# Pirate

# A Compiler for Lua targeting the Parrot Virtual Machine

Klaas-Jan Stol

June 6, 2003

# Abstract

The Lua programming language is a dynamically typed language with a simple syntax. It was designed to be extensible and embedded. Lua can be extended by the programmer so that events that otherwise would result in an error, can then be caught by user-defined functions.

Beside extending Lua, it is also possible to embed a Lua program within a host program. This way, the host program can invoke a Lua script for higher level tasks that are more easily programmed in Lua than in a lower level language such as C, although a Lua script may also be executed as a stand-alone program.

The original distribution of Lua offers the programmer a Lua interpreter, or virtual machine. The interpreter can be used to execute a Lua script directly. Before actually executing the script, it is compiled into Lua byte-code. The interpreter can also be used to compile a script only, without actually running it. One way or the other, the Lua language and virtual machine are integrated. To be able to run a Lua script on another virtual machine, the Lua language must be separated from its original run-time environment.

Parrot is a virtual machine that is currently being designed and implemented to execute dynamically typed languages. Because Parrot is able to execute dynamically typed languages, then it should also be able to host the Lua language.

This paper describes the design and implementation of Pirate, a Lua compiler that targets the Parrot Virtual Machine. Not only is this an exploration of compiler construction theory and practice, it also demonstrates the re-implementation of a programming language to a new virtual machine.

Pirate was not designed to execute a Lua script as an embedded program, but rather as a stand-alone application consisting of one or more modules (although it is possible to embed a Lua script compiled with Pirate). Because of the fact that Parrot will be able not only to run Perl 6 but other languages as well, it is interesting to investigate how these modules written in different languages can cooperate. This offers a new use of the Lua language.

Although Parrot is still under construction at this moment, many things are already operational and it was possible to implement a working Lua compiler. However, because certain features of the Lua language need rather specific support of Parrot that are not implemented yet, not all features of Lua have been implemented. Other features have a temporary implementation that will be changed at a later stage when the needed features of Parrot are fully operational.

# Contents

# Preface

This paper presents the design and implementation of Pirate, a Lua compiler that targets the Parrot virtual machine. This work was done in the last semester of my undergraduate education at the Hanzehogeschool Groningen. Pirate was created to explore the possibilities of re-targeting an existing programming language to a new platform, and to obtain a better understanding of compilers in general. For this purpose, the Lua language was re-targeted to the Parrot virtual machine, which will be the new virtual machine for the Perl 6 programming language.

## Content of this paper

This paper consists of four parts. Part one is an introduction to the Lua language, the Parrot virtual machine and compilers in general. Chapter 1 introduces the Lua language, virtual machines and the Parrot virtual machine in particular. Chapter 2 gives an overview of compiler construction in general.

Part two describes the implementation of the front-end of the compiler. The front-end is the part of the compiler that does syntax and contextual checking (such as scope checking) of the source program. Implementation of the syntactical analyzer is described in Chapter 3. Implementation of the scope checker is discussed in Chapter 4.

Part three deals with the back-end of the compiler. The back-end of a compiler does not have to do any more checks on the source program. At this point of compilation, all errors should have been discovered. Chapter 5 first discusses some issues that must be dealt with before actual generating code. After that, Chapter 6 describes how machine instructions are generated. Chapter 7 describes the optimizer that tries to optimize the generated code.

Part four finally describes how Pirate can be used. Chapter 8 describes what Lua libraries are available in Pirate and which restrictions are present for using the libraries. After that, Chapter 9 explains in what ways Pirate can be used.

Finally, the conclusion recapitulates the presented work and provides a list of known issues in the current implementation. After that, a list of potential future improvements is provided, that can be implemented at a later stage.

## Obtaining Pirate

Pirate is written in ANSI C, so it should be possible to run the program on any platform that has a corresponding compiler. The software is available at:

`http://sourceforge.net/projects/luaparrot/`

## Readership

This report is aimed at people that have some experience in programming in a higher level programming language, such as Java, C or Pascal. I assume the reader understands what a compiler does, and has a basic knowledge of scripting languages. No knowledge of compilers or compiler theory is assumed.

## Acknowledgments

At this point I would like to express my gratitude towards Martijn de Vries for his supervision and helpful tips. I would also like to thank the people of the Perl 6 community for their responsive reactions and contributions.

Klaas-Jan Stol

# Part I

# Overview

# Chapter 1

# Introduction

## 1.1 Compilers

A compiler is the basic tool for any modern programmer. Any program written in a high-level programming language must be translated to some form that can be understood by a computer. For example, it is not possible for a computer to understand the following Lua program:

```
do
  x = 1 + 2
end
```

For the computer to be able to execute such a program, it must be translated to some form of machine code, like this:

```
 new           P0, .LuaNumber  # create a new number variable
 set           P0, 1           # set this variable to the value 1
 add           P0, 2           # add 2 to this number
 store_global  "x", P0         # store the value with name "x"
```

The instructions above are examples of the machine code for the Parrot Virtual Machine. Actually, the *mnemonics* for the instructions are printed here, for ease of reading. The actual machine code is in binary form. To be able to execute the above instructions, these mnemonics should be translated to a binary format by the assembler.

The translation of a programming language to machine instructions (or its mnemonics) is done by a compiler. A compiler does not necessarily translate a programming language. There are more types of compilers. For example, the program that formatted this report, LaTeX, is in fact a compiler too. The text is typed in as plain text, and all kinds of commands are used to indicate which text is a chapter title, paragraph title, normal text, et cetera.

Chapter 2 gives a more detailed overview of the tasks of a compiler.

## 1.2 The Lua language

This section gives an overview of the Lua language. This is to get an idea of the possibilities of the language, and potential problems that might arise during the implementation. Implementation of Pirate is described in detail in parts II and III. A more detailed description of the Lua language can be found in [12] and [13].

### 1.2.1 Usage of Lua

Lua was originally designed as an extensible embedded language. It is extensible, in the sense that the programmer can extend its semantics and change the behaviour of the language (see 1.2.5). An embedded language means that a Lua program can be executed within another program. This can easily be done using the C language.

Lua does not necessarily have to be run as an embedded language, it can also be executed as a stand-alone language.

### 1.2.2 Lua Data Types

Lua has six built-in data types. These are: *number*, *string*, *userdata*, *table*, *function* and *nil*. Numbers are used to store floating-point or integer numbers. The string type is used to store strings. The userdata type can be used to store C pointers (for embedded Lua scripts). Tables are associative arrays, so they may be indexed with ordinary numbers, but also with other values like strings. Tables are discussed in more detail in 1.2.3. Functions are blocks of statements that may be called. Functions are discussed in more detail in 1.2.4. Nil is a special type that only has one value (called **nil**) that is different from any other value. It is similar to NULL in C.

Lua is a dynamically typed language which means that variables do not have a fixed type, but that their types can change during run-time. Any variable can hold any value. One moment a variable can hold a number, the next moment it can hold a string. Variables do not have to be declared, like in most statically typed languages.

### 1.2.3 Lua Tables

As noted above, tables in Lua are implemented as associative arrays. This means that these arrays can be indexed not only with numbers but also with strings. In fact, they can be indexed with *any* non-nil value. Lua uses tables to create user-defined data types, just as Pascal has *records* and C has structures.

### 1.2.4 Lua Functions

Lua functions are *first-class* values. This means that functions can be stored in variables and passed around as arguments or return-values. Functions in Lua do not have a name, they are just entities representing a block of statements. These entities may, of course, be assigned to some variable. Calling a function is done by invoking the variable to which the function was assigned to.

### 1.2.5 Extensible Semantics with Tag Methods

Lua is extensible in the sense that the Lua programmer can change the behaviour of the language. This is done through *tag methods*. Each variable in the Lua run-time environment has a *tag*. One could think of it as a label that contains a number.

The tag is used when *events* occur. The complete list of events is shown in figure 1.1. When an event occurs, the tag is used as an index into a table containing functions, the so-called tag methods. For example, in figure 1.1, the first event listed is **gettable**. When an indexed variable is accessed (such as in `a[b]`), then the tag of the table being indexed is used as an index into the tag method table. This yields a tag method, which will be called. This event is quite innocent, but suppose a programmer tries to add two tables together. Obviously, this is normally not possible. But Lua allows the programmer to define a tag method for such an event. Another way to look at a tag method is as an exception handler.

| **Event** | **Cause** |
|---|---|
| **gettable** | an indexed variable is accessed. |
| **settable** | an indexed variable is assigned. |
| **index** | an attempt is made to access a non-existent index. |
| **getglobal** | a global variable is accessed. |
| **setglobal** | an assignment is made to a global variable. |
| **add** | a + operation is applied to non-numerical operands. For **sub**, **mul**, **div** and **pow** there are similar events. |
| **unm** | a unary minus operation is applied to a non-numerical operand. |
| **lt** | an order operation is applied to non-numerical or non-string operands. The remaining order operations can be expressed using the regular equivalents (For example, **gt** can be implemented using **lt** with operands switching place). |
| **concat** | a concatenation is applied to non-string operands. |
| **gc** | a userdata object is garbage collected. |
| **function** | a non-function value is called. |

Figure 1.1: The complete list of tag events.

Defining tag methods is a way to extend the semantics of Lua. More information and applications of tag methods can be found in [13].

## 1.3 Virtual Machines

### 1.3.1 What is a Virtual Machine?

A Virtual Machine (VM), or *interpreter*, is a computer in software. This means that instructions are not executed by a hardware processor, like a Pentium and the like, but by a program that *emulates* a processor. The machine code for a virtual machine is often called *byte-code*. Byte-codes are often just numbers, each representing a particular instruction. One could think of a virtual machine as one big switch statement, such as in the C fragment below:

```
while(more_bytecodes) {
   switch(bytecode) {
      case LOAD:
      /* load a variable from memory */
      case STORE:
      /* store a variable in memory */
      case ADD:
      /* add a number to some other number */

      ( for each bytecode a case )
   }
   bytecode = next_bytecode();
}
```

This is a very simplistic view on a virtual machine, and implementations can differ from machine to machine, but the key point is that instructions are executed by software[1].

### 1.3.2 Stack-based vs Register-based Machines

Many virtual machines are stack-oriented. This means that most instructions operate on values on a stack, normally the two topmost values. For example, the `ADD` instruction would pop the two topmost values, add them together, and push the result back onto the stack. The Java Virtual Machine and the original Lua interpreter are stack-based virtual machines.

There are also register-based virtual machines. These machines do not use a stack to store operands for an operation, but the operands are stored in processor registers. This is much more like real hardware machines. The Parrot Virtual Machine is a register-based machine.

### 1.3.3 Why a Virtual Machine?

An instruction executed by hardware is usually executed much faster than if it were executed in software. This brings up the question of why a virtual machine

[1]Eventually all software is executed by a hardware processor, so the software emulating a processor (the VM) itself is, too.

is preferred to a real machine. There are a few reasons for this.

- **Portability** When code is generated for a real (hardware) machine, then the generated code can almost certainly not be run on another type of machine. When a virtual machine is used, there is an extra layer of abstraction inserted. In that case, the code generator of a compiler can target the virtual machine. To run the code on another type of machine, only the virtual machine has to be ported. Because most virtual machines are written in a (portable) high-level programming language, this porting is easy. Often, only a minimum of machine-specific code has to be rewritten, sometimes, no rewriting has to be done at all. For example, the original Lua interpreter and compiler are written in ANSI C, which means the source code can be compiled on any platform that has an ANSI C compiler available.
- **High-level operations** Some concepts found in programming languages are not easily implemented on a real machine. This is especially true for functional or logic programming languages. Real machines are largely oriented towards imperative languages, like Pascal, C and Fortran. This means that the operations of an imperative language are easily translated to instructions on a real machine.
- **Ease of implementation of a language** Targeting a virtual machine is often easier, because a VM often has some higher level of abstraction. When targeting a real machine, one has to take much more details in account. A VM often has some high-level instruction set. Because implementing instructions in hardware is an expensive matter, the instruction set of a real machine is not likely to be of such a high level.

