Programming in Meruem

Melvic C. Ybanez

September 2015

Contents

Introduction

0.1 About the book

This book is a tutorial for the Meruem programming language, written by the people who developed the current version of Meruem. Our goal is to teach you the introductory concepts of functional programming and (to some extent) metaprogramming using the Meruem language.

0.2 Who should read this book

If you are someone looking for the next popular object-oriented programming language to master and doesn't feel like learning new and more mathematical ways of solving problems for now, then this tutorial is not for you. There are many options for new such languages out there, but Meruem is not (yet) one of them.

That said, if you are someone willing to spend a lot of time mastering not just a new programming language but also different programming paradigms, hoping that you will be able to apply all the knowledge you can gain from this book with any other programming languages you already know, then this book is for you.

0.3 Programming background required

This book is written primarily for imperative and/or object-oriented programmers who want to learn functional programming and metaprogramming, or people who don't know programming at all. If you are already familiar with functional programming and metaprogramming, then most of the contents here will not be new to you, but this book can still serve as a review material.

0.4 How to read this book

Most of the chapters in this book are not self-containing, so I recommend you read them in the proper order starting from chapter 1, especially if you are new to Lisp-like languages like Meruem.

Chapter 1

Starting Out

1.1 What is Meruem?

Meruem is a dynamically-typed, interpreted programming language that supports both *functional programming* and *metaprogramming*, and runs on top of the *Java Virtual Machine* (JVM).

Meruem is also a *Lisp* dialect. That means it has most, if not all, of the characteristics common to all Lisps, like *homoiconicity*, *macros*, and a small, simple and elegant core.

1.2 Why learn Meruem?

Meruem will change the way you think about programs, programming, and problems in general. The things that you will learn from this book will still be applicable to your day-to-day job as a programmer, even if you will be using a different and more mainstream programming language. This is because learning Meruem is not just learning a new programming language, it's learning completely new programming paradigms. Knowing different programming paradigms (imperative, OOP, FP, etc) is always a good thing since it would give you different ways of solving problems. After you've learned Meruem, you'd realize that there's more to programming than just *imperative* and/or *object-oriented programming*.

1.3 Overview of Programming Paradigms

Before we continue, let us first make a brief discussion about the different programming paradigms. We are not going to talk about all of them, though. We are just going to talk about the ones most programmers are familiar with (imperative, OOP), and the ones this book are going to focus on (functional programming and metaprogramming).

In the following subsections, I am going to show you some code samples from different programming languages. If you are not familiar with these languages, don't worry. Knowing them is not required. That said, I strongly recommend that you try reading these subsections (or at least the explanation parts).

1.3.1 Imperative Programming

In **Imperative programming**, you give the computer a sequence of statements for it to perform. Each of these statements can cause side effects. **Side effects** are changes (on a state or something) that occur in some place (like outside of a function being invoked) when a function, command or statement is invoked or executed. For example, the following code will print the string `Hello World` to the screen:

```
1 print "Hello World"
```

That is a Python snippet. It is a side effecting statement, because something is printed when that line of code is run. The state of the console has changed. Another example is the modification of variable values or references:

```
1 x = function_that_returns_int()
2 if x < 20:
3   x = 7
```

The above code calls a function named `function_that_returns_int`, and store the result to the variable `x`. If `x` is less than 20, then set it to 7. Line 3 is a side effecting assignment statement since you are destroying the old value and replacing it with a new one, making the value of `x` different from before. This is called a *desctructive assignment*. Line 1, however, is not a side effecting statement (assuming that `function_that_returns_int` is side-effect free) since assigning an initial value to a variable is not the same as changing it. This kind of assignment is known as *initialization*.

You'll learn more about side effects later in the book. (Though you won't learn much about imperative programming in general here.)

1.3.2 Object-oriented Programming (OOP)

In **Object-oriented programming**, you focus on designing data structures that contain both the data and the functions that can operate on these data. Most object-oriented languages also support imperative programming, and in some cases, I only think of them as either imperative or OO languages, even though they are (to some extent) both. Many languages are actually multi-paradigms (i.e. they support more than one paradigm), but they tend to favour one paradigm over the others.

Java is a good example of a language that supports both imperative and OOP. In Java 8, you can even do a little bit of functional programming. Java is usually thought of as an object-oriented language. If you program in Java, you are expected to do it the OOP way. For instance, if you want to create a data structure that represents a person, you can do it like this:

```
1 public class Person {
2     private String name;
3     private int age;
4
5     public void setName(String name) {
6         this.name = name;
```

```
7       }
8
9       public void setAge(int age) {
10          this.age = age;
11      }
12
13      public String getName() {
14          return name;
15      }
16
17      public int getAge() {
18          return age;
19      }
20  }
```

The code above is called a *class*, and most OO languages have it. Classes can contain both the variables (in this case, `name` and `age`) and the functions (in this case, `setName`, `getAge`, etc.) or *methods*. In order to use a class, you usually have to instantiate it. **Instantiation** is the process of converting a class into an object or *instace of a class*. To instantiate a class in Java, you use the `new` operator, as follows:

```
3  Person bob = new Person();
4  Person juan = new Person();
```

As you can see, you can create more than one object using a single class. That is because a class is just a blueprint of an object, and you can use the same blueprint many times to create its instances. Now you can use `bob` like this:

```
5  bob.setName("Bob");
6  bob.setAge(24);
7  System.out.println(bob.getName());
8  System.out.println(bob.getAge());
```

There are so many things to learn in object-oriented programming, such as *inheritance*, *polymorphism*, *encapsulation*, etc. The in-depth discussions of these topics are unfortunately outside the scope of this book.

1.3.3 Functional Programming

Functional Programming is about writing programs that consist mostly of functions and/or expressions. Functions in functional languages are *first-class citizens*, which means that you can consider or treat them like any other values. First-class functions can be passed as values to variables, or as actual arguments to other functions. You can also return functions from other functions. Basically whatever it is you can do with an ordinary value like an integer `107` or a string `"Hello World"` can be done with first-class functions.

I am not going to show any examples here, since this is one of the main things this book will be focusing on, anyway. You'll see a lot of examples through-out the book.

1.3.4 Metaprogramming

Metaprogramming is the writing of code that takes other code as input values, or produces other code. In other words, in metaprogramming, programs can be treated as data. So you can pass/return code to/from other functions. It's like in functional programming, except that you don't have to wrap things in functions in order to pass them around.

Lisps are quite known for their support for metaprogramming using *macros*. In this book, you will learn the macro system of Meruem. Remember, Meruem too is a Lisp.

1.4 Installing Meruem

To program in Meruem, you need to install Java and download the Meruem interpreter.

1.4.1 The Java Virtual Machine

As I've said above, Meruem runs on the Java platform, which is a JVM (sometimes I just refer to it as "the JVM"). To be more accurate, the current version of Meruem actually gets ran by the Scala programming language, which runs on top of the JVM. What I mean by that is that the interpreter of Meruem is written in Scala.

But, just what is a JVM?

Simply put, a JVM is a program that makes it possible for a computer to run your Java program. Essentially, without a JVM, we can't run Java programs. The Java compiler generates Java bytecode, the instruction set that the JVM understands and translate to machine code.

So how do Scala programs run on the JVM? Simple, the Scala compiler generates the same instruction set as the Java compiler. In other words, the Scala compiler would generate the same (or almost the same) bytecode as Java's. The JVM wouldn't even probably know (or care) if the bytecode it's translating to machine code were generated by the Java compiler or by the Scala compiler. And since Meruem is written in Scala, then a Meruem code will eventually be converted to Java bytecode.

So we need to install a JVM in order to run our Meruem interpreter. To do that, we install a *Java Runtime Environtment* (JRE). Installing a JRE was what I meant earlier by installing Java. A JRE contains the JVM, libraries, and some other things we shouldn't worry about in this book. There are many instructions on the web on how to install a Java runtime environment on different platforms, such as this one: `https://www.java.com/en/download/help/download_options.xml`. I recommend you complete the instructions first before proceeding.

Note: T here is also what is known as a *Java Development Kit* (JVM). You have to install it if you want to develop Java programs and not just being able to run them. A JVM already contains a JRE so you don't need to install both.

1.4.2 Downloading the interpreter

The next thing you need you do is to download the Meruem interpreter.

As you already know, Meruem is an interpreted programming language. That means it needs an interpreter, and not a comipler, so programs written in this language can be run. It is important to know the difference between an interpreter and a compiler. An **interpreter** is a program that executes the code on-the-fly without having to convert them to machine code beforehand, while a **compiler** is a program that translates a source code to something else like a machine code or (in the case of Java, or Scala) bytecode, without running them. More generally, a compiler translates one form of code to another.

To download the Meruem interpreter, go to `https://github.com/melvic-ybanez/Meruem/releases` and download the zip file (meruem.zip) of the latest release. Then, extract the zip file into the directory of your choice. Make sure the meruem folder and the meruem.jar are located in the same directory. You can also download the source codes (Source code (zip) or Source code (tar.gz)) if you want to view the code for the interpreter itself. After that create a new environment variable `MERUEM_HOME` and set its value to the path of the meruem installation.

