CX Programming Language

Amaury Hernandez-Aguila

Contents

1. Getting Started with CX

This Chapter works as an introduction to Skycoin's programming language: CX. In the following Sections you will learn about the objectives and philosophy of the language and about the features that make CX unique.

In this first Chapter you can find instructions on how to install CX, and how to write and run your first program using the language. We will check how CX programs are internally represented in Chapter 2, so we can understand some debugging features and the CX REPL. We'll then review some basic programming concepts in Chapter 3, like what is a variable and the types of data these variables can represent, how you can group different values in arrays and slices, how you can group different types of values in structures, and how you can change the scope of a variable in a program. In Chapter 4 you will learn how to use functions and methods, and we'll talk a bit about side effects. The different control flow mechanisms that CX currently offers are covered in Chapter 5, such as *if* and the *for loop*. The last fundamental piece is packages, which help us modularize our programs, and they are covered in Chapter 6.

After Chapter 6, we'll start covering more complex subjects, such as pointers in Chapter 7 and how to use CX with OpenGL and GLFW in Chapter 8. Chapter 9 covers how CX can work both as an interpreted and as a compiled language, and what advantages bring each mode. Chapters 10 and 11 describe CX's garbage collector and affordances.Chapter 12 describes CX's serialization capabilities, and we'll learn how we can serialize a full, running program, store it in a file, and later deserialize it to continue its execution.

CX uses its affordance system to create a genetic programming algorithm that can be used to create programs that create programs, and this feature is explained in Chapter 13. Talking about creating programs that create programs, are you interested on creating your very own CX? If you are, we'll cover that subject in Chapter 14. Lastly, we'll cover some advanced techniques you can use while using the REPL to create a program in Chapter 15, and Chapter 16 teaches us how to create unit tests to make sure everything works as intended while your programs grow larger.

1.1 What is CX?

CX is an interpreted and compiled, garbage collected, general purpose programming language, which means that it can be used to create any type of program, such as web, desktop and command line applications. However, at the time of writing this book, the current version of CX is best suited to create command line applications and, surprisingly, video games! In a not so distant future, CX will be capable of handling all sorts of platforms, for the development of many kind of applications. But do not despair, as the current capabilities of the language are enough to try the fundamentals of programming, video game programming and other very powerful and interesting features, such as affordances, which are unique to CX.

You may be wondering about the objectives of CX – why create another language anyway? One of the main objectives of the language is to lower the software development costs by increasing the programmer's productivity. CX achieves this by providing well-known programming tools, such as a Read-Eval-Print Loop (REPL) for interactive programming and interactive debugging mechanisms. But CX goes further by extending the common REPL's capabilities. Those extensions will be reviewed in Chapter 15.

Another objective of CX is to provide the developer with many high-level tools that are part of the core language, i.e., the programmer won't need to install any external packages and the language will interact seamlessly with these features. Clear examples are CX's interactivity with the Skycoin ecosystem for creating decentralized applications and CX's affordance system and genetic programming functions that help the programmer create applications in a more interactive manner.

1.2 Installing CX

Eventually, we'll have a bootstrapped version of CX, and you'll be able to compile CX using CX, but in the meantime, you need to have a working Go installation to compile CX, as CX is implemented in this language. Although providing instructions on how to install Go is out of the scope of this book, we can give you some guidelines:

- At the time of writing, you can find instructions on how to install Go here: `https://golang.org/doc/install`
- Make sure you get a version of Go superior to 1.8
- Correctly setting a Go environment – particularly the GOPATH variable – usually decreases the chances of getting errors with the installation of CX

After getting your Go installation ready, you will need to install some libraries or programs, depending on your operating system.

In the case of Linux distributions, you might need to install some OpenGL libraries, if you haven't done already. CX has been tested in Ubuntu, and the commands to get the required libraries for this distribution are shown in Listing 1.1.

```
sudo apt-get install libxi-dev
sudo apt-get install libgl1-mesa-dev
sudo apt-get install libxrandr-dev
sudo apt-get install libxcursor-dev
sudo apt-get install libxinerama-dev
```

Listing 1.1: Installing Required OpenGL Libraries in Ubuntu

As there are dozens of Linux distributions, it'd be hard to give instructions on how to get the correct libraries for each of them. Nevertheless, using your favorite search engine to find out the names of those libraries for your distribution, and how to install them should be easy.

If you are using Windows, you might only need to install *GCC*. If you already installed GCC through Cygwin, you might run into trouble, as Go apparently doesn't get along with Cygwin. If you haven't installed GCC, you should install it either through *tdm-gcc* (`http://tdm-gcc.tdragon.net/`) or Mingw (`http://www.mingw.org/`).

At the moment, most users of CX have installed it on MacOS systems, and in all of the cases the installation of the language has been straightforward.

And finally, you'll need *Git* installed, regardless of your operating system. If you find any problems with the installation, we'll be grateful if you can open an issue at CX's GitHub repository (`https://github.com/skycoin/cx`), so we can improve the installation process!

After going through the hassles of installing Go and the required libraries, you should be able to install CX by running either the *cx.sh* (for *nix users) or the *cx.bat* (for Windows users) installation scripts, which can be found in CX's GitHub repository (`https://github.com/skycoin/cx`). If you are running a *nix operating system, you can also try the command shown in Listing 1.2.

```
sh <(curl -s https://raw.githubusercontent.com/skycoin/cx/master/cx.sh)
```

Listing 1.2: One-liner CX Installation Script for *nix Systems

If everything went well, you should be able to see CX's version printed in your terminal after running **cx -v**.

1.3 Hello, world!

Now it is time to write your first program in CX! And as the tradition dictates, this first program has to be printing *Hello, world!* to the terminal. You can find this program in Listing 1.3.

```
package main

func main () {
    str.print("Hello, world!")
}
```

Listing 1.3: "Hello, world!" Example

We can see the essential parts of a CX program in the aforementioned program. Every CX program has to be organized in packages (you can learn more about them in Chapter 6), and, specifically, every CX program must declare a *main* package. Additionally, in this *main* package, you must declare a *main* function, which will work as your program's *entry point*. The entry point of any program is the function, subroutine or instruction that will be run first, and which will tell the operating system how to continue with the program's execution.

After writing the program using your favorite text editor, save it to your computer using the name *hello-world.cx*. You can then run it by using either **cx hello-world.cx** or **cx hello-world.cx -i**. After executing either instruction, you should see the text *Hello, world!* printed to your terminal.

In case you're curious about the *-i* flag, it instructs CX to interpret the program, instead of compiling and then running it. You can learn more about this in Chapter 9. Also, there's actually a third way of running your program: **cx hello-world.cx -r**, but we'll learn more about it in Chapter 15, and it's related to the next Section.

1.4 Introduction to the REPL

A Read-Eval-Print Loop (REPL) is a terminal tool for interactive programming. The programmer can enter an expression, statement or declaration, and they will be evaluated instantaneously. REPLs are usually found in dynamically typed languages and interpreted languages, but this is not a rule. For example, Go is a statically typed language and there's a REPL for it called *gore*. As another exception, Common Lisp has a REPL, and Common Lisp is both an interpreted and compiled language. You can have a look at a REPL session in Listing 1.4.

```
1   CX 0.5.2
2   More information about CX is available at http://cx.skycoin.net/ and https://github.com/skycoin/c
3
4   :func main {...
5       * str.print("Hello, world!")
6
7   :func main {...
8       * :step 1
9   Hello, world!
10
11  :func main {...
12      *
```

Listing 1.4: REPL Session Example

We can see that this REPL session example is another way of creating a *Hello, world!* program in CX. The first thing to explain in a REPL session is that the asterisk or multiplication sign (*) is telling the programmer that CX is awaiting for an instruction to be entered. This is called a *REPL prompt* At **line 5**, we decide to enter an expression: **str.print("Hello, world!")**. But where does this expression go? How does CX know what is the entry point in a REPL session? To answer this question, we need to look at **line 4**. This line is telling us that we're currently inside function *main*, and that any expression that we write is going to be added to that function. This means that the entry point of a program written using the REPL is still the *main* function.

Now, if we want to run the program, we need to use the **:step** meta-command, which is explained further in Chapter 15. At **line 8** we are telling CX to advance the program by 1 instruction, which results in executing the **str.print("Hello, world!")** expression and prints the message to the terminal.

Something that you might have noted is that we writing *str* in front of *print*. This is explained in the next Section.

1.5 Strict Typing System

One of the features of CX is its strict typing system. Although the language provides some type-generalized functions, such as $len()$, there is usually a type-specific function for achieving the task. For example, in Listing 1.5, we can see that we can print a string either by using **str.print** or **print**.

```
package main

func main () {
    str.print("type-specific function")
    print("type-generalized function")
    i32.print(i32.add(10, 10))
    i32.print(10 + 10)
}
```

Listing 1.5: Type-Specific Functions

Another kind of type-generalized functions are the infix arithmetic symbols, e.g., **+**, *****, etc. The parser will infer the type of its arguments and translate the arithmetic statement to an expression that uses a type-specific function.

The objective of having a strict typing system like this is to promote *safety*. If the programmer misinterprets data in a program and, for example, tries to send an **i32** value to **str.print()**, this error can be caught early at compile-time instead of being caught at run-time.

2. CX Programs Representation

When you create a CX program and run it with the **cx** command, the first thing that happens is that the code gets parsed. Every statement, declaration and expression in your code is translated to a series of *adders*, *removers*, *selectors*, *getters* and *makers* (these are covered in Chapter 14). The trans-compiled version of a CX program is a series of these instructions that generate a structure that holds all the necessary information for the CX runtime to execute the program. It is worth to note that both interpreted and compiled versions of CX can read the same structure, so CX can have some parts of its programs compiled while other parts run in an interpreted way. The programmer can decide to compile certain functions that need to be fast, while having other functions to be interpreted, so they can be modified interactively by the user or the affordance system, for example.

The structure that represents a CX program is generated by a parser, which reads the code that you, the programmer, has written using your favorite text editor. This structure can be considered as the program's Abstract Syntax Tree (AST). CX's REPL (introduced in Section 1.4) has a meta-command that prints the AST of a program. This meta-command can be called by writing **:dp** or **:debugProgram** in the REPL, and will print something similar to Listing 2.2.

If you want to try it out, you can save the program in Listing 2.1 to a file called **ast-example.cx**, and load it to a REPL by executing the command **cx ast-example.cx -r**. Then in the REPL prompt, just enter the meta-command **:dp**, and it should print the AST shown in Listing 2.2.

```
package main

var global i32

func main () {
    var foo i32
    foo = 5

    str.print("Hello World!")
    i32.print(55)
    i32.print(i32.add(global, 10))
}
```

Listing 2.1: Abstract Syntax Tree Example - Code

```
1   Program
2   0.- Package: main
3       Globals
4           0.- Global: glblVariable i32
5       Functions
6           0.- Function: main () ()
7               0.- Expression: foo = identity(5 i32)
8               1.- Expression: str.print("Hello World!" str)
9               2.- Expression: i32.print(55 i32)
10              3.- Expression: lcl_0 = i32.add(global i32, 10 i32)
11              4.- Expression: i32.print(lcl_0 i32)
12          1.- Function: *init () ()
```

Listing 2.2: Abstract Syntax Tree Example

Let's go line by line of Listing 2.2. **Line 1** is first telling us that we are showing the AST of a program. **Line 2** then tells us that what follows are the contents of a package, which is named *main*. We can then see that all the global variables declared in the package are going to be printed after **Line 3**, which in this case is only one. Then we are presented with the last part of the package: the functions. The first function is our *main* function, which is declared to not have any input parameters nor output parameters, as seen at **Line 6**.

Before continuing with the analysis of the *main* function, let's briefly discuss that **init* function at **Line 12**. This function is actually the first function to be called in a CX program. Yeah, we lied to you, *main* is not the one called first. This function initializes all the global variables in your program, and in future versions of CX you'll be able to put other expressions you wish to run first, before your program starts (this behavior is present in languages like Go).

Now, we can see something strange happening on *main*'s list of expressions: there is a function call that we never wrote in our original CX source code (*identity*), and we can see a variable that we never declared (*lcl_0*). The *identity* operator is used when we want to "make a copy" of a value to a variable, and the variables called *lcl_N*, where *N* is an integer, are used as temporary variables that hold intermediary calculations in nested function calls. There are other weird things that happen when parsing a CX program, which we will see in later Chapters when dealing with programs' ASTs, but for now it's enough for you to understand that there is not necessarily a one-to-one relationship between your CX source code elements and the resulting AST. Actually, in more complex programs the compiler will heavily modify the resulting AST in order to optimize your code. Nevertheless, there is an important point that should be understood before continuing with the rest of the book, and this is discussed in the next Section.

2.1 Everything in a Function is an Expression

Everything in a function is an expression! This is an approach adopted from functional languages. For example, your *if/else* is transformed to a series of expressions, something that doesn't happen in imperative languages like C. Why is this important to notice? Well, you could have an *if/else* statement returning a value. In the CXGO implementation this doesn't happen, as we try to mimic as much as possible the behavior of Go. Nevertheless, it is important to take into account if you decide to create your own CX implementation. You could, for example, implement a CX-based language where the code in Listing 2.3 is valid, and it is allowed by the CX specification.

```
val = if 5 > 4 then 10 else 20
print val        // This will print 10!
```

Listing 2.3: Example of if/else Statement as an Expression

In some Chapters 5 we will see how CX transforms all the control flow mechanisms to a series of *jmp*s, where *jmp* (from the word "jump") is just an operator that takes a number of lines of code (expressions, actually) to skip.

The reason behind this design choice is convenience: it's easier to build a program structure using this approach, and implementing some of the CX features, such as affordances, is a breeze if you only have to deal with expressions. Another example is using a genetic programming algorithm (see Chapter 13 to change a CX program's structure: you only have to add, remove, change and move around the same type of component: expressions.