## 1.4 The Parrot Virtual Machine

To get an idea of what the Parrot VM is, it is interesting to look at some internals of the VM. Below is a description of the main components of the Parrot virtual machine.

### 1.4.1 Built-in Data Types

Parrot has four built-in data types. These are: integer, floating-point, string and Parrot Magic Cookie, or PMC. The integer type will be used to store integers or pointers. The floating-point type is designed to store architecture-independent floating-point numbers. The string type is an abstracted, encoding-independent string type. PMCs will represent any other type, including objects. They are discussed in 5.2.1.

### 1.4.2 Registers

Parrot has four sets of 32 registers. This is one set for each built-in data type. Registers are used by all instructions that operate on data. That is, when an instruction is to be executed on some operands, these operands are first loaded into registers.

### 1.4.3 Stacks

Although Parrot is a register-based virtual machine, that does not mean it has no stacks. Stacks are very convenient to have around, because they are a relatively fast way to store and restore data. Many machines have special instructions for manipulating stacks. Parrot has six stacks, one for each built-in data type, one call stack and one generic (user) stack. The stacks for the built-in types are used for saving all registers during function calls. The call stack is used to store return addresses and the like when calling functions. The generic, or user stack can be used to store arbitrary data.

### 1.4.4 Globals and Lexicals

Parrot has two places for storing PMCs. First, there is a global symbol table, that can easily be accessed by special instructions. Second, Parrot has so-called *scratchpads*. A scratchpad can be used to store local PMCs. Whenever a new scope block is entered, a new scratchpad can be created. The scratchpads themselves are stored on a special stack. So, when a scope block is closed, the topmost scratchpad is popped off of the stack.

### 1.4.5 Other Features

Parrot has some other important features. These features are quite high level; they will not be implemented on a real machine (that is, in hardware).

- **Garbage collector** The Parrot VM has a garbage collector (GC). A GC is a system that reclaims memory that is used by objects (not necessarily class objects) that are not used or referenced anymore. This memory would be lost if it wasn't for the GC. Having a GC greatly simplifies memory management for the programmer: she[2] does not have to take care of it at all. Parrot does have an instruction to explicitly destroy an object, which can be used if it is obvious that a particular object will not be used anymore.

- **Support for Threads** Parrot has support for threads. This means that Parrot has special operations for dealing with threads. This makes implementing threads much easier, because these instructions are atomic, so when the VM is executing such an operation, the thread executing it cannot be preempted.

[2]This means he or she.

- **Support for Objects** Parrot has support for objects. This means that the Parrot VM has instructions to do object operations, such as an instruction for calling a method. This kind of support makes implementing an object oriented language easier. At the moment of this writing, objects are not fully implemented.
- **Support for Exceptions** Parrot has special instructions that handle exceptions. These are among others instructions that set an exception handling routine in place, and throw an exception. Having these special instructions makes implementing exceptions easier. At the moment of this writing, exceptions are not fully implemented.
- **Support for Events** Parrot will have built-in support for events. At the moment of writing, there is no information available about events.

## 1.5 Running Lua on Parrot

The original Lua language and run-time environment is distributed as a whole. That is, no distinction is made between the Lua language and its run-time environment. But in reality, one should make this distinction. Even when one runs a Lua program in the original distribution, the program is first compiled to (Lua) byte-code. This byte-code is executed by the Lua virtual machine. Virtual machines were described in more detail in section 1.3 .

This distinction between language and interpreter must be made when re-targeting Lua. When Lua is executed by Parrot, the Lua interpreter no longer plays any role. This has implications for the complete implementation of the language. The Lua interpreter was specially designed for the Lua language, so anything that was needed for the language could be put into the interpreter. This is not the case for Parrot. Parrot is meant to be an interpreter that can host dynamically typed languages, but has no special support for a number of features that are specific for Lua. This may imply that the implementation of some Lua features is not as efficient as in the original Lua VM. An example of this is tag methods, which were described in 1.2.5. Other features are present in Parrot, but may have a different behaviour. An example of this is the garbage collector. The Lua interpreter has a garbage collector that works in a specific way, like the Lua designers thought it would be best. Parrot also has a garbage collector, but the implementation is different. All these factors have to be taken into account while implementing Lua for Parrot.

# Chapter 2

# Overview of Compiler Organization

## 2.1 Introduction

A compiler is a complex program. To understand how a compiler is built, it is useful to have a look at its components. This chapter gives an overview of how a typical compiler is constructed, and later on, how Pirate is constructed. It is divided into two parts. The first part will describe the modules that together form a typical compiler. Each module is a self-contained part that has a specific function. Section 2.2 describes these modules.

The second part will describe the data-flow within Pirate. When the compiler will be running, it starts with reading the source program, that is, the program being compiled. The compiler will create a data structure representing this source program. This intermediate representation is processed by the different phases of the compiler. Section 2.3 will discuss how the different phases of the compiler process this intermediate representation.

## 2.2 Compiler Organization

This section describes the overall structure of a typical compiler. Details of the implementation of Pirate can be found in parts II and III.

A typical compiler consists of a number of modules. The main modules are:

- Lexer
- Parser
- Symbol Table
- Contextual Analyzer

- Code Generator
- Code Optimizer

The task of each module is described in more detail in the rest of this section.

### 2.2.1 Lexer

The lexer reads the source program and groups characters together to form *tokens* or *symbols*. The lexer is called every time the parser needs a token. For example, the following Lua code snippet:

```
function hello()
   print("Hello, World!")
end
```

will result in the following tokens:

| token | type |
|---|---|
| `function` | `keyword 'function'` |
| `hello` | `identifier 'hello'` |
| `(` | `a left parenthesis` |
| `)` | `a right parenthesis` |
| `print` | `identifier 'print'` |
| `(` | `a left parenthesis` |
| `"Hello, World!"` | `a literal string` |
| `)` | `a right parenthesis` |
| `end` | `keyword 'end'` |

Another task of the lexer is eliminating unneeded text such as comments. The lexer for Pirate is discussed in more detail in section 3.1.

### 2.2.2 Parser

The parser reads the tokens as delivered by the lexer and groups them together into units called *phrases* or *sentences*. The syntactic structure of a programming language is often described by a *context-free grammar* (CFG). This is a formal specification that describes how phrases in a particular language may be constructed. A very convenient notation for writing grammars is Backus-Naur Form (BNF), so this notation is used here.

If a particular phrase is not valid, then the parser submits a syntax error. For example, the following is a small part of the grammar for Lua, that defines the structure of a while statement:

`statement` → `while` ***`expression`*** `do` ***`block`*** `end`

A while statement begins with the keyword `while`, then an expression, after that the keyword `do`, after which a block should follow (a block should be interpreted as a block of statements). At the end there should be the keyword `end`.

According to this rule, the following program is correct:

```
while x > 0 do
    x = x - 1
end
```

whereas the following is not:

```
while x > 0    -- syntax error:  "do" expected
    x = x - 1
end
```

During parsing, it builds an intermediate representation (IR). In Pirate, this IR is an Abstract Syntax Tree. (AST). This tree is used in later phases of the compiler. Figure 2.1 shows the AST for the while statement above.

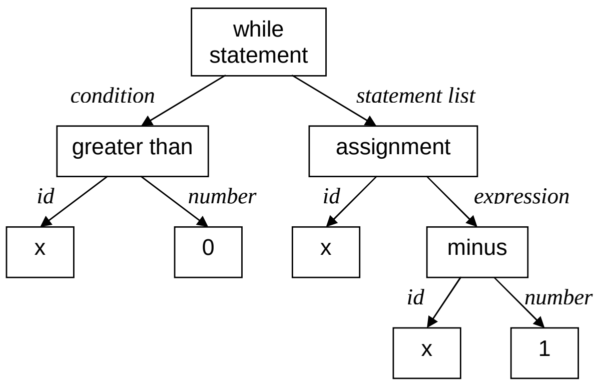


Figure 2.1: The AST for a `while` statement.

Section 3.2 will discuss the parser in more detail. Constructing an IR is formally not a task of a parser, but because the parser in Pirate constructs an IR, the AST is discussed in this chapter too.

### 2.2.3 Symbol Table

The symbol table is used to store symbols when necessary. In programming languages that have statically typed variables, the type (and often some other data, too) is saved as an attribute of the variable, when the symbol is entered into the symbol table. In Lua this is not the case, because Lua variables do not have a fixed type. The Lua compiler uses symbol tables in two phases: the scope checking phase and the code generating phase. The scope checker is described in Chapter 4. The symbol table and its implementation are described in appendix D.2.

### 2.2.4 Contextual Analysis

In most compilers there is a component that performs *contextual* or *semantic analysis*. Contextual analysis may consist of several different checks. Typical for this phase are:

- Declaration checking
- Type checking
- Scope checking

In many programming languages variables have to be declared before they can be used. If a variable is used before it is declared, the contextual checker can emit an error. In Lua, variables cannot be declared (except for local variables), so this check cannot be carried out in Pirate. Type checking makes sure that a variable of some type is only assigned values of that type. For example, the following fragment of C code would generate an error:

```
int i;         /* declare a variable of type integer */
i = 1.234;     /* error!  i is integer, no floating-point!  */
```

However, because Lua variables do not have a type (only values do), this kind of check is not possible in Pirate. Furthermore, it is possible in Lua to extend the semantics of the language. For example, in normal cases adding two tables will result in an error. In Lua it is possible to prevent this error by creating functions that *do* add two tables. These functions are called *tag methods* and are discussed in detail in Chapter 5.

Scope checking can be seen as a part of checking variables for declaration, but in Lua, this is not the case. Lua has an unconventional feature, called *upvalues*. The scope checker in Pirate checks if upvalues are used in the correct way. In short, an upvalue is a special copy of a variable in an enclosing function. Upvalues are discussed in detail in Chapter 4

### 2.2.5 Code Generator

The code generator module is, of course, responsible for generating code. During a traversal of the Abstract Syntax Tree, code is generated based on the information that is saved in the tree. For each node, instructions are selected that represent the meaning of that node. For example, a `while` statement can be represented as a particular node in the tree, that has pointers to a `condition` node and a `block` node. The *code template* for a `while` statement is shown in figure 2.2. The target language of Pirate is *Intermediate Machine Code* (IMC). This IMC can then be compiled to *Parrot Byte Code* (PBC) by the IMC compiler (IMCC). In fact, it is possible to feed IMC to IMCC and run it immediately after compiling. In that case, no separate call to the Parrot interpreter is needed. The code generator is described in more detail in Chapter 6.

```
while condition do statements end →
    begin:
        code for condition
        if condition == false goto end
        code for statements
        goto begin
    end:
```

Figure 2.2: The code template for a `while` statement.

### 2.2.6 Code Optimizer

The code optimizer module performs some optimizations on the generated code. It is implemented as a peephole optimizer. This means that no global optimizations are done, but only small instruction sequences are optimized. The code optimizer is described in more detail in Chapter 7.

## 2.3 Compiler Data-flow

As compilation progresses, the representation of the source program changes. This section gives an overview of the data-flow in the Lua Compiler.

In general, a compiler organization can be designed to process the source program in one of two ways. The rest of this section describes these possibilities and the chosen organization for Pirate.

### 2.3.1 One-pass compilers

The first possibility is to read the source program only once and immediately generate code. In this kind of organization there is no intermediate representation (IR) of the source program. This kind of organization is commonly known as a *one-pass compiler.*

### 2.3.2 Multi-pass compilers

The other possibility is to read the source program and generate an explicit intermediate representation (IR). Compilers with this type of organization are called *multi-pass compilers.*

In this kind of organization the source program is processed in multiple phases. Each phase has one or more tasks. The representation of the source program may change as the different phases process it. There are many different types of intermediate representations possible. Well known are:

- Abstract Syntax Tree (AST)
- Postfix notation
- Three-address code

An AST is just a tree data structure that contains nodes that correspond to objects in the programming language. The Lua compiler uses an AST as its IR. Section 3.3 describes the AST in more detail.