That's it. Installing the Meruem interpreter simply means downloading the zip file, extracting it, and setting the value of the `MERUEM_HOME` variable.

1.5 Installing Winter

We can write Meruem code either by entering them on the REPL or by writing them to a file. The second method essentially creates a *source file*. The REPL (to be discussed later) is already included in the Meruem distribution that you downloaded earlier, so you don't have to download anything to use it. On the other hand, making source files requires the presence of a *text editor*. You can use any text editor that you want, but Winter is the more recommended one.

Winter is the second component of "Project Meruem" (the first component is the Meruem language). It is a text editor designed primarily for Meruem source files. It supports syntax highlighting, smart indents, a projects tree, and many more. You'll see more of its features when you start using it.

To download Winter, go to `https://github.com/melvic-ybanez/Winter/releases/tag/v1.0` and download the jar file (Winter.jar). After that you can run the Winter by double-clicking on the jar file. If that doesn't work, you can try running it by running `java -jar Winter.jar` on the terminal or command prompt. If everything worked fine, you should see a text editor program opened. It should look like this:

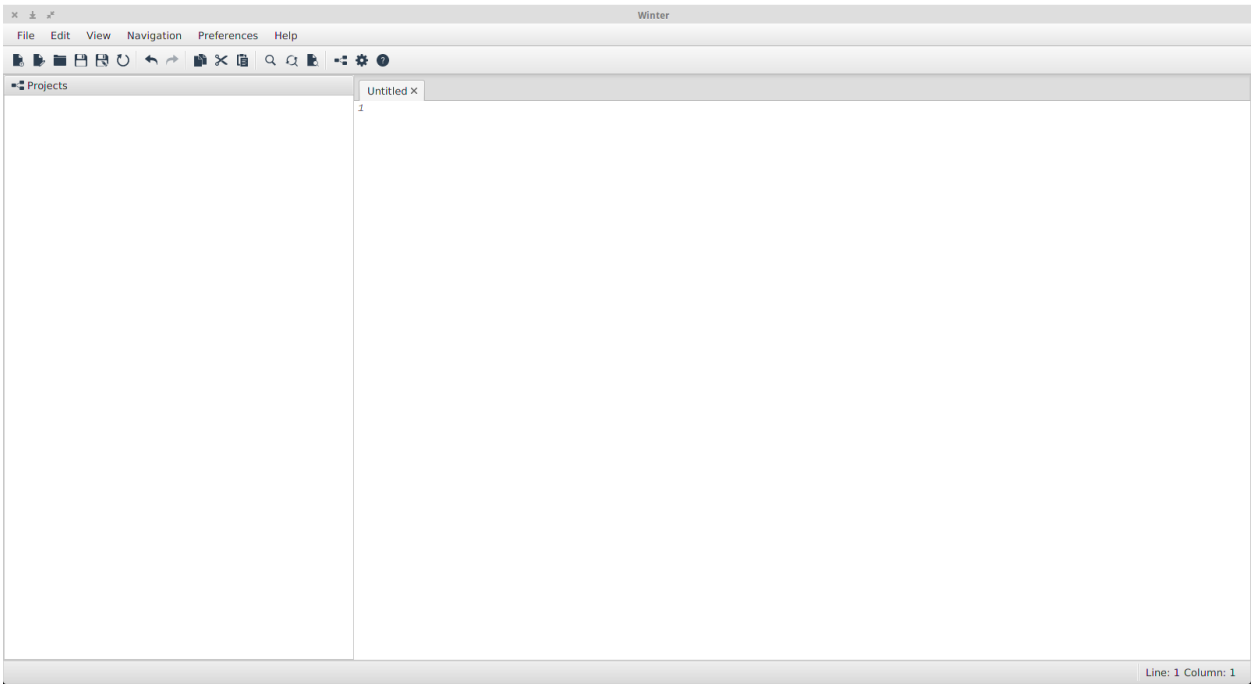

Figure 1.1: The Winter text editor

That's it. Installing Winter simply means downloading the jar file.

Note that you can't run both the Meruem interpreter (we can also just refer to it simply as "Meruem") and Winter by installing Java first.

1.6 Running the REPL

Many programming languages like Scala, Haskell, Clojure (another Lisp that runs on the JVM), Python, Ruby, and more, have what is known as a REPL. A **REPL** (Read-Eval-Print-Loop), also known as *interactive shell*, is a program that reads input from the user (Read), evaluate it (Eval), prints the result (Print) and ask for another input (Loop). The REPL decides how to print a value based on that value's "string representation". Every Lisp has a REPL (as far as I know). And since Meruem is a Lisp, it has a REPL too. It's called The Meruem REPL (surprise!).

The Meruem REPL is already packaged with the Meruem distribution that you have downloaded earlier. To run it, first, fire up the terminal (or command prompt, but from now on we'll just refer to it as "Terminal"). Then use the `cd` command to go to the directory where you installed Meruem. (If you don't want to keep on doing the previous step, you can make use of the MERUEM_HOME variable). When you're already inside the installation directory, just type `java -jar meruem.jar` and press the enter key. You should be able to see the following on the terminal:

```
meruem>
```

This signifies that the REPL has successfully started and that it is waiting for you to enter something. Try entering the expression `(+ 1 2)`, and then press enter:

```
meruem> (+ 1 2)
3
meruem>
```

The answer `3` has been printed. Now the REPL is waiting for another input. Don't worry if you don't understand what the code `(+ 1 2)` does. For now, what's important is to confirm that your REPL works fine.

To exit the REPL, run `exit`:

```
meruem> exit
Bye!
```

If you don't want to keep on `cd`ing into the meruem installation directory, you can make use of the MERUEM_HOME environment variable. Your command would then look like this: `java -jar $MERUEM_HOME/meruem.jar` (on Unix, or `java -jar %MERUEM_HOME%\meruem.jar` on Windows).

The REPL is useful if all you want to do is to try out some expressions. Instead of creating a new source file for that, you can just evaluate things inside the interactive shell and immediately see the results of the evaluations.

1.7 Hello World in Meruem

It is almost customary for any programming language tutorial to have a `Hello World` program as the very first complete program to run. We are going to follow such practice here.

First, run Winter (again by running the `java -jar Winter.jar` command.). Next, type the following in the editor area:

```
1  (defun main (args) (println "Hello World"))
```

Save it as `hello_world.mer`. Then, `cd` into where you saved `hello_world.mer` and run the following command: `java -jar meruem.jar hello_world`. You should be able to see the following output:

```
$ java -jar meruem.jar hello_world
Hello World
```

The `hello_world` is the name of the source file you want to run. A Meruem source file ends with a `.mer`. Basically, if the source file is not provided, Meruem will fire up the REPL for you.

If the code worked, then congratulations, you've just run your first complete Meruem program. This also means that your set up is complete and working well. Now you are ready to start learning the Meruem programming language.

Chapter 2

Data Types

In this chapter, we are going to make use of the REPL only. We are not going to use Winter (or any text editor you have right now). If you haven't already installed the REPL, please go back to section 1.4 and follow the instructions on how to install Meruem before proceeding. Remember, the Meruem distribution is already bundled with a REPL.

Programming always involves the manipulation of data. For instance, writing a program that adds two random numbers involves working on numbers. Reading the contents of a file involves the manipulation of files and strings. An enrolment system requires the presence of data that represent the student information, the class schedules, and others. Even the `Hello World` program that we wrote earlier wouldn't even be completed if we didn't know what data to print to the screen. Whatever it is you want to do, you need some data.

Now, the thing about data is they don't all have the same classifications, and the operations that you can perform on a data depend on the classification of that data. For example, you can add a number to another number, but you can't add a number to a student information. (That wouldn't really make sense.) This classification of data is known as a *data type*.

A **data type** tells you how a thing is classified, what set of values belongs to this type, and what operations can be performed on it. Meruem has a short list of supported data types. Let's discuss each of them, starting with the `Number` types.

2.1 Number

`Number` types are types whose instances hold numeric values. You can perform mostly mathematical operations on them. Meruem has four number types: `Integer`, `Long`, `Float`, and `Double`. The differences of these types will be discussed in the following subsections.

2.1.1 Integer

`Integer` types (or simply, `Integers`) are signed whole numbers (including 0). The "signed" indicates the inclusion of negative numbers. An integer can also be referred to as an `int`. To see what integers look like, fire up the REPL and enter the following expressions:

```
meruem> 10
10
meruem> -28
-28
meruem> 0
0
```

These are examples of integer literals. A **literal** is a value that was written directly in the source code, and not one shown to be stored in a data structure or variable, or treated as a return value from a function or expression.

Integer literals evaluate to themselves. This is why when you entered `10`, you also got the result of `10`.

2.1.2 Long

`Long` numbers are just like integers, except that they are `longer`:

```
meruem> 456@L
456
meruem> 456@l
456
meruem> 46466464@L
46466464
```

The only thing that makes `long`'s syntax different than `int`'s is the `@L` or `@l` at the end of the numbers. The `@L` tells the interpreter that the number preceding number should be interpreted as a long, and not an integer. The `L` can be either in lower or upper case. These suffixes are removed when `long` literals are evaluated.

