.

$match :: Pattern \rightarrow Value - Env$

Here are some notes on special interpretation of singleton tuples/patterns, as in (x).

In the grammar ".y" file:

1) In the *Primary* case for '(' `Exp` ')' you must change this to '(' `ExpList` ')' to allow for lists of expressions which will create *Tuple*s in the program. Define *ExpList* as a new rule for lists of expressions. Note that the type of *ExpList* is $[Exp]$, which doesn't match *Exp* as required by *Primary*. You'll have to apply a function to convert it to the right type.

   But special handling is needed for singleton tuples, like (exp). Based on the change above, this will create $Tuple\ [exp]$, which is not wanted. That is a tuple with a single expression. It must be interpreted as just the expression, *exp*.

   The easiest way to do this is to put a condition that tests if the *ExpList* has *length* 1 and return its single *Exp*, otherwise return it as a tuple. The condition goes into the $\{\lambda ldots\}$ code for that *Primary* rule.

2) A similar problem happens with *function*'(' `PatList` ')'. You can either fix that in the grammar or (more easily) in your new *match* function. You need a add a special case for $TupleP\ [VarP\ x]$, where there is only one variable in a tuple pattern. In this case (x) must work that same as x (without parentheses).

3) As a hint, change the *Declare* rule to be *var Pattern*‘ = ’ `Exp` ‘`;`’ *Exp* where *Pattern* is your new rule that parses patterns (either a *VarP* or a *TupleP*).

4) I’ll leave it as an *optional* challenge to figure out to handle empty tuples ().

You must write and include test cases that amply exercise all of the code you’ve written. You can assume that the inputs are valid programs and that your program may raise arbitrary errors when given invalid input (except as mentioned above). Here are some examples that must signal errors:

$var\ (a, b) = 3; a + b$
$var\ (a, b) = (3, 4, 5)$

The first case is an error because 3 is not a tuple. The second case is an error because the lengths are not the same. This must be checked in the *match* function discussed above.

The files you need are in the First Class Functions zip file.

Chapter 5

Recursive Definitions

One consequence of using a simple *var* 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.

recursive function A recursive function is a function that calls itself within its own definition.

For example, consider this definition of the factorial function:

> **let** $fact = \lambda n \to$ **if** $n \equiv 0$ **then** 1 **else** $n * fact\ (n - 1)$
> **in** $fact\ (10)$

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 *var* 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 *var* 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,

> **let** $x = x + 1$ **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:

let $x = y + 1$
  $y = 99$
in $x * y$

This example includes two bindings at the same time (which we do not currently support. 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.

5.1 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.

infinite structure An *infinite structure* is a structure that is conceptually infinite, but cannot be represented explicitly in its entirety.

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 $n * \lambda ldots$. 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.

5.1.1 Three Analyses of Recursion

In what follows we will explore three ways to understand recursion. The first explanation just allows us to define recursive *var* 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.

5.2 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:

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

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:

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

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.

5.2.1 Using Results of Functions as Arguments

Another interesting use of recursion and laziness is the ability to use the result of a calling a function as one of the arguments to the function call itself.

A type for trees:

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

```
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.

Computing minimum and maximum at the same time.

```
minMax (Leaf n) = (n, n)
minMax (Branch a b) = (min min1 min2, max max1 max2)
  where (min1, max1) = minMax a
    (min2, max2) = minMax b
```

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

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

```
repTree x (Leaf n) = Leaf x
repTree x (Branch a b) = Branch (repTree x a) (repTree x b)
```

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

```
repMin' (Leaf n, r) = (n, Leaf r)
repMin' (Branch a b, r) = (min min1 min2, Branch newTree1 newTree2)
  where (min1, newTree1) = repMin' (a, r)
    (min2, newTree2) = repMin' (b, r)
```

Finally to do the replacement with the minimum:

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

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.

5.2.2 Implementing Recursive Variable Declarations with Haskell

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

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

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:

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

The only change is the replace *env* with *newEnv* in the call to *evaluate* on *exp*. 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 *var* 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.

5.2.2.1 Recursive Definitions in Environment/Closure Trees

For the case of recursive bindings, a special case must be defined for a *var* binding that defines a function:

- Case **Recursive** *var f = function (x) { exp }; body*
  1. Create a binding $f = C$ where C is the name of a new closure.
  2. Create a *CLOSURE* $C : x, exp$ to define the closure
  3. Proceed with the ECT for the *body*

Note that the *CLOSURE* is within scope of the binding, not above it as before.

Here is an example an example that uses this approach.