In postfix notation, the operator of an expression is placed *after* its operands. Using postfix notation removes the need for parentheses. Postfix notation is useful when a machine with a stack-based architecture is targeted.

Three-address code is an abstracted form of assembly code. Each instruction has an operator and three addresses; two addresses for the operands and one address for the result location. The target language of Pirate (IMC) is a kind of three-address code, so the instructions generated are of this kind.

### 2.3.3 Choosing an Organization

When an organization must be chosen, there are a number of issues that should be addressed. Some of these are:

- **Need for multiple passes** Sometimes, it is not possible to do compilation in one pass. This is the case when identifiers may be used *before* they are declared.
- **Compilation speed** A one-pass compiler will normally be faster, because only one pass over the source program is made. No data-structures have to be created (during run-time).
- **Memory usage** If the complete source program is represented in memory by an intermediate representation, such as an AST, then this may require much memory. On the other hand, if the compiler itself is large, then this may require a lot of memory too.
- **Compiler modularity** Obviously, a multi-pass compiler will be more modular than a one-pass compiler. If the language that is compiled is changed, adapting the compiler is easier, because finding the parts of the compiler that must be changed is simpler.
- **Quality of the generated code** The generated code in a multi-pass compiler can be of a higher quality, but this is not necessarily the case. Because all information about the usage of variables is in the intermediate representation, it is possible to take advantage of that. It is also possible for the compiler to do a separate optimizing pass, to remove redundant instructions.

For Pirate, modularity was the most important motive. That is why Pirate was designed to be a multi-pass compiler. The rest of this section will discuss the phases of Pirate in more detail.

### 2.3.4 Phases of Pirate

Pirate is a multi-pass compiler. Most probably, it would have been possible to write it as a one-pass compiler. However, one of the main design decisions was to keep it modular. The compiler was not built to be a production compiler, nor was it meant to be highly efficient.

The Lua compiler uses four passes to generate an executable file from the source program. This means that some form of the source program is processed 4 times before an executable file is generated. The four phases in Pirate are:

- Parsing Phase
- Scope checking Phase
- Code generation Phase
- Optimizing Phase

This organization is shown in figure 2.3. Each phase will now be described in short.

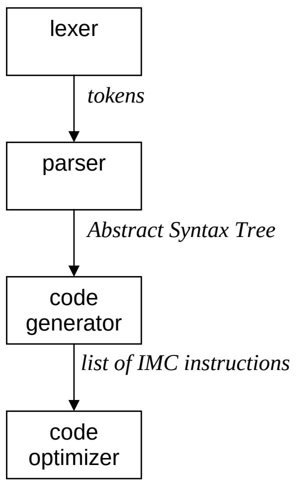


Figure 2.3: Data-flow in the Lua compiler

### Parsing Phase

In the first phase of the compiler, the source program is being processed by the parser and lexer. This is done interleaved, so every time the parser needs a new token, the Lexer is invoked. During parsing, an intermediate representation of the source program is built, in the form of an Abstract Syntax Tree. For every language construct, one or more nodes are created. This AST can then be processed by the next phases.

### Scope checking Phase

In the scope checking phase, the AST is traversed. All *local* identifiers are entered into a symbol table. With help of this symbol table, all identifiers are checked for their scope. After this phase has completed, the symbol table is discarded. The AST is still needed in the next phase.

### Code generation Phase

Code is generated during a traversal of the AST. The generated instructions are organized in functions. All functions are stored in a list. For each function, there is one node, and this function node has a list of instructions. The *main* program is represented as a function called `main`. After this phase, the AST can be discarded, and the source program is now represented as a list of functions, each function keeping a list of instructions.

### Optimizing Phase

The code generator is designed to be as simple as possible. That has implications for the generated code. It is possible that some instructions are redundant, because the code generator does not do any optimizations. For example, if a value is stored in some variable, and the next instruction fetches that same variable, the fetch instruction can be removed.

Part II

# Compiler Front-end

# Chapter 3

# Syntactical Analysis

During syntactical analysis the source program is parsed. The parser verifies that there are no errors in the structure of the source program. This phase can be further divided into two sub-phases. The first sub-phase is the *lexer*. The lexer reads the characters as they are typed by the programmer. The lexer groups this character stream into units called *tokens*. The tokens are then processed by the second sub-phase of syntactical analysis: the *parser*.

This chapter will describe the syntactical structure of the Lua programming language and how syntactical analysis is implemented. First, in section 3.1, the implementation of the Lexer is described. Then in section 3.2, implementation of the parser is discussed. After that, section 3.3 describes the construction of the Abstract Syntax Tree that is built during parsing. Section 3.4 will discuss error-handling during parsing.

## 3.1 The Lexer

The task of the lexical analyzer (often called *scanner* or *lexer*) is to read in the source file and produce a stream of tokens. The produced stream of tokens will be used by the parser to do syntactical analysis. The lexer also removes white space and comments. In this way, the parser can assume that all tokens are valid in a way, which makes constructing a parser easier. For example, the parser does not have to check for comments anymore, because the lexer removes them.

The rest of this section will discuss what is needed to create a lexer. First, it is necessary to decide which tokens must be recognized This is done in subsection 3.1.1. After that, the lexer can be built.

### 3.1.1 Specifying Tokens

For the lexer to be able to recognize groups of characters as tokens, it is necessary to set up some rules. Not any group of characters is a valid token. For example, in many programming languages a variable identifier should begin with a letter (as opposed to a digit). These rules can easily be expressed using *regular expressions.* Regular expressions are a formal way to describe the set of strings that may be generated. Below is a short and informal introduction to regular expressions.

**Regular Expressions**

A regular expression is a description of a pattern. In describing these patterns, some formal notation is used. For example, the following regular expression denotes all strings that consist of one or more letters:

```
[a-zA-Z]+
```

The brackets denote a class of characters that may be matched. So, the letters a to z lowercase and uppercase may be matched. The plus sign denotes that one or more letters may be matched.

Lua identifiers may consist of letters, digits and underscores, but it must start with a letter or underscore. The regular expression that expresses these rules is shown below.

```
[a-zA-Z_][a-zA-Z_0-9]*
```

The asterisk sign means zero or more copies of the preceding expression.

A more formal and complete description of regular expressions can be found in any good book on compilers, for example [11].

**Keywords**

Other tokens may have special meaning in the source code. These are *keywords.* For example, in Lua the word `while` is a keyword. In fact, all keywords in Lua are reserved. That means a keyword may not be used as an identifier for some kind of object.

### 3.1.2 Building the Lexer

The lexer was automatically generated using the program Flex. Flex takes an input file in a specific format and generates a C source file that can be linked to the main program. Although writing a lexer by hand is quite straightforward, an automatically generated lexer is easier to adapt. Moreover, when one knows how to use Flex, a lexer can be built within only a few hours.

When the lexer recognizes a token, depending on the type of token it does one or two things. If the lexer recognized a keyword, it just returns the *type* of the token. For example, if it recognizes the keyword `while`, it returns a code representing that keyword. On the other hand, if the lexer recognized an identifier, it first saves the actual value of the identifier in a variable that the parser can access, and then returns a code indicating that it recognized an identifier. The same is true for numbers.

More information on Flex can be found in [14].

## 3.2 The Parser

The parser is responsible for finding the syntactic structure of the source program. Each sentence of the source program consists of a number of tokens. These tokens are delivered by the scanner. As the syntactic structure of the program is recognized, an internal representation is built that will be processed by later phases of the compiler. In a one-pass compiler this is of course not necessary, because there is only one pass through the source program.

### 3.2.1 Grammars

The syntactic structure of a programming language and programs written in it can be defined by a *context-free grammar.* A formal definition of a context-free grammar will now be given, to introduce some formal terminology. A context-free grammar is defined by four components:

1. A finite terminal vocabulary. The terminals are the tokens that are returned by the scanner.
2. A finite nonterminal vocabulary. These are intermediate symbols or units, that can consist of other nonterminals and terminals.
3. A start symbol where all derivations of the grammar rules begins.
4. A finite set of production rules. A production rule describes how some set of terminals and nonterminals can be rewritten as another nonterminal. In 2.2.2 an example of a production rule was given, that defined the structure of a `while` statement.

Figure 3.1 shows the definition of a very simple language, which will be used in the rest of this section. In this way, discussion of parsing techniques can be more concrete. The language is *very* limited, because a program can only contain one statement: a print statement or an assignment statement. The grammar is kept small, because it would get too big when more features were added. In this grammar, the word *expression* for example is a *nonterminal*, that is, that symbol cannot be delivered by the lexer, but it is an intermediate symbol used in a larger context. The `+` symbol on the other hand, is a *terminal.* That symbol can be delivered by the lexer. The start symbol in this little grammar is the symbol *program.*

| | |
|---|---|
| *program* | → `begin` *statement* `end` |
| *statement* | → *printstatement* \| *assignstatement* |
| *printstatement* | → `print (` *string* `)` |
| *assignstatement* | → *id* `=` *expression* |
| *expression* | → *number operator number* |
| *operator* | → `+`[a] |

[a] Only one operator is defined to prevent the issue of precedence.

Figure 3.1: Grammar for a very simple language.

Let us assume that there is a lexer that can recognize *id*s and *number*s. Some example 'programs' are:

```
begin print("Hello, World!") end
begin x = 10 + 20 end
```

There are many parsing algorithms, but there are only two basic strategies. The first is *top-down* parsing. One could think of top-down parsing as constructing the syntax tree starting at the root of the tree. Top-down parsing is discussed in 3.2.2.

The other basic parsing algorithm is *bottom-up* parsing. Using this algorithm, the syntax tree would be constructed starting at the leaves of the tree. Bottom-up parsing is discussed in 3.2.3.

It is important to realize that it is not necessary to actually create a syntax tree. In a one-pass compiler, this will not happen. However, because it makes the discussion somewhat easier and Pirate does create a syntax tree, in this discussion the assumption is made that a syntax tree will be created.

### 3.2.2 Top-down Parsing

This section describes the basic ideas behind top-down parsing. Let us see how the second example program will be parsed using a top-down parsing algorithm. Let us assume that keywords are recognized and processed by the parser, but they are not considered in this example.

```
begin x = 10 + 20 end
```

Top-down parsing starts creating the AST at its root, so the parser starts with creating a *program* node (figure 3.2(a)). Then, the parser must choose which production for *statement* must be applied. The statement in this program starts with `x`, which is recognized by the lexer as an *id*, so the parser will recognize the statement as an *assignstatement* (figure 3.2(b)).

Then, a node for this *id* will be created (figure 3.2(c)). The other part of an assignstatement is an *expression*, so an *expression* node is created (figure 3.2(d)).