Now you might be wondering when to use `long` instead of `int`. The answer is simple: data types have limitations, not only in their structure but also on the range of values they can take. Integers can only take values ranging from -2^{32} to $2^{31} - 1$. Try inputing a value larger than the maximum value of an integer and you will get an error:

```
meruem> 545345453535353
Exception in thread "main" java.lang.NumberFormatException:
    For input string: "545345453535353"
        at java.lang.NumberFormatException.forInputString(
           NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:583)
        ............
```

That is only a part of a longer error message. The error occurred because you provided a number that is greater than the maximum value of an integer. If you wanted to manipulate on numbers that big you should have used a long:

```
meruem> 5453454533535353@l
545345453535353
```

2.1.3 Float

A `float` is a number type that can contain decimal digits. You can write `float` by appending `@f` (or `@F`) to the a number:

```
meruem> 454@f
454.0
meruem> 453.454@F
453.454
```

Notice that even if you don't include some decimal digits, the interpreter will still treat the number as if there's a "`.0`" at the end.

2.1.4 Double

A `double` is just like a float, except that it has twice the precision and can hold a much bigger value. As you've probably already guessed, a `double` can with either `@d` or `@D`:

```
meruem> 5453.@d
5453.0
meruem> 654654.899898@D
654654.899898
meruem> 34.20
34.20
```

Notice that even if you didn't put a suffix (like `@d`), the value was still valid. That is because a number with decimal digits is by default interpreted as double. I encourage you to do more experiments on your own. After all, the best way to learn a programming language is to used it.

2.2 Boolean

A `boolean` accepts only one of the two possible values. In Meruem, these values are `true` and `false`: (In other languages that don't support boolean types, they are simulated using the values `0` and `1`).

```
meruem> true
true
meruem> false
false
```

The example shows that boolean literals, like integer literals, evaluate to themselves.

2.3 Character

A `character` (also referred to as `char`) is any alpha-numeric (letter or number) character or special symbol that is preceded by a \ character:

```
meruem> \a
\a
meruem> \6
\6
meruem> \?
\?
```

One thing to notice is that even a number gets turned into a `char` when preceded by a \ character. A limitation of this type is that it can't hold multiple characters:

```
meruem> \hello
An error has occurred. Parse Failure: string matching regex
    '\z' expected but 'e' found
Source: .home.melvic.meruem.meruem.prelude [11:53}]
(defun truthy? (expr) (and (!= expr false) (!= expr nil)))
```

Here, the interpreter is telling you that it can't parse the input properly. (In this chapter, we shouldn't worry too much about the details of the errors.)

Note: T he evaluated form of a character in the REPL contains a \ at the beginning. However, when you run a source file that prints a character to the console, you shouldn't see the \ printed in the output.

2.4 List

`List` is a collection of things. It is written as a set of space-separated elements enclosed in a pair of parentheses. Each of the elements of a list is a value with it's own type. Since `List` is a type that can be composed of some other types, it is considered as a *compound type*. Here are some examples of `lists` in Meruem:

```
meruem> (list 1 2 3 5)
(1 2 3 5)
meruem> (list \h \e \l \l \o)
(h e l l o)
meruem> (list 5@f 66@L)
(5.0 66)
```

The `list` element that comes before any of the items in each of the sample expressions is a `function` (functions will be discussed later) that constructs a list whose elements are the rest of the items inside the parentheses. The constructed list gets returned and printed.

`List` is the most important data type (or data structure, in some sense) in Meruem. Unlike in non-lisp languages where `lists` are usually used only to hold a certain data (e.g a collection of things), `lists` in Meruem can be considered not only as a data structure, but also as a function application, a function definition, or more generally, (a part of) your whole code. Lisp was originally an acronym, LISP, which means "list processing". In other words, programming in a Lisp like Meruem, is about working primarily on lists.

Meruem programs are composed only of `atoms` and `lists`. In Meruem, an **atom** is any data or value that is not a list. It is the smallest component of a program. In other words, numbers, chars, and booleans are all atoms. *Symbol*s, which will be explained in the next few sections, are also considered `atoms`. The rest of the Meruem code are just lists, and you can perform on them any manipulation that you can do on a list.

Let's have a few more examples:

```
meruem> (list 34 \t 56@l)
(34 t 56)
meruem> (list \h \e \l \l \o (list \w \o \r \l \d))
(h e l l o (w o r l d))
meruem> ()
()
```

The first expression shows that you can combine different kinds of values (integer, char, and long, in this case) in a list. The second one shows that you can nest lists (I didn't say that lists can only contain atoms :)). When evaluating a list with another lists as some of its elements, the inner lists get evaluated first before evaluating the outer one. The third expression in the example is an empty list. Let's have a more complicated example:

```
meruem> (cons (head "hello") (++ (list \w \o) (list \r \l \
   d)))
(h w o r l d)
```

This example uses types and operators that we haven't discussed yet, so we don't expect you to fully understand it now. However, this should at least show you that you can have more than two levels of nesting in a list, and that the order of evaluation goes from the innermost lists to the outermost ones.

2.5 String

A `string` is a series of characters. You can express a string literal by enclosing the characters that it contains in double quotes:

```
meruem> "I am a string"
"I am a string"
meruem> "hello world"
"hello world"
```

The double quotes get displayed as well in the REPL, but when you print a string via source file the quotes shouldn't appear.

A string can act as both an atom and a list. You can treat it as a list of characters. You will see later on that many list operations can be applied to strings as well. However, a string is technically not a list. You can argue that it is an atom, pretending to be a list. Or perhaps we can just say that a string is a type of it's own. Anyway, let's have another example:

```
meruem> "hello\nworld"
```

```
"hello
world"
```

The output looks weird, right? That's because the \n is an escape sequence (not a character literal inside a string). An **escape sequence** is a sequence of characters that get translated into something else during evaluation. In this case, the \n gets translated into a newline character, causing the next character to appear in the next line. The escape sequence \n is useful if you want your string output to span multiple lines. There are more escape sequences supported by Meruem strings. Each of them is a combination of the *escape character* (\) and a character. Let's see a couple more of them:

```
meruem> "hello\tworld"
"hello  world"
meruem> "h\bello world"
"ello world"
meruem> "My name is \"Bond\""
"My name is "Bond""
```

A \t is used to insert tab, and a \b means backspace. A \" inserts a double quote. Normally, you can't insert a double quote inside a string literal (without escaping it). There are many more escape sequences. Discussing all of them is already beyond the scope of this book. Fortunately, you can find plenty of discussions about them on the web (most languages tend to have the same set of supported escape sequences, after all).

2.6 Symbol

In Meruem, a `symbol` is a series of characters that means something. It is a name that maps to some other value, like a *variable*, *function*, or *macro*. The evaluated form of a symbol shows the object or value it is mapped to. To make things clear, let's try entering a few symbols in the REPL:

```
meruem> quote
quote
meruem> cons
<function>
meruem> head
<function>
meruem> cond
cond
```

In the last example, `quote` and `cond` are symbols that evaluate to themselves. These two are built-in operations that you are going to use frequently when writing Meruem programs. `cons` and `head` are functions. Instead of showing the real objects the symbols point to, the REPL shows their string representations. If you try to evaluate a symbol that doesn't point to anything, the interpreter will yell at you:

```
meruem> foo
An error has occurred. Unbound symbol: foo.
```

```
Source: .home.melvic.meruem.meruem.prelude [1:1}]
foo
^
```

The interpreter is telling you that `foo` isn't bound to anything, so it doesn't know how to evaluate it. Symbols are case-sensitive. That means `cons` and `Cons` are not the same:

```
meruem> Cons
An error has occurred. Unbound symbol: Cons.
Source: .home.melvic.meruem.meruem.prelude [1:1}]
Cons
^
```

We've already seen `cons` evaluate to `<function>`. However, `Cons` is a different symbol, and in this case isn't bound to any value (just like `foo`), so it returns an error.

2.7 Nil

The last type I'm going to discuss is the `Nil` type. `Nil` indicates the absence of value. This type only has one possible value or instance, and it is `nil`. Nil also evaluates to itself, as shown in the following example:

```
meruem> nil
nil
meruem> Nil
An error has occurred. Unbound symbol: Nil.
Source: .home.melvic.meruem.meruem.prelude [1:1}]
Nil
^
```

You will get to see its real usage later on.

Note: I n some Lisps, `nil` refers to an empty list. In other words, `()` and `nil` are just the same. In Meruem, that is not the case. `Nil` is an atom of it's own.

Chapter 3

Language Syntax

In this chapter, we are going to focus the discussion on the structure of Meruem code. We are going to talk about it's odd-looking syntax, and why it is something you should try to embrace rather than avoid. Let us start the discussion with the main (and probably the only) syntax component of a Meruem program: *S-expressions*

3.1 S-expressions

The syntax of Meruem code is composed only of s-expressions. In fact the syntax for the whole program itself is an s-expression. So the key to mastering the syntax of Meruem is to learn how s-expressions work. The good news is it's actually pretty easy to learn s-expressions. So what is an s-expression?