2.2 Elements Redefinition

Unlike some other programming languages, CX will always allow a declaration to be re-declared. In future versions of CX, an option to print warnings if this happens will be included, but in the meantime the language will not complain about this. If you want to see it by yourself, save the code in Listing 2.4 to a file and execute it using **cx**.

```
package main

func main () {
    str.print("Hello!")
}

func main () {
    str.print("Bye!")
}
```

Listing 2.4: Example Function Redefinition

Re-declarations allow CX to be an interactive programming language. If you notice an error in one of your functions, you can simply change that function in your source code, re-evaluate the function, and the program structure will be changed accordingly. If you have thousands of objects of certain struct type, and you'd like to change that object's definition to include another field, you can stop your CX program, redeclare the type, and every object of that type will be updated to include that field.

3. Data Structures

Although you could use CX as a calculator and work with literal numbers all the time, it would be a waste of power. In order to create more robust applications, you need to work with more complex types of data, such as arrays and structures. Nevertheless, before learning about these complex data structures, we need to review the different primitive types of data that CX offers at the moment.

3.1 Primitive Types

All data handled by a computer is stored as 0s and 1s[1], which are called bits. Eight of these bits comprise a byte, and a byte can represent up to 256 or 2^8 different values. Although 256 values are enough to solve a wide variety of problems already, you can always use more bytes to hold more values. For example, the traditional approach to represent an integer is to use 4 bytes, and this is why they are sometimes called 32-bit integers. For many applications, using 32-bit integers are more than enough, as these integers can hold up to 4,294,967,296 or 2^{32} different values. But if you happen to need more than that, another common type of integer is the one that uses 8 bytes to represent its value, which can hold up to 2^{64} different values (the actual number is so big that it's not even going to make sense to you if you see it printed in here).

You may or may not be wondering – depending on your curiosity and your professional background – how many bytes do you need to represent real numbers, e.g., 3.14159 or 2.41? In computer science parlance, real numbers are called floating-point numbers, and similarly to integers, floating-point numbers also require either 4 bytes or 8 bytes[2], depending on the precision you want to work with (the number of digits after the decimal point). We could discuss how we can make a computer interpret these bytes as either an integer or a floating-point number, but that's out of the scope of this book. The true objective behind this explanation is to make you realize how a *type* in a computer program is just a bunch of bytes being interpreted in a particular way.

[1]There are actually computers that use ternary logic instead of binary logic, and instead of bits you have trits, but the vast majority of the computers use binary logic.

[2]This is the common convention. There's nothing stopping you from using an arbitrary number of bytes to represent a floating-point number.

CX at the moment provides the following primitive types: *byte*, *bool*, *str*, *i32*, *i64*, *f32*, *f64*. All of the integer and floating-point number types are signed, which means that half of the possible values that they can represent are used to represent negative numbers. For example, a *byte* type in CX is able to represent any integer number from -128 to 127, for a total of 256 different values. In the future other primitive types will be incorporated, such as i16 (16-bit integer) and ui64 (unsigned 64-bit integer).

But this doesn't mean that you are limited to only those types. They are called primitive types because other more complex types are derived from them. These complex types are reviewed in the following Sections of this Chapter.

3.2 Variables

Variables have been used in code examples in previous Chapters already, but they have not been formally introduced. As was mentioned at the beginning of this Chapter, you could create programs where you only use literal numbers, but you'd be extremely limited on what you can create. Variables are one of those features that are very easy to understand and use, and yet, they greatly expand your development capabilities. You can see how you can declare variables of the different primitive types in CX in Listing 3.1.

```
package main

func main () {
    var optionCode byte
    var isAlive bool
    var name str
    var number i32
    var bigNumber i64
    var area f32
    var epsilon f64

    name = "John Cole"
    number = 14
}
```

Listing 3.1: Variable Declaration

As you can see, you can tell CX that you're going to declare a variable by using the keyword **var**, followed by the name of the variable, and finally the type that you desire that variable to have. If you want to assign a value to that variable, you just write the name of the variable, then the equal symbol (**=**) followed by the desired value.

It is interesting to note that variables are not actually needed in order to create a program, but most – if not all – of the enterprise-level programming languages provide something similar to the concept of variable. If you are curious about this, you can check some purely functional programming languages like Haskell, and also learn about lambda calculus.

3.3 Arrays

If you have to create a program where you have to store three telephone numbers, you could just create three different variables to hold each of them. But what if you had to store thousands of telephone numbers? Using variables to accomplish that task would be inefficient. The answer to this problem is to use arrays.

Arrays are fixed length collections of elements of the same type. To store or access an element in an array, you just need the name of the array and an index where you want to store the value to. To declare an array you have to put square brackets before the type in a variable declaration, and the number of elements that you want the array to hold must be inside the brackets. You can see an example of an array of three 32-bit integers shown in Listing 5.8.

```
1   package main
2
3   func main () () {
4       var foo [3]i32
5       foo[0] = 10
6       foo[1] = 20
7       foo[2] = 30
8
9       i32.print(foo[2])
10  }
```

Listing 3.2: Array Example

At **Line 4** we can see the array declaration, at **Lines 5**, **6** and **7** the array gets initialized, and finally at *Line* 9 we print the last element of the array, as arrays are zero-indexed in CX.

If you are curious enough (if you're already a programmer, it doesn't count), you could be asking yourself: can you have arrays of arrays? The answer is: yes! You only need to put the extra pair of brackets you need until you achieve the number of dimensions you want. An example of multi-dimensional arrays is shown in Listing 3.3.

```
package main

func main () {
    var foo [3][3]i32

    foo[1][2] = 40

    i32.print(foo[1][2])
}
```

Listing 3.3: Multi-dimensional Arrays

Before continuing to slices, it's worth mentioning the existence of **len**. **len** is a type-generalized function that accepts an array as its first and only input argument, and returns a 32-bit integer that represents the number of elements that that array is capable of holding. This function is especially useful when using arrays in combination with the **for** loop, which will be covered in Chapter 5. An example of **len**'s usage can be seen in Listing 3.4. Please note that there are type-specific versions of **len** for each of the primitive types.

```
package main

```

```
func main () {
    var foo [10]i32

    i32.print(len(foo))
}
```

Listing 3.4: Printing Array Length

Please be careful with the sizes you choose for your arrays. If you create an array larger than 2^{32}, you'll get an error because 2^{32} is the maximum array size or because you could exceed the maximum memory allocated to CX by your operating system. Also, if you are working with very large arrays, you'll most likely want to create a pointer to it to send the array to the heap memory. CX passes its arrays by value to other functions, which means that if you send a very big array to a function as its argument, you'll be creating a copy of it to be sent, which will be a very slow and memory consuming operation. You'll learn more about functions in Chapter 4 and about pointers in Chapter 7.

3.4 Slices

Under the hood, slices are just arrays. This means that a slice has the same performance in read/write operations as an array. The advantage of using slices over arrays is that slices are incremented in capacity automatically if it ever exceeds it. However, this can also be considered a disadvantage. A slice in CX starts with a capacity of 32 elements. If this limit is reached, CX creates a copy of that slice, but with an increased capacity of $2x$ its previous limit, which is 64 in its second iteration. As you can see, most of the time a slice will be wasting memory, and time whenever CX creates a copy of it in order to increase its limit.

It must be noted that capacity is not the same as size or length. Capacity represents the reserved memory space for a slice, while size represents the actual number of slots in a slice that are being used. You can understand better the difference if you run the code in Listing 3.5. Although any slice will start with 32 slots reserved in memory, e.g., $32 * 4$ bytes for a **[]i32** slice, this doesn't mean that all of those slots have an actual value in there. Capacity is a concept related to performance rather than to practicality.

```
package main

func main () {
    var slice []i32
    slice = append(slice, 1)
    slice = append(slice, 2)

    i32.print(len(slice)) // prints 2, not 32
}
```

Listing 3.5: Difference Between Capacity and Size

There are three native functions that are specifically designed to work with slices: **make** creates a slice of a type and size that you specify, initializing the elements to the specified type's *nil* representation, e.g., *0* for an i32 and *""* or an empty string for an *str*; **append** takes a slice and an element of the type of that slice, and puts it at the end of the slice; and lastly, **copy** creates a copy of each of the elements of a slice, and puts each of the elements, in order, to the second slice until every element has been copied to it or until the capacity of the second slice runs out.

```
1  package main
2
3  func main () {
4      var slice1 []i32
5      var slice2 []i32
6
7      slice1 = make("[]i32", 32)
8      slice1 = append(slice1, 1)
9
10     slice2 = make("[]i32", 32)
11
12     copy(slice2, slice1)
13 }
```

Listing 3.6: Slice-specific Native Functions

Listing 3.6 shows the declaration of two slices of type **i32** at **Lines 4** and **5**. The first slice then gets initialized using the **make** function, which creates a slice of size 32 in this case. This means that **slice1** now has a size of 32 elements and a capacity of 32 elements too. At **Line 8**, we append a 1 to **slice1**, which makes the slice have now a size of 33 and a capacity of 64. After initializing **slice2** at **Line 10**, we **copy** the contents of **slice1** to **slice2**. What do you think that are the elements of **slice2** now?

As a final note, slices are always allocated in the heap in CX due to their scalability nature. It would be disastrous to have a slice grow in the stack, as it would make programs run very slow – CX would need to juggle with the objects in the stack, making copies and moving them to different positions. If slices are allocated in the heap, we can delegate all of these operations to CX's garbage collector, and keep the stack clean. This behavior will slightly change in the future, though. If CX's compiler can detect that a slice is never going to grow during a function call, we can then flag that slice to be put in the stack for better performance. For more information about CX's heap and stack, you can read Chapters 7 and 10.

3.5 Structures

Structures allow the programmer to create more complex types. For example, you may want to create a type *Person* where you can store a name and an age. This means that we want a mix of an **i32** and a **str**. A structure that solves this problem is presented in Listing 3.7

```
package main

type Person struct {
    name str
    age i32
}

func main () {
 9      var p1 Person
10      var p2 Person
11
12      p1.name = "John"
13      p1.age = 22
14
15      p2 = Person{
16          name: "Gabrielle",
17          age: 21
18      }
19
20      str.print(p1.name)
21      i32.print(p1.age)
22
23      str.print(p2.name)
24      i32.print(p2.age)
25  }
```

Listing 3.7: Type *Person* using Structures

The syntax for declaring a new structure or type is shown at **Line 3**, and **Lines 4** and **5** show the structure's *fields*. The fields of a structure are the components that shape the type being defined by a structure. In order to use your new **Person** type, we first need to declare and initialize variables that use this type. This can be seen at **Lines 9-13**. **Lines 12** and **13** show that we can initialize the struct's fields one by one, by using a *dot notation*, while **Lines 15-18** show a different way of initialization: the struct literal. A struct literal is created by writing the name of the type we want to initialize, followed by the name of the struct fields' names and their values separated by a colon. Each of these field-value pairs need to be separated by a comma.

Both of these initialization approaches has its advantages. The *dot notation* has the advantage of versatility: you can initialize different fields at different points in a program. For example, you can initialize one field before a loop, and another field after that loop. On the other hand, the *struct literal* approach has the advantages of readability and that it can be used as a function call's argument directly. For example, you can send a **Person** struct instance to a function call this way: `PrintName(Person {name: "John"})`.

3.6 Scope of a Variable

The type and name are two of the properties of a variable. There is one property that we haven't mentioned so far: *scope*. The scope of a variable dictates where a variable can be seen. A *local* variable is only accessible in the function where it was declared, while a *global* variable can be accessed by any function of a package.

```
package main

var global i32

func foo () {
    i32.print(global)
    // i32.print(local) // this will raise an error if uncommented
}
```

```

func main () {
    var local i32

    local = 10
    global = 15

    i32.print(global)
    i32.print(local)
}
```

Listing 3.8: Usage of Local and Global Variables

If you want to create a global variable, you only have to declare it outside any function declaration. If you want a local variable, declare it inside the function you want it to have access to. Listing 3.8 shows an example that declares a global variable that is accessed to by two functions: **main** and **foo**, and a local variable that is only accessible by the **main** function.

As a last note, global variables can also be accessed by other packages that import the package containing said variable. You'll learn more about packages in Chapter 6.

4. Functions

Unless you are learning an esoteric programming language, chances are that that language is going to have some sort of subroutine mechanism. A subroutine is a named group of expressions and statements that can be executed by using only its name. This allows a programmer to avoid writing that group of expressions and statements again and again, every time they are needed. In CX, subroutines are called functions, because they behave similarly to how mathematical functions behave.

In CX, a function can receive a fixed number of input parameters, and like in Go, it can return a fixed number of output parameters. These parameters must be of a specific type, either a primitive type or a complex type. At the moment, both input and output parameters must have a name associated to them, but this will change in the future and anonymous output parameters will be possible. Parameters are a very powerful feature, because they allow us to have a function behave differently depending on what data we send to it. Listing 4.1 shows how we can create a function that calculates the area of a circle, and another function that calculates the perimeter of a circle.

```
package main

var PI f32 = 3.14159

func circleArea (radius f32) (area f32) {
    area = f32.mul(f32.mul(radius, radius), PI)
}

func circlePerimeter (radius f32) (perimeter f32) {
    perimeter = f32.mul(f32.mul(2.0, radius), PI)
}

func main () () {
    var area f32
    area = circleArea(2.0)
    f32.print(area)
    f32.print(circlePerimeter(5.0))
}
```

Listing 4.1: Determining Area and Perimeter of a Circle using Functions

If you needed to calculate the area of 20 circles, you'd only need to call **circleArea** 20 times, instead of having to write `f32.mul(f32.mul(radius, radius), PI)` 20 times (although you'd probably be using a **for** loop instead; see Chapter 5).

4.1 Lexical Scoping

Variables in CX are lexically scoped, which means that they are only accessible in the function where they were declared. This was reviewed in Chapter 3, but there's a situation that was not covered, and that is more appropriated to be covered in this Chapter.

```
1   package main
2
3   func foo () {
4       i32.print(x)
5   }
6
7   func main () {
8       var x i32
9       x = 15
10      foo()
11  }
```

Listing 4.2: Lexical Scoping

If CX was dynamically scoped, the code shown in Listing 4.2 would print 15, because the call to **foo** at **Line 10** would capture the value of the variable **x** declared in **main**. Instead, it will raise an error because **Line 4** is trying to access a variable that has not been previously declared.