Now, the next input token is a number, so a node for this number is created

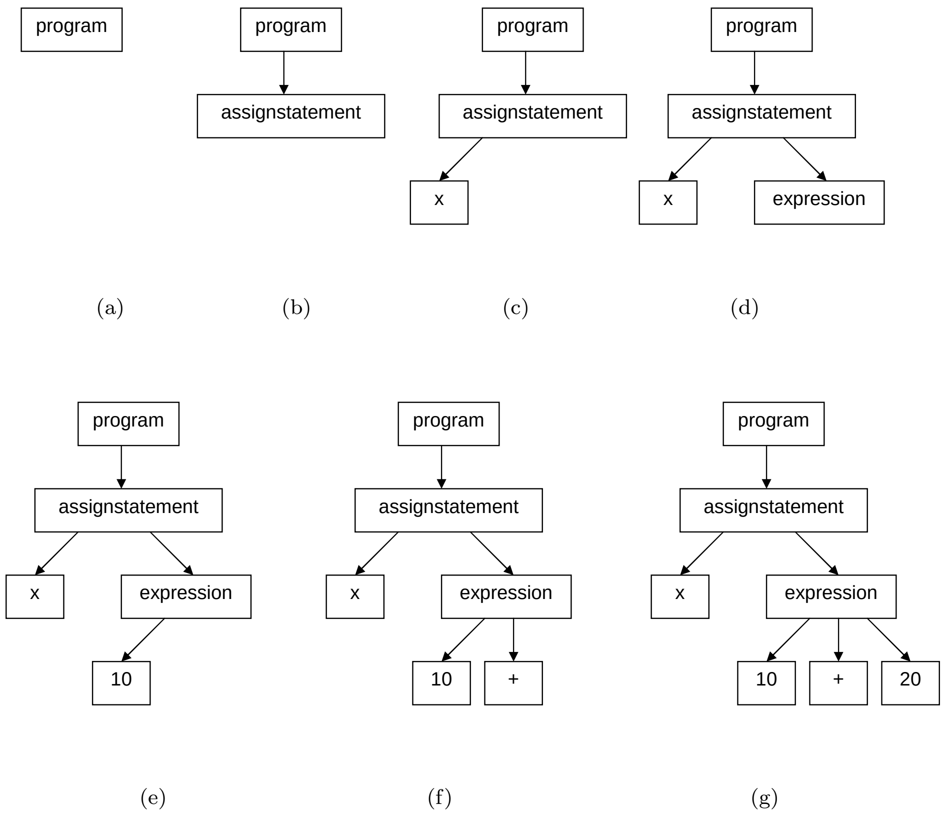


Figure 3.2: Top-down parsing.

(figure 3.2(e)). After that, the operator is recognized, so a node for the operator is created (figure 3.2(f)). Then, another number is recognized, so a node for that number is created too (figure 3.2(g)).

Parsers that use a top-down strategy can easily be implemented as a *recursive descent* parser. A recursive-descent parser has a subroutine for each nonterminal of the grammar. Then, the parsing starts with invoking the subroutine that parses the program. Whenever a production rule is applied, the parse subroutine for that production rule is called. For a more detailed example with code, see appendix B.

If the production rules are mutual recursive, then the parsing functions will be too. The parser will call a subroutine for each production rule it applies. So, the parser will get 'deeper' in the function call chain every time a function is called. It *descends* down the syntax tree of the program. Hence the name of this type of parsers.

Recursive-descent parsers cannot parse just any grammar. The grammar must be in the class of LL*(1) grammars.* LL(1) stands for a class of grammars that can be parsed, reading the source from left to right, producing a leftmost derivation, and using only one symbol of lookahead. This symbol of lookahead is used to decide on which production rule should be applied.

Recursive-descent parsing is not the only top-down algorithm. However, it is easy to understand for beginners. Other top-down algorithms can be found in some more comprehensive books on compiler theory, such as [10].

### 3.2.3 Bottom-up Parsing

A bottom-up parser constructs a syntax tree starting at the leaves of the tree. So, instead of *starting* with creating a *program* node, now the parser *finishes* with creating this node. We will parse the same program as in 3.2.2, but now using a bottom-up technique. Again, consider the program:

```
begin x = 10 + 20 end
```

Again, keywords are ignored in this example. They are processed by the parser, but nothing is done with them. Now, the first symbol is `x`, which is recognized by the lexer as an identifier. So, a corresponding node is created (figure 3.3(a)). The parser cannot apply any production rules, so the node is stored somewhere. The next input symbol is `10`, which is a number. So, a number node is created (figure 3.3(b)). Again, no production rule can be applied, so this node is stored too. Then, the addition symbol is recognized, so an operator node is created for that (figure 3.3(c)). Still no production rule can be applied. After that, another number is parsed, so again a number node is created (figure 3.3(d)). Now, the last three symbols, `10 + 20`, match the right-hand side of the production rule for *expression.* So, the created nodes for these symbols are combined, or *reduced*, into one node for expression (figure 3.3(e)). But now, the complete right-hand side of the production rule for *assignstatement* is matched. So, again the symbols representing this right-hand side are reduced to one node representing an assignment statement (figure 3.3(f)).

And again, the node representing an *assignstatement* can be reduced to a *statement*, and eventually a *program* symbol (figure 3.3(g)).

Grammars that can be parsed by a bottom-up parser belong to the LR*(k)* class of grammars. The LR technique reads the input left to right, producing a rightmost derivation. During parsing, there are $k$ symbols of lookahead that are used in making parsing decisions. In practice, $k$ is usually 1.

Now, a short description of the fundamental internals of an LR parser is given. A bottom-up parser has a parse stack. Tokens are parsed and pushed, or *shifted* onto the stack until some series of tokens matches the right-hand side of a production rule. If this is the case the right-hand side, often called a *handle*, is said to be *reduced* to the nonterminal of which the handle was matched.

The tokens that are parsed are not actually pushed onto the parse stack themselves, but rather a *state* representing them in a certain context. This is because some terminals may occur in several contexts, so a state combines

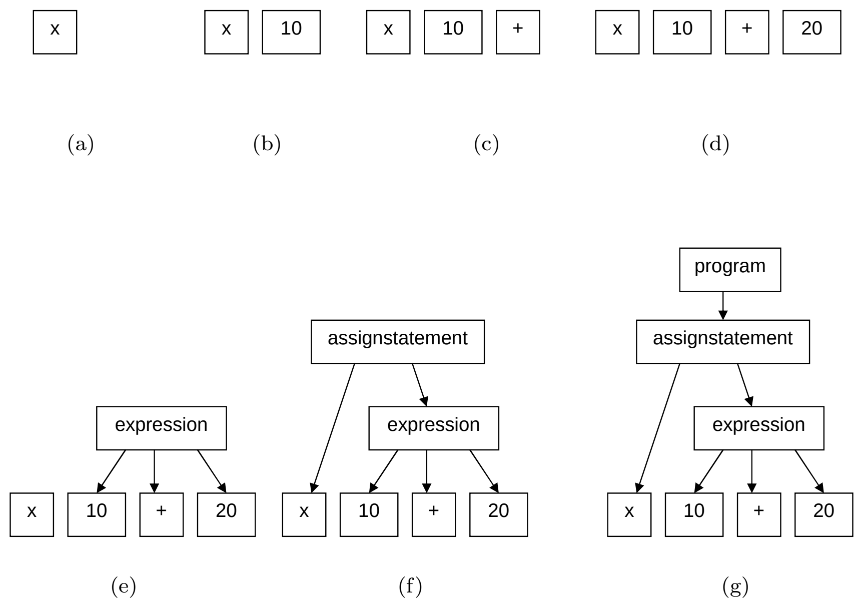


Figure 3.3: Bottom-up parsing.

the type of token with the context it appears in. Now, for the parser to decide whether to shift or reduce, the parser keeps an *action* table. A *goto* table defines the next state based on a certain current state.

This description of LR parsers is a global one, it does not discuss all kinds of details. These details, as well as a more comprehensive discussion of LR grammar restrictions, can be found in [10] and [11]. Constructing an LR parser is usually done with an LR parser generator. This is because writing an LR parser by hand would be too much work. A well-known parser generator is YACC. This program takes a grammar file and creates C source code for a parser for that grammar.

### 3.2.4 Building the Parser

The parser of Pirate is an LR(1) parser. Actually, it is a LALR(1) parser. LALR is an acronym for Look Ahead LR. In reality, all LR parsers, except LR(0), are 'Look Ahead' LR parsers.

**Bison**

The parser was generated using Bison, a GNU replacement for YACC[1]. The file processed by Bison is a specification of the grammar in a particular format.

For a compiler to be useful, the parser must do something more than just parse the source of a program. In Bison (or YACC for that matter), a grammar rule can have a semantic action associated with it. When a certain handle is reduced, the semantic action for that rule is executed. In the Lua parser, the semantic actions construct the Abstract Syntax Tree. The AST is discussed in 3.3. In figure 3.4 a small part of the Lua grammar is shown. Note that this

```
primary  :  LITERAL  { /* action to handle this literal */ }
         |  NUMBER   { /* action to handle this number */ };
```

Figure 3.4: Small part of the Lua grammar input for Bison.

production rule in this figure was simplified. Also, the semantic actions are left out, these are discussed in section 3.3. This production rule states that a `primary` is either a `literal` or a `number`. When a `literal` is reduced to the nonterminal `primary`, the action for that rule is executed.

To build the parser for Pirate, all production rules are entered into a file in a format that Bison accepts. The output of Bison is a C source file representing the parser. This parser can then be invoked by calling the function `yyparse`[2]. More information on Bison can be found in [2] and [14].

## 3.3 The Abstract Syntax Tree

The Lua compiler constructs an Abstract Syntax Tree during the parsing phase. This section describes the details of this AST and how it is implemented.

### 3.3.1 Structure of the Tree

The structure of the AST corresponds to the syntactic structure of Lua. For example, a node for an assignment statement has two child nodes: a node for the left-hand side, and a node for the right-hand side of the assignment. So, for each type of statement and expression there is a special type of node.

### 3.3.2 TreeCC

The code for the AST and its operations is generated by a program, called `TreeCC` (Tree Compiler Compiler). TreeCC was designed to assist in developing compilers and other language-based tools. Because the source code for an AST

[1]Bison is not 100% compatible with YACC; the programs have some very small differences. See [2] for more details.

[2]This is the standard name of the parse function. It is possible to change this.

can get quite extensive, it is easy to make errors or forget things. For each type of node, there must be an implementation for every operation that is defined for the AST. The TreeCC program checks that for every type of node, there is such an implementation. More information about TreeCC can be found in [3].

TreeCC works more or less like Bison. A specification for the tree and its operations is written and fed to the program. The program creates a C source file that can be linked to the main program. For each type of node there is an appropriate constructor function.

### 3.3.3 Building the Abstract Syntax Tree

The last section mentioned semantic actions associated with production rules. In Pirate, the AST is built as a semantic action. Figure 3.5 shows how the tree nodes are built within the parser.

```
primary  :  LITERAL         { $$ = Literal_create($1); }
         |  NUMBER          { $$ = Number_create($1); };

var      :  PRIMARY INDEX   { $$ = Var_create($1, $2); }
         |  (...)
```

Figure 3.5: Part of the grammar with semantic actions.

The symbols `$$` and `$1` are identifiers that represent the values on the left-hand side and the right-hand side, respectively. If a `literal` is reduced to a `primary`, then a node for a `literal` is created. The constructor for a `literal` node is called with the actual value of the `literal` as an argument. After that, the node representing the primary is used in the constructor for a `var` node. This node is referenced as `$1` in the production rule for `var`. All nodes of the tree are built in a similar way.

## 3.4 Reporting Syntax Errors

Error reporting is a very important issue in compiler design. This is because a compiler will not only compile correct programs. On the contrary, many programs it will process are erroneous.

The parser generated by Bison automatically has an error function that is called when a syntax error is discovered. The global Compiler State structure, that holds all important global variables of the compiler, has some fields for error reporting. The layout of this data structure can be found in appendix D.1. The fields that are important for reporting syntax errors are shown in figure 3.6.

The field `linenumber` keeps track of the current linenumber. This field is updated by the lexer. The field `linebuffer` is a pointer to a buffer holding the current source line. The field `tokenpos` keeps track of the position in the current source line. Each time a token is parsed successfully, `tokenpos` is increased by

| field | meaning |
|---|---|
| `long linenumber;` | current line number during parsing |
| `char *linebuffer;` | buffer holding the current line being parsed |
| `int tokenpos;` | current position in `linebuffer` |

Figure 3.6: Variables for error reporting

the length of that token. When a syntax error is discovered, the error function displays an error message. This message prints the current linenumber, the current source line and it can indicate where in the source line the error was discovered. An example is shown below.