An **s-expression** (or `sexprs`) is a recursive tree-like data structure whose *leaf sub-nodes* are *atoms* and the non-*leaf sub-nodes* are themselves s-expressions. (A *leaf* node is a node that doesn't have any children.) In other words, an s-expression can be either an atom (the smallest unit of expression in Meruem), or a composition of other s-expressions (like I said, it's recursive).

I'm sure you're already familiar with `atoms`, but just to refresh your memory let's see some of them again:

```
meruem> 10
10
meruem> 30@l
30
meruem> 50.79@f
50.79
meruem> 70@d
70.0
meruem> \c
\c
meruem> cons
<function>
```

Again, atoms are the first type of s-expression. The second one has the following structure:

```
(elem1 elem2 elem3)
```

where `elem1`, `elem2`, and `elem3` are all s-expressions. Does it look familiar? That's right! It looks like a `list`. This is why in the previous chapter, I mentioned that the Meruem program itself can be considered as a list. The following expressions are s-expressions that are not atoms:

```
meruem> (list 1 2 3)
(1 2 3)
meruem> (list "the quick brown" (list \f \o \x))
(the quick brown (f o x))
meruem> (++ "I want" "cookies")
"I wantcookies"
```

It is worth noting that this type of s-expressions and lists are not always the same. A list is a data structure, while an s-expression refers to the syntax itself. For instance, consider the expression `(+ 1 2 4)`. If you run it in the REPL, you'll get `7` as a result. So, the expression `(+ 1 2 4)` is an s-expression but you wouldn't call it a list. It is a number, just like $1 + 3$ is a number expressed as the sum of 1 and 3. In other words, the type of the expression is the type of the value returned when we evaluate it. On the other hand, the expression `(list 1 2 3)` is an s-expression and a list, because the value it returns when evaluated, which is `(1 2 3)`, is a list. The difference between s-expression and list, however, does not invalidate the claim that Meruem programs are just lists. Internally, everything is still a list. Also, you can capture an entire expression (an s-expression) without evaluating it, effecting treating it as an ordinary list (see section). For this reason, the two terms can be used almost interchangeably. Also, in this book, I usually refer to s-expressions (or list) as just "expressions".

3.2 Executable expressions

Not all s-expressions (or lists) can be evaluated. For instance, if you are going to evaluate `(1 2 3)`, you'll get an error:

```
meruem> (1 2 3)
An error has occurred. 1 can not be converted to a function
   .
Source: .home.melvic.meruem.meruem.prelude [1:2}]
(1 2 3)
 ^
```

So why can we evaluate `(list 1 2 3)` and not `(1 2 3)`? The error message from the last example made it sound like the interpreter was expecting for the first element of the list to be a function. And that's, indeed, what was happening. To be more accurate, the interpreter requires that the first element of a list should be a *callable*. A callable is a thing that can be called and/or applied to some arguments, like a function, operator,

special commands, macros, and more. An expression whose first element is a callable is an executable expression. The syntax of a executable expression looks like this:

```
(callable-name arg1 arg2 arg3 ...)
```

`callable-name` refers to the name of the callable (like `list` or `++`). The `...` indicates that the number of arguments (or *args*) are not limited to only 3. Let's have some examples (you've already seen some of them):

```
meruem> (list \c \d)
(c d)
meruem> (+ 6 7 (- 9 20))
2
meruem> (cons \m "eruem")
"meruem"
```

In Meruem, most of the callables are effectively functions. An operator like + is just a function who happened to have that operator-looking name. (In many languages, naming a function like this is not possible.). Macros are also just special functions, internally. Most of the special commands behave like functions as well. In fact, I rarely use the term "callable" in Meruem. I just refer to these things as "function"s. So does the interpreter (see the last error message). For that reason, executable expressions are usually just called *function calls* or *function applications*. You'll learn more about functions in the later part of the book.

3.3 Why S-expressions

S-expressions might look so uncool and unreadable at first, but there are actually reasons why Lispers prefer them over the C-like syntax.

One reason is **simplicity**. All you have to do is think of atoms, lists, and how to evaluate them. Atoms are easy to understand and many of them just evaluate to themselves. List are evaluated by treating the first element as a function and apply it to the rest of the elements. And that's it. That's most of what you need to know about the syntax of Meruem. Compare that to the syntax of, say, Java. In Java, there are so many things to remember, like the syntax for defining variables, arrays, classes, objects. Also, there are so many special symbols like `()`, `[]`, <>, {} and others. In Meruem, you only need to watch out for the parentheses, and the more you learn about Lisps, the more all these nested parentheses will look good to you.

Another reason is **consistency**. For instance, the operators, special commands, functions and macros can usually all be considered a function, and they all appear in the same place.

Finally, there are some advantages offered by the *prefix notation* (or *polish notation*). A **prefix notation** is a notation where the operator appears before the operands. That's exactly the case in our lists. One advantage of this is that you can add as many operands as you want and specify only the operator once. The expression `(+ 1 2 3 4 5)` is much easier to write than the *infix* equivalent `1 + 2 + 3 + 4 + 5`. Another advantage is that you don't have to worry about *operator precedence* anymore. The inner expressions always get evaluated first. For instance, in `(* 56 9 (+ 5 5) (- 4 5))`, it is quite clear that + and − will be invoked before the `*`.

I could go on, but I think I've already made my point. You will see more advantages of s-expressions as you continue using the Meruem language, anyway.

3.4 Special Syntaxes

There are some parts in a Meruem code that are not in a s-expressions form. The following subsections will focus on each of them.

3.4.1 Comments

A **comment** is a part of the code that the interpreter just ignores. Comments are useful when you want to write something in plain English without letting the interpreter throw some *parse errors*. Usually, you comment things for documentation purposes. Comments in Meruem are any lines of code that start with the `;` character. Any character following the `;` up to the end of the line will be ignored by the interpreter. The following example shows that commented parts of the code will not be included in the evaluation:

```
meruem> (+ 56 7) ; this is a comment
63
meruem> "hello" ; world
"hello"
```

Becareful when writing comments though. If you insert it inside an expression, it might cause a parse error:

```
meruem> (* 50 20)  ; this will run just fine
1000
meruem> (* 50 ; this will return an error 20)
An error has occurred. Parse Failure: `)' expected but end
   of source found
Source: .home.melvic.meruem.meruem.prelude [11:53}]
(defun truthy? (expr) (and (!= expr false) (!= expr nil)))

                                                        ^
```

To the eyes of the interpreter, the second example is just equivalent to `(* 50`, and that is not a valid s-expression. You can, however, do it like this:

```
1  (defun main (args)    ; This is the main function.
2    ; The main functions is the entry point of a Meruem
       program.
3    (println "I will still be printed"))
```

The code shows that the elements in a list can be separated by multiple spaces, tabs, and newline characters, and not just by single spaces in between. Since this code spans multiple lines (not to mention it actually contains a *main* function, but more on this later), it is a good idea to save it as a source file using the Winter text editor (instead of running it directly in the REPL). Then run it via `java -jar meruem.jar filename` like you

did in section 1.7. I trust that from now on, you can tell which code samples are not to be run in a REPL.

3.4.2 Quotes

As I've mentioned earlier, you can prevent an expression from being evaluated. To do so, you'll just have to add `'` at the beginning of the expression:

```
meruem> '456
456
meruem> '(1 4 5)
(1 4 5)
meruem> '(list 3 5 6)
(list 3 5 6)
meruem> 'foo  ; this works, even if foo is an unbound
   symbol
foo
```

The `'` is known as the *quote* operator. Actually, the above expressions still got evaluated, but the resulting values are just the same as the corresponding inputs but without the quotes. This operator is one of the most important syntactic sugars (syntax that make things more pleasant to read) in Meruem, so expect to see more uses of it from now on. There is a more advanced form of quote, called *quasiquote*, but we are not going to discuss it in this chapter.

3.4.3 List of Pairs

It is not uncommon for an expression to be a list of pairs (where a pair is a list that holds two items). In fact it is so common that I decided to add a *syntactic sugar* for it. The following example shows how to use it:

```
meruem> '((1 "one") (2 "two"))  ; this expression
((1 one) (2 two))
meruem> '{ 1 "one" 2 "two" }  ; is the same as this
((1 one) (2 two))
```

Notice the use of the quote operator. Without the quote, the first expression would have been written like `(list (list 1 "one") (list 2 "two"))`, which is quite verbose. But more importantly, notice how the two expressions in the example evaluate to the same values. That is because they are just the same expression, except that the second one used a syntactic sugar. In general, the form `((a b) (c d))` (where `a`, `b`, `c`, and `d` are all s-expressions) can also be written as {`a b c d`}.

Chapter 4

Standard Functions

Meruem is a functional programming language. So when you program in Meruem, you don't only rely on the power of lists, you rely heavily on the power of functions too. Creating and using functions are one of the most common tasks in functional programming, so we better know how to do both. In this chapter we are going to learn how to use some of the existing standard functions of the Meruem programming language.