4.2 Side Effects

CX is an imperative language and not purely functional, unlike, for example, Haskell. This means that functions can have side effects, i.e., they can change the state of objects outside of the function's scope. Side effects include actions like modifying the value of a global variable or, if you are a purist, even printing text to a terminal. Functional programming has some advantages, like easier debugging, but CX alleviates this by providing debugging tools like its REPL (see Chapter 15).

4.3 Methods

Methods are a special type of functions that can be associated to user-defined types. Although methods are not strictly necessary, as their functionality can be replaced by normal functions, they provide some useful advantages. The first advantage is that different methods can have the same name as long as they are associated to different types. This can help the programmer start thinking only about the action that needs to be performed, instead of thinking about a name for that specific structure. For example, instead of having to call functions named **printPlayerName()** and **printRefereeName()**, you can simply call the structure instance's method name **printName()**. This situation is shown in Listing 4.3.

```
package main

type Player struct {
    name str
}

type Referee struct {
    name str
}

func (p Player) printName () {
    str.print("Player information")
    str.print(p.name)
}

func (r Referee) printName () {
    str.print("Referee information")
    str.print(r.name)
}

func main () {
    var p Player
    p.name = "Michael"

    var r Referee
    r.name = "Edward"

    p.printName()
    r.printName()
}
```

Listing 4.3: Methods Example

Another advantage of methods is that they promote safety, as they are associated to a particular user-defined type. If a method is not defined for a type, this error will be caught at compile-time.

5. Control Flow

A program in CX is executed from top to bottom, one expression at a time. If you want a group of expressions to not be executed, or executed only if certain condition is true, or executed a number of times, you'll need control flow statements. In CX, you have access to these control flow statements: **if** and **if/else**, **for** loop, **goto**, and **return**.

In the following Section you'll review the **jmp** function, and you'll see that in CX, every control flow statement is transformed to a series of **jmp**s.

5.1 jmp and goto

Although **jmp** exists in the CX native function repertoire, it can't really be used. If you want to write a function call to **jmp**, you'll need to send it a boolean argument that represents a predicate. In case of the predicate evaluating to *true*, **jmp** will make the program skip or go back a certain number of instructions. If the predicate evaluates to *false*, nothing will happen. The tricky part here is: how do we specify how many instructions the program will move? You can't. **jmp** is designed to be used exclusively by the parser, or any program that is in charge of constructing a CX program structure. As the default number of expressions to be moved by **jmp** in any case is 0, **jmp** will do nothing if used directly by a programmer. In future versions of CX, the compiler will raise an error if you try to use **jmp** directly, so, aside from being totally useless for a programmer, it's a pretty bad idea to include it in your programs at the moment, as it will make your programs incompatible with later versions of CX.

You could now be arguing that it was bad idea that you learned about **jmp**, as you now will be tempted to include meaningless function calls to it all around your code – we hope you don't do this. We want to accomplish two things by introducing **jmp** to you: 1) you'll understand CX ASTs better, and 2) in case you want to build your own CX (see Chapter 14), you need to know that **jmp** can be used to create control-flow statements. In the following Sections, each of the examples that depict the use of the different control-flow statements will be accompanied by their corresponding AST in a different Listing, where it can be seen that all of them are translated to a series of **jmp**s.

Nevertheless, if you want to have access to some simple instruction jumping mechanism, you can use **goto**. **goto** will always perform an instruction jumping if encountered, and the number of instructions that the program will be jumped to will be determined by a *label*. Listing 5.1 shows an example where *goto* is used to jump directly to a **print** expression, and Listing **??** shows its AST.

```
package main

func main () {
    goto label
label1:
    str.print("this should never be reached")
label2:
    str.print("this should be printed")
}
```

Listing 5.1: Using *goto* for Control Flow

It is important to note that *labels* are only used by *goto* statements and affordances (see Chapter 11). If a label is encountered by the CX runtime, it will be ignored. Actually, if you check the AST of the program in Listing 5.1 you will see that labels don't appear: the parser read the labels and transformed them to the number of expressions required by a **jmp** to make the CX program arrive at that expression. In the case of the code shown in Listing 5.1, the number of instructions to be skipped by **goto label1** is $+1$ (it could be a negative number if it had to make a jump to an early instruction).

5.2 if and if/else

There will be plenty of situations where you will need to execute a number of expressions or statements only if certain condition is true. For example, you only want to allow a website user to login using a username if the password that they provide is the one that matches the given username. In order to handle this kind of situations, you can use the **if** statement.

Listing 5.2 shows some examples of how the **if** statement can be used. The first case, starting at **Line 6** takes **false** as a predicate. As **false** is not variable–it will always evaluate to the boolean false value–the lines of code between the curly braces will never execute. Similarly, in the second case, starting at **Line 12**, as the predicate is **true** the **str.print** expression will always be executed. In the last case, we use the *greater than* relational operator **i32.gt** to decide if the enclosed expression will be executed or not. In this case, as 5 is greater than 3, the **str.print** expression is executed. Listing 5.3 shows its AST.

```
 1  package main
 2
 3  func main () {
 4      if false {
 5          var err i32
 6          err = i32.div(50, 0)
 7          str.print("This will never be printed")
 8      }
 9
10      if true {
11          str.print("This will always print")
```

```
    }

    if i32.gt(5, 3) {
        str.print("5 is greater than 3")
    }
}
```

Listing 5.2: Using If for Control Flow

```
Program
0.- Package: main
    Imports
        0.- Import: main
    Functions
        0.- Function: main () ()
            0.- Expression: jmp(false bool)
            1.- Expression: err = i32.div(50 i32, 0 i32)
            2.- Expression: str.print("This will never be printed" str)
            3.- Expression: jmp(true bool)
            4.- Expression: jmp(true bool)
            5.- Expression: str.print("This will always print" str)
            6.- Expression: jmp(true bool)
            7.- Expression: lcl_0 = i32.gt(5 i32, 3 i32)
            8.- Expression: jmp(lcl_0 bool)
            9.- Expression: str.print("5 is greater than 3" str)
            10.- Expression: jmp(true bool)
        1.- Function: *init () ()
```

Listing 5.3: Listing 5.2's Abstract Syntax Tree

If you want to execute certain block of code if the predicate is true, and a different block of code if the predicate is false, you can extend the **if** statement to its **if/else** form. Listing 5.4 shows an example of how to use **if/else**, and the AST for this example is shown in Listing 5.5.

```
package main

func main () {
    var out i32

    if i32.lteq(50, 5) {
        out = 100
    } else {
        out = 200
    }

    i32.print(out)
}
```

Listing 5.4: Using If/Else for Control Flow

```
Program
0.– Package: main
    Imports
        0.– Import: main
    Functions
        0.– Function: main () ()
            0.– Expression: jmp(false bool)
            1.– Expression: err = i32.div(50 i32, 0 i32)
            2.– Expression: str.print("This will never be printed" str)
            3.– Expression: jmp(true bool)
            4.– Expression: jmp(true bool)
            5.– Expression: str.print("This will always print" str)
            6.– Expression: jmp(true bool)
            7.– Expression: lcl_0 = i32.gt(5 i32, 3 i32)
            8.– Expression: jmp(lcl_0 bool)
            9.– Expression: str.print("5 is greater than 3" str)
            10.– Expression: jmp(true bool)
        1.– Function: *init () ()
```

Listing 5.5: Listing 5.4's Abstract Syntax Tree

The syntax of **if** and **if/else** is similar to Go's syntax: you don't need to enclose the predicate in parentheses, unlike other languages like C, and the curly braces need to start after the condition, or the parser will complain. The reason behind this is that in order to not be required to write a semicolon after each expression, some tweaks needed to be implemented (just like in Go). As a consequence of these tweaks, you are required to start your curly braces after the predicate. This has the disadvantage of losing a bit of flexibility in how you are allowed to write your code, but it's also an advantage because the code now looks cleaner and more standardized.

5.3 for Loop

Another very useful control-flow statement is the **for** loop. This statement allows us to repeat the execution of a block of code until a predicate is false. In some languages the **for** loop is strict on its syntax, and only allows the traditional

Most C-like languages only allow the traditional *initialization*, *condition* and *increment* syntax for the **for** loop, but in CX you can use it similarly to how you would use a **while** loop in other languages. Listing 5.6 reviews this syntax, and Listing 5.7 shows its AST.

```
package main

func main () () {
    for true {
        str.print("Infinite loop!")
    }
}
```

Listing 5.6: Infinite Loop Example

```
Program
```

```
0.- Package: main
    Imports
        0.- Import: main
    Functions
        0.- Function: main () ()
            0.- Expression: jmp(true bool)
            1.- Expression: str.print("Infinite loop!" str)
            2.- Expression: jmp(true bool)
        1.- Function: *init () ()
```

Listing 5.7: Listing 5.6's Abstract Syntax Tree

First of all, don't run the code above, as it's an infinite loop. Although it's essential to know how to create an infinite loop, this infinite loop is particularly useless–it only prints "Infinite loop!" to the terminal. This example illustrates how you can use a single argument as the predicate of a **for** loop, as long as it evaluates to a boolean value.

```
1   package main
2
3   func main () () {
4       var foo [5]i32
5       foo[0] = 10
6       foo[1] = 20
7       foo[2] = 30
8       foo[3] = 40
9       foo[4] = 50
10
11      var c i32
12      for c = 0; c < 5; c++ {
13          i32.print(foo[c])
14      }
15  }
```

Listing 5.8: Traditional Syntax of For Loop

```
Program
0.- Package: main
    Imports
        0.- Import: main
    Functions
        0.- Function: main () ()
            0.- Expression: jmp(true bool)
            1.- Expression: str.print("Infinite loop!" str)
            2.- Expression: jmp(true bool)
        1.- Function: *init () ()
```

Listing 5.9: Listing 5.8's Abstract Syntax Tree

The second example in Listing 5.8 shows the traditional syntax of a **for** loop, i.e., at **Line 12** we first initialize a variable, which is usually used as the counter, then we provide a predicate expression, and finally an expression that is usually used to increment the counter. Listing 5.9 shows its AST.

5.4 return

The last control-flow statement is **return**. The only purpose of **return** is to make a function stop its execution as soon as it is encountered. As it was mentioned in Chapter 4, **return** can't be used to return anonymous outputs, as they are not implemented yet. This means that you can't use **return** like this: `return 5, "five";` in a function that returns an **i32** and a **str**, in that order. The correct way is to first assign the desired values to the named outputs, and then call **return** whenever you want a function to end prematurely.

```
package main

func foo () (out1 i32, out2 str) {
    out1 = 5
    out2 = "five"

    return

    out1 = 10
    out2 = "ten"
}

func main () {
    var num i32
    var text str

    num, text = foo()
}
```

Listing 5.10: Usage of return

The code shown in Listing 5.10 demonstrates how **return** prevents the function **foo** from reassigning values to the output parameters.

```
Program
0.- Package: main
    Functions
        0.- Function: foo () (out1 i32, out2 str)
            0.- Expression: out1 = identity(5 i32)
            1.- Expression: out2 = identity( str)
            2.- Expression: jmp( bool)
            3.- Expression: out1 = identity(10 i32)
            4.- Expression: out2 = identity( str)
        1.- Function: main () ()
            0.- Expression: num, text = foo()
            1.- Expression: i32.print(num i32)
            2.- Expression: str.print(text str)
        2.- Function: *init () ()
```

Listing 5.11: Usage of return

The AST shown in Listing 5.11 demonstrates how a **jmp** is used to skip all the remaining expressions. The parser calculates the number of expressions that follow the **return** statement, and then makes the **jmp** expression always skip all of them.

6. Packages

If your project grows too big, you'll need a better way to organize your code. A solution to this would be to separate your functions into different files, but this is not a good solution as you could still encounter problems if you end up naming another function with the same name. To make things worse, as it was mentioned in Chapter 2, CX won't complain if you redefine a function or a global variable somewhere else in your code. The solution to this problem is modularization.

Modularization is a technique where you isolate groups of declarations in your source code under a common module name. This module name works as a "last name" for all the declarations grouped in that module, and gives every declaration a unique "full name" across all the source code files.

Each programming language has its own way of calling these isolated units of declarations. For example, in C# they are called *namespaces* and in Python they are called *modules*. In CX, we call these modules *packages*.

In Listing 6.1 we can see a program that got organized into three different packages: **foo**, **bar** and **main**. Package **foo** declares 3 definitions: a structure named **Point**, a global variable named **num**, and a function named **bar**. Package **bar** imports package **foo** and declares a single definition: a function named **returnPoint**. As you can see, importing a package is handled by the **import** keyword, followed by the name of the package that you want to import. Something interesting in the function **returnPoint** is that it is using definitions defined in package **foo**. As we can see, in order to access something from an imported package, you first need to write that package's name, then a period followed by the name of the definition of interest.

```
package foo

type Point struct {
    x i32
    y i32
}

var num i32 = 15

```

```
func bar () {
    str.print("From foo package")
}

package bar
import "foo"

func returnPoint () (resPoint foo.Point) {
    var resPoint foo.Point
    resPoint = foo.Point{x: 10, y: 20}
}

package main
import "foo"
import "bar"

func main () {
    var aPoint foo.Point
    aPoint.x = 30
    aPoint.y = 70
    aPoint = bar.returnPoint()

    var check i32
    check = 10
    i32.print(check)

    i32.print(aPoint.x)
    i32.print(aPoint.y)

    var foo1 foo.Point
    foo1.x = 20
    foo1.y = 30
    i32.print(foo1.x)
    i32.print(foo1.y)

    i32.print(foo.num)
    foo.bar()
    i32.print(foo.num)
}
```

Listing 6.1: Importing Packages Example

The different packages and their definitions can be placed altogether in a single file (unlike in other languages, where you have to use a file or a directory for a single package or module), but this can become unpractical sooner than later, so it is advised that you use a single package per directory, as in the programming language Go. Also, CX projects behave similarly to Go projects, where you have to place your files in a directory in a CX workspace. CX workspaces are described in Section 6.1.

6.1 CX Workspaces

Dividing your code into different files is essential as your projects grow bigger. CX takes an approach similar to Go for handling projects: a package in a directory can split into a number of files, but you can't use more than one package declaration in these files inside this directory. In other words, a directory represents a package. An exception to this rule would be declaring several packages in a single file. The purpose of this exception is to allow the programmer to test ideas quickly without them being required to create packages in different directories and another directory for their application (which will contain the **main** package).

