# 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.

That's it. Installing the Meruem interpreter simply means downloading the zip file and extracting it.

## 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). 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 add the path to Meruem's installation directory to your environment paths). 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!
```

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. 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> 545345453535353@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` ends with either `@d` or `@D`:

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

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. 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.

## 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,

19

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)
```

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 a named piece of reusable code that takes a set of inputs and return an output. The inputs are know 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 parameters of the function.

## 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 of this list) to the evaluated arguments. In other words, the result of evaluating the first argument becomes the first parameter of the function, the result of evaluating the second argument becomes the second parameter, and so on. Note that some of these arguments can be function calls too.

Lists are normally just evaluated as function calls. This is why the expression `(1 3 4)` is not executable, because while it is a list, its first element is not a function.