```
var fact = function(n) {
  if (n == 0)
    1
  else
    n * fact(n - 1)
};
fact(4)
```

Below is an Environment/Closure Tree for evaluating this program. Note that the binding of *fact* has moved from after the closure to before the closure. This means that the binding is in scope for the body of the function.

$*\, fact = C1$
|
$+--+\ CLOSURE\ C1 : n, function - 1$
| |
| | $INVOKE\ I1$
| $+--*\ n = 4$
| | | $CALL\ I2 : C1\ (3)$
| | | $RESULT : 24$
| | $RESULT : 24$
| |
| | $INVOKE\ I2$
| $+--*\ n = 3$
| | | $CALL\ I3 : C1\ (2)$
| | | $RESULT : 6$
| | $RESULT : 6$
| |
| | $INVOKE\ I3$
| $+--*\ n = 2$
| | | $CALL\ I4 : C1\ (1)$
| | | $RESULT : 2$
| | $RESULT : 2$
| |
| | $INVOKE\ I4$
| $+--*\ n = 1$
| | | $CALL\ I5 : C1\ (0)$

| | | *RESULT* : 1
| | *RESULT* : 1
| |
| | *INVOKE I5*
| + − −∗ $n = 0$
| *RESULT* : 1
|
| *CALL I1* : *C1* (4)
| *RESULT* : 24
RESULT : 24

5.3 Understanding Recursion with Fixed Points

Another way to explain recursion is by using the mathematical concept of a fixed point.

fixed point A *fixed point* of a function f is 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".

5.3.1 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
$square(x) = x^2$	$0, 1$
$g_\phi(x) = 1 + 1/x$	1.6180339887...
$k_4(x) = 4$	4

function	*fixed point(s)*
$id(x) = x$	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.

generator A function that is passed as an argument to the fixed-point function, with the intent of creating an infinite value.

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?

[2]The function *fix* is often called Y. For further reading, see @ScottDataTypes, @GunterPL, @WhyY and @thomas2006end.

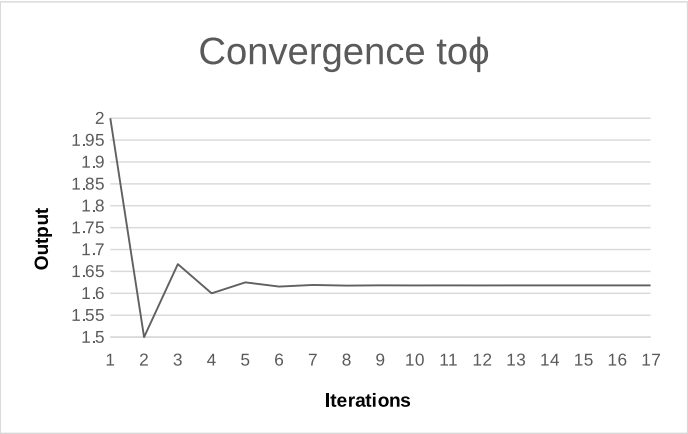

Figure 5.1: Plot of convergence of ϕ

5.3.2 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:

$g_\phi(x) = 1 + 1/x$

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 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.

Here is a plot of how the function converges:

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
$g_\phi(x) = 1 + 1/x$	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.

5.3.3 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
$[2, 2, 2, \ldots]$	$[2, 2, 2, \ldots]$	*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 than 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.

5.3.4 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...

5.3.5 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 *fix* 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.

5.3.6 A Non-Recursive Definition of *fix*

It is also possible to define *fix* non-recursively, by using *self application*.

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)
  -- returns id
```

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.

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

The composition ($\circ$) operator composes two functions:

$$f \circ g = \lambda x \to f\ (g\ x)$$

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.

diverge A function diverges if it doesn't return a value.

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.

5.4 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 == 0)
    return 1;
  else
    return n * f(f, n - 1);
}
```

To call *fact_s* to compute a factorial, it is necessary to pass *fact_s* 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. The actual *fact* function must still be defined:

$$fact = function\ (n)\ \{fact\_s\ (fact\_s, n)\}$$

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

Chapter 6

Computational Strategies

In previous sections the Exp language was extended with specific kinds of expressions and values, for example the *var* 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.

6.1 Error Checking

Errors are an important aspect of computation. They are typically a pervasive feature of a language, because 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 zip 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*

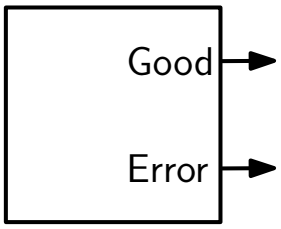

Figure 6.1: A computation that may produce an error.

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 abstract illustration of a *Checked* value, which represents a computation that may either be a good value or an error.

6.1.1 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 (Literal v) env = 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:

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

6.1.2 Error Checking in Multiple Sub-expressions

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

```
evaluate (Binary op a b) env =
  binary op (evaluate a env) (evaluate b env)
```

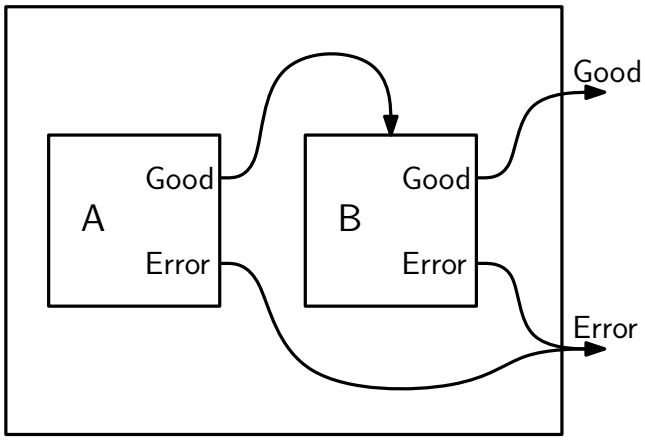

Figure 6.2: Composing computations that may produce errors.

The problem is that either *evaluate a env* or *evaluate b env* 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 *evaluate a env*, in *evaluate b env*, or in *binary*. This definition for *evaluate* of a binary operator handles the first two situations:

```
evaluate (Unary op a) env =
  case evaluate a env of
    Error msg → Error msg
    Good av → checked_unary op av
evaluate (Binary op a b) env =
  case evaluate a env of
    Error msg → Error msg
    Good av →
      case evaluate b env of
        Error msg → Error msg
        Good bv →
          checked_binary op av bv
```

Now it should be clear why programmers do not always check all error return codes: because it is tedious and requires lots of code! 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:

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 (BoolV b) = Good (BoolV (¬ b))
checked_unary Neg (IntV i) = Good (IntV (−i))
checked_unary op  v        =
  Error ("Unary " ++ show op
    ++ " called with invalid argument " ++ show v)
checked_binary :: BinaryOp → Value → Value → Checked Value
checked_binary Add (IntV a) (IntV b) = Good (IntV (a + b))
checked_binary Sub (IntV a) (IntV b) = Good (IntV (a − b))
checked_binary Mul (IntV a) (IntV b) = Good (IntV (a * b))
checked_binary Div _        (IntV 0) = Error "Divide by zero"
checked_binary Div (IntV a) (IntV b) = Good (IntV (a `div` b))
checked_binary And (BoolV a) (BoolV b) = Good (BoolV (a ∧ b))
checked_binary Or  (BoolV a) (BoolV b) = Good (BoolV (a ∨ b))
checked_binary LT  (IntV a) (IntV b) = Good (BoolV (a < b))
checked_binary LE  (IntV a) (IntV b) = Good (BoolV (a ⩽ b))
checked_binary GE  (IntV a) (IntV b) = Good (BoolV (a ⩾ b))
checked_binary GT  (IntV a) (IntV b) = Good (BoolV (a > b))
checked_binary EQ  a        b        = Good (BoolV (a ≡ b))
checked_binary op  a        b        =
  Error ("Binary " ++ show op ++
    " called with invalid arguments "
    ++ show a ++ ", " ++ show b)
```

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

6.1.3 Examples of Errors

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

```
x
# Error "Variable x undefined"
```

Or for divide by zero:

```
3 / 0
# Error "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.

6.1.3.1 Exercise 6.1: Complete Error Checking

Extend the evaluator with error checking for the remaining expression cases, including **if**, non-recursive *var*, 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.

Start with the files for First Class Functions and Error Checking and combine them and complete the error cases. The files you need are Error Checking zip file.

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

6.1.3.2 Exercise 6.2: Error Handling

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. The syntax is *try* {*Exp*} *catch* {*Exp*}, and the meaning is to evaluate the first *Exp* and return its value if it is *Good*, otherwise evaluate the second *Exp* and return its value (or an *Error*).

6.1.3.3 Exercise 6.3: Multiple Bindings and Arguments

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

6.2 Mutable State

A second common pervasive computational strategy, besides error handling, is the use of *mutable state*.

mutable state Mutable state 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.

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 5. On each iteration of the loop the variable x is multiplied by i. The result of x at 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 zip file.

6.2.1 Addresses

Imperative languages typically allow everything to be mutable by default: all variables are 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.

address An address identifies a mutable container that stores a single value, but whose contents can change over time. Addresses are sometimes called *locations*.

The storage identified by an address is sometimes called a *cell*. You can think of it as a *box* that contains a value. (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
@e	Accesses the contents stored at address e
$a = e$	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.

dereferencing A address is *dereferenced* when the contents of the cell associated with the address is looked up and returned.

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 = IntV Int
   | BoolV Bool
   | ClosureV String Exp Env
   | AddressV 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 a "pound 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.

6.2.1.1 Memory

The current value of all mutable cells used in a program is called *memory*.

memory 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:

```
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:

$$[\mathit{IntV}\ 120, \mathit{IntV}\ 6]$$

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

$$[120, 6]$$

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	[]
$x = \mathit{mutable}\ 1;$	[1]
$i = \mathit{mutable}\ 2;$	[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]

6.2.2 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.

6.2.2.1 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"?

6.2.2.2 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.

```
update :: Int → Value → Memory → Memory
update addr val mem =
  let (before, _ : after) = splitAt addr mem in
    before ++ [val] ++ after
```

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:

```
mul10 addr mem =
  let IntV n = access addr mem in
    update addr (IntV (10 * n)) mem
```

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

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

The result is

$$[IntV\ 3, IntV\ 40, IntV\ 5, IntV\ 6]$$

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

$$mul10 :: Int \rightarrow Memory \rightarrow Memory$$

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

6.2.3 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:

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

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

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

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:

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

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

$$\textbf{type}\ Stateful\ t = Memory \rightarrow (t, Memory)$$

This is a *generic* type for a memory-based computation which returns a value of type t.

stateful computation A stateful computation is represented functionally as a function that takes an initial state and returns a value and an updated state.

Just as in the case of errors, it is useful to give a visual form to the shape of a stateful computation:

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

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

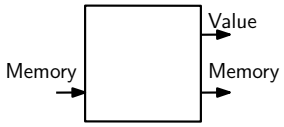

Figure 6.3: Shape of a stateful computation.

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.

6.2.4 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       -- mutable e
   | Access  Exp       -- @a
   | Assign  Exp Exp   -- a = e
```

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.

```
evaluate (Mutable e) env mem1 =
  let (ev, mem2) = evaluate e env mem1 in
    (AddressV (length mem2), mem2 ++ [ev])
```

The access expression @*a* 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.