Listings 6.2 and 6.3 show the code for two packages: **math** and **main**. The **math** code needs to be in a file named whatever you want, inside a directory that you should name the same as your package. It's not mandatory to do so, but the consistency helps other programmers that are reading your code.

```
package math

func double () (out i32) {
    out = i32.add(5, 2)
}
```

Listing 6.2: Package to be Imported

```
package main
import "math"

func main () {
    str.print("hi")
    var foo i32
    foo = math.double()
    i32.print(foo)
}
```

Listing 6.3: Main Package

```
Program
0.- Package: math
    Functions
        0.- Function: double () (out i32)
            0.- Expression: out = i32.add(5 i32, 2 i32)
1.- Package: main
    Imports
        0.- Import: math
    Functions
        0.- Function: main () ()
            0.- Expression: str.print("hi" str)
            1.- Expression: foo = double()
            2.- Expression: i32.print(foo i32)
        1.- Function: *init () ()
```

Listing 6.4: Resulting Abstract Syntax Tree

The AST for the full program can be seen in Listing 6.4 . As you can see, each package lists the packages that were imported. The names of these packages are the names that were given to the **package** declaration. In other words, if you name your package's directory **foo** but you declare your package in your code as **bar**, CX will handle all the calls to this package through the latter instead of the former name.

But where exactly do you have to put all this code? CX, as mentioned before, follows the same philosophy as Go: you work in workspaces. A workspace is a directory dedicated solely to manage your projects, dependencies, executables and shared libraries. A workspace can be any directory in your file system that contains these three directories: *bin*, *src* and *pkg*. *bin* is used to store the binary files of your projects and/or libraries; *src* is used to store the source code of your projects and their dependencies; and *pkg* stores object files that are used to create the executables stored in *bin*.

After installing CX for the first time, the installation script will create a default workspace for you located at $HOME/cx or %USERPROFILE%\cx, depending on what operating system you are using: unix-based systems or Windows, respectively. If you want to override this, you can set the environment variable $CXPATH or %CXPATH% to a file system path where you want your CX workspace to reside.

A way to get started quickly with a new CX project is to use the CX executable to create one for you. You only have to write **cx -n** or **cx –new** and a series of questions about your new project will be asked to you that will be used to initialize it.

Just like in Go, a project without **main** package or function is considered a library to be imported by other packages or applications, while a project with a **main** package and function is considered an application that is going to be calling the other projects in the **src/** directory of your workspace as libraries.

If you're working in a single file, you can just import your packages using the name you used in the package declaration statement, like in Listing 6.1. If you are dealing with packages from different directories in your workspace, then you need to make sure that you write the full path to the desired package. For example, if the package you want to import is located in `$CXPATH/src/math_stuff/stats` or `%CXPATH%\src\math_stuff\stats`, you'd need to import the package like this: `import "math_stuff/stats"`. As you can see, you have to omit the `src` part because all of the libraries need to be there anyway.

7. Pointers

Programming languages that use a stack to pass values to function calls can pass the actual value or a reference to it. Passing by value means that all the bytes that represent that data structure are copied to the function call. In the case of a simple integer or a floating-point number, this isn't a big problem, because you're copying at most 8 bytes. The real problem arises when you try to pass a really big data structure, like an array or a string (which is basically an array). Copying all these bytes every time a function is called creates two problems: 1) it is slow; imagine that you have to execute a for loop that iterates *N* times, where *N* is the size of your data structure, and you have to do this every time you call that function; and 2) you are more prone to encounter a stack overflow error, as you are filling your stack with all these copies of your data structure.

A solution to the pass-by-value problem is to use pass-by-reference. In pass-by-reference, instead of copying the actual value, you send the address of the value that you want to use. A reference is just a number that represents the index where you can find the actual value in memory, and as such, a reference only needs 4 bytes, as it's just a normal 32-bit integer. This also means that creating a pointer to a 32-bit integer is useless if your purpose is to increase your program's performance (actually, using a pointer would make your program a tiny bit slower, because it needs to dereference the pointer).

```
package main

type Cell struct {
    id i32
    drawable i32
    alive bool
    aliveNext bool
    x i32
    y i32
}

func main () {
    var cells *[900]Cell
14      cells = makeCells()
15
16      for bool.not(glfw.ShouldClose("window")) {
17          draw(cells, "window", program)
18      }
19  }
```

Listing 7.1: Pointer to a Structure Instance

The code in Listing 7.1 presents a situation where using a pointer drastically improves the performance of a program. **Line 13** shows the declaration of a variable of type pointer to an array of structure instances. This variable is then used to hold the output of **makeCells**, and the for loop draws all the cells to a window. If we weren't using a pointer, we'd need to pass by value all the 900 cells, which sum a total of 16,200 bytes. In contrast, by using a pointer we're only sending 4 bytes that represent the other 16,200 bytes.

This Listing shows an excerpt of an OpenGL example present in the CX git repository (https://github.com/skycoin/cx). The example is currently located at examples/opengl/conways−game−of−life−gc.cx, but this path could change in the future. If you try to run this example using your local CX installation, you'll find out that it doesn't run, so download the full example from the CX repository.

7.1 Memory Segments

You may be wondering where are those 16,200 bytes from the example in Listing 7.1. CX handles four types of memory segments: *code*, *data*, *heap* and *stack*.

The *code* segment holds all the program's elements, like functions, expressions, packages, etc. In many programming languages this segment is "read only", i.e it can't be modified. In CX this is not the case, as the code segment can be changed through affordances and in the REPL.

The *data* segment is special because the data elements stored in there are never going to be moved or destroyed; the only thing that can change are their values. For this reason, global variables are stored in here, as they are never going to be destroyed and their addresses are constant.

In order to understand the *heap*, we first need to understand the *stack*. The *stack* holds all the local variables that are declared in functions. The first function that is called in a program is the **main** function, and this is why it's also called the *entry point*. The first bytes in a *stack* are many times going to be dedicated to the main function. Exceptions to this are having a **main** function without any variable declarations and multi-threaded programs. The next bytes in the *stack* are going to be used as the program runs, and other functions are called.

```
package main

var epsilon i32

func bar () (w i32) {
    w = 5
}

func foo (num1 i32, num2 i32) (res i32) {
    var weight i32

    weight = bar()
13
14      res = (num1 + num2) * weight * epsilon
15  }
16
17  func main () {
18      epsilon = 5
19      epsilon = foo(10, 10)
20  }
```

Listing 7.2: Pointer to a Structu

Listing 7.2 helps us to understand the different memory segments in CX. **Line 3** declares a global variable, which will be set in the *data* segment. In this particular program, the data segment is only going to be 4 bytes long, as it only needs to store one 32-bit integer. Just after compiling the program, these 4 bytes will be set to 0s, but as soon as the program is run, the very first instruction to be run at **Line 18** is an assignment to epsilon, which will modify the data segment to hold `5 0 0 0`.

As the **main** function does not declare any variables, the *stack* segment will not be used until we call **foo** at **Line 19**. Before starting the execution of this function call, CX reserves a certain amount of bytes for that call in the *stack*. This amount of bytes needs to be constant throughout a program's execution, i.e. CX knows how many bytes to allocate for any function call after compile time. In this case, **foo** needs 16 bytes, because it has two *i32* input parameters, one *i32* output parameter and one *i32* local variable declaration. Before **foo** ends its execution, it makes a function call to **bar**. This means that CX needs to keep **foo**'s bytes "alive," as the function call has not finished yet. Instead, CX needs to reserve 4 more bytes for **bar** for its *i32* output parameter. Once **bar** finishes its execution, the 4 bytes reserved for it can now be discarded, and the program's execution returns to **foo**. After **Line 14** finishes, **foo**'s execution will also finish, and the bytes reserved for it can now be discarded. Some details about this process were not mentioned on purpose, but the general idea should be clear now.

As you can see, the *stack* is always growing and shrinking, and it does this in a linear manner, i.e. you're never going to be discarding bytes in the middle or at the beginning, only the most recent reserved bytes are the ones that get discarded. This behavior avoids *fragmentation*, which is a problem when using the *heap* segment (we'll review this topic in Chapter 10).

CX does not support multi-threading yet, but it is interesting to note that multiple stacks need to be used for multi-threaded programs. Every time you create a new thread, a new stack must be assigned to that thread.

```
package main

func greetings (name str) (g str) {
    g = sprintf("Greetings, %s", name)
}

func main () {
    var name str
    name = "William"
    name = greetings(name)
    str.print(name)
}
```

Listing 7.3: Pointer to a Structu

To begin understanding the *heap* segment, we can have a look at Listing 7.3. This program creates a *str* variable, the string "William" is assigned to it, it is sent to **greetings**, and its result is re-assigned to **name** to later be printed to the terminal. You may be wondering what this program has to do with the *heap*, as no pointers are ever declared. Well, first you need to keep in mind that the *stack* needs to grow/shrink in constant "chunks" of bytes and the *data* segment never grows or shrinks. Now pay attention to **Lines 9** and **10**. First, name is holding the value "William" and then it will hold the value "Greetings, William." Do you see the problem here? If these strings were handled only by the stack, we would have a variable-sized function, which is not allowed.

Strings in CX basically behave as pointers. Whenever a string needs to be allocated, it is allocated in the *heap*, not in the *stack* or *data* segments. After its allocation, a pointer to this string is assigned in the *stack*. This way, functions that handle strings can be fixed-sized, as pointers always have a size of 4 bytes. Going back to the example in Listing 7.3, **name** is first assigned the address of the string "William" allocated in the heap, then a new string, "Greetings, William", is allocated in the call to **greetings**, and its address is returned as its output and re-assigned to **name**. This means that you can allocate whatever object you need at any point in the *heap*, in any order and wherever you want.

```
package main

type Point struct {
    x i32
    y i32
}

func CreatePoints () {
    var points [5]Point
    var ptr *[5]Point

    var c i32
    for c = 0; c < 5; c++ {
        points[c] = Point{x: c, y: c + 1}
    }

    ptr = &points
}

func main () {
    for true {
        CreatePoints()
    }
}
```

Listing 7.4: Pointer to a Structu

But allocating anything you want and wherever you want isn't problematic? Indeed, it is so problematic that in some programming languages you need to personally take care of what and when you want to allocate a new object in the *heap*, and even when you need to destroy that object. These languages are said to have "manual memory management," and perhaps the most popular language of this type is C. For example, Listing 7.4 executes an infinite loop that repeatedly calls **CreatePoints**, which creates an array of 5 **Point** instances, and allocates them in the *heap*. As you can notice, nothing else happens with the pointer to this array, **CreatePoints** simply allocates this array of **Point** instances, and then returns. Now, as we are doing this an indefinitely number of times, wouldn't this program cause a *heap* overflow eventually? Not really, CX's garbage collector will be activated each time the *heap* is full, and remove the objects that are no longer being used. The resulting dead objects could be anywhere in the *heap*, which will cause *fragmentation*, but don't worry as the garbage collector deals with this problem too. As can be noted, the *heap* is the most flexible memory segment.

The last memory segment is the *code* segment. This segment can be modified at will, unlike in other programming languages. This segment holds all the program's elements, such as functions, expressions and structure declarations. Modifying this memory segment will be discussed in Chapter 11.

8. OpenGL and GLFW with CX

In the Skycoin team we believe that a bright future exists for blockchain technologies in video game development. For this reason, one of the first libraries that was developed for CX was the OpenGL library. This Chapter presents some video game examples that should help you get started with video game development in CX. In order to use the OpenGL and GLFW libraries in your CX programs, just `import "gl"` and `import "glfw"` after a package declaration.

The current OpenGL library does not implement all of the OpenGL functions and constants, but it should implement everything in the future. The OpenGL version that the CX library targets is 2.1.

CX also provides a GLFW library that helps the programmer set up things like windows and input devices. The GLFW version targeted by the CX library is 3.2.

The examples in this Chapter are not explained thoroughly, as the purpose of this book is to explain the features of the CX programming language, not to explain how OpenGL and GLFW work.

```
package main

import "gl"
import "glfw"

var width i32 = 800
var height i32 = 600

func main () {
    glfw.Init()
    glfw.WindowHint(glfw.Resizable, glfw.False)
    glfw.WindowHint(glfw.ContextVersionMajor, 2)
    glfw.WindowHint(glfw.ContextVersionMinor, 1)

    glfw.CreateWindow("window", width, height, "Window Example")
    glfw.MakeContextCurrent("window")

    gl.Init()
    var program i32
20        program = gl.CreateProgram()
21        gl.LinkProgram(program)
22
23        for bool.not(glfw.ShouldClose("window")) {
24            gl.Clear(i32.bitor(gl.COLOR_BUFFER_BIT, gl.DEPTH_BUFFER_BIT))
25
26            gl.UseProgram(program)
27
28            glfw.PollEvents()
29            glfw.SwapBuffers("window")
30        }
31    }
```

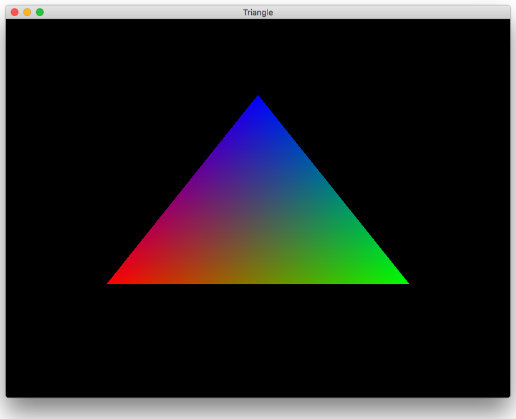
Listing 8.1: Creating a Window using OpenGL

The first step to creating a video game is to create the window where everything is going to be displayed. Listing 8.1 shows a bare-bones example that only displays an empty window. You could think that it's a lot of instructions to only accomplish a simple task such as creating a window, but it's the OpenGL way. This example can be used as a template to start a new OpenGL project in CX.