4.1 What is a Function?

A **functoin** is piece of code that is designed to perform a specific computation. A function takes a set of inputs (which may be empty), does something with it, and return an output. The inputs are known as *parameters* and the output is known as the *return value*. In the previous chapter, you've learned that the syntax of a function call looks like this:

```
(function arg1 arg2 arg3 ...)
```

Note that I've replaced `callable` with `function`, to emphasize that most of the callables are effectively just functions. The elements starting from the second one are the *arguments* of the function. Function arguments are additional data that you pass to the function in order for it to complete its computations and return the result. The function receives the values passed as arguments by storing them to *parameters*.

4.2 Evaluating a Function

In order to evaluate a function call, the arguments need to be evaluated first. Then, you apply the function (first element) to the evaluated arguments. The result of evaluating the first argument gets passed to the first parameter of the function, the result of evaluating the second argument gets passed to the second parameter, and so on. Since function calls are s-expressions, and s-expressions are recursive in nature, some of the arguments of a function can be complex expressions or function calls too.

Note: T here are some functions that don't evaluate all of their arguments first before evaluating the function itself. You will learn about them later.

Lists are normally just evaluated as function calls. The expression `(1 3 4)` is not executable, because while it is a list, its first element is not a function. If you want to call a function that doesn't take any arguments, just wrap the function in parentheses, effectively creating a list with only one element (e.g. `(some-function)`).

Now that you know what functions are and how function evaluation works, it's time to test some standard and built-in functions of Meruem.

4.3 Type-converting Functions

In many cases, it is very useful to convert one type to another. For that reason, Meruem provides some type converting functions that do exact that kind of conversion:

```
meruem> (to-long 4)
4
meruem> (to-int 30.23)
30
meruem> (to-double 67)
67.0
meruem> (to-float 45@l)
45.0
meruem> (to-string 56)
"56"
meruem> (to-string 56@d)
"56.0"
meruem> (to-char 45)
\-
meruem> (to-char 97)
\a
```

The names of these functions are self-explanatory. For instance, `to-long` takes a number and tries to convert it to a `long`. If you apply a function to argument whose type is not compatible with the type the function is expecting, you'll get an error:

```
meruem> (to-char "the")
An error has occurred. Invalid Type. Not a Number: the
Source: .home.melvic.meruem.meruem.prelude [1:10}]
(to-char "the")
         ^
```

In this example, the interpreter was expecting a number, and you gave it a string. If you are not sure what type an expression has, you can use the `type` function to retrieve it:

```
meruem> (type 20)
Integer
meruem> (type 45.456)  ; double, even without the @d
Double
```

4.4 Functions as Operators

As mentioned in the previous chapter, operators in Meruem are actually just functions whose names are composed of operator (or special) symbols. In other words, there is nothing special about them. In the following subsections you are going to see the different operators supported by Meruem.

4.4.1 Arightmetic Operators

Arightmetic operators are the basic mathematical operators that we have learned since primary school. Let's review them using s-expressions:

```
meruem> (+ 1 2 3)
6
meruem> (- 56 78)
-22
meruem> (* 34 2 3 5)
1020
meruem> (/ 100 20)
5
meruem> (% 19 4)
3
```

The only operator there not that was never taught in school was the last one (`%`). It is known as the *modulus* operator. The modulus operator will return the remainder after dividing the first operand with the second one.

Each of these operators or functions receives a varying number of arguments. For instance, the + operator can be applied to more any number of arguments you want:

```
meruem> (+ 34 65 6 7 87 5 4 3)
211
meruem> (+ 4 5)
9
```

What if you don't pass any arguments at all? Well, the addition function returns the additive identity number, the multiplication function returns the multiplicative identity number, and the substraction and division operators return errors:

```
meruem> (+)
0
meruem> (*)
1
meruem> (/)
An error has occurred. Incorrect number of arguments: 0
Source: .home.melvic.meruem.meruem.prelude [9:28}]
(defun lazy (expr) (lambda () ,expr))

                                ^
```

```
meruem> (-)
An error has occurred. Incorrect number of arguments: 0
Source: .home.melvic.meruem.meruem.prelude [9:28}]
(defun lazy (expr) (lambda () ,expr))

                                    ^
```

Since they are all just s-expressions, you can nest them to your heart's content:

```
meruem> (+ 5 (- 788 89 0) 54  9 (/ 4 5 (* 4 6)))
767
```

You can create the most complex mathematical expressions using only these simple contructs. Now, take a lot at this example:

```
meruem> (/ 100 7)
14
```

We all know that 100 is not divisible by 7, so why did the expression return a whole number? The reason is that the interpreter performs an *integer division*. That is, it disregards the decimal digits after the division. If you want to include the decimal digits in the result, you need to convert one of them into a float or double:

```
meruem> (/ 100@f 7)
14.285714
meruem> (/ (to-double 100) 7)
14.285714285714286
```

4.4.2 Relational Operators

What if you want to know which of the two numerical expressions is greater? What if you want to check if the two function calls return the same values? In programming, comparing things is very common. That is why Meruem was designed to have sufficient number of *relational operators*.

Relational operators are operators that are used to check the relationship between two or more expressions. The relational operators supported by Meruem includes the *equality* and *inequality* operators:

```
meruem> (= 45 3)
false
meruem> (> 55 7)
true
meruem> (< 345 6)
false
meruem> (>= 54 3)
true
meruem> (<= 67 45)
false
meruem> (> 5 6 6)
```

```
false
meruem> (= () '(1 2 3))
false
meruem> (< () ())
An error has occurred. Invalid Type. Not a Number: ()
Source: .home.melvic.meruem.meruem.prelude [9:28}]
(defun lazy (expr) (lambda () ,expr))

                                    ^
```

Notice that relational functions can take varying number of arguments too. Also, while the equality operator can take non-numeric arguments, the others can't.

4.4.3 Logical Operators

The last types of standard operators in Meruem are the *logical operators*. There are 3 logical operators supported in the language: `and`, `or`, and `not`. All three of them can take varying number of arguments. They are also one of those functions that don't necessarily evaluate all of their arguments prior to the actual application.

And

The **and** operator checks the truthfulness of all of the arguments. If an argument is falsy, it returns that argument, and the rest of the argument won't even be evaluated. If all arguments have been evaluated, it would return the last one. If no arguments are provided, the `and` operator returns true.

Before we jump to examples, it is worth discussing what values are considered "truthy" and what values are considered "falsy" in Meruem. Well, it's actually pretty simple. The values `false` and `nil` are considered falsy values. The rest are thruthy.

Now let's see some sample usages of the `and` operator:

```
meruem> (and 45 false 6)
false
meruem> (and 45 nil 5)
nil
meruem> (and 45 89)
89
meruem> (and)
true
meruem> (and nil 55 3 true)
nil
```

Or

The **or** operator also checks the truthfulness of the arguments. It also behaves just like the `and` operator. However, unlike `and`, `or` will return the first truthy (and not the falsy) argument found. The following examples show the difference:

```
meruem> (or 45 false 6)
45
meruem> (or 45 nil 5)
45
meruem> (or nil false 7)
7
meruem> (or nil 7 false)
7
meruem> (or nil false nil)
nil
meruem> (and 1 2 false)
false
```

Not

The last logical operator is **not**. It simply negates an expression, returning the opposite of an expression's truthfulness:

```
meruem> (not true)
false
meruem> (not false)
true
meruem> (not nil)
true
meruem> (not 20)
false
```

To determine whether a value or expression is truthy or falsy, we can use the `truthy?` and `falsy?` functions respectively.

```
meruem> (truthy? true)
true
meruem> (truthy? false)
false
meruem> (truthy? nil)
false
meruem> (truthy? 1)
true
meruem> (falsy? true)
false
meruem> (falsy? 1)
false
meruem> (falsy? false)
true
```

4.5 Conditional Functions

Sometimes you may want to decide at runtime what code to execute (or expression to evaluate) based on a given condition. You can do this using *conditional expressions*.

4.5.1 If-expressions

The simplest form of conditional expressions is the `if-expression`. An if-expression takes three arguments. First, it checks the truthfulness of the first one. If the first argument is truthy, it evaluates the second argument and return the result. Otherwise, it evaluates the third argument and return the result. Here's are some if-expressions in action:

```
meruem> (if (> 10 4) "greater" "lesser")
"greater"
meruem> 'foo
foo
meruem> (if 'foo 5 7)
5
meruem> (if nil false true)
true
meruem> (if (= () ()) (if true 10 9) nil)  ; nested
   if-expressions
10
```

4.5.2 The Cond Expression

The other, more general, conditional expression is the *cond expression*. It takes a list of pairs. In each of the pairs, `cond` would evaluate the first element. If the first element of the pair is truthy, it would evaluate the second element, halt the evaluation process (meaning the rest of the pairs would no longer be evaluated), and returns the result of evaluating the second element. You'll understand this better with a few examples:

```
meruem> (cond (true 5) (false 6))
5
meruem> (cond (nil "ignore this") (false 'nope) (5 "yaay"))
"yaay"
meruem> (cond (nil 5) (false 5))        ; returns nothing
nil
meruem> (cond)  ; no conditions to check
nil
```

4.6 Predicates

A **predicate** is a function that maps an expression to a boolean value. Basically, all they do is just take an argument and check if that argument is something that falls in a certain

category. Here are the standard predicates currently present in Meruem (you've already seen some of them):

atom? checks if the given argument is an atom.

symbol? checks if the given argument is a symbol.

list? checks if the given argument is a list.

truthy? checks if the given argument is truthy.

falsy? checks if the given argument is falsy.

even? checks if the given argument is an even number. It doesn't accept non-numeric types.

odd? checks if the given argument is an odd number. It doesn't accept non-numeric types.

empty? checks if a string or list is empty. It doesn't accept any other types of values.