```
evaluate (Access a) env mem1 =
  let (AddressV i, mem2) = evaluate a env mem1 in
    (access i mem2, mem2)
```


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

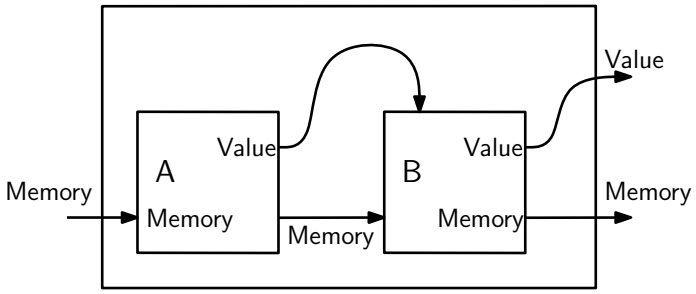

Figure 6.4: Composing stateful computations.

evaluate (*Assign a e*) *env mem1* =
  let (*AddressV i*, *mem2*) = *evaluate a env mem1* **in**
    let (*ev*, *mem3*) = *evaluate e env mem2* **in**
      (*ev*, *update i ev mem3*)

6.2.4.1 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:

evaluate (*Binary op a b*) *env mem1* =
  let (*av*, *mem2*) = *evaluate a env mem1* **in**
    let (*bv*, *mem3*) = *evaluate b env mem2* **in**
      (*binary op av bv*, *mem3*)

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

6.2.5 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
  | Declare        String Exp Exp
  | Function       String Exp
  | Call           Exp Exp
  | Seq            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 (Literal v) env mem = (v, mem)
evaluate (Unary op a) env mem1 =
  let (av, mem2) = evaluate a env mem1 in
    (unary op av, mem2)
evaluate (Binary op a b) env mem1 =
  let (av, mem2) = evaluate a env mem1 in
    let (bv, mem3) = evaluate b env mem2 in
      (binary op av bv, mem3)
evaluate (If a b c) env mem1 =
  let (BoolV test, mem2) = evaluate a env mem1 in
    evaluate (if test then b else c) env mem2
evaluate (Variable x) env mem = (fromJust (lookup x env), mem)
evaluate (Declare x e body) env mem1 =
  let (ev, mem2) = evaluate e env mem1
      newEnv = (x, ev) : env
  in
    evaluate body newEnv mem2
evaluate (Function x body) env mem = (ClosureV x body env, mem)
evaluate (Call f a) env mem1 =
```

```
    let (ClosureV x body closeEnv, mem2) = evaluate f env mem1
        (av, mem3) = evaluate a env mem2
        newEnv = (x, av) : closeEnv
    in
      evaluate body newEnv mem3
  evaluate (Seq a b) env mem1 =
    let (_, mem2) = evaluate a env mem1 in
      evaluate b env mem2
```

Here are the mutation-specific parts of the evaluator:

```
  evaluate (Mutable e) env mem1 =
    let (ev, mem2) = evaluate e env mem1 in
      (AddressV (length mem2), mem2 ++ [ev])
  evaluate (Access a) env mem1 =
    let (AddressV i, mem2) = evaluate a env mem1 in
      (access i mem2, mem2)
  evaluate (Assign a e) env mem1 =
    let (AddressV i, mem2) = evaluate a env mem1 in
      let (ev, mem3) = evaluate e env mem2 in
        (ev, update i ev mem3)
```

6.3 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 affect 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.

6.3.1 Abstracting Simple Computations

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

Checked	Stateful
evaluate (*Literal v*) *env* = *Good v*	*evaluate* (*Literal v*) *env m* = (*v*, *m*)

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
$return_C :: Value \rightarrow Checked\ Value$ $return_C\ v = Good\ v$	$return_S :: Value \rightarrow Stateful\ Value$ $return_S\ v = \lambda\ m.(v, m)$

Using these return functions, the original *evaluate* 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

evaluate (*Literal v*) *env* =
  $return_C$ *v*
	evaluate (*Literal v*) *env* =
	  $return_S$ *v*

6.3.2 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 evaluate	Stateful evaluate

evaluate (*Binary op a b*) *env* =
 case *evaluate a env* **of**
  Error msg → *Error msg*
  Good av →
  case *evaluate b env* **of**
	evaluate (*Binary op a b*) *env* = λ*m1*.
	 let (*av*, *m2*) = *evaluate a env m1* **in**
	 let (*bv*, *m3*) = *evaluate b env m2* **in**
	 (*binary op av bv*, *m3*)

Checked evaluate	Stateful evaluate
Error msg → *Error msg* *Good bv* → *checked_binary op av bv*	

Note that the memory *m1* argument has become a lambda! This is an instance of the *Rule of Function Arguments*. It was done to allow the first lines to be equivalent *evaluate* (*Binary op a b*) *env*.

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 *first* and *next*, the core of the pattern is:

Checked	Stateful
case *first* **of** *Error msg* → *Error msg* *Good v* → *next v*	$\lambda m1$.**let** $(v, m2)$ = *first m1* **in** *next v m2*

This *first* corresponds to *evaluate a env* or *evaluate b env* in both the original versions. The *next* represents the remainder of the computation. It is 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 *next* is the variable *v* that comes form the *Good* case. For the *Stateful* case, in addition to *v* the *next* also requires access to *m2*

These patterns can be made explicit as a special operator named *bind* $\gg=$ 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* is named *A* and the *next*, which is a function, is named *F*:

Checked	Stateful
$A \gg=_C F$ = **case** *A* **of** *Error msg* → *Error msg* *Good v* → *F v*	$A \gg=_S F$ = $\lambda m1$.**let** $(v, m2)$ = *A m1* **in** *F v m2*

These generic operators for *Checked* $\gg=_C$ and *Stateful* $\gg=_S$ computations abstract away the core pattern composing two *Checked* or *Stateful* computations.

The family of operators $\gg\!=$ are called *bind* operators, because they bind together computations. This is unrelated to bindings discussed in earler chapters.