The window has a resolution of 800x600, as defined by the global variables **width** and **height**, AT **Lines 6**, respectively. The function that actually creates the window to be displayed is created at **Line 15**, and it is constantly being re-drawn in the loop that begins at **Line 23**.

```
package main

import "gl"
import "glfw"

var width i32 = 800
var height i32 = 600

func main () () {
    glfw.Init()

    glfw.CreateWindow("window", width, height, "Triangle")
    glfw.MakeContextCurrent("window")

    gl.Init()

    var program i32
    program = gl.CreateProgram()

    gl.LinkProgram(program)

    for bool.not(glfw.ShouldClose("window")) {
        gl.Clear(gl.COLOR_BUFFER_BIT)

        gl.UseProgram(program)

        gl.MatrixMode(gl.PROJECTION)
        gl.LoadIdentity()
        gl.MatrixMode(gl.MODELVIEW)

```

Figure 8.1: Triangle in OpengGL window



```
31              gl.Begin(gl.TRIANGLES)
32              gl.Color3f(1.0, 0.0, 0.0)
33              gl.Vertex3f(-0.6, -0.4, 0.0)
34              gl.Color3f(0.0, 1.0, 0.0)
35              gl.Vertex3f(0.6, -0.4, 0.0)
36              gl.Color3f(0.0, 0.0, 1.0);
37              gl.Vertex3f(0.0, 0.6, 0.0);
38              gl.End();
39
40              glfw.PollEvents()
41              glfw.SwapBuffers("window")
42          }
43      }
```

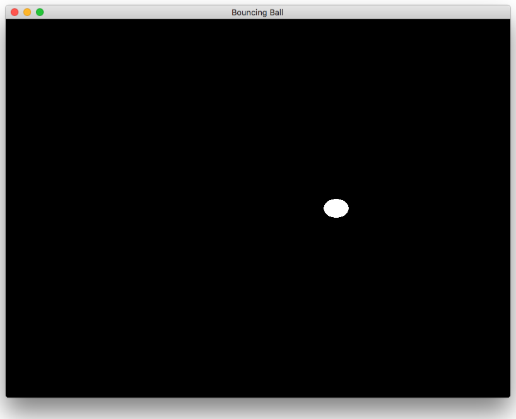
Listing 8.2: Drawing a Triangle to a Window

Now that we can create a window and display it, let's draw something on it. Listing 8.2 adds some lines of code to the code in Listing 8.1 (**Lines 27 - 38**). Functions **gl.Color3f** and **gl.Vertex3f** are used to assign a color and coordinates to a vertex for a triangle, enclosed by calls to **gl.Begin** and **gl.End**. After running the code in this Listing, you should see a window with a triangle like in the one displayed in Figure 8.1.

```
1   package main
2
3   import "gl"
4   import "glfw"
5
6   var width i32 = 800
7   var height i32 = 600
8
9   type Ball struct {
10      x f32
11      y f32
12      vx f32
```

```
    vy f32
    gravity f32
    radius f32
}

func drawBall (ball Ball) () {
    var full_angle f32
    full_angle = f32.mul(2.0, 3.141592654)
    var x f32
    var y f32

    gl.Begin(gl.POLYGON)
    gl.Color3f(1.0, 1.0, 1.0)

    var i f32
    for i = 0.0; f32.lt(i, 20.0); i = f32.add(i, 1.0) {
        x = f32.add(ball.x, f32.mul(ball.radius, f32.cos(f32.div(f32.mul(i, full_angle), 20.0))))
        y = f32.add(ball.y, f32.mul(ball.radius, f32.sin(f32.div(f32.mul(i, full_angle), 20.0))))

        gl.Vertex2f(x, y)
    }

    gl.End()
}

func main () () {
    glfw.Init()

    glfw.CreateWindow("window", width, height, "Bouncing Ball")
    glfw.MakeContextCurrent("window")

    gl.Init()
    var program i32
    program = gl.CreateProgram()
    gl.LinkProgram(program)

    var ball Ball
    ball = Ball{
        radius: 0.05,
        x: 0.0,
        y: 0.0,
        vx: 0.01,
        vy: 0.01,
        gravity: 0.01}

    for bool.not(glfw.ShouldClose("window")) {
        gl.Clear(gl.COLOR_BUFFER_BIT)

        gl.UseProgram(program)

        gl.MatrixMode(gl.PROJECTION)
        gl.LoadIdentity()
        gl.MatrixMode(gl.MODELVIEW)

        if f32.lteq(f32.add(ball.y, ball.radius), -1.0) {
            ball.vy = f32.abs(ball.vy)
```

Figure 8.2: Bouncing ball in OpengGL window



```
69            } else {
70                ball.vy = f32.sub(ball.vy, ball.gravity)
71            }
72
73            ball.x = f32.add(ball.x, ball.vx)
74            ball.y = f32.add(ball.y, ball.vy)
75
76            drawBall(ball)
77
78            glfw.PollEvents()
79            glfw.SwapBuffers("window")
80        }
81    }
```

Listing 8.3: Bouncing Ball Example

As the final example of this Chapter, Listing 8.3 presents a little more complex situation. We use a structure that will represent a ball to be drawn on screen, declared at **Line 9**. In the for loop that updates the screen (**Lines 58-80**) we update the coordinates (x and y) of the ball, and draw the ball's new position to the window. The function **drawBall** uses the coordinates of the ball structure instance as a center, and uses its radius to draw a circle using polygons, which represents the ball.

After running this last example, you should see a ball that starts at the center of the screen, and starts falling and bouncing to the right of the screen. It should display something similar to Figure 8.2.

9. Interpreted and Compiled

As has been noted in previous Chapters, CX is both an interpreted and a compiled language. But this doesn't only mean that you can run a program by interpreting it or compile it and then run it; CX goes further. CX can work with both compiled and interpreted code at the same time, just like some languages, such as Common Lisp. The reason behind this design decision is that it maximizes the number of features CX can provide. For example, a function that is constantly being constructed by affordances is far easier to be evaluated if it's purely interpreted, instead of recompile the function every time (or maybe even the whole program).

CX started as being purely interpreted, mainly because the Skycoin team was still testing some ideas on what direction the language was going to take. As the language progressed in complexity, and we wanted to test programs that were more expensive regarding computational resources, it was clear that CX needed to implement optimization techniques to the code it was generating. However, we realized that the current design had reached certain limit. The generated programs were very flexible, as many features of the language were managed by the underlaying language: Go. This flexibility allowed CX to implement affordances, an integrated genetic programming algorithm and other features in a short amount of time. Nevertheless, its speed was comparable to Martz's Ruby (no, not Ruby, Martz's Ruby is about 5 times slower than Ruby). As a consequence to this, CX had to take another direction in its design, and some core optimizations were implemented.

Nowadays CX is pretty fast, even if a plethora of optimizations still need to be implemented. At some benchmark tests CX scored a similar speed to Python, but we still need to perform more benchmarks to get a more objective conclusion. Even if the resulting speed is actually 5 times slower than Python, it's far better than before and, as stated above, many optimizations can still be implemented.

```
$ cx hello-world.cx
Hello, world!
$ cx hello-world.cx -i
Hello, world!
```

Listing 9.1: Interpreting and Compiling the same Program

You may be wondering what happened to the interpreted version of CX. It's still in use and it is faster now than before. We realized that some of the optimizations that were implemented for the compiled version can work with the CX interpreter, and it got benefited from them. It is still slow, but it retained all its flexibility. If you open a CX REPL, you'll be running the CX interpreter, and if you run a `$ cx example.cx` command, without the *-i* flag, you'll be running the compiled version of CX (this is shown clearer in Listing 9.1).

Having both interpreted and compiled code results in a workflow you can follow to maximize productivity and performance. You can use the CX interpreter to test code without having to be re-compiling your code every time, and when you're done testing and fine-tuning your code, you can compile for speed and better memory management.

9.1 Interpreted CX Features

The first and most notorious feature is the REPL. Having a REPL would be cumbersome if we had all the code compiled every time a change was made to the program, and the REPL basically does this, unless you're just testing a function call. In the CX REPL you can not only redefine full functions, global variables or other "higher scope" elements, but you can also fine-tune these elements. For example, if you only want to change the type of a structure declaration's field, you can do it in CX. You don't need to re-write all the structure declaration with your changes into the REPL's prompt. This same feature applies to functions, packages, statements inside functions, expressions, anything. If you're interested on finding out how you can do all of this in the REPL, check Chapter 15. In order to provide all the previously described functionalities, the REPL uses a feature called *meta-commands* to mimic some of the functionalities of affordances (see Chapter 11).

Similarly to meta-commands, affordances are more easily implemented using the CX's interpreter. As mentioned before, affordances are similar to meta-commands. The main differences are: affordances can be called without having to be in the REPL, i.e. you can create calls to affordances in your source code files; and affordances can work with a set of rules to have an automated behavior.

Lastly, a genetic programming algorithm is provided as a native function in CX (see Chapter 13. This algorithm can be used to create functions that meet some criteria. The most traditional objective is to solve some sort of curve-fitting problem [LS86]. Theoretically, you can construct any type of function using genetic programming, and we want to achieve that in CX in later stages of the language. Just imagine setting up a rule set that defines a website or a mobile application, and the genetic programming algorithm takes care of the rest. We believe that affordances will allow to create this type of solutions in conjunction with genetic programming.

And that's it, the REPL and meta-commands, affordances and genetic programming are all features that can exist in the CX environment thanks to its interpreter. Now let's review the compiler in the next Section.

9.2 Compiled CX Features

In the interpreted version of CX we had Golang managing all the memory allocations of a program. CX didn't have any of the memory segments discussed in Section 7.1, all the values were held in Go structures, and the stack was just a sequence of structures containing all the information form each of the function calls. This is very practical because we could focus all the development to researching interesting features, but the downside was performance, both in memory and speed. A simple for loop that iterated 1 million times was taking something in the order of the dozens of seconds. This performance can be acceptable for some kind of programs or for testing some ideas, but it's definitely unacceptable for most programs in their production stage.

The CX's compiler is not exactly a compiler in the traditional sense of the word, but it definitely will become one soon. We call it a compiler for now because it will become one and because of the optimizations it makes to the generated code. In a sense, that can be already considered a compiler, as the code is not run line-by-line as in an interpreter. We are only lacking a proper way to create executables targeted to a platform (operating system and CPU).

As stated in the previous Section, CX's compiled code performs similarly to Python in some tests. Python should beat CX in other benchmarks, as it's a language that has been optimized since 1991, but at least it's not super slow as CX's interpreted programs.

Another feature of CX's compiler is that it has its own garbage collector now. Go's garbage collector is remarkable, but it was not working well with CX. Now that CX has its own memory segments, we can optimize very well how that memory is allocated.

In conclusion, the compiler was not necessary in terms of features, but it was definitely necessary as performance is almost always a critical aspect of any programming language. Even interpreted languages are often discarded or chosen because of their speed or how well they manage memory.

10. Garbage Collector

CX is a garbage collected language, unlike other languages like C, where you have to manually manage memory allocations and deallocations, or languages like Rust that adopt other techniques to manage memory. Manual memory management brings the advantage of efficiency in memory deallocations, but at the expense of possible memory leaks. If you define a routine where you allocate some objects and then you forget to properly deallocate them when they're no longer being used, you could end up exhausting your heap memory, and the program could crash. Another problem is that it could not necessarily crash immediately, but after some days or weeks of use. Maybe the program is not properly deallocating a single object every hour, so exhausting your heap memory will take some time, but it will definitely happen if the program is meant to be run forever, such as a web service. For this reason, programs made in C, for example, are usually used to solve problems where efficiency is far more important than reliability, and garbage collected languages, such as Java or Go, are usually used to write software systems meant to run for large periods of time, where reliability is preferred over resource efficiency. Manual memory management is less important nowadays that computing resources are cheaper than ever (although this statement can not be treated as a fact, we can clearly see a tendency to opt for automatic memory management in the present). Additionally, many garbage collectors are now extremely efficient and the impact on a program's performance could be regarded as negligible in many situations.

For the reasons stated above, we decided to make CX a garbage collected language, although in the future you'll be able to handle memory manually too. One of the platforms that we want to target in the future is micro-controllers, and manual memory management is usually preferred in this situation. But for now, all programs made in CX are garbage collected.

10.1 What is Garbage Collection

If you read Section 7.1, you'll have a better idea of what garbage collection is (or if you have a background in software development, of course). If your programs always use fixed-sized data structures, such as integers, floating-point numbers and structures containing this kind of data, you can always store your data in the *stack* segment, as your program's objects can be stored and destroyed in a sequential manner, and they'll always have the same relative address in the *stack*. However, programs limited to fixed-sized data structures are not going to be able to solve many situations, or not conveniently at least. For this reason, it is practical to have another segment of memory called the *heap*, where variable sized data structures can be stored. Objects in the *heap*, in CX, start being allocated sequentially, just like in the *stack*, but they can be destroyed in an arbitrary order. This behavior leaves fragmented chunks of memory being used, and other fragments that are no longer being used. A garbage collector's mission is to manage these fragmented chunks of memory.

10.2 CX's Garbage Collector

There exist many types of garbage collector. CX's garbage collector is of a type called mark-and-compact, which is a variation of another algorithm called mark-and-sweep. The mark-and-sweep is arguably the most common form of garbage collector, and the first garbage collector was of this type. The basic idea behind a mark-and-sweep garbage collector is to traverse all the pointers in the stack, and mark their pointees as alive objects. After marking all the alive objects, we can consider all the other objects as dead objects (sweep them!). This marking process usually involves changing a bit in the object's header from 0 to 1. Now if we want to allocate a new object, we can use the bytes of dead objects (marked with a 0 in their header).

As you can imagine, a lot of fragmentation is going to occur in the mark-and-sweep algorithm. Allocating new objects become an issue if the *heap* is too fragmented. An allocation algorithm must search in the *heap* for enough free bytes for the object that you want to allocate. In fact, you could end up with enough free bytes to allocate a new object, but they could be all scattered around in the *heap*. In CX we wanted to avoid this fragmentation, as fragmentation causes a program to be "using" more memory than it should. As a consequence, we decided to implement a mark-and-compact algorithm, as was stated before. In the mark-and-compact algorithm, instead of just marking the objects as either alive or dead, we move every alive object to the beginning of the *heap*, and thus, the objects are compacted by squashing out the dead objects. The way you organize the remaining alive objects is arbitrary, but the most common way of doing this is by moving the objects to the beginning of the *heap* in the order that they were created. This form of compaction is called "sliding compaction."