Input program:
(line numbers are displayed only for the purpose of this example)

```
1:  function f()
2:     x == 1 + 2
3:  end
```

The resulting error message will display:

```
Error in line 2:
syntax error

x == 1 + 2
   ^
```

So, the erroneous source line is printed, together with a mark pointing at the erroneous token.

When an error is detected, it is usually desirable to attempt to recover from the error and continue parsing to discover more errors (if any). Bison has a mechanism for recovering from errors. This mechanism uses special *error* productions that may be inserted into the grammar. When an error production is applied, the parser can try to synchronize and continue parsing. However, these error tokens cannot be inserted just anywhere, because they may introduce errors in the grammar[3]. The Lua grammar has only one error token. In the future, a better error recovering mechanism is planned.

Because the error recovering mechanism in Pirate is not very good, it may happen that one syntax error may be caught as an error multiple times, so that the compiler will report two or more errors, while there was just one error in the source.

[3]These errors are called **reduce/reduce** conflicts. More information about this type of conflicts can be found in [14].

# Chapter 4

# Scope Analysis

Scope analysis verifies that all used variables are valid. In languages that demand declaration of variables before use, this analysis will be more complex. Because Lua does not have declarations of global variables, this analysis would not have been necessary. However, Lua does have some other construct, called *upvalues*. In section 4.1 this feature and its effects will be discussed.

During scope analysis, it is necessary to store the names of identifiers, and possibly some other information about these identifiers. In Pirate, a *symbol table* is used to store this information. Section 4.2 will discuss symbol tables.

## 4.1 Lua Scope rules

Lua has few but unconventional scope rules. A variable in Lua is global, unless it is explicitly declared local. Parameters work as local variables. A function body has access to its own local variables and all global variables. However, a global variable can be hidden by a local variable with the same name, as is shown below.

```
a = 123
function f()
    local a = "hello"
    print(a) -- will display "hello", not 123
end
```

A function cannot access local variables from an enclosing function directly. That is, a nested function cannot access local variables from the function in which it is nested. So, the following is not allowed:

```
function f()
    local a
    function g()
       print(a) -- cannot access local in enclosing scope!
    end
end
```

A global variable called 'a' (if any) would be hidden by the local declaration `local a`.

However, a function can access the value of a local variable from the immediately enclosing function, using an *upvalue*. Accessing an upvalue is done with special syntax: '%' *name*. Thus, the following is legal:

```
function f()
    local a
    function g()
       print(%a) -- OK!
    end
end
```

Note that an upvalue can only be used for accessing variables in the *immediately enclosing* function. For transferring values over more levels of nesting, each level needs a copy of this upvalue, as is shown below:

```
function f()
    local a = 123
    function g()
       local a = %a
       function h()
          local a = %a
          function i()
             print(%a) -- this will print "123"
          end
       end
    end
end
```

Using an upvalue is a method to access the *value* of a variable. When the function in which it appears is instantiated, the value of the upvalue variable is *frozen*. This means that the variable is *read-only*. So, the following is not allowed:

```
function f()
   local a
   function g()
    %a = 123 -- cannot assign to upvalue!
   end
end
```

However, assigning to fields of an upvalue of a table is allowed. This is because not the actual table is stored in the table variable, but a *reference* to it. Although this reference is fixed when function g was instantiated, the data it references is not. This means that its fields can be accessed in the normal way. So, the following is legal:

```
function f()
   local a = {} -- create a table
   function g()
    %a.some_field = 123 -- %a is fixed, but its fields are not!
   end
end
```

## 4.2 The Symbol Table

The symbol table is used during scope analysis. Its use is somewhat unconventional, because in most compilers, a symbol table is used to store *all* identifiers, whereas Pirate uses it only for *local* identifiers. A symbol is stored as a string, together with the current scope level.

### 4.2.1 Use of the symbol table

If Lua did not have the upvalue feature, then no scope checking would be needed. A variable would be either global, if no local declaration was present, or if there was a local declaration, the variable would be local. However, because Lua has upvalues, and there is a restriction when accessing upvalues, this restriction must be checked.

Scope checking is done as described in the rest of this section.

One can view the main program as having a base scope level. Let us call this base scope level `L` for a moment. A function that is 'nested' in this main program, has scope level `L+1`. A function that is nested within another function that has scope level `n`, then has scope level `n+1`. Below is a simple example to make this clear.

```
local x = 1 + 2 -- x has base scope level L (x is local)
function f()
    local y = "hello" -- y has scope level L+1
    function g()
       local z = 123 -- z has scope level L+2
    end
end
```

Note that *global* variables, unlike *local variables*, always have base scope level.

Understanding this scope level organization, implementing a scope checker is very easy. The scope checker does a traversal of the abstract syntax tree, that was created during parsing. Whenever a new function is defined, a new scope level is created in the symbol table. When an identifier is seen, its name is looked up in the table. If it is not found, then it must be a global (as global identifiers are not entered). On the other hand, when the identifier is found, then the scope level of the symbol is compared to the *current* scope level. If they are *not* equal, then the scope checker found a scope error.

Upvalues are handled in a similar way. An upvalue is looked up in the symbol table. If it is not present, then that will result in an error. Otherwise, the scope level of the symbol should be 1 less than the current scope level. If this is not the case, then an error is found.

### 4.2.2 The Symbol Table

There are many possible implementations for a symbol table. In choosing an implementation, one should consider speed of operations and the number of identifiers that should be stored. Some possible implementations are listed below.

- List
- Binary Tree
- Hash Table

The listed alternatives will now be discussed in short.

#### List

A list is one of the simplest data-structures. It can be implemented using an array, but in most languages arrays are of a fixed size, which limits the number of entries. To avoid that, a linked list can be used. When using a list as a symbol table, two possible organizations are possible: *unordered* and *ordered.*

*Unordered List*
Entering a new identifier is straight forward and takes constant time: the identifier is placed into the next available position. However, finding an identifier is too slow when there are many entries. The running time of finding an identifier is $O(N)$.

*Ordered List*
When using an ordered list, finding an identifier can be done using a binary search. The running time for a binary search is $O(logN)$, so finding an identifier is faster than in an unordered list, especially when there are many entries. However, now entering a new identifier is costly. The identifier must be entered into the right position, and, if an array is used, all following entries should be moved forward one position.

### Binary Tree

A binary tree is a conceptually simple data-structure. An entry is stored as a *node*, and each node (except for the leaf nodes) keeps a pointer to two *child* nodes (hence, a *binary* tree). A binary tree is interesting, in that it combines fast insertion with fast searching of identifiers. Both operations have an average running time of $O(logN)$, assuming the tree is *balanced*! However, if the tree grows like a linked list, then performance will suffer. In that case, both entering and searching will use more time than using a list.

### Hash Table

A hash table is a table of fixed size, in which each entry is mapped to a particular position. Mapping is done using a hash function. A hash function should be fast and should avoid clustering. That is, similar (but not equal) names should map to different positions in the table.

Because there is a limited number of entries, it is unavoidable that multiple items will yield the same hash value (with enough identifiers being entered, of course). There are a number of methods to resolve these *collisions*. One of them is to rehash the result with some other hash function. However, the chance that the new position is already taken, too, is not small. A simpler solution is to link all entries that have the same hash value as a linked list. However, this results in a small performance hit, especially when the number of identifiers is significantly bigger than the size of the table. On the other hand, if each bucket contains a list with a constant number of entries, then access time is more or less constant, which is better than $O(logN)$.

Implementations of these data structures, their advantages, disadvantages and their properties can be found in any book on data structures.

The symbol table in Pirate is implemented as a hash table. The compiler can keep track of the number of identifiers returned in the lexer. Because the symbol table is created after the parsing phase, it can be created with a size that is based on the number of identifiers counted in the lexer. This way, the number of collisions can be kept small, so that entries can be accessed in more or less constant time.

Appendix D.2 describes the implementation of the symbol table used in Pirate.

## 4.3 Reporting Scope Errors

Unlike the parser, the scope checker has no access to a variable keeping track of the current linenumber. That is, the variable in the global compiler state object is invalid, because its value is the number of the last source line. So, there must be some other way to find out the linenumber in which a scope error was detected. Fortunately, during the construction of the AST, this information is saved in the tree. Each node has a field that keeps the linenumber of the syntactic unit for which the node was created.

If a scope error was detected, a function is called that displays the error. The format of the error message is a bit different than the error message that is displayed for a syntax error. An example is shown below.

Input program (saved in a file called 'error.lua'):
(line numbers are displayed only for the purpose of this example)

```
 1:  function f(y)
 2:     x = %y -- attempt to assign the upvalue y to x
 3:  end
```

The resulting error message will display:

```
Error in file:  'error.lua', line 2:
Upvalue must be global or local to immediate outer scope
Inaccessible variable:  'y'
```

When reporting scope errors, it is not possible to print the original source line, because the source is no longer available. It would cost too much memory if the source was saved in the tree as well. This is one disadvantage of implementing the scope checker as a separate pass in the compiler. However, simplicity of implementation was considered more important than preventing this disadvantage. Furthermore, the source line number *is* printed, so that should help the Lua programmer on her way.

## Part III

# Compiler Back-end

# Chapter 5

# Parrot Run-Time Environment

## 5.1 Introduction

Before generating code, it is necessary to have a look at the run-time environment in which the generated code must be executed. Because Lua is targeted to a 'strange' machine, that was not specially designed for Lua, we must design support routines that emulate a Lua Virtual Machine.

First, we need to decide how variables will be represented in the Parrot Virtual Machine. This is discussed in section 5.2 Then decisions must be made on where to store these variables. This is done in section 5.3.

## 5.2 Implementation of Lua Data Types

This section will discuss how Lua variables will be represented in the Parrot Virtual Machine. Parrot is designed to handle dynamic languages, such as Lua. For that purpose, there is a special built-in data type. This section gives an introduction to Parrot Magic Cookies (PMC) and how they work. After that, implementation of the Lua data types as PMCs is described.

### 5.2.1 Parrot Magic Cookies

A Parrot Magic Cookie (PMC) is a completely abstracted data type. It can be anything, and it has a standard way to handle data in the PMC. A PMC should be seen as a data container and a set of operations to operate on that data. The data can represent a number, a string, a function or something else. The data may also be some kind of aggregate, like an array or a hash.

A PMC behaves as it is programmed by its implementor. When a new language is targeted to Parrot, it may be necessary for the compiler writer to create new data types that behave in a specific way. Section 5.2.3 discusses the PMC types that are created for the Lua language.

PMC objects can change their own type during run-time. This makes implementing a dynamically typed language much easier. For example, when one assigns a string to a LuaNumber, the LuaNumber PMC will change its type to a LuaString object.

Unlike 'primitive' data types, like numbers, PMCs are not stored by *value*, but by *reference*. This is important to realize, because this has consequences for the code that must be generated to get the desired behaviour.

### 5.2.2 PMC operations

The thing that makes all PMC objects equal is their set of operations. All PMC objects have the same set of operations. However, a PMC does not necessarily have an implementation for some operation. For example, the LuaString PMC type does not have an implementation for the `invoke` operation, but the LuaFunction PMC type does. When a programmer tries to invoke a LuaString object, a run-time error will result.

Each PMC has a table with functions that can be invoked. This table is called a *vtable*. When a vtable function is called, the appropriate function is called for that PMC. For example, this means that when the increment operation is called for a PerlNum object (which will represent a Perl number in Parrot), its numeric value will be incremented. When the same operation is called for a PerlString object, the value of the string is incremented, i.e. "a" will become "b", because "b" is the successor of "a". This behaviour is specific for Perl and is defined by the PMC implementor.

A PMC does not have to be a scalar, it can also be a more complex type, such as an array or hash table. For these kinds of objects, there are special keyed instructions. In this way it is possible to index a PMC with numbers (for arrays) or with strings (for hash tables). These operations are, of course, not implemented for data types like a number.