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

Checked	Stateful
(*evaluate a env*) $\gg\!=_C$ (λav. (*evaluate b env*) $\gg\!=_C$ (λbv. *checked_binary op av bv*))	(*evaluate a env*) $\gg\!=_S$ (λav. (*evaluate b env*) $\gg\!=_S$ (λbv. λm. (*binary op av bv*, m)))

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

$return_S$ x $\equiv$ $\lambda m.\ (x, m)$

Using $return_S$ the result is:

Checked	Stateful
(*evaluate a env*) $\gg\!=_C$ (λav. (*evaluate b env*) $\gg\!=_C$ (λbv. *checked_binary op av bv*))	(*evaluate a env*) $\gg\!=_S$ (λav. (*evaluate b env*) $\gg\!=_S$ (λbv. $return_S$ (*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 a way to hide the complexity of error checking and mutable memory. This complexity has been hidden in two new operators, *return* and bind $\gg\!=$. The type of the bind operators is also interesting:

Checked: $\gg\!=_C$ ::*Checked Value* $\rightarrow$ (*Value* $\rightarrow$ *Checked Value*) $\rightarrow$ *Checked Value*
Stateful: $\gg\!=_S$ ::*Stateful Value* $\rightarrow$ (*Value* $\rightarrow$ *Stateful Value*) $\rightarrow$ *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.

6.3.3 Monads Defined

monad 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 $\gg\!=_m :: m\ t \to (t \to m\ s) \to m\ s$

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.

return 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.

$\gg\!=$ The *bind* function $\gg\!=_m$ specifies how m-computations are combined together.

In general the behavior of $A \gg\!=_m F$ 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 no problem.

TODO: mention the monad laws.

6.4 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.

6.4.1 The Monad Type Class

Haskell allow monads to be defined very cleanly using *type classes*.

type class A type class is a Haskell mechanism for overloading functios based on their type.

The *Monad* class has the following definition:

```
class Monad m where
  (>>=) :: m t → (t → m s) → m s
  return :: t → 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 type *Checked* is an instance of the *Monad* class:

```
instance Applicative Checked where
  pure val = Good val
  (<*>) = ap
instance Monad Checked where
  return = pure
  a >>= f =
    case a of
      Error msg → Error msg
      Good v → f 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.

6.4.2 Haskell do Notation

Haskell also supports special syntax for writing programs that use monads, which simplifies the use of the bind operator. The 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:

```
evaluate (Binary op a b) env =
 (evaluate a env) >>=_C (λav.
  (evaluate b env) >>=_C (λbv.
   checked_binary op av bv))
```

The effect here is for *av* to be bound to the value produced by the *evaluate a env*, and for *bv* to be bound to the result of *evaluate b env*. 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** notation, 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:

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

Here is the basic pattern for **do** notation:

```
do
  x ← e1
  e2
```

This is equivalent to this form, using bind:

```
e1 >>= (λ x . e2)
```

The expressions *e1* and *e2* must be expressions that produce values in the same monad m. To be precise, if *e1* has type $m\ t_1$ where m is a data type declared as an instance of *Monad*, then the variable x will be assigned a value of type t_1. Then *e2* must have type $m\ t_2$ for some type t_2. Note that the $\leftarrow$ symbol must be understood differently from $=$. What it means is that x is bound to the simple value produced by the computation *e1*. The $\leftarrow$ is there to remind you that x is not bound directly to the monadic computation produced by *e1*, but is bound to the value that the computation generates.

For a concrete example, if m is *Checked* then *e1* must have type *Checked t1* for some type *t1*. The value of expression *e1*, which is a *Checked t1*, could be a good value or an error. If *e1* produces an error then the computation stops, x is never bound to any value, and *e2* is not called. But if *e1* produces a good value v, then x will be bound to v (which is the value that was labeled *Good*) and the computation will proceed with *e2*.

One 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.

TODO: mention **let** in **do**, and the case where no variable is used.

6.5 Using Haskell Monads

The messy evaluators for error checking and mutable state can be rewritten much more cleanly using monads. In the format of the previous chapter's comparison of *Checked* and *Stateful* computations, here they are using monads:

Checked	Stateful
$av \leftarrow evaluate\ a\ env$	$av \leftarrow evaluate\ a\ env$
$bv \leftarrow evaluate\ b\ env$	$bv \leftarrow evaluate\ b\ env$
$checked\_binary\ op\ av\ bv$	$return\ (binary\ op\ av\ bv)$

Look closely, and they are nearly identical!

6.5.1 Monadic Error Checking

Here is a version of error checking using the *Checked* monad defined above: The code for error checking using monads is given in the Checked Monad zip file.

```
evaluate :: Exp → Env → Checked Value
evaluate (Literal v) env   = return v
evaluate (Unary op a) env = do
  av ← evaluate a env
  checked_unary op av
evaluate (Binary op a b) env = do
  av ← evaluate a env
  bv ← evaluate b env
  checked_binary op av bv
evaluate (If a b c) env = do
  av ← evaluate a env
  case av of
    BoolV cond → evaluate (if cond then b else c) env
    _ → Error ("Expected boolean but found " ++ show av)
  -- variables and declarations
evaluate (Variable x) env =
  case lookup x env of
    Nothing → Error ("Variable " ++ x ++ " undefined")
```

```
    Just v -> return v
evaluate (Declare x e body) env = do    -- non-recursive case
  ev <- evaluate e env
  let newEnv = (x, ev) : env
  evaluate body newEnv
  -- function definitions and function calls
evaluate (Function x body) env =
  return (ClosureV x body env)
evaluate (Call fun arg) env = do
  funv <- evaluate fun env
  case funv of
    ClosureV x body closeEnv -> do
      argv <- evaluate arg env
      let newEnv = (x, argv) : closeEnv
      evaluate body newEnv
    _ -> Error ("Expected function but found " ++ show funv)
```

Note that code involving errors only occurs where an error is actually raised. Other parts of the code, for example the case for *Binary* and *Declare* do not explicitly mention errors. This is very different from the code given in the Section on Error Checking.