Each of the garbage collection algorithms has its advantages and disadvantages. The most prominent disadvantages of CX's current garbage collector are that it needs to stop any process in a program to start the marking and compacting in the *heap*, and that it needs to check all the pointers in the *stack*. The best type of garbage collectors are hybrid solutions, where a mix of the most common garbage collectors are used to attenuate the disadvantages of each of the base algorithms used. In the future, CX's garbage collector will move to this direction in order to provide a better solution. In the meantime, the CX's current solution provides a garbage collector that is efficient in memory space, as it completely avoids fragmentation.

Additionally, allocating a new object in CX is actually faster than in C, for example. The mark-and-compact algorithm can implement a form of allocation called "sequential allocation." In this algorithm, a pointer is always pointing to the end of the used *heap*, and thus knowing where a new object should be allocated will be immediately known. In contrast, C's malloc function needs to traverse the *heap*, searching for a chunk of memory that can store the object. This search process is unnecessary in sequential allocation.

11. Affordances

The concept of affordance was developed by the psychologist James J. Gibson, and it was first presented in [Gib66]. The traditional explanation of what an affordance is can be found in the aforementioned work, and it is as follows:

> The affordances of the environment are what it offers the animal, what it provides or furnishes, either for good or ill. The verb to afford is found in the dictionary, the noun affordance is not. I have made it up. I mean by it something that refers to both the environment and the animal in a way that no existing term does. It implies the complementarity of the animal and the environment.

In other words, an affordance is whatever an environment allows an object to do or to be. Affordances can act in both ways, they describe what the object can do with its environment and what it can "receive" from its environment. For example, a person can push or pull a door, and the door can be pushed or pulled by something.

In CX, affordances describe the allowed actions that can be performed on program elements and what actions can an element perform on other program elements. These program elements are functions declarations, global variables, expressions, function parameters, packages, expression arguments, structure declarations and their fields. If you read Section 7.1, you will notice that all of these are elements that are present in the code segment. The kind of actions that affordances describe in CX are, for example, adding an argument to an expression, removing a function parameter, adding a whole new expression to a function declaration, and modifying a global variable to hold a new value. You can notice that these are all actions that you usually perform when creating a program. In fact, you can create a full program only by using affordances. The first affordance of an empty program is to add a package; it can either be an arbitrary package or a **main** package, depending if you want to create a library or an application. After adding a package, the options increase dramatically, as you could, first of all, delete or rename the previously created package, add another empty package, or add declarations to your first package. As you begin to add more elements, the number of affordances increases exponentially.

There are obviously some rules or limitations on what it can be done with affordances. For example, you cannot add a structure field to a function declaration. A less obvious example is that you cannot send a 32-bit integer to **str.print()**, as this function is expecting an argument of type *str*. These limitations help reduce the number of affordances in a program, but they can still be a lot, even in relatively small programs. The solution to handling this problem is to implement some mechanism that allows us to get only those affordances that are useful. This mechanism is a rule set that you can define before asking CX what affordances are available at certain point during a program's execution. These rules will examine the elements that can be part of an affordance, and check if they meet some criteria. For example, in a video game we could reject any player that has hit points lower than certain quantity, or allow a boss to appear to the screen if the player has completed certain quests. Although these examples could have been solved using simple if/else statements, affordances can solve more obscure and complex problems, as they have a true global scope. For example, if you wanted to check if an object has already been discarded by CX's garbage collector, you can do that with affordances. Or what about if you want to have access to values of variables in previous function calls, that's right, you can do that with affordances.

Affordances were created with the purpose of increasing security in a program. There are certain types of attacks where a function call can access other parts of memory. In this case, affordances add an extra layer of security, assuring that only a limited number of elements can interact with other elements of a program.

It is worth noting that affordances not only act at compile-time, but also at runtime. You can create a function that is constantly evaluating what is allowed in the interaction among a program's elements.

As a last note before looking at the examples that follow, please bear in mind that CX's affordance system is still under development and many of its features could change in the future. For example, the rule set was previously managed by an embedded Prolog interpreter, and you had to know some Prolog in order to use it. This was obviously a very bad idea, but it allowed us to experiment with many of the possibilities of affordances. Now the rules are created using a very simple syntax. At the moment, affordances can only work with expressions, but most of the code to manipulate other program elements is almost complete.

```
package main

func main () {
    foo1 := 1
    foo2 := 2
    foo3 := 3

    target := ->{
        pkg(main) fn(main) exp(message)
    }

    rules := ->{
        allow(* > 1)
    }

    affs := aff.query(target, rules)
    aff.print(affs)
    aff.execute(target, affs, 0)

20  message:
21      i32.print(0)
22  }
```

Listing 11.1: Using Affordances on an Expression

Listing 11.1 shows a basic program that uses affordances to filter among the possible values that the expression at **Line 21** can take. As this is a small program, the only possible values are those being held by **foo1**, **foo2** and **foo3**. In order to know what expression we want to target, we need to label it first. To do this, we can simply use to-do labels, as seen at **Line 20**, where we label our target expression as "message." The next step is to create a variable to hold the target expression. To do this, we use the affordance mini programming-language, which is called by writing **->**, and we write the desired statements inside of the braces. Creating a target is done at **Line 8**. Targets are constructed by going down in levels of scope: the package is specified first, then the function, and lastly the expression. To specify the desired package, you use (pkg) followed by the name of the package enclosed by parentheses. For functions, you use **fn**, followed by the name of the function enclosed by parentheses. Lastly, to specify the expression, you use **exp** followed by the label given to the expression, again, enclosed by parentheses.

Rules, as mentioned before, are used to filter the possible options. In this example, rules are defined at **Line 12**, and they contain only one clause: allow anything that is greater than 1. The asterisk in here represents the initial allowed objects to be sent to the targeted expression. As the expression is waiting for a 32-bit integer, the asterisk will be of type i32. Think about it like how the x in mathematics can mean any number but, in this case, it can mean any program element. If the targeted expression can receive a structure instance as its input, we could create predicates of the form $*.\text{field} == \text{something}$, for example.

Now that we have both the target and the rule set, we can query CX's affordance system using **aff.query**, as shown at **Line 16**. The results returned by **aff.query** can be pretty-printed to the console by calling **aff.print**, as shown at **Line 17**. This is useful if you want the user to be involved on what affordance to execute. For example, you could use affordances to create an entire program just by selecting the options that you want, and **aff.print** would be used to let the programmer know what affordance to execute. When you have chosen an appropriate affordance, either manually or automatically, you can execute it by calling **aff.execute**, as shown at **Line 18**. **aff.execute** takes three arguments as inputs: a target, the set of affordances, and an index representing the desired option to execute. As you can see, you could execute the same affordance to several targets, and execute several affordances by specifying different indexes. In the case of the above example, we simply execute the first option, represented by index 0. After running the whole program, the number 2 should be printed to the console, as is the first element that is greater than 1.

```
package main

var goNorth i32 = 1
var goSouth i32 = 2
var goWest i32 = 3
var goEast i32 = 4

func map2Dto1D (r i32, c i32, w i32) (i i32) {
    i = w * r + c
}
```

```

func map1Dto2D (i i32, w i32) (r i32, c i32) {
    r = i / W
    c = i % W
}

func robot (row i32, col i32, action i32) (r i32, c i32) {
    if action == 1 {
        r = row - 1
        c = col
    }
    if action == 2 {
        r = row + 1
        c = col
    }
    if action == 3 {
        c = col - 1
        r = row
    }
    if action == 4 {
        c = col + 1
        r = row
    }
}

func getRules (row i32, col i32, width i32, wallMap [25]bool, wormholeMap [25]bool) (rules aff) {
    rules ->= allow(* == *)

    if wallMap[map2Dto1D(row - 1, col, width)] {
        rules ->= ->{reject(* == 1)}
    }

    if wallMap[map2Dto1D(row + 1, col, width)] {
        rules ->= ->{reject(* == 2)}
    }

    if wallMap[map2Dto1D(row, col + 1, width)] {
        rules ->= ->{reject(* == 3)}
    }

    if wallMap[map2Dto1D(row, col - 1, width)] {
        rules ->= ->{reject(* == 3)}
    }

    if wormholeMap[map2Dto1D(row - 1, col, width)] {
        rules ->= ->{allow(* == 1)}
    }

    if wormholeMap[map2Dto1D(row + 1, col, width)] {
        rules ->= ->{allow(* == 2)}
    }

    if wormholeMap[map2Dto1D(row, col + 1, width)] {
        rules ->= ->{allow(* == 3)}
    }

    if wormholeMap[map2Dto1D(row, col - 1, width)] {
        rules ->= ->{allow(* == 3)}
    }
}

func main () (out str) {
    var wallMap [25]bool = [25]bool{
        true, true,  true,  true,  true,
        true, false, true, false, true,
        true, false, true, false, true,
        true, false, false, false, true,
        true, true,  true,  true,  true}

    var wormholeMap [25]bool = [25]bool{
        false, false, false, false, false,
        false, false, false, false, false,
        false, false, false, false, false,
        false, false, false, false, false,
        false, false, false, false, false}

    var width i32 = 5
    var row i32 = 1
    var col i32 = 1

    var target aff
    var rules aff

    target = ->{pkg(main) fn(main) exp(robot)}

    for c := 0; c < 6; c++ {
        wallMap[map2Dto1D(row, col, width)] = true
        wormholeMap[map2Dto1D(row, col, width)] = false
        rules = getRules(row, col, width, wallMap, wormholeMap)

        affs := aff.query(target, rules)
        aff.execute(target, affs, 0)
    robot:
        row, col = robot(row, col, 1)
    }
}
```

Listing 11.2: Using Affordances on an Expression

Listing 11.2 shows a much more complex example. The program is an extremely naive representation of a robot moving on a map. The map is built using two arrays, where each of the indexes represents a room, and the indexes "surrounding" them are used as the contiguous rooms. One of the arrays has walls, and the other one wormholes. If the robot encounters a wall on the map, it can't move to that direction, but if a wormhole is on the wall, it can move to the other side. In the arrays, a true value means that a wall or a wormhole is present there, and a false means there is not. In the example, there is no wormhole, so you can play with the values to see the different results.

12. Serialization

In CX, every program object and piece of data can be serialized at any moment, preserving any state in which the program is. You can choose to serialize all the program or only certain parts of it, such as structure instances or functions. These serialization features are very useful, as you can save a program to a file or a database.

The serialization process not only involves the program structure, e.g. function declarations and structure instances. Other parts of a CX program are also serialized, such as the call stack and the different memory segments in CX. This means that a program can be totally or partially serialized, and it can resume its execution later on. A program could be paused, serialized and sent over a network to different computers to be executed in there. A common example of CX's serialization combined with other of CX's features is as follows.

Imagine you want to evolve programs to predict a financial market's price movement. You can start evolving functions inside of a CX program using its integrated genetic programming system (see Chapter 13). At certain points in time you can save these serializations to a database, for example, programs which achieved a very good performance. You can then send some of these serializations to other workstations or servers where they will initialize a separate evolutionary process. This is something similar to taking some monkey from Earth to different planets in the Universe: wait a few millions of years, and then check how they evolved in each of these planets (only if you believe in the theory of evolution, though; otherwise, they will still be monkeys).

Evolutionary algorithms can often be manually manipulated (imagine aliens interfering with the DNA of a planet's species). A person can log in to one of the workstations or servers in this evolutionary network, and check some of the individuals being evolved in CX. This person will just have to pause the program using the CX REPL, and check the program's structure using the **:dp** meta-command (see Chapter 15). But maybe this person doesn't know what can be added or removed from the function being evolved. This is not a problem, because the function is evolving according to a rule set established in CX's affordance system (see Chapter 11, and you'd only need to call the affordance system in the CX REPL and start selecting options from a menu. After being happy with the changes, the program can be resumed again by issuing the meta-command **:step 0**, so the program continues its execution (see Chapter 15).

12.1 Serialization

Now let's see how you can serialize the different program elements and data in CX.

```
package main

func main () {
    var target aff
    var result []byte

    target = ->{}
    result = serialize(target)
}
```

Listing 12.1: Serialization of a Program

Listing 12.1 shows how to serialize a full program using the function **serialize**, which is the one that we're going to be using in all the subsequent examples. We can tell the function **serialize** what to serialize by using the affordance operator (**->**) (see Chapter 11). In the case of serialization, we're only going to be using the affordance operator to specify a target to be serialized. In the case of Listing 12.1, we're leaving the target empty. This is a special case that instructs CX to serialize everything or, in other words, the full program.

```
package main

func main () {
    var target aff
    var result []byte

    target = ->{pkg(main)}
    result = serialize(target)
}
```

Listing 12.2: Serialization of a Package

If your program only has one package, as in Listing 12.2, you could end up with a similar serialization as in Listing 12.1, but with some differences. `->{}` instructs CX to serialize *everything*, including CX's memory segments (see Chapter 10), whereas `->{pkg(main)}` is only going to cause a serialization of the program segment (the program structure).

```
package main

func main () {
    var target aff
    var result []byte

    target = ->{mem(heap)}
    result = serialize(target)

    target = ->{mem(stack)}
    result = serialize(target)

    target = ->{mem(data)}
    result = serialize(target)
}
```

Listing 12.3: Serialization of the Memory Segments

To serialize the other memory segments, you can use the affordance target **mem()**, and give either *heap*, *stack* or *data* as its argument, as seen in Listing 12.3.

It is worth noting that if you serialize your stack, you are actually serializing *all* the stacks. CX, at the time of writing, is still not a multi-threaded programming language. Nevertheless, it should soon become one, and the data contained in all of the stacks will be serialized. Additionally, CX manages its call stack and stack separately, but both of these segments are serialized together when calling **serialize** with **mem(stack)**.