### 5.2.3 PMC types

As said before, a PMC can be anything. A PMC is not written by an application programmer, but more likely by a language implementor and by the designers of Parrot. The Parrot distribution has a number of PMC classes included, for example a Sub class to represent a subroutine. Because the standard PMC classes that come with the Parrot distribution do not behave as Lua variables, it was necessary to implement some PMC classes specially for Pirate. It may be that a PerlNum PMC for example behaves exactly like a LuaNumber, but all Lua variables should be initialized with a *tag* property. This is programmed into the PMC, so all Lua data types must be implemented as special PMC classes to allow for that.

The rest of this section will describe the PMC classes that are written for Pirate.

### 5.2.4 LuaFunction

A function in Lua is not only a representation of a function, but also a container for its upvalues. When a function is instantiated, all upvalues are fixed and stored somewhere. Because each instance of a function has its own instances of upvalues, it makes sense to store these upvalues in the function object. So, a LuaFunction PMC consists of two parts, a part that holds a function, and a part that holds the values of its upvalues. The function is just an instance of the built-in Sub PMC. The upvalues are stored in a Hash table.

The interface for the LuaFunction PMC is in fact the same as the combined interfaces for a Sub PMC and a Hash PMC.

### 5.2.5 LuaTable

A LuaTable is an associative array. This means that it can be indexed with numbers and with other values, like strings. Parrot does have an aggregate type that can be indexed with numbers (PerlArray) and an aggregate type that can be indexed with strings (PerlHash), but neither of them can be indexed with both[1].

A LuaTable is implemented as a PMC that keeps two fields. One field is a PerlArray, the other is a hash table. Whenever a LuaTable is indexed with a string, it stores the value in the hash part. Likewise, if it is indexed with a number, the array is used.

### 5.2.6 LuaString

The LuaString PMC is implemented as a subclass of a PerlString PMC. Some methods were redefined to get the desired effects. For example, when assigning a number to a LuaString variable, it will change its type to a LuaNumber instead of a PerlNum.

### 5.2.7 LuaNumber

The LuaNumber PMC is implemented as a subclass of a PerlNum PMC. Just as in the LuaString PMC, some methods were redefined. If a LuaNumber is assigned a string, it will change its type to a LuaString.

### 5.2.8 LuaNil

Any variable that is used in Lua, but was not given a value, has the value `nil`. The LuaNil PMC can be seen as an uninitialized variable. It changes its own type, whenever a value is assigned to it. This means that it will change to a LuaString when a string is assigned to it, and to a LuaNumber if a number is assigned to it.

[1]A PerlHash can be indexed with a Key PMC that represents a number or a string, but then a Key must be created first.

### 5.2.9 LuaUserdata

In the original Lua interpreter, the *userdata* type of Lua represents a C pointer. At the moment of writing, it is not clear if it is possible and makes sense to store C pointers in Parrot. If it does, a pointer can easily be implemented as a LuaUserdata PMC. However, at the moment there is not such an implementation.

## 5.3 Run-Time Storage Organization

All values in the program should be stored somewhere. Where some value should be stored, depends on what its scope is and whether the value is an actual parameter (often called *argument*). This section discusses the different types of storage that are available on Parrot, and how they are used.

### 5.3.1 Registers

Whenever there must be something done with some variable, it must be stored in some place that the processor[2] can easily access, so the operation can be executed fast. For this purpose, Parrot has a set of registers. All instructions processing data use registers as their operands.

### 5.3.2 Storing Global Variables

All variables are global, unless declared local or as a parameter. This means there must be some space to store these global variables, a place that can be accessed throughout the whole program. On a real (hardware) machine, these globals would be stored in main memory.

The Parrot Virtual Machine stores its global variables in a global symbol table. Parrot can access this global symbol table fast.

### 5.3.3 Storing Local Variables

Local variables are different from global variables in that they have limited scope to some block. Parrot has facilities to store local variables in an area called a *scratchpad*. Whenever a new scope block is entered, a new scratchpad is created. All local variables can be stored into such a scratchpad. When the block is closed, the scratchpad can easily be disposed. These scratchpads are stored on a stack like structure.

### 5.3.4 Storing Parameters

When a function is called with a set of actual parameters, often called *arguments*, these arguments should be stored in a place that the called function can easily access. There is a small number of alternatives for storing parameters.

[2]Parrot is a virtual machine, which is in fact a processor in software.

First, if the machine has registers, arguments may be passed using these. It can happen that there are more arguments than registers, so there must be some way to handle overflow arguments.

The second technique is storing the arguments onto the run-time stack. This is a simple technique to implement, the caller just pushes its arguments onto the stack. The callee can then pop them off of the stack and store them into registers, or, if the machine instructions allow it, access the values on the stack directly.

The third technique that is mentioned here, is putting the arguments in some static allocated storage. This is a technique that is very impractical, because this way it is impossible to write recursive functions. However, it is a very simple technique to implement.

The next section will discuss parameter handling and the calling conventions for Parrot in more detail.

## 5.4 Parameter-Passing Modes

There are several parameter-passing modes. Two will be discussed here in short. These are: passing parameters by *value* and by *reference* (or by *address*). When a parameter is passed by value, the called function gets a copy of the value of the argument. When a parameter is passed by reference, a reference to the value of the argument is passed, so both the caller and callee have a reference to the same object.

All arguments in Lua are passed by value. This means that when a function is called with some parameters, then the function receives copies of those values. However, one thing is very important to note: LuaTables objects are stored as references. This means that when a LuaTable object is passed as an argument, the *reference* to the table must be copied, not the table itself.

Since all Lua data types are implemented as PMCs, some attention must be paid to how PMCs are handled by Parrot. In section 5.2, it became clear that PMCs are stored as references. That is, the actual object is not stored in a register, but a reference to it is stored. When assigning one PMC to another PMC register, not the actual object is copied, but only the reference pointing to it. So, when a PMC is passed as an argument, it is effectively passed by reference. For LuaTables this is fine, but not for all other Lua data types. This has implications for the instructions that must be generated for a parameter list. Chapter 6 will discuss code generation.

## 5.5 Tag methods

As described in section 1.2.5, tag methods can be seen as some kind of events ore exception handlers. How tag methods will be implemented definitively, is not decided yet. Parrot will have built-in support for events and exceptions. However, how nothing is done at the moment for events, so no design for that can be made. Implementing tag methods as exceptions is also an option, although exceptions are not really meant for this. If the implementation of events in Parrot will allow it, the event system will be used for implementing tag methods. However, at the moment of writing, events are not implemented in Parrot. The same is true for exceptions.

Although it is quite an important issue to implement tag methods, tag methods will be left out in the current implementation of the compiler. This means that Lua code that uses tag methods cannot be executed on Parrot *at this moment*. However, when support for events or exceptions is implemented, it can.

# Chapter 6

# Generating Code

When all compile-time checks are done, then instructions can be generated that instruct the computer to do what the programmer wants it to do. Code generation is done during a traversal of the Abstract Syntax Tree. During a visit of each node, instructions are selected that represent the meaning of that node. Because each construct in Lua is represented by a particular node in the AST, the code generation algorithms are isolated in the functions that belong to these nodes. So, the function that generates instructions for an `if` statement has no knowledge of how code is generated for function calls, and it does not have to. In this way, the code generating functions act like black boxes.

The code generator has a number of internal stacks. These are used to store intermediate results that will be used in other parts of the AST. The stack module is described in more detail in appendix D.4. There is also a great number of flags in the code generator. These are used to keep track of the context.

In section 6.1 the Intermediate Machine Code (IMC) language is described. This is the language that Pirate targets. The rest of the chapter discusses how the most important Lua constructs are translated to the target language. Some translation templates are not shown here because they are trivial. These details can be found in the source code.

## 6.1 The Target Language

### 6.1.1 The role of IMC

Imcc is the intermediate code compiler for Parrot. The most important reason to use IMCC is that it takes care of register allocation. Because Parrot has only a limited number of registers, it is harder to target Parrot directly than to target *imcc*. Imcc offers the Parrot programmer unlimited symbolic registers. This makes code generation much easier. Imcc maps these symbolic to real registers.

Whenever there is no real register available, IMCC takes care of register *spilling*. This means that the contents of a register, that is not needed at the moment, is temporarily stored somewhere else.

IMCC can also do some optimizations. Optimizing is the subject of Chapter 7.

Detailed information about IMCC can be found in appendix E.1. Knowing some details of the target language is interesting for two reasons. Firstly, it gives insight in the possibilities of the (virtual) machine that will execute the instructions. Knowing the features and restrictions of the available instruction set is necessary to understand the design decisions that were made during construction of the compiler.

Secondly, it makes reading the sections that discuss translation much easier. Sometimes, it is not quite clear what is meant by a particular instruction name, or the syntax of that instruction.

### 6.1.2 Parrot instructions

Sometimes, there is no IMC equivalent for a particular Parrot instruction. For these cases, it is possible to use Parrot instructions directly inside an IMC block. The Parrot instructions used in the code generator are described in appendix E.2. The following sections will describe how each construct in the Lua language is translated to IMC.

## 6.2 Translating Control Structures

Translating control structures is quite easy. Control structures are basically a set of labels and `jump` instructions that must be emitted at the right position. Because the nodes for expressions and statements are independent of the control structures, the latter do not need any knowledge of the former.

### 6.2.1 `break` Statements

A `break` statement is used to terminate the innermost enclosing loop. A loop is a `while`, `repeat` or `for` statement. An example of a break statement is given below.

```
x = 1
while x < 10 do
   if x == 5 then
      break -- execution will continue at the print statement
   end
end
print(x)
```

Conceptually, a `break` statement is quite simple. Control of execution just branches to the end of the while loop (or any other kind of loop, for that matter). However, a `break` breaks out of the *current* loop. It is of course possible to write nested loops. When translating a `break` statement, the jump instruction should jump to the right position, that is, the end of the current loop. A simple pointer to the end of the loop is not enough, because it would be overwritten when a new (nested) loop is translated.

Fortunately, because of the nature of a nested structure, the solution to this is quite simple. The natural flow of control in nested structures is Last In, First Out (LIFO). The perfect data-structure for that is a *stack*. Each time a loop statement is translated, the end label of that loop is pushed onto the stack. When a `break` statement is translated, the end label is popped off the stack, and a `jump` instruction is emitted. In this way, it is possible to handle situations like the one below.

```
x = 1
while x < 10 do

    y = 1
    while y < 5 do
       if y == 2 then
          break -- jump to "if x== 5 then"
       end
       y = y + 1
    end

    if x == 5 then
       break -- jump to "print(x)"
    end

    x = x + 1
end
print(x)
```

### 6.2.2 `return` Statements

A return statement consists of the keyword `return` and an optional expression list. When there is an expression list, then code is generated for this expression list first. The registers holding the result of the expressions are pushed onto an internal stack in the code generator. Then, when code for all expressions has been generated, a `save` instruction for each of the saved registers is generated. For each `save` instruction, a register is popped off of the internal stack to be emitted as an argument for the save instruction

```
return [exprlist] →
    code for each expression in exprlist
for each expression in exprlist:
    save $Px
    ret
```

### 6.2.3 `if` Statements

The translation of `if` statements is not as easy as it first seems. One would say it is easy, because for every `if` statement, there are only two options: either a statement is *true*, or it is *false*. However, because Lua also allows for optional `elseif` parts, it is somewhat more complex.

The code template is straightforward, but a few things must be noted. First, Lua allows for unlimited `elseif` parts in an `if` statement. At the end, there may be an `else` part. Before code generation for an `if` statement starts, all labels are generated. This is done because at a number of places there may be jumps forward. Because the names of these labels are known, these jump instructions may be emitted before the actual label is emitted.