Each of these names is actually self explanatory. Let's see some examples of some of them:

```
meruem> (atom? 6)
true
meruem> (list? ())
true
meruem> (even? 67)
false
meruem> (empty? ())
true
meruem> (empty? "hello")
false
```

4.7 More Utility Functions

This section will discuss a few other commonly used functions.

4.7.1 Apply

The **apply** function takes a function and a list and apply the function to the unpacked contents of the list as if the contents were passed directly to that function:

```
meruem> (apply + '(1 2))
3
meruem> (apply list '(1 2 4))
(1 2 4)
meruem> (apply if '((< 5 6) true nil))
true
```

The `apply` function is useful when you want to apply a certain function to a set of arguments whose size and values aren't known until runtime (i.e, the arguments were dynamically generated). For instance, supposing that there is a function `foo` that takes an integer and returns a list of integers. What is the easiest way for you to compute their sum? You can't do `(+ (foo 10))` because the + function takes a list (as in a set) of integer arguments, not an argument that is a list (as in lisp list) of integers. The way to do what you want is to use the `apply` function, like this: (apply + (foo 10)).

4.7.2 Lazy

The **lazy** function (or macro) takes a single argument, wrap this argument in a function that doesn't take any arguments, then return that function. The argument being passed to the `lazy` function does not get evaluated prior to the function call. The purpose of `lazy` is simply to "delay" the evaluation of an expression. Let's see some examples:

```
meruem> (lazy 56)  ; returns a function that returns 56
<function>
meruem> ((lazy 56))  ; evaluates the function returned by
   lazy
56
```

Note that `<function>` is the string representation of a function. The *string representation* of an expression is the string value returned by applying the `to-string` function to that expression.

4.7.3 Identity

The **identity** function takes a single expression and immediately return that expression:

```
meruem> (identity 45)
45
meruem> (identity "hello")
"hello"
meruem> (identity (+ 5 3))
8
```

It looks so useless, right? Well, you'll understand it's purpose when we get to *higher-order functions*.

4.7.4 Increment and Decrement

The last functions we are going to discuss in this section are the increment and decrement functions. The **increment** function, denoted by `inc`, takes a number and returns that number increased by one. It is essentially the same as `(+ 1 arg)`, where `arg` is a numeric argument. The **decrement** function, denoted by `dec`, is the opposite of the increment function (i.e it is the same as `(- 1 arg)`).

Finally, here is the last set of examples for this chapter:

```
meruem> (inc 67)
68
meruem> (dec 30)
29
meruem> (inc 9.8)
10.8
meruem> (inc \a)
An error has occurred. Invalid Type. Not a Number: a
Source: .home.melvic.meruem.meruem.prelude [1:6}]
(inc \a)
     ^
```

Chapter 5

Defining Variables and Functions

In the previous chapter, you learned how to use some of the standard and built-in functions in Meruem. In this chapter, you are going to learn how to create your own functions and how to assign names to them. Not only that, you are also going to learn how to name s-expressions in general, by storing them to *variables*.

5.1 Lambdas

Technically, Meruem supports only one way to create a function, and that is through the use of the `lambda` command. More specifically, the `lambda` command is used to create a *lambda*. A **lambda** refers to anonymous functions. In other words, lambdas are functions whose names you don't know or care.

To use the `lambda` command, you need to pass two arguments to it. (Remember, even commands in Meruem behave like functions too.) The first argument is a list. This list would represent the parameters for the function (which in this case, a lambda) you are creating. The second argument is any expression. This expression would be the body of your anonymous function. The result of evaluating the body of a function would also be the value the function would return. Here's an example:

```
meruem> (lambda () 6)
<function>
```

This function takes no arguments and just returns the integer `6`. If you evaluate it, you'd see `<function>` as the result. That is because the `lambda` command creates a function instance, and (as stated in the previous chapter) the string representation of a function is `<function>`. To evaluate a lambda, just enclose it in parentheses:

```
meruem> ((lambda () 6))
6
```

Pretty simple, right? Let's try creating a lambda that takes two integers and return their sum:

```
meruem> (lambda (a b) (+ a b))
<function>
```

In this example, a and b would be the parameters of the lamba or function. You can give a parameter any name that you want. By the way, is the difference between parameters and arguments already clear to you? Let's clarify it, either way. **Parameters** are the ones that appear as part of the function definition (like a and b in the last example), while **arguments** are the expressions or values that you pass to the function during function call (like 1 and 2 in (+ 1 2)). In other words, the arguments would substitute the parameters inside the function. Now, let's try applying the last function to the values 10 and 20:

```
meruem> ((lambda (a b) (+ a b)) 10 20)
30
```

This expression is a list. Again, when you evaluate a list, the interpreter would treat it as a function call, and therefore ask for the first element to be a function. The first element in this example is (lambda (a b) (+ a b)), which is a lambda. The second and third elements of this list would be the first and second arguments respectively.

It is important to know that the number of arguments passed should be the same as the number of accepting parameters:

```
meruem> ((lambda () 5) 4)
An error has occurred. Extra arguments: (4)
Source: .home.melvic.meruem.meruem.prelude [1:16}]
((lambda () 5) 4)
               ^

meruem> ((lambda (a b c) a) 1 2)
An error has occurred. Not enough arguments. Expected
   values for (c).
Source: .home.melvic.meruem.meruem.prelude [1:15}]
((lambda (a b c) a) 1 2)
              ^
```

In the first expression, you created a lambda that doesn't accept any parameters, but you provided it with one argument (in this case, 4). So the interpreter complained because there's an extra argument of 4. The second example is the opposite. This time, you created a lambda with 3 parameters, but passed only two arguments. The interpreter complained because it is expecting a value for c.

Another important thing to remember is that the types of values passed to a function should also be correct:

```
meruem> ((lambda (a) (inc a)) 5)
6
meruem> ((lambda (a) (inc a)) \5)
An error has occurred. Invalid Type. Not a Number: 5
Source: .home.melvic.meruem.meruem.prelude [1:23}]
((lambda (a) (inc a)) \5)
                      ^
```

An error occured in the second input because we passed a character to a function that passes its parameter directly to inc. Remember, the inc function only takes numbers as arguments. Therefore, the parameter a could only be a number.

The order of the arguments matter too:

```
meruem> ((lambda (a b) (- a b)) 20 30)
-10
meruem> ((lambda (a b) (- a b)) 30 20)
10
```

Lambdas can take a varying number of parameters too, just like some of the standard functions introduced in the previous chapter. You can tell a lambda to accept any number of arguments by placing the `&` symbol before the name of the parameter that will serve as the list to store the arguments:

```
meruem> ((lambda (& xs) (apply + xs)) 1 2 3)
6
```

This function takes any number of arguments and computer their sum. In this case, `xs` would be a list with the elements `1`, `2`, and `3`. You can also add some parameters before the `&`. In that case, those parameters would still take the values from the arguments (as usual), but the remaining argument values who don't have any receiving parameters will be stored in the parameter following the `&`.

```
meruem> ((lambda (a b & rest) (list a b rest)) 1 2 4 5 6)
(1 2 (4 5 6))
```

5.2 Variables

A **variable** is a symbol that is bound to a specific expression or value. To define a variable, you use the `def` command. The `def` command takes two arguments. The first argument is the name of the variable, and the second one is the value the variable is bound to. Here are some examples:

```
meruem> (def x 10)
nil
meruem> (def one-plus-two (+ 1 2))
nil
```

The first expression sets `x` to the integer `10`. The second one sets `one-plus-two` to the result of evaluating `(+ 1 2)`. Both expressions return `nil` because all each of them do is define a variable. As mentioned before, `nil` is used to indicate the "absence of value". In this case, we return `nil` because returning a value wouldn't really make sense.

To use the variables we just defined, we can simply refer to their names:

```
meruem> x
10
meruem> one-plus-two
3
meruem> (+ x one-plus-two)                ; add the two
   variables
13
```

You can also set a value of a variable to another variable:

```
meruem> (def y x)
nil
meruem> y
10
```

The name of a variable should be a symbol. Therefore, the following definitions wouldn't work:

```
meruem> (def \c 20)
An error has occurred. Invalid Type. Not a Symbol: c
Source: .home.melvic.meruem.meruem.prelude [1:6}]
(def \c 20)
     ^
meruem> (def "x" 100)
An error has occurred. Invalid Type. Not a Symbol: x
Source: .home.melvic.meruem.meruem.prelude [1:6}]
(def "x" 100)
     ^
```

5.2.1 Immutability and Variable bindings

In imperative languages, you can both mutate the value a variable is bound to, or reassign a new value to a variable. In functional languages like Haskell or Meruem, you can't do both. This is because functional languages follow the more mathematical (rather than algorithmic) approach to solving problems.