In later versions of CX, we might introduce native functions to process information from these serialization results, but for now, you can only deserialize this information into another instance of a CX program (see the next Section) or process the byte slice byte by byte to do whatever you require to do.

```
package main

var foo i32

func bar () {
    str.print("Hi.")
}

type foobar struct {
    foo i32
}

func main () {
    var target aff
    var result []byte

    target = ->{pkg(main) var(foo)}
    result = serialize(target)

    target = ->{pkg(main) fn(bar)}
    result = serialize(target)

    target = ->{pkg(main) strct(foobar)}
    result = serialize(target)
}
```

Listing 12.4: Serialization of Declarations

Listing 12.4 shows how to serialize declarations in packages. Note that the structure and function serializations are serializing the code representation of these declarations, and not instances of these. In other words, we're serializing the Person structure, not John or Ana, which would be instances of this structure. In the case of the function declaration, CX does not have functions as first-class objects yet, so there should not be any confusion, but it's good to notice that we're referring to the function declaration itself, and not an instance of this function.

```
1  package main
2
3  type Point struct {
4      x i32
5      y i32
6  }
7
8  func main () {
9      var target aff
10     var result []byte
11
12     var foo i32
13     var bar Point
14
15 foobar:
16     i32.print(foo)
17
18     target = ->{pkg(main) fn(main) var(foo)}
19     result = serialize(target)
20
21     target = ->{pkg(main) fn(bar) var(bar)}
22     result = serialize(target)
23
24     target = ->{pkg(main) fn(bar) expr(foobar)}
25     result = serialize(target)
26 }
```

Listing 12.5: Serialization of Expressions

Lastly, you can also serialize function statements, expressions and variable declarations. As you can see in Listing 12.5, function variables are targeted by specifying the package, the function and the name of the variable using **pkg**, **fn** and **var** in the affordance operator, respectively. The case of targeting an expression is a bit more complex, as you need to label it first (**Line 15**), and then use that label to target the expression in the affordance operator.

12.2 Deserialization

After serializing program elements or data using the procedures described in the last Section, you may now want to deserialize the resulting slices of bytes.

```
package main

func main () {
    var target aff
    var result []byte

    target = ->{}
    result = serialize(target)
    deserialize(result)
}
```

Listing 12.6: Deserialization of a Program

We're not going to be deserializing all of the examples from the last Section, as it'd be pointless. You're always going to have a slice of bytes, and they are always going to be deserialized by the **deserialize** function. Listing 12.6 shows **deserialize** in action, which is deserializing a slice of bytes representing the whole program.

What **deserialize** does is something similar to a *patch*. If a declaration in the slice of bytes already exists in the current program, it will simply redefine it; if it does not exist, it will create it. In the case of function declarations, statements and expressions, they can be applied using the affordance system, although this functionality has not been implemented yet.

13. Genetic Programming

After the first version of affordances was implemented in CX, it seemed natural to use it for creating a genetic programming algorithm. Genetic programming (GP) is an evolutionary algorithm that automatically creates programs (programs creating programs!). In theory, you could only tell a GP algorithm a set of goals and GP will generate the program for you, so you could tell it "I want an operating system that does this and this and this" and it could arrive to that solution. But of course in practice this would be extremely difficult. In reality, GP is usually used to find solutions to problems that are relatively hard for a human being, but relatively easy for a computer to solve. For example, you can use GP to find a mathematical model that describes a financial market (such as SKY price movements!), and it will find it in minutes or maybe seconds, depending on your hardware. However, obtaining such model by hand would be very difficult, as it could take you days, weeks or even months to create such a model.

GP is pretty easy to understand. Imagine that you have a set of operators, such as +, -, / and *. Now imagine that you have a set of input variables, such as x. Lastly, imagine that you have the plot of a curve that curiously enough resembles the curve generated by plotting $f(x) = x^2 + x$. When run the GP algorithm, it will start making random combinations of operators and variables, such as $x+x$, $x+x+x$, $x*x*x$, $x*x+x+x$, and so on. These combinations of operators then will be evaluated to see how well they perform. For example, $x*x+x+x$ will throw a curve that is closer to our target function than, say, $x+x$ (as this isn't even a curve). Those combinations that behave well are kept, while the ones that perform poorly are destroyed, just like in natural selection where the strong individuals are the ones that survive (and hence the name *genetic* programming). Again, as in natural selection, the strong individuals are the ones to reproduce and share their genetic material among them to create stronger individuals. In the case of GP, the genetic material represents mathematical terms. So, for example, if we reproduce $x*x$ and $x*x+x+x$, we can end up with these combinations: $x*x+x$ and $x*x+x+x$ (depending on how you design your crossover operators, i.e. how you want the individuals to be reproducing), where the former corresponds to the terms contained in an equivalent function to our target function.

As said at the beginning of this Chapter, CX's GP is entirely based on affordances. If you read Chapter 11, you now know that affordances can list all the possible actions that can be performed on a program's element, such as functions. Well, we can use this functionality to list all the operators that can be used to create expressions for the target functions (the one that we want to simulate). Then we can also use affordances to determine what we can send to these expressions. Also, if we want to reproduce individuals, we can use affordances to know what expressions from each individual can be obtained, and how they can be added to their offspring. You can do everything in a GP using affordances.

```
1   package main
2
3   func realFn (n f64) (out f64) {
4       out = n * n + n
5   }
6
7   func simFn (n f64) (out f64) {}
8
9   func main () (out f64) {
10      var numPoints i32
11      var inps []f64
12      var outs []f64
13
14      var c i32
15
16      for c = 0; c < numPoints; c++ {
17          inps = append(inps, i32.f64(c) - 10.0D)
18      }
19
20      for c = 0; c < numPoints; c++ {
21          outs = append(outs, realFn(inps[c]))
22      }
23
24      var target aff
25      target = ->{pkg(main) fn(realFn)}
26
27      var fnBag aff
28      fnBag = ->{fn(f64.add) fn(f64.mul) fn(f64.sub)}
29
30      evolve(target, fnBag, inps, outs, 5, 100, 0.1D)
31
32      str.print("Testing evolved solution")
33      for c = 0; c < numPoints; c++ {
34          printf("%f\n", simFn(inps[c]))
35      }
36  }
37  o
```

Listing 13.1: Using Genetic Programming to Evolve a Function

But enough about theory, let's see an example in action. Listing 13.1 shows how to use CX's GP to find a function that curve-fits $f(x) = x^2 + x$. This target function is defined at **Line 3**, and the function that will try to simulate the curve defined by the real function, **simFn**, is defined at **Line 7**. As you can see, **simFn** starts as an empty function declaration. This is because the GP is going to fill this function with expressions.

After defining our **simFn**, we now need our data. In curve-fitting algorithms you usually need two sets of data: the inputs and the outputs of the target function. For example, if you input 1, you'll get a 2, if you input a 2, you'll get a 3, etc. In this case, the inputs are constructed at **Line 17**, while the outputs at **Line 21**. The inputs range from -10 to 10, and the outputs are obtained by evaluating **realFn** with these inputs.

The next step is to set a "bag" of operators. These operators are the ones that will be used to create the CX expressions that will be inside **simFn**. In previous versions of CX we used a string to define these operators, e.g. "i32.add|i32.mul|i32.sub", but now we have integrated affordances with the GP even more, and we specify the functions using the affordance operator, as can be seen at **Line 28**. Similarly, the function to be evolved was previously defined with a string, e.g. "simFn", but now we also use the affordance operator, as seen at **Line 25**.

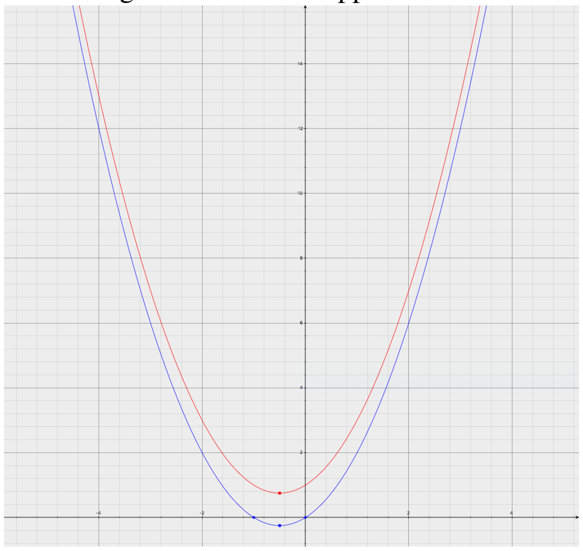
After having defined all the data mentioned in the previous paragraphs, we only need to decide how many expressions should our simulated function have, for how many generations should our algorithm run, and what's our threshold error. "What the..." you may be saying to yourself at this point, but the cure for this is to explain these concepts in the following paragraphs.

First, CX's GP is of a certain type called cartesian genetic (CGP) programming, which was devised by Miller and Thomson in [MT00]. In CGP you limit the number of expressions or statements that can be defined in a function to be evolved. This is a simple method that completely eliminates bloat, which is a major problem in traditional GP. In traditional GP, you can end up with evolved functions having thousands and thousands of expressions, and many of them might not even make any sense. For example, you could have expressions such as `x + x + x - x - x - x` or `x * x / x`. CGP has been proved in several research works that limiting the number of expressions forces GP to improve its solutions, while completely eliminating bloat, and use less computing resources as an extra.

Next, we have the number of generations. This parameter is clearly understood once we remember that programs are reproduced or crossed over, just like in biological evolution. The number of generations tell the GP how many times individuals are going to reproduce among them. The first generation will create sons and daughters (this is a book that supports gender equality after all), the next generation will create grandsons and granddaughters, the next will create great-grandsons and great-granddaughters, and so on.

The last parameter is a threshold error, often called *epsilon*. In most problems trying to be solved by any evolutionary algorithm, it will be very hard to achieve an exact solution. However, in all of these problems, a close-enough solution is usually a sufficient solution. For example, take a look at Figure 13.1. We can see two plots, and maybe our target function is represented by the blue line, while our evolved solution is the red one. Maybe this is enough, depending on where we want to use this function. For example, maybe we want to evolve a program that manages the cruise control of a car, and if we don't get an exact solution, the car might go 2 miles per hour faster or slower than our desired speed limit, and this could be a very good solution. In other cases, any error is unacceptable, such as in determining if a patient requires an amputation or not. Now, epsilon tells the GP algorithm how bad a solution can perform while maintaining us happy with the results. In CX, this error is obtained by calculating the mean-squared error (MSE), which is pretty easy to understand: you only need subtract every simulated data point to each of its real counterpart, sum all of these numbers and average them, and then get that number's square root. If you are wondering why provide both a number of generations and an epsilon, the answer is that the program will stop when any of these criteria are met. These stop criteria can be interpreted as: "I'm willing to wait for 100 generations or until the solution achieves this performance error," and this makes perfect sense once you realize

Figure 13.1: Curve approximation



that many optimization problems can take weeks or months to finish.

Having explained all the input parameters, we can now fully understand the single most important Line in Listing 13.1: **Line 30**. In this Line, we can see a call to **evolve**, which will contain all of the previously discussed input parameters. In order, we're sending: the function to be evolved (*target*), the operators to be used in the evolutionary process (*fnBag*), the inputs (*inps*), the outputs (*outs*), the limit number of expressions our evolved function can have, the number of generations the algorithm will run, and epsilon or the good-enough error.

After waiting a few milliseconds (it's an extremely easy problem to solve, after all), we'll have a **simFn** filled with expressions that will hopefully simulate the target function $f(x) = x^2 + x$. To test this, we can evaluate the evolved function with each of the inputs (the integers from -10 to 10), and check if these outputs correspond to the outputs from the target function. This testing is performed at **Lines 33-35**.

14. Understanding the CX Base Language

In Chapter 1 we mentioned that CX is actually a programming language specification, which means that you could create your own CX. The CX that we have been using throughout the book to run all the examples is called CXGO (something similar to how the most popular Python implementation is actually called CPython).

At the moment, the specification file has not been finished and it needs to be heavily updated, which means that creating your own CX from scratch is not possible, as you wouldn't know what conditions need to be met by your language. But fear not, because you can use the official Skycoin implementation to create it!

CX does not specify any syntax or grammar to be followed, which means that you could even create a CX in Minecraft using redstone. In order for a language to be called CX, it only needs to implement the necessary native functions and follow the same program structure as any other CX. For example, your CX can't implement classes, as they don't exist in CX. Also, you'd be required to implement an affordance system, as this feature is required in all CX programming languages.

How programs are executed in CX is also unspecified, which means that you could build your own runtime, compiler, linker, etc. However, you are required to compile to the same program structure. The objective with this approach is to have any program created in any CX to run using any runtime system. This is similar to the approach that some languages like Clojure or F# take, where they run using the JVM or the CLR, respectively.

Skycoin's implementation is divided in two parts: a generalized library that can be used to construct programs to be run in CX, called CX base, and the actual programming language, which was created using the CX base language and a parser built in *goyacc* and Go. If you don't mind programming in Go, you can import the CX base language to create a CX.

```go
package main

import (
	. "github.com/skycoin/cx/cx"
)
```

```
6
7  func main () {
8      prgrm := MakeProgram()
9      mainPkg := MakePackage("main")
10     mainFn := MakeFunction("main")
11
12     prgrm.AddPackage(mainPkg)
13     mainPkg.AddFunction(mainFn)
14
15     prgrm.RunCompiled()
16 }
```

Listing 14.1: Writing a program using CX base language

Listing 14.1 shows how you can use the CX base language to create a very basic program that can be run using Skycoin's CX runtime. First we need to import the CX base language package, as shown at **Line 4**. This package will give us access to *makers*, *adders*, *removers* and other utility functions that will help us construct a compliant CX program. **Line 8** creates the minimal CX program you could create: a null program. A null program is one that does not have any package, functions or anything in it. If you try to run this program, CX will complain because it doesn't have a *main* package nor a *main* function, but this doesn't mean it's not a valid CX program; if you were developing a library, neither of these two components are required.

As we are interested on running a program, not just creating a library, we create a *main* package and function, at **Lines 9** and **10**, respectively. As you can see, the naming convention for the functions that are going to be creating the program elements is *MakeXXX*, and these functions will usually require the essential properties to be sent as input parameters, such as the name of a package.

We already have the elements created, but they have not been added to the main program structure yet. We can do this by calling an adder method on the elements that are going to hold these new elements. In this case, we are interested on adding a function to a package, and adding that package to the main program structure. To do this, we're calling the program's method **AddPackage**, and sending the created package as an argument, and we do the same with the *main* function by calling the package's method **AddFunction**, and sending it as an argument. These operations are seen at **Lines 12** and **13**.