6.5.2 Monadic Mutable State

The full code for the stateful evaluator using monads is given in the Stateful Monad zip 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 -> (t, Memory)
```

To define a monad, Haskell requires a **data** type that labels the function with a constructor. In this case, the constructor is named *ST*:

```
data Stateful t = ST (Memory -> (t, Memory))
```

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

The *Stateful* monad is an instance of *Monad*. It is fairly complex, mostly because of the need to deal with the type tag.

It defines *return* to return a value without changing memory. It does this by returning a *ST* with a function that takes a memory and returns the value with the memory unchanged.

```
instance Applicative Stateful where
  pure val = ST (λm → (val, m))
  (<*>) = ap
instance Monad Stateful where
  return = pure
  (ST c) >>= f =
    ST (λm →
      let (val, m') = c m in
        let ST f' = f val in
          f' m'
      )
```

It defines the bind operator $\gg\!=$ to take a stateful value c and a function f. It returns a new *Stateful* value that accepts a memory m, it then passes this memory to c and captures the result as a *val* and a new memory m'. It then applies the function f to the value *val* and captures the resulting *Stateful* value. This function f' is applied to the new memory m' to create the *Stateful* result.

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

```
evaluate :: Exp → Env → Stateful Value
  -- basic operations
evaluate (Literal v) env = return v
evaluate (Unary op a) env = do
  av ← evaluate a env
  return (unary op av)
evaluate (Binary op a b) env = do
  av ← evaluate a env
  bv ← evaluate b env
  return (binary op av bv)
evaluate (If a b c) env = do
  cond ← evaluate a env
  case cond of
    BoolV t → evaluate (if t then b else c) env
  -- variables and declarations
evaluate (Declare x e body) env = do    -- non-recursive case
  ev ← evaluate e env
  let newEnv = (x, ev) : env
  evaluate body newEnv
evaluate (Variable x) env =
  return (fromJust (lookup x env))
  -- first-class functions
evaluate (Function x body) env =
  return (ClosureV x body env)
evaluate (Call fun arg) env = do
```

```
    closure ← evaluate fun env
    case closure of
      ClosureV x body closeEnv → do
        argv ← evaluate arg env
        let newEnv = (x, argv) : closeEnv
        evaluate body newEnv
    -- mutation operations
evaluate (Seq a b) env = do
    evaluate a env
    evaluate b env
evaluate (Mutable e) env = do
    ev ← evaluate e env
    newMemory ev
evaluate (Access a) env = do
    addr ← evaluate a env
    case addr of
      AddressV i → readMemory i
evaluate (Assign a e) env = do
    addr ← evaluate a env
    ev ← evaluate e env
    case addr of
      AddressV i → updateMemory ev i
```

Note that the expression forms that don't involve memory, including unary and binary operations, function calls and function definitions, don't explicitly mention any memory operations, as they did in the code given in the Section on Mutable State. The evaluate function depends on three helper functions that provide basic stateful computations to create memory cells, read memory, and udpate memory.

```
newMemory val = ST (λmem → (AddressV (length mem), mem ++ [val]))

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

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

Assignment 3: Defining a Monad for State and Error handling

Combine the monads and interpreters for Error Checking and Mutable State into a single monad that performs both error checking and mutable state. You must also combine the evaluation functions.

The type of your monad must combine *Checked* and *Stateful*. There are several ways to do this, but then one needed for this assignment is:

$$\textbf{data}\ \mathit{CheckedStateful}\ t = \mathit{CST}\ (\mathit{Memory} \to (\mathit{Checked}\ t, \mathit{Memory}))$$

Chapter 7

Abstract Interpretation and Types

So far we have been focused on writing interpreters for small languages. An interpreter is a meta-program that evaluates a program in a written in the interpreted language. When evaluating an expression such as:

$$\textit{evaluate}\ (3 + 5) ==> 8$$

we cannot be more precise about the result of this particular program: the expression 3+5 evaluates only to the (concrete) number 8. The evaluate function implements what is called a *concrete interpreter*. However it is possible to write interpreters that return *abstract values*. Those interpreters, called abstract interpreters, return some abstraction of the result of executing a program.

abstract value An abstract value is a value that represents a collection of concrete values.

concrete interpreter A *concrete interpreter* is a normal interpreter that evaluates with normal values.

abstract interpreter A *abstract interpreter* is an interpreter that operates over abstact values using abstract operations.

validity An abstract interpreter is *valid* if the concrete value result is a member of the set of values of the abstract value evaluation result, for all evaluations.

The most common and familiar example of abstract interpretation is *type-checking* or *type-inference*. A type-checker analyses a program in a language, checks whether the types of all sub-expressions are compatible and returns the corresponding type of the program. For example:

$$\textit{typeCheck}\ (3 + 5) ==> \textit{Int}$$

A type-checker works in a similar way to a concrete interpreter. The difference is that instead of returning a (concrete) value, it returns a type. A type is an abstraction of values. When the type of an expression is *Int*, it is not known exactly which concrete number that expression evaluate to. However it is known that that expression will evaluate to an integer value and not to a boolean value.

Type-checking is not the only example of abstract interpretation. In fact abstract interpretation is a huge area of research in programming languages because various forms of abstract interpretation are useful to prove certain properties about programs.[TODO: more references]

7.1 Languages with a Single Type of Values

In a language with a single type of data type-checking is trivial. For example, in the language of arithmetic, which only allows integer values, type-checking would be defined as follows:

```
data Type = TInt

check :: Exp → Type
check e = TInt
```

In other words, all expressions have type *TInt* and type-checking cannot fail, since there are no type-errors. Therefore, type-checking only really makes sense in a language with at least two types of data.