Do take notice of the fact that some parts of this template may be left out. This is because the `elseif` and `else` parts are optional. If these are not present, then some instructions are not emitted.

```
if expr then block {elseif expr then block} [else block] end →
        code for condition
        if condition == false goto falselabel
        code for block (if true)
        goto end -- if there is an elseif or else statement
    falselabel: -- if there is an elseif or else statement
        code for elseif statements -- see 'Translating elseif statements'
        code for else block -- if available
    end:
```

#### Translating `elseif` Statements

```
elseif condition then block →
        code for condition
        if condition == 0, falselabel
        code for block
        goto end -- this end label was generated in the main if statement
    falselabel:
        code for other elseif statements
```

### 6.2.4 `while` Statements

A while statement is one of the loop constructions that is available in Lua. When a while loop is entered, first the condition is checked. If it is false, control jumps to a label at the end of the while loop. Otherwise, the next instructions, which are instructions generated for the block, may be executed. At the end of these instructions is a `goto` instruction, to jump to the beginning of the while loop.

```
while condition do block end →
    begin:
        code for condition
        if condition == false goto end
        code for block
        goto begin
    end:
```

### 6.2.5 `repeat` Statements

A `repeat` statement is almost the same as a while statement, except for the fact that the instructions for *block* are executed at least once. In fact, a repeat statement is the same as a *do-while* statement in C or Java.

First a label at the top is emitted. Then, code is generated for the block of the repeat statement. After that, code is generated for the condition. If the condition is *true*, then control jumps to the label at the top. The last code that is emitted is an end label. It seems this end label is redundant, because no jump instruction to this label is generated. However, because it is possible to `break` out of the repeat statement, this label can be jumped to from within the block. (see 6.2.1 for details on breaking out of loops.)

```
repeat block until condition →
    begin:
        code for block
        code for condition
        if condition == true goto begin
    end:
```

### 6.2.6 `for` Statements

`For` loops are the most complicated loop statements. This is because code for initializing condition variables is integrated into a for loop, whereas this is separated at other types of loop statements, such as while statements.

```
for name = initexpr , limitexpr [, stepexpr] do block end →
      .namespace namespace id
      .local LuaNumber <name>
      code to evaluate initexpr, result in init
      name = init
      code to evaluate limit expression, result in limit
      code to evaluate step expression, result in step
   begin:
      if step <= 0, test2
      if name  <= limit, block
      goto end
   test2:
      if name < limit , end
   block:
      code for block
      name = name + step -- if stepsize is 1, then "inc"
      goto begin
   end:
      .endnamespace namespace id
```

## 6.3 Translating Assignments

This section describes how assignment statements are translated. However, it must be noted that the given implementation is not how it should be. That is, as section 1.2.5 described, when a global variable is accessed or assigned to, then that should be handled as an event. But because there is no easy way to implement that at the moment, a more basic implementation without using tag methods is chosen.

Assignment statements have an unconventional form in Lua. Not only is it possible to do an assignment to one variable, a whole list of variables can be assigned a value in one statement. Thus, the following is legal:

```
i = 3
i, a[i] = 4, 100
```

In the example above, `i` is given the value `4` and `a[i]` is assigned the value `100`. In assignment statements, first all values on the right side and indices (if any) on the left side are evaluated. Then the variables on the left are assigned these values.

A naive translation would translate this example as two separate assignment statements:

```
i = 4
a[i] = 100
```

However, this is not correct. Now, the value `100` is assigned to a field at index `4`, but it should be stored at the index denoted by the old value of `i`, which is `3`. To handle these cases correctly, the code generator must first generate instructions to evaluate all expressions on the right-hand side, *and* all index expressions on the left-hand side.

Because the number of expressions may be different from the number of variables, first the number of variables is counted. If there are any index expressions present, then code for evaluating these expressions is generated. The result registers are stored on an internal stack of the code generator. After that, the expression list is traversed, decrementing the number of variables at each expression node. This way, no expression will be evaluated if it is not necessary.

## 6.4 Translating Functions

Translating functions is one of the hardest parts in the code generator. Basically, a function is just a branch instruction, but should it be called by other languages, then some conventions must be obeyed. Because Lua running on Parrot should be able to be called by other languages, and also call functions written in other languages, Lua functions should adhere to the Parrot calling conventions. Furthermore, Lua has a special feature, called upvalues. Upvalues were discussed in detail in Chapter 4. Because upvalues are owned by a function, the implementation of upvalues is discussed in this section, too.

The rest of this section will discuss how code is generated for functions. First, the Parrot calling conventions are discussed. Then, translation of function *definitions* will be described. After that follows the discussion of the translation of function calls. Last but not least is the discussion concerning upvalues.

### 6.4.1 Parrot Calling Conventions

Calling conventions are rules that specify how a function is called, and where the parameters are passed. Calling conventions make sure that functions compiled with different compilers, or even written in different languages, are able to call each other.

For example in Chapter 5, several different ways for passing parameters were discussed. Let's suppose a function `f` is compiled with a compiler that would emit instructions to store all parameters on the run-time stack. If then another function `g` is compiled by another compiler that would emit instructions to get parameters from registers, then these functions `f` and `g` are not able to work together.

So, it's good to have some standard way for passing parameters. That way, code that is written in different modules, or even different languages, can cooperate.

The Parrot calling conventions consist of two parts: calling functions and returning from functions. The calling conventions are described in appendix C.1. The conventions for returning from functions are described in appendix C.2.

### Exporting Functions

For a function to be able to be called by another module, it is necessary to obey some rules. Most importantly, the function must adhere to the Parrot Calling Conventions. All function calls in Lua do follow the *calling* conventions, but not the *returning* conventions. This is because of the way how Lua handles assignment statements. According to the calling conventions, return values should be stored into registers. However, Lua functions save their return values onto the run-time stack. This way, another module will not be able to receive values returned by a Lua function.

Another point of attention is the function names. All global labels and subroutine entry points should start with an underscore. So, if a Lua function should be called from another module, it should behave completely according to the Parrot calling conventions and its name should start with an underscore. The easiest way to accomplish this, is to create function stubs. These are wrapper functions that call the actual function. These wrapper function have the same name as the original function names, except that they start with an underscore. Furthermore, they restore all return values from the run-time stack, and store them into registers, so the calling function can find them where they are expected.

Only global functions will be exported, so no stubs will be created for locally declared functions. During the code generating phase, all names of global functions are collected into a data structure. For that purpose a symbol table is used. However, the implementation of the symbol does not support looking up symbols without knowing their names. For that purpose, an iterator structure was designed. This allows for iterating over all entries in a symbol table. The iterator and its operations are described in appendix D.3.

Below is an example of how a function is exported. The function returns some numbers.

```
function f()
    return 1, 2, 3
end
```

The stub for this function should be (in IMC):

```
.sub _f
    P0 = global "f"   # get current value of f
    invoke            # invoke it
    restore P5
    restore P6
    restore P7
    ret               # return to calling function
 .end
```

As described earlier, return values are pushed onto the run-time stack in *reversed* order, so the first return value is at the top of the stack. This value will be restored first, so it will be restored into the first return register.

However, the above is not quite complete. Because there may be only 2 or less return values on the run-time stack, the code above will run not correctly[1]. The number of returned values is, according to the Parrot calling conventions, in register I3. So, for each restore instruction, we must check if that is valid. Unfortunately, it is not possible to generate a run-time loop for this, because the register to which must be restored must be incremented each iteration. For that there is no instruction (that would implicate there should be some kind of variable containing the name of a register).

So, the solution is a somewhat inefficient list of instructions with checks and branches. The above example would be translated to this:

```
.sub _f
   P0 = global "f"
   invoke
   $P20 = I3          # copy number of return values into counter.
   if $P20 == 0, end  # check for enough return values
   restore P5         # restore value from stack into register
   dec $P20           # decrement counter
   if $P20 == 0, end
   restore P6
   dec $P20
   if $P20 == 0, end
   restore P7
 end:
   ret
.end
```

If assignment statements are reimplemented, then this sort of code will not be necessary anymore. However, at the moment it is a solution that works. There may be a better solution in the future, if Parrot will get instructions to handle these kinds of situations. (However, this may not happen).

### 6.4.2 Translating Function Definitions

The translation of a function involves translating all statements in the function. The function will get an internal name, which is automatically generated by Pirate. This is because a function in Lua does not really have a name. A function may be assigned to a variable, but that variable may also be set to some other value. If no variable points to the function, then the function is inaccessible. At another point, more than one variable may point to the same function. Below is the code generation template for a function definition.

[1]It will run, but an error message is displayed.

`function(` [*parameter list*]) *block* `end` →
    `.sub` *auto-generated id*
    *code for parameter list*
    *code for block*
    `.end`

### Translating the Parameter List

When Parrot assigns a PMC to another PMC, then the reference to the pmc is copied, because PMCs are stored as references. An example:

```
P0 = P1
```

Now, P0 points to the same object as P1, so when something is done with P0, then that will affect the object where both register point to. Because all Lua data types are implemented as PMCs, it is not enough for a function to have a set of local variables, one for each parameter, and then copy the actual parameters into those local variables. Somehow, a 'deeper' copy must be made. Parrot has an instruction for that: `clone`.

So, all actual parameters are being cloned when a called function starts executing. However, LuaTables should be stored as references, so for them copying references is the right thing to do.

Now it is clear how parameters are being passed, and how to solve this for Lua variables. Still, there is one problem. Declared parameter lists do not have any types associated with parameters. That is, when a function is declared like this:

```
function f(a)
```

then `a` may be a number, a string, or any Lua type including a table. So, at compile-time, it is not evident if a parameter is of a 'basic' type, that should be cloned, or if it is a table that should not be cloned. The only solution to this is doing a run-time check. So, instructions must be generated that do a check on the type of the parameters.

The resulting code template for parameters is thus:

*parameter list* → `[`*namelist*`]` `[`*varargs*`]`

```
namelist →
    .local LuaNil id
    typeof $Ix, id
    if $Ix == LuaTable, goto copy
    id = clone current_reg
    goto next
    copy:
    id = current_reg
    next:
```

Lua allows for a variable argument list, denoted by "...". The actual arguments are collected into a table at run-time. At the moment of writing, this feature has not been implemented completely, but the code template should look something like this:

```
varargs →
    $Px = new LuaTable
    $Iy = 1
    start:
    if num_args_left == 0, goto end
    restore $Pz
    $Px[$Iy] = $Pz
    dec num_args_left
    inc $Iy
    goto start
    end:
```

### 6.4.3 Translating Function Calls

When a function is called, first a `saveall` instruction is emitted, to save the current contents of the registers. Then, if there are any arguments, code is generated to evaluate the expressions. The result-registers of these expressions are pushed onto an internal stack in the code generator. After that, the first 11 arguments are stored in registers P5 to P15. If there any arguments left, then they are stored into an array.

Then before doing the actual call, some other registers must be set, according to the calling conventions.

```
name([exprlist]) →
    saveall
    code for exprlist, results are stored in registers
for each expression in exprlist:
    Pr = register (r in range 5-15)
if there are any arguments left, then:
    P3 = new Array
```

```
for each argument that is left:
    P3[i] = arg j (i starting at 0, j from argument 12 to n)
    $Px = global "name"
    P0 = $Px
    invoke
    restoreall
```

Lua has a special syntax for calling so-called *methods*. These are functions stored in tables. This allows for a kind object oriented programming (be it somewhat artificial). When such a function is called, the table that in which the function is stored, is passed as the first parameter. This does not change the way parameters are passed, though.

### 6.4.4 Translating Upvalues

Upvalues belong to the function in which they occur. So, an upvalue is stored as a field of the function object that represents the function. Thus, the translation becomes:

```
%name →
    $Px["name"] -- $Px holding the function PMC
```

Of course, an upvalue can occur as a r-value (when its value is "read"), or as a l-value. The latter case is only allowed if a member field is accessed, so the upvalue must be a table.