Finally, even though the program does nothing, we run the program by calling the program's method **RunCompiled** at **Line 15**. Save that code to a file, run it by executing `go run example.go`, and you should see... nothing. Let's now create a more interesting program: let's calculate 10 + 10!

```
1  package main
2
3  import (
4      . "github.com/skycoin/cx/cx"
5  )
6
7  func main () {
8      prgrm := MakeProgram()
9      mainPkg := MakePackage("main")
10     mainFn := MakeFunction("main")
11     initFn := MakeFunction(SYS_INIT_FUNC)
12
13     prgrm.AddPackage(mainPkg)
14      mainPkg.AddFunction(mainFn)
15      mainPkg.AddFunction(initFn)
16
17      sum := MakeExpression(Natives[OP_I32_ADD], "", 0)
18
19      num := MakeArgument("", "", 0)
20      num.AddType("i32")
21      num.Offset = 0
22
23      WriteToStack(&prgrm.Stacks[0], 0, []byte{10, 0, 0, 0})
24
25      sum.AddInput(num)
26      sum.AddInput(num)
27
28      result := MakeArgument("result", "", 0)
29      result.AddType("i32")
30      sum.AddOutput(result)
31
32      prnt := MakeExpression(Natives[OP_I32_PRINT], "", 0)
33      prnt.AddInput(result)
34
35      mainFn.AddExpression(sum)
36      mainFn.AddExpression(prnt)
37
38      prgrm.RunCompiled(0)
39  }
```

Listing 14.2: Summing 10 + 10 using CX base language

Listing 14.2 shows how you can double a number, and then print the result using the CX base language. It's quite a shock right? Similarly to the last example, we need to create a program, a *main* package, and a *main* function. In addition to that, we need to create a **init* function, which, as explained in Chapter 2, initializes some parts of a CX program such as global variables. These components are created and added to the program at **Lines 8-15**. Then we continue with the expression that will perform the sum at **Line 17**.

As mentioned before, we'll be doubling a number. This is important to note, as we're not going to be adding, for example, a 10 and another 10; instead, we're using the same number to double it. This might not make sense, but let's see how we create the inputs. First, we need to create the argument at **Line 19**. The first input argument to **MakeArgument** is the name of the argument, which only makes sense if we're creating a variable or symbol. In this case we just want to hold a reference to the number 10, so we don't really need to name the argument. Additionally, we already have the internal name of this argument, as we're assigning it to the Go variable **num**. **num** will be pointing to the *offset* 0, as seen at **Line 19**. This means that whatever is stored at the beginning of a stack frame, that'll be the value of our **num** argument, and as it is of type *i32*, it'll have a size of 4, which means that it'll read the first 4 bytes of the stack frame. Before continuing with the next parts, the second and third arguments of **MakeArgument** are the file name where the argument is declared and line number respectively, which we don't need for this example.

Next we will write our information to the stack. To do this, we'll use the function **WriteToStack**, which first takes a stack as its argument, then the offset at which it should start writing bytes, and lastly the sequence of bytes to write. As has been mentioned in other Chapters, CX is currently single-threaded, but it will become multi-threaded in the future. As a consequence of this, you need to send **WriteToStack** a reference to the stack to which you want to write to. For this example, we are using the first stack, we'll start writing our bytes at the index 0 of the stack, and we'll write `10, 0, 0, 0`, which corresponds to the 32-bit integer 10.

After creating the argument and writing the bytes our argument will be pointing to, we can now add this argument to our expression. This is done at **Lines 25** and **26**. As you can see, we are adding the same argument twice to the expression (we want to be efficient with our memory, after all).

If we run the program until this point, CX will complain about evaluating the expression and not using the result, similarly to what Go would throw. Let's now assign the result to a variable to avoid this error. To do this, we create another argument, as seen at **Line 28**, and then we add this argument as the **sum** expression's output, as seen at **Line 30**.

Now if we run the program until this point, we'll have CX doubling 10, and overwriting this number with 20. The reason behind this is that the output variable **result** has a default offset of 0 too, so it's pointing to the same memory address as **num**.

The next expression does a call to **i32.print**, and is defined at **Line 32**. After defining the expression, we can add an input argument to it which, in this case, is the **result** argument, as seen at **Line 33**.

It has been an exhaustive journey, but we now only need to add the expressions to our *main* function, as it's done at **Lines 35** and **36**, and call our program's **RunCompiled** method. Finally, if we run our file with `go run example.go`, we'll see a 20 being printed to the terminal. Feel proud about your achievement!

As a final comment, if you want to create your own CX, you'll need to be generating all of these commands automatically. For example, you can use a parser such as *goyacc* (the one used by CXGO) to generate the program's structure.

15. CX's Read-Eval-Print Loop

The REPL has been used to certain extent in the preceding Chapters, but its features have not been thoroughly discussed. This Chapter aims to explain all of the currently developed features for CX's REPL.

Most of the features that are presented here are related to *meta-commands*, which are commands that you can enter in the REPL that affect a program, but are not actual expressions, statements or declarations.

15.1 Selectors

Let's first discuss *selectors*, which are meta-commands that allow us to navigate a program's structure and target elements to be affected by other meta-commands.

```
CX 0.5.7
More information about CX is available at http://cx.skycoin.net/ and https://github.com/skycoin/c

:func main {...
    *

* func foo () {}

* :dp
Program
0.- Package: main
    Functions
        0.- Function: main () ()
        1.- Function: *init () ()
        2.- Function: foo () ()

* :func foo {}

:func foo {...
```

```
20      * i32.print(5 + 5)
21
22  :func foo {...
23      * :dp
24  Program
25  0.- Package: main
26      Functions
27          0.- Function: main () ()
28          1.- Function: *init () ()
29          2.- Function: foo () ()
30              0.- Expression: *lcl_0 i32 = add(5 i32, 5 i32)
31              1.- Expression: printf( str, *lcl_0 i32)
```

Listing 15.1: REPL function selection meta-command

Listing 15.1 shows a REPL session where we start inside the function **main** at **Line 5**, and then we exit that scope using Ctrl-D. At any moment, you can know in what scope you are in by looking at the line above the prompt, and you can go up one level in scope by hitting Ctrl-D. If you are in the global scope and hit Ctrl-D, you'll leave the CX REPL, so be careful.

After exiting **main**, we declare a new function in the global scope: **foo**, at **Line 7**, and we check that it was actually created by debugging the program structure using the **:dp** meta-command (which stands for "debug program"), at **Line 9**.

To change scope, we'll use our first *selector* **:func**. **Line 17** shows the meta-command in action, and we can see that it changed the scope to **foo**'s at **Line 20**. At that same Line, we add an expression to **foo**: a call to **printf**, which will only print 10 to the terminal. We again check that the expression was correctly added by calling **:dp**.

The other selectors are **:package** and **:struct**, which change the scope to another package or to another struct declaration, respectively.

15.2 Stepping

Let's continue the REPL session from Listing 15.1 in Listing 15.2.

```
1   :func foo {...
2       *
3
4   * :func main {}
5
6   :func main {...
7       * foo()
8
9   :func main {...
10      * :dp
11  Program
12  0.- Package: main
13      Functions
14          0.- Function: main () ()
15              0.- Expression: foo()
16          1.- Function: *init () ()
17          2.- Function: foo () ()
18              0.- Expression: *lcl_0 i32 = add(5 i32, 5 i32)
```

```
              1.- Expression: i32.print(*lcl_0 i32)

:func main {...
    * :step 0
10

:func main {...
    * :step 1
in:main, expr\#:1, calling:main.foo()

:func main {...
    * :step 1
in:foo, expr\#:1, calling:add()

:func main {...
    * :step 1
in:foo, expr\#:2, calling:i32.print()
10

:func main {...
    * :step 1
in:terminated

:func main {...
    * :step 1
in:main, expr#:1, calling:main.foo()

:func main {...
    * :step 1
in:foo, expr#:1, calling:add()

:func main {...
    * :step 1
in:foo, expr#:2, calling:i32.print()
10

:func main {...
    * :step -1

:func main {...
    * :step 1
in:foo, expr#:2, calling:i32.print()
10

:func main {...
    * :step -1

:func main {...
    * :step 1
in:foo, expr#:2, calling:i32.print()
10
```

Listing 15.2: REPL Session Example

We want to add a call to **foo** in our **main** function, and we do this by simply writing **foo()** while being in the scope of the **main** function. To do this, we first need to exit **foo**'s scope by hitting Ctrl+D, and using the **:func** selector, as seen at **Line 4**. After this, we can add the call to **foo**, and we can check our new program's structure using the **:dp** meta-command.

In order to test our program, we can use CX's *stepping* features. First, if we want to run all the program until the end, we can use the **:step 0** meta-command, as seen at **Line 22**. But sometimes we'll need to check a program's execution step by step, and we can do this by giving the **:step** meta-command a different argument than 0. Starting at **Line 26**, we can see how the REPL tells us at what line number we are in what function call. After issuing enough **:step 1** meta-commands, we finally see that the program finalizes at **Line 39**, with the REPL printing the message `in : terminated`.

An even more interesting feature of *stepping* is that you can give it negative arguments. If this is the case, CX will create a behavior similar to a *for* loop, where the stepped back expressions will be executed again. An example of negative stepping starts at **Line 56**. You can see how we step back and forth to keep printing the number 10 to the terminal.

16. Unit Testing in CX

As CX grew, a mechanism to test all the features of the language was needed. Sometimes adding a new feature to CX breaks other features of the language. For example, once methods were added to the language, bugs related to accessing structure instance fields arose. The parser was getting confused, as it didn't know how to differentiate between, for example, `instance.field` and `instance.methodCall()`. We were not noticing these errors until we actually run code involving method calls or accessing fields. The solution to this problem is to unit test each of the features of the language every time the language gets considerably modified.

At the time of writing, CX's unit testing library consists of a single function: **assert**. As in other languages, **assert**'s objective is to check if two arguments are equal. In CX, this test is performed byte by byte, so a 32-bit integer is never going to be equal to a 64-bit integer, even if they represent the same real number, because they have different sizes.

```
package main

func main () {
    var correct []bool

    correct = append(correct, assert(i32.add(10, 10), 20, "Add error"))
    correct = append(correct, assert(10 - 10, 0, "Subtract error"))
    correct = append(correct, assert(i32.f32(10), 10.0, "Parse to F32 error"))
    assert(5 < 10, true, "I32 Less than error")
}
```

Listing 16.1: Testing with assert

Listing 16.1 shows an example on how to use **assert** to test different arithmetic operations. The first and second input arguments to assert are the ones that get compared byte by byte, while the third argument is a custom error message that is appended to the default error message. In CX it's conventional to start with the expression to be tested as the first input argument, and then use the second input argument as the desired result of the first input argument. The custom error message is helpful to understand what expression raised an error, in addition to the usual file name and line number thrown by CX.

Also, notice that **assert** returns a boolean argument, which indicates if the test was successful or not. This might seem like it does not make sense, as **assert** will stop a program's execution if the test is not successful, but this behavior is there for two reasons: 1) you can count the number of tests performed, and 2) CX will implement in the future a function, **test.error**, which tests if an expression raised an error in a particular situation, while avoiding halting the program. For example, `i32.div(0, 0)` has to raise a *divide by 0* error, and if it doesn't, then this is an error. After re-implementing this function (most likely with a different name, as the test package no longer exists), we will be able to count how many tests return true and how many return false.

```
package main

func main () {
    var check i32
    check = 999

    if 2 < 3 {
        check = 333
    }

    assert(check, 333, "not entering IF error")

    if  3 < 2 {
        check = 555
    }

    assert(check, 333, "entering IF error")

    if 2 < 3 {
        check = 888
    } else {
        check = 444
    }

    assert(check, 888, "entering else in IF/ELSE error")

    if  3 < 2 {
        check = 111
    } else {
        check = 777
    }

    assert(check, 777, "entering if in IF/ELSE error")

    if 3 > 0 {
        if 25.0 > 29.0 {
            check = 0
            assert(check, 10, "entering nested IF/ELSE 2nd level error")
        } else {
            if 30L < 60L {
                check = 999
42              } else {
43                  check = 0
44                  assert(check, 10, "entering nested IF/ELSE 3rd level error")
45              }
46          }
47      } else {
48          check = 0
49          assert(check, 10, "entering nested IF/ELSE 1st level error")
50      }
51
52      assert(check, 999, "entering nested IF/ELSE error")
53
54      var i i32
55      for i = 0; i < 10; i = i32.add(i, 1) {
56          check = i
57      }
58
59      assert(check, 9, "FOR loop error")
60
61      for i = 1; i32.lteq(i, 10); i = i32.add(i, 1){
62          if i32.eq(i32.mod(i, 2), 0000) {
63              check = i
64          } else {
65              check = i
66          }
67      }
68
69      assert(check, 10, "FOR-IF/ELSE loop error")
70  }
```

Listing 16.2: Testing control flow statements

Listing 16.2 shows a more complex situation, where we are testing if the different control flow statements of CX are behaving as intended or not. For example, in an *if/else* statement, if the predicate is true, the *then* clause needs to be executed, not the *else* clause. To test this behavior, we can create a "check" variable that is going to be changing its value, just like it can be seen at **Line 8**. If this *if* statement is successful, the **check** variable will change its value from 999 to 333. As a consequence, we need to use **assert** to check if **check**'s value is now 333. If this is not the case, we can now be sure that there's an error with how the *if* statement is implemented, and we need to correct it. Likewise, at **Line 14** we check if the *if* statement is correctly not entering when its predicate evaluates to false. If the *if* statement enters in this case, the value of **check** will be changed to 555, so we need to test using **assert** that **check**'s value is still 333.

References

[Gib66] James Jerome Gibson. "The senses considered as perceptual systems." In: (1966) (cited on page 50).

[LS86] Peter Lancaster and Kestutis Salkauskas. *Curve and surface fitting: an introduction*. Academic press, 1986 (cited on page 46).

[MT00] Julian F Miller and Peter Thomson. "Cartesian genetic programming". In: *European Conference on Genetic Programming*. Springer. 2000, pages 121–132 (cited on page 62).