7.2 A Language with Integers and Booleans

In this tutorial we are going to write a type-checker for a language with Integers and Booleans. The language with integer and booleans that we are going to use is the same one from the Section on More Kinds of Data:

```
data BinaryOp = Add | Sub | Mul | Div | And | Or
  | GT | LT | LE | GE | EQ

data UnaryOp = Neg | Not

data Exp = Literal  Value
  | Unary           UnaryOp Exp
  | Binary          BinaryOp Exp Exp
  | If              Exp Exp Exp
  | Variable        String
  | Declare         String Exp Exp
```

(NOTE: The language in the tutorial files includes an additional constructor Call. For this question you can ignore that constructor and there is no need to have a case for Call in the type-checker function.)

For this language types are represented as:

$$\textbf{data}\ \mathit{Type} = \mathit{TInt} \mid \mathit{TBool}$$
$$\quad \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})$$

This data type accounts for the two possible types in the language.

Type-checking can fail when the types of subexpressions are incompatible. For example, the expression:

$$3 + \mathit{true}$$

should fail to type-check because addition (+) is an operation that expects two integer values. However in this case, the second argument is not an integer, but a boolean.

In a language with variables a type-checker needs to track the types of variables. To do this we can use what is called a type environment.

Type environments A *type environment* is a mapping from variable names to types.

$$\textbf{type}\ \mathit{TEnv} = [(\mathit{String}, \mathit{Type})]$$

A type environment plays a similar role to the environment in a regular interpreter: it is used to track the types of variables during the type-checking process of an expression.

7.2.1 Type of the type-checker

We are going to use the following type for the type-checker:

$$\mathit{typeCheck} :: \mathit{Exp} \rightarrow \mathit{TEnv} \rightarrow \mathit{Type}$$

Note how similar this type is to the type of an environment-based interpreter except for one difference:

1) Where in the concrete interpreter we used *Value*, we now use *Type*.

If we look at *typeCheck* as an abstract interpreter, then types play the role of abstract values.

7.2.2 Typing rules for expressions

Most expressions in the language it are fairly obvious to type-check. For example, to type-check an expression of the form:

e1 + e2

we proceed as follows:

1) check whether the type of *e1* is *TInt*
2) check whether the type of *e2* is *TInt*
3) If both types are *TInt*, return *TInt* as the result type (*Just TInt*); otherwise fail to type-check (*Nothing*)

7.2.2.1 Typing Declare Expressions

To type a declare expression of the form

var $x = e$; *body*

we proceed as follows:

1) type-check the expression e
2) if e has a valid type then type-check the body expression with an type-environment extended with $x -- > typeof\ e$; otherwise fail with a type-error.

For expressions with unbound variables: for example:

var $x = y; x$

you should throw an error. In other words the type-checker works only for valid programs.

7.2.2.2 Typing *If* Expressions

The only slightly tricky expression to type-check is an **if** expression. The type-checking rule for an **if** expressions of the form:

if (*e1*) *e2* **else** *e3*

is

1) check whether the type of *e1* is *TBool*
2) compute the type of *e2*

3) compute the type of *e3*
4) check whether the types of *e2* and *e3* are the same. If they are the same return that type; otherwise fail.

The code for the *typeCheck* function is as follows:

```
typeCheck :: Exp → TypeEnv → Type
typeCheck (Literal (IntV _)) env = IntT
typeCheck (Literal (BoolV _)) env = BoolT
typeCheck (Unary op a) env =
  checkUnary op (typeCheck a env)
typeCheck (Binary op a b) env =
  checkBinary op (typeCheck a env) (typeCheck b env)
typeCheck (If a b c) env =
  if BoolT ≢ typeCheck a env then
    error ("Conditional must return a boolean: " ++ show a)
  else if typeCheck b env ≢ typeCheck c env then
    error ("Result types are not the same in "
      ++ show b ++ ", " ++ show c)
  else
    typeCheck b env
typeCheck (Variable x) env = fromJust (lookup x env)
typeCheck (Declare x exp body) env = typeCheck body newEnv
  where newEnv = (x, typeCheck exp env) : env
```

Here are the two helper functions, which are the abstract versions of binary and unary:

```
checkUnary Not BoolT = BoolT
checkUnary Neg IntT = IntT
checkUnary op a      = error ("Mismatched argument for " ++
                         show op ++ " " ++ show a)

checkBinary Add IntT IntT = IntT
checkBinary Sub IntT IntT = IntT
checkBinary Mul IntT IntT = IntT
checkBinary Div IntT IntT = IntT
checkBinary And BoolT BoolT = BoolT
checkBinary Or  BoolT BoolT = BoolT
checkBinary LT  IntT  IntT  = BoolT
checkBinary LE  IntT  IntT  = BoolT
checkBinary GE  IntT  IntT  = BoolT
checkBinary GT  IntT  IntT  = BoolT
checkBinary EQ  a     b     | a ≡ b = BoolT
checkBinary op  a     b     =
  error ("Mismatched binary types for " ++
    show a ++ " " ++ show op ++ " " ++ show b)
```

Note that if type-checking any subexpressions fails with a type-error then the type-checking of the **if** expression will also fail.

7.3 Type-Checking First-class Functions

In order to support type-checking we need to modify some of the datatypes with additional type-annotations:

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

In expressions functions need to have parameters annotated with types. Similarly in closures, the parameters need a type annotation. These are marked in bold above.