In mathematics, a variable is just like an alias to a value or expression (that eventually resolves to a value). Mutating a value (like 100) or expression (like $a + b$) wouldn't really make sense. You also can't bind an already bound variable to something else in the middle of a solution. For instance, in algebra, you usually see problem solutions like the following:

$$\begin{aligned} a &= 2 \\ b &= 3 \\ x &= (a+b)(a-b) \\ &= (2+3)(2-3) \\ &= -5 \end{aligned}$$

In the example above, a remains 2 and b remains 3 until the end of the solution. Both a and b are variables because they could be anything prior to the evaluation of $(a+b)(a-b)$, and not because they could be anything while $(a+b)(a-b)$ is being evaluated.

Same thing can be said to variables in Meruem. Once a variable has been initialized, you can no longer reassign a new value to it:

```
meruem> (def foo 10)
nil
```

```
meruem> (def foo 20)
An error has occurred. foo is already defined.
Source: .home.melvic.meruem.meruem.prelude [1:6}]
(def foo 20)
     ^
```

5.3 Defining Functions

You already know how to define variables. Now let's see how to define functions. It's actually pretty simple. Remember, `def` assigns the result of evaluating its second argument to the first one, which is symbol, denoting the name of the variable. The second argument can be any s-expression, including lambdas. Since a lambda is a function, then by storing a lambda to a variable, we've already given name to a function.

Let's try that idea by creating our own function `sum` that takes a list of numbers and compute their sum:

```
meruem> (def sum (lambda (& xs) (apply + xs)))
nil
meruem> (sum 1 2 3)
6
meruem> (sum)
0
meruem> sum
<function>
meruem> (type sum)
Function
```

Storing lambdas to variables is so common that there is a macro created just for that. It's called `defun`. The **defun** macro takes three arguments. The first one would be the name of resulting function, the second argument would be the list of arguments this function will take, and the third argument would be the body of this function. Here's the equivalent of the `sum` function that we created earlier implemented using `defun` instead of `def`:

```
meruem> (defun sum (& xs) (apply + xs))
nil
meruem> (sum 43 54)
97
```

Let's try something that is a bit more complicated. How about a function that takes an integer representing a person's age, and returns the stage of the life cycle that person is currently in. Here's the code:

```
1  (defun life-stage (age)
2    (cond
3      ((< age 0) nil)
4      ((< age 3) "Infancy")
5      ((< age 6) "Early Childhood")
```

```
6       ((< age 8) "Middle Childhood")
7       ((< age 11) "Late Childhood")
8       ((< age 20) "Adolescene")
9       ((< age 35) "Early Adulthood")
10      ((< age 50) "Midlife")
11      ((< age 80) "Mature Adulthood")
12      (true "Late Adulthood")))
```

This function is not meant to be coded in the REPL due to its size. We better write it in a source file and run it as a complete program. In order to do that, we need a *main function*. The following subsection will talk about it (including how to run the `life-stage` function).

The **main function** is the entry point of a Meruem program. That means every Meruem programs starts by calling the main function. Without it, a program is not runnable. Here's the code for the `Hello World` program again:

```
(defun main (args)    ; This is the main function.
  ; The main functions is the entry point of a Meruem
     program.
  (println ("Hello World")))
```

As you can see, `main` has one parameter (in this case, named `args`). It is a list of command line arguments, which are all strings. You can pass command line arguments by specifying them after the name of the program to run. For instance, if the name of your source file is `hello_world.mer`, and you want to pass the arguments `foo`, `bar`, and `baz`, your command would look like this: `java -jar meruem.jar hello_world foo bar baz`.

Now, let us try printing the result of invoking the `life-stage` function. Save the following code in a Meruem file and run it:

```
1   (defun life-stage (age)
2     (cond
3       ((< age 0) nil)
4       ((< age 3) "Infancy")
5       ((< age 6) "Early Childhood")
6       ((< age 8) "Middle Childhood")
7       ((< age 11) "Late Childhood")
8       ((< age 20) "Adolescene")
9       ((< age 35) "Early Adulthood")
10      ((< age 50) "Midlife")
11      ((< age 80) "Mature Adulthood")
12      (true "Late Adulthood")))
13
14  (defun main (args)
15    (println (life-stage 20)))
```

As you've probably already guessed, the `println` function prints the string representation of its argument to the console. Without it, we wouldn't see any output. That's because

running a program is different than evaluating expressions in the REPL, where every results get printed immediately.

5.4 Scoping

Scope determines the part of the code where a particular bound symbol is accessible. For instance, `x` in the following example can be accessed inside the function `addBy10` but not outside of it:

```
meruem> (defun addBy10 (x) (+ 10 x))
nil
meruem> (addBy10 67)
77
meruem> x  ; x is undefined in this scope
An error has occurred. Unbound symbol: x.
Source: .home.melvic.meruem.meruem.prelude [1:1}]
x  ; x is undefined in this scope
^
```

The reason the REPL can't accessed `x` outside of `addBy10` is because it's only visible inside that function. In other words, `x` is a bound symbol that is local to `addBy10`. We can even create a new variable `x` outside of `addBy10` and there won't be any conflicts. This particular type of scoping is known as *lexical scoping*.

A function can accessed variables declared outside of it:

```
meruem> (def x 10)
nil
meruem> (defun incX () (inc x))
nil
meruem> (incX)
11
```

If a bound symbol exists both within and outside of a function, the one within it prevails:

```
meruem> (def a  100)
nil
meruem> (defun identity1 (a) a)  ; "a" is local to
   identity1
nil
meruem> (identity1 20)  ; 20 or 100?
20
```

5.5 Let-expressions

Let-expressions are used to simplify complex expressions by breaking them down into multiple parts, and giving names to those parts, making them more readable and/or reusable.

You construct a let-expression using the `let` command. The `let` command takes two arguments. The first argument is a list of pairs. Each of the pairs is composed of a variable followed by the value that variable would be bound to. The second argument would be an expression that can make use of the variables defined in the first argument. The result of evaluating the second argument would be the result of the let-expression itself. The following shows an example of a let-expression:

```
meruem> (let ((a 10) (b 20)) (+ a b))
30
```

We can also use the syntactic sugar for the lists of pairs that was discussed in section 3.4.3:

```
meruem> (let { a 10 b 20 } (+ a b))
30
```

Let-expressions also have a *function scope*, which means that variables declared inside them can't be accessed by any outsiders:

```
meruem> (let { msg "hello" } msg)
"hello"
meruem> msg
An error has occurred. Unbound symbol: msg.
Source: .home.melvic.meruem.meruem.prelude [1:1}]
msg
^
```

As a way of showing a more complicated example, let us refactor the code for the `life-stage` function using let-expression by giving a name to the task of comparing the given `age` to a particular number. The following example shows that a let-expression can also define a variable that holds a lambda, essentially creating a function within the expression:

```
1  (defun life-stage (age)
2    (let { up-to (lambda (limit) (< age limit)) }
3      (cond
4       ((up-to 0) nil)
5       ((up-to 3) "Infancy")
6       ((up-to 6) "Early Childhood")
7       ((up-to 8) "Middle Childhood")
8       ((up-to 11) "Late Childhood")
9       ((up-to 20) "Adolescene")
10      ((up-to 35) "Early Adulthood")
11      ((up-to 50) "Midlife")
12      ((up-to 80) "Mature Adulthood")
13      (true "Late Adulthood"))))
```

Chapter 6

Recursion

6.1 What is Recursion?

Recursion is an approach to solving a problem by using a function that is composed of simpler instances of the same function. It is essentially similar to constructing a function that calls itself.

One common example is the recursive, mathematical definition of *factorial*. The **factorial** of a positive integer n is the product of n and the *factorial* of 1 less than n. This is a recursive definition because the term being defined (which is $factorial$) appears, as an important element, in it's definition.

Here's a sample implementation of a factorial function in Meruem:

```
1  (defun factorial (n)
2    (if (= n 0)
3      1
4      (* n (factorial (dec n)))))
5
6  (defun main (args)
7    (println (factorial
8               (to-int (head args)))))
```

Before we discuss the `factorial` function itself, let's first take a look at our main function here. The main function applies the `head` function — which takes a list and returns its first element — to `args`. Then it converts the result (which, in this case, is a string) to `int`. Next, it applies the `factorial` function to the integer result. (`factorial` can only be applied to integers so we had to convert the `head` of `args` to `int` first.) Lastly it prints the result of calling `factorial`. Basically, what we're doing is telling the user to pass as a command line argument the value `factorial` is to be applied to. With this, we can run the program multiple times with different arguments without having to modify the code.