Types and type-environments are defined as follows:

```
data Type = IntT
  | BoolT
  | FunT Type Type   -- new
  deriving (Eq, Show)

type TypeEnv = [(String, Type)]
```

The interesting thing is that we now have a type for functions denoted by the constructor *TFun*. These types are similar to Haskell function types like $Int \to Int$ or $Bool \to (Int \to Bool)$. Function types are useful to have function arguments. For example:

```
function (f : Int → Int) {
  function (x : Int) {
    f (f (x))
  }
}
```

is a function that given a function f and an integer x, applies f twice to x.

Type checking a function definition is straightforward, given the argument type as part of the definition. It type checks the body of the function in a type environment extended with a declaration of the argument.

```
typeCheck (Function (x, t) body) env =
  FunT t (typeCheck body newEnv)   -- new
    where newEnv = (x, t) : env
```

Type checking a function call is also straightforward: it matches the actual argument type with the declared function argument type.

```
typeCheck (Call fun arg) env =
  case typeCheck fun env of
    FunT a b → if a ≢ typeCheck arg env
      then error "Invalid argument type"
      else b
    _ → error "Expected function"
```

The file FirstClassFunctionsTyping.hs adds a function *typeCheck* for doing type-checking.

```
typeCheck :: Exp → TypeEnv → Type
```

You can also optionally modify the Happy file to extend parsing so that we can deal with type-annotations in the language.

Here is a package of Haskell files for Typing.

Chapter 8

Data

So far we have focused on how to define and use static-scoped binding and higher-order functions, how to use *monads* to define pervasive computational strategies, like error checking and mutable state, and static type-checking. While these are important topics, they are certainly not all that you need to know in order to understand programming languages. This chapter delves into the possibility for defining and refining data in many different ways.

While we have defined primitive data types, including *int* and *bool*, and *closures* and *addresses* for functions and mutable state, we haven't examined the nature of data itself. The most complex data we have encountered is the use of **data** in Haskell to define the structure of abstract syntax *Exp* and values *Value*, and function types mentioned in this paragraph. We have not addressed how to define data within the languages that we have been studying.

The main topic of the chapter is how to define and implement *data abstraction*. By *data* we mean the information that appears in a program. Data is a complex topic, encompassing primitive data, data structures, algebraic data, abstract data types, objects in object-oriented programming, and databases. *Abstraction* is a concept that has appeared in several important places in this book already. It was used to describe *function abstraction*, or the definition of reusable transformations from inputs to outputs. It was used in the chapter on typing in the concept of *abstract interpretation*, where all values are abstracted to combine similar values together to operate on categories of values.

Abstraction applied to *data* means that data is constructed, manipulated, and observed without being tied to a particular representation. Thus there is an abstract view of data, which is used by *clients* of the data. And there is a concrete representation of data, which is created and managed by an *implementor*.

There are several benefits to having an abstract view of data. The first benefit is that only the essential properties of the data are visible to the programmer – all non-essential details are hidden. This allows a programmer to focus on what

is important, and ignore implementation details. A second benefit is that the implementation of the data can be changed, for example to be more reliable, smaller, or more efficient, without having to change a program that uses the data. This works as long as the abstract view of the data doesn't change, only the hidden implementation details change. It is also possible to choose different representations for data structures, for example, to change a program from small-scale data to managing large data. If these forms of data are sufficiently abstract, changing representation does not require the client program using the data to change.

Some of the key questions to be addressed here are 1) How can users define data types that look the same as primitives? 2) What are objects? 3) What are data structures? 4) Why do most popular languages, like Python and Java, have both primitive types and objects? 5) Is it possible to have a practical language without primitive data?

8.1 Data in Programming Languages

Primitive data types were introduced in the very first programming languages. They include integers, floating point numbers, booleans, and strings. Languages like Algol, FORTRAN, COBOL, and Lisp were defined with these primitives. Primitives sometimes included date/times, fixed-point numbers, and complex numbers. Early languages also had built in *data structures*, like records, arrays, unions, hash tables, and lists. These languages supported *user-defined data types* that were constructed using these primitive data structuring mechanisms.

However, the fact that primitives were different from user-defined data types began to be an issue. For example, if your language didn't support complex numbers, or date/time values, it was impossible for the user to define these types and have them work the same as primitive types. It was a challenge to figure out how to allow user-defined data types that look like primitive types.

This challenge was eventually solved by the discovery of *Abstract Data Types* (ADT). The key idea is in understanding primitive types. They work by having a *type name* together with *operations* on values of that type. Since they are primitive types they could be created by *literals* in the language, like -32 and 99.75. An Abstract Data Type is a user-defined type that has its representation *hidden* from the client. It includes operations for construction, manipulation, and observation of values. For example, an abstract data type for complex numbers might look like this:

```
type complex

val fromPolar : real, real -> complex
```

```
val + : complex, complex -> complex
val - : complex, complex -> complex
val * : complex, complex -> complex
val / : complex, complex -> complex

val magnitude : complex -> real
val phase     : complex -> real
val toString  : complex -> string
```

There are two primary forms of data abstraction: Abstract Data Types (ADTs) and Objects (OO). Primitive data is usually a form of abstract data type. Most object languages

For example, primitive types are abstract: a programmer using them doesn't *really* know how they are implemented or represented. This applies to booleans, integers, floating point values, strings and dates. For example, integers could be implemented as 32 or 64 bit representations, or they could be implemented as arbitrary-precision arithmetic, which would require variable amounts of storage and more complex operations.

All the programmer needs to know is that there is a way to create primitive values, operations that manipulate them, and ways to view some textual representation of the data. In a statically typed language, there will also be a *name* associated with each type of primitive data.

Chapter 9

Bibliography