Now, let's talk about the `factorial` function. It takes an integer and checks if its value is `0`. If it is `0`, then the function returns 1. Otherwise, the function returns the product of the integer `n` and the result of applying itself to the value of `n` decreased by 1. In this case, the `factorial` function is performing a recursive call. Let's try running it a few times to

see if it will behave as we wanted it to:

```
$  java -jar $MERUEM_HOME/meruem.jar factorial 5
120
$ java -jar $MERUEM_HOME/meruem.jar factorial 10
3628800
$ java -jar $MERUEM_HOME/meruem.jar factorial 4
24
```

In order to have an idea of what's really happening when we ran `factorial`, you can substitute each of the argument values for every parameter `n` in each call to `factorial`. For instance, here's what happens when you execute `(factorial 5)`:

$$\begin{aligned} factorial(5) &= 5 \times factorial(4) \\ &= 5 \times (4 \times factorial(3)) \\ &= 5 \times (4 \times (3 \times factorial(2))) \\ &= 5 \times (4 \times (3 \times (2 \times factorial(1)))) \\ &= 5 \times (4 \times (3 \times (2 \times (1 \times factorial(0))))) \\ &= 5 \times (4 \times (3 \times (2 \times (1 \times 1)))) \\ &= 120 \end{aligned}$$

As you can see, we couldn't compute for the factorial of 5 without knowing the factorial of 4 first. And we couldn't get the factorial of 4 without calling the factorial of 3, and so on. When we finally got to $factorial(0)$ — or `(factorial 0)` in Meruem — the recursive calling stopped. That is because we hit the *base case* (I'll explain this in a minute), which says that the factorial of 0 is 1. When $factorial(0)$ returned 1, $factorial(1)$ resumed it's computation. When $factorial(1)$ completed and returned, $factorial(2)$ resumed, and so on. Eventually, the original call — $factorial(5)$ — returned and we had our result.

A recursive function should have a base case. A **base case** is the case in which the result can be computed without resorting to a recursive call. In our factorial function, the base case happens when the input is 0. You can say that $factorial(0)$ is the simplest instance of the factorial function. If we remove the base case of the factorial function, calling it would cause an *infinite recursion*, because calling $factorial(0)$ would result to $0 \times factorial(-1)$, instead of 1, $factorial(-1)$ resolves to $-1 \times factorial(-2)$, and so on.

6.2 Recursion and StackOverflowErrors

Here's a function that adds all the positive integers from 1 up to the integer n:

```
1  (defun sum-1-to-n (n)
2    (if (<= n 0)
3      0
4      (+ n (sum-1-to-n (dec n)))))
```

First, it checks if the parameter is bound to `0` or a negative number. If it is, the function returns `0`. Otherwise, it returns the sum of `n` and the `sum-1-to-n` of `1` less than `n`.

Make changes to `main` so that it will print the result of calling `sum-1-to-n` instead of `factorial`. Save it as `sum12n.mer` (or any filename you want). You can run it just like you did `factorial.mer`:

```
$ java -jar $MERUEM_HOME/meruem.jar sum12n 5
15
$ java -jar $MERUEM_HOME/meruem.jar sum12n 10
55
$ java -jar $MERUEM_HOME/meruem.jar sum12n 70
2485
$ java -jar $MERUEM_HOME/meruem.jar sum12n 0
0
$ java -jar $MERUEM_HOME/meruem.jar sum12n -3
0
```

Works fine! Now what if we try inputting a bigger number:

```
$ java -jar $MERUEM_HOME/meruem.jar sum12n 400
Exception in thread "main" java.lang.StackOverflowError
        at meruem.Evaluate$.apply(Evaluate.scala:15)
        at meruem.Evaluate$.apply(Evaluate.scala:27)
.........
```

Tadaah! It's an error. More specifically, it's a *stackoverflow error*. (This error is supposed to display a very long message or stack trace, but I simplified it to conserve space.) Allow me to explain the issue.

Every function call we make requires the allocation of a new *stack frame*, which means that the deeper the recursion gets, the more *stack frames* are needed. The problem is, we are only provided with a limited number of stack frames. So, if the recursion requires more stack frame than necessary, you'd get a `StackOverflowError`. Here's what the stack looks like when we execute `(sum-1-to-n 5)`:

```
(sum-1-to-n 5)
(+ 5 (sum-1-to-n 4))
(+ 5 (+ 4 (sum-1-to-n 3)))
(+ 5 (+ 4 (+ 3 (sum-1-to-n 2))))
(+ 5 (+ 4 (+ 3 (+ 2 (sum-1-to-n 1)))))
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum-1-to-n 0))))))
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1)))))
.....
(+ 5 10)
15
```

Now, that's pretty exhausting! Imagine if we tried tracing the stack of `(sum-1-to-n 400)`. It would take us a decade to finish it. In the next section I am going to show you how to fix this kind of problem.

6.3 Tail-recursion

Tail-recursion is a recursion in which the recursive call is the last thing the function does before returning. Here's the tail-recursive version of `sum-1-to-n`:

```
1  (defun tail-sum1ton (n acc)
2    (if (<= n 0)
3      acc
4      (tail-sum1ton (dec n) (+ n acc))))
```

We changed the name to `tail-sum1ton` so we can easily distinguish the older version from the new one. This function has two parameters instead of one. The first parameter is the same `n` we had before. The second parameter is called an *accumulator*. In our base case, we return `acc` instead of 0. Then, if `n` is greater than `0`, `tail-sum1ton` would call itself, but this time the result won't be added to `n`. What would happen now is that `acc` would be added to `n` and the result would be passed as the second argument to the recursive call. In other words, we no longer return the "current sum" as we did before. We just pass it as the second argument to the recursive call. After the recursive call, `tail-sum1ton` have nothing else to do but to return the result immediately.

If Meruem was a language that supported *tail-call optimization* — a method used by a compiler or interpreter to avoid allocating a new stack frame for a function during a *tail call*, the stack trace of `tail-sum1ton` would have looked like this:

```
(tail-sum1ton 5 0)
(tail-sum1ton 4 5)
(tail-sum1ton 3 9)
(tail-sum1ton 2 12)
(tail-sum1ton 1 14)
(tail-sum1ton 0 15)
15
```

Unfortunately, Meruem doesn't support tail-call optimization. So, if we execute `(tail-sum1ton 400 0)`, we'll still get a `StackOverflow` error.

6.3.1 `tail-rec` and `recur`

The more recommended way of solving the stack overflow issue (that we saw earlier) in Meruem is to use the `tail-rec` and `recur` functions.

`tail-rec` takes two arguments. The first one is a list of key-value pairs. Each of the keys would be a variable that is local to the `tail-rec` function, and each of the values would be the initial value of the corresponding key. The second argument of `tail-rec` could be any expression that, when evaluated, would serve as the return value of `tail-rec` itself.

`recur` is usually used only within `tail-rec`. It's purpose (when used within `tail-rec`) is to make `tail-rec` to call itself (essentially making it recurse).

When `tail-rec` calls itself for the second time, its variables would be bound to the arguments to `recur`, and not to their initial values. This is not the same as reassigning

new values to the variables of `tail-rec`. Just think of it as recursion. In recursion, every recursive call creates a new instance of the same function, whose parameters might be bound to different values. So no reassignments really happen.

Enough chit-chat, let's now reimplement `sum-1-to-n` using `tail-rec` and `recur`. Here's the new code:

```
1  (defun tail-sum-1-to-n (n)
2    (tail-rec { n n acc 0 }
3      (if (<= n 0)
4        acc
5        (recur (dec n) (+ n acc)))))
```

In this case, `tail-rec` has two local variables, `n` and `acc`. `n` gets initialized to `n` (the outer one) and `acc` gets initialized to `0`. The arguments to `recur` — `(dec n)` and `(+ n acc)` — gets passed to `tail-rec`'s `n` and `acc` respectively, in every recursive call.

We can now pass large values to `sum-1-to-n` without getting a `StackOverflow` error:

```
$ java -jar $MERUEM_HOME/meruem.jar sum12n 400
80200
$ java -jar $MERUEM_HOME/meruem.jar sum12n 500
125250
$ java -jar $MERUEM_HOME/meruem.jar sum12n 600
180300
```

How does `tail-rec` get optimized? Well, internally the interpreter would perform a tail-recursion. Since the interpreter is written in Scala, and Scala's tail-recursions get converted to *loops* at *compile-time*, then Meruem's tail-rec is essentially like loops in imperative languages.

Chapter 7

Module System

When your program gets really big and complicated, it is not a very good idea to put everything into one source file. You need to split the program into smaller pieces, and make each of these pieces as reusable as possible so different parts of the program can use it. Such a piece is called a *module*.

7.1 What is a Module?

A **module** in Meruem is a reusable named component of a program, just like a function. One difference between a module and a function is that a function contains expressions while a module can contain functions. In other words, a large program can be composed of several modules, and each of these modules can be composed of functions. One source file serves as a single module, and the functions and variables within that source file are the members of that module. From now on, we are going to use the term "module" instead of "source file" when we refer to Meruem files.

In other to use a module you need to *import* it. You can do that using the `import` command. The `import` command takes the name of the module to import as a string argument. Let us create a simple module that has one variable and one two functions:

```
1  (def x 10)
2
3  (defun squared (x) (* x x))
4
5  (defun cubed (x) (* x x x))
```

Save it as `module_demo.mer`.