## 6.5 Translating Local Declarations

Local variables are stored in so-called *scratchpads*. These work like small symbol tables. Each time a new block is entered, a new scratchpad is created. Scratchpads are organized in such a way, that variables in an enclosing scope are visible, too. This is exactly the case in Lua, for example:

```
do (create a new block)
   local x = 123
   do (another new block)
      print(x) -- will print "123"
   end
end
```

The code template for local declarations is:

```
local namelist [= exprlist] →
for each name in namelist:
    $Px = new LuaNil
if there is an expression list for initializing variables:
    code for exprlist, results are stored in registers
    $Px = result register
for each name in namelist:
    store_lex -1, "name", $Px
```

## 6.6 Translating Data Structure References

This sections describes how data references are translated. Simple references are references to variables without fields (variables other than LuaTables). Translation of simple references is described in 6.6.1. Translation of references to data fields is described in 6.6.2.

### 6.6.1 Simple References

Translating simple references is nothing more than fetching a value of a variable and storing it in a register. So whenever a variable is referenced, its value (or in the case of a LuaTable, a reference to it) will be loaded with the instruction in the following code template:

```
name →
    $Px = global "name"
```

### 6.6.2 Field references

The syntax of Lua allows for two forms for referencing fields: *name*`["`*field*`"]`, and *name.field*. Both forms are translated to the same code.

```
name.field →
    $Px = global "name"
    $Px["field"]
```

## 6.7 Implementing tags

A tag is just a property of an entity. Each type of entity has its own tag. This means, for example, that all numbers in Lua have the same tag. For numbers, in fact, for all types, except *tables*, the tag cannot be changed. Because tables can have different tags, it must be possible to access the tag, so it can be changed at run-time. This means the tag cannot be implemented as some kind of constant field, but as a property that can be set and changed. Parrot has built-in support for setting and getting properties of PMC objects. Special machine instructions allow for fast access.

# Chapter 7

# Optimizing Code

This chapter discusses optimization of the generated code. There are many different types of possible optimizations. Section 7.1 describes what optimizations are done in the IMC compiler. After that, section 7.2 describes what optimizations are done by Pirate. Finally, section 7.3 discusses the implementation of the optimizer in Pirate.

## 7.1 Optimizations in the IMC Compiler

This section describes which types of optimizations are done by the IMC compiler. It is interesting to have a look at the optimizations that are done by IMCC, because these types of optimizations do not have to be implemented by Pirate anymore. However, there are some cases in which this is not necessarily true as will become clear in section 7.2.

All compilers targeting IMCC benefit from the fact that IMCC does these optimizations, because this way these optimizations have to be implemented only once.

The following are optimizations that are done by IMCC:

- **Removal of unused labels** A label that is never jumped to may be removed.
- **Jump optimization (jumps to jumps)** If some jump instruction is succeeded by another jump instruction, then the first jump target may be changed to the succeeding target.
- **Strength reduction** Expensive operations are replaced by cheaper ones. An example (in equivalent C code) is: `x = x + 2`.
  This may be replaced by `x += 2`. The compiler now knows that one of the operands has the same location to store the result.

- **Deletions of unnecessary assignments** If a variable is used as the left-hand side of an assignment, but the variable is never used after that, then this assignment instruction may be removed.
- **Dead-code elimination** Code that is never executed can be removed.
- **Code motion** Expressions that can be evaluated safely before a loop, are placed before the loop. This way the amount of code in the loop itself is reduced.

## 7.2 Optimizations in Pirate

The optimizer in Pirate is a *peephole optimizer*. A peephole optimizer examines a small 'window' (or 'peephole') of two or three instructions. If the instructions match a particular pattern, then the instructions are replaced by faster or less instructions that have the same effect. According to the documentation of IMCC, these optimizations are done by IMCC, but there are cases that can be optimized even more. An example is shown below.

The Lua statement `x = x + 1` will translate to: `P0 = P0 + 1` (Assuming P0 holds the value of x). This will translate to the following Parrot instruction:

```
add P0, P0, 1
```

IMCC can optimize this to the more efficient instruction:

```
add P0, 1
```

However, this can be further optimized to the more efficient instruction:

```
inc P0
```

So, it may seem that the optimizer in Pirate does some redundant work, but this is not the case. However, because IMCC is still being developed, it may be that the optimizations in Pirate are no longer necessary in the future.
The following optimizations will be implemented in Pirate.

- (Further) Strength reduction
- Removal of useless instructions

Useless instructions are instructions that have no effect.An example is:

```
P0 = P0 + 0
```

Other types of optimizations can easily be added later. The next section will discuss the global structure of the code optimizer.

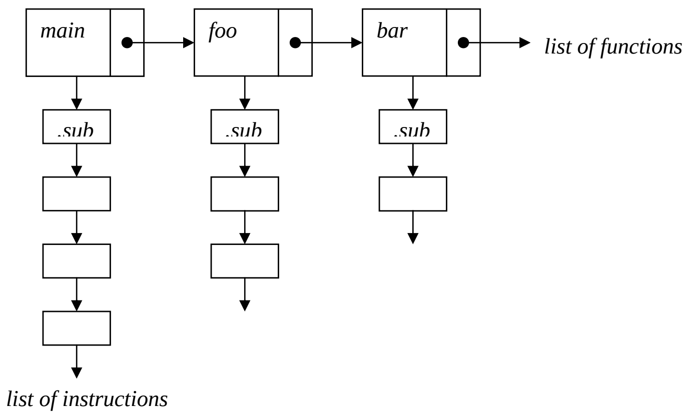
## 7.3 Implementation of the Code Optimizer

When the code generator is finished, the code is organized in functions. Each function keeps a list of instructions. The functions are linked together in a list, too. An example is shown in figure 7.1. Note that there is always a `main`

```
function foo()
    -- do something
end

function bar()
    -- do something else
end
```

(a) The source code



(b) Organization of the instructions for (a)

Figure 7.1: Organization of instructions.

function. Lua instructions do not necessarily have to be written in functions. In fact, the first statements being executed must be outside any function, otherwise they are never executed. When code is generated, these 'first' statements are put into the `main` function.

The organization of the generated instructions is very straightforward. Designing a peephole optimizer for this organization is quite easy. It is just an iterator that iterates over all functions, and within each function, all instructions will be visited one by one. When an 'interesting' instruction is visited, then the optimizer may inspect the next one or two instructions and do the appropriate optimizing actions.

# Part IV

# Using Lua

# Chapter 8

# Lua Libraries

Lua is a small language, so there are not many constructs and built-in functions. The language can be used without its libraries. However, many Lua programmers use the libraries, because they offer much functionality. The libraries in the original distribution are provided as separate C modules that can easily be loaded into the virtual machine[1].

This chapter takes a closer look at the standard libraries that Lua offers and how they can be implemented in Pirate. Section 8.1 examines what libraries Lua has. After that, section 8.2 discusses functions to execute Lua code. These functions are special because before actually executing the code, it must be compiled. So, somehow Pirate must be invoked to translate the code.

## 8.1 The Standard Libraries

This chapter describes gives more information about the Lua libraries. Lua has several libraries, listed below.

- **Basic library** The basic library contains general functions, such as an `error` function but also functions for handling tags and the like. Most functions can be implemented in Parrot instructions directly. The basic library also contains some special functions for running Lua code. These are discussed in section 8.2.
- **String library** The string library contains functions for manipulating strings. Again, most functions can be implemented in Parrot instructions. Parrot has a few special string instructions, such as one for retrieving the length of a given string.
- **Math library** The math library is in fact an interface to some functions of the standard C math library. Typical are for example **sin**, **abs** and the

[1]Explicit loading the libraries is not necessary when using Lua stand-alone.

like. Parrot has most of these instructions built-in, so implementing them in Parrot assembler instructions is straightforward.

- **I/O library** The I/O library contains functions for reading and writing files. There are three global variables that are initialized with file descriptors for `stdin`, `stdout` and `stderr` (from the C language). Parrot has some instructions for handling files, so implementing these functions will not be troublesome.
- **System library** The system library contains some system facilities, such as a function for retrieving the current date and time. In the future, it will be possible to implement many of these functions. At this moment, for example, there is already a **time** operation in Parrot (although it is not documented, yet).

Implementing the Lua libraries for Pirate is for the greater part quite straightforward. This is because Parrot is a virtual machine with a lot of 'high level' operations (see section 1.3). Many functions can be implemented with only a few, and sometimes only a single instruction.

The original Lua language also has a debug interface. These are functions that can be used to debug a Lua program. Because the functionality of the debug interface is built into the Lua interpreter, there is no equivalent for Pirate at the moment. However, if the implementation of Pirate will be improved, it should be possible to create some debugging functions for Lua specific oriented towards Parrot.

When a Lua program is compiled in the normal way with Pirate (without -c option), then all libraries are included. It would be desirable to have the libraries in a pre-compiled binary form. Then, the pre-compiled Parrot byte-code (PBC) file can be linked to the main program, or it may be loaded dynamically during run-time. Both linking and loading of PBC files will need some kind of support of Parrot. At the moment of writing, Parrot does not offer support for this. So, now it is still necessary to insert all library code into the main file. This way, the main file will be quite large, even for a trivial 'Hello World' program. However, because this solution is only temporary, this is acceptable.

## 8.2 Functions to Execute Lua Code

Lua has two functions to execute Lua code. These are:

- dostring(*s*) The string *s* is executed as a Lua program.
- dofile(*f*) The file *f* is executed. The file may contain Lua code or it may be a pre-compiled file.

These functions are special, because they are somewhat more complex. When one of these functions is called, then the string or source file must be compiled **first** before Parrot can actually execute them. So, the byte-code must be created during *run-time*. Fortunately, Parrot has special instructions for this, because this feature can be found in many scripting languages. However, these functions have not been implemented at the moment, so implementing them for Pirate is not possible. A short description of how compiling during run-time is done is given below.

To be able to compile during run-time, a compiler for a language, in this case Lua, should be registered. A compiler will be represented as a PMC. The built-in compiler PMC is an extension of a Native Call Interface PMC. This is an object that takes care of invoking *native* subroutines, that is, subroutines written for example in C. So, when the `compile` instruction is being executed, then a native subroutine call is done. The native subroutine should be a call to a compiler.

The description above is only an overview, and details are left out. However, it does give an idea of how compilation during run-time can be implemented.

# Chapter 9

# The Lua Compiler for Parrot

This chapter describes how the Lua compiler for Parrot should be used and how the generated files can be executed. Section 9.1 describes how Pirate should be invoked. Then, section 9.2 describes the possibilities for executing a Lua program on Parrot. Finally, the possibilities to use Pirate with other languages are explored in section 9.3.

## 9.1 Compiling Code

The Lua compiler for Parrot is called *Pirate*. Pirate has a small number of run-time options. These will be displayed when invoking the compiler with the `-h` option. These options are:

- **-c**: Compile only. No main function will be created and no run-time support routines are inserted.
- **-o**: Specify the output file. The default output file is 'lpc.imc'.
- **-v**: Enable verbose output.

When the **-c** option is activated, then no `main` function will be created and no support routines are inserted into the file. This option should be used for compiling a file that only contains function definitions only. This option is for creating modules that can be used with other modules.

The **-o** option should be used when the code should be written to a file with a specific name. If this option is not used, then the code is written to a file called 'lpc.imc'.

The last option, **-v**, enables verbose output. The compiler will output messages during compiling, indicating in which phase the compiler is.

## 9.2 Running Lua code

Pirate can be used in one of three ways. These are listed below.

- **Lua stand-alone** Lua may be used stand alone. In that case, it is just used as a scripting language, just like Perl.
- **Lua with other languages** Lua may be used with other languages. As long as all modules are compiled with a compiler that emits code that adheres to the Parrot calling conventions, there is no problem doing this.
- **Lua embedded** Just as the original Lua language, it can be used as an embedded language. However, now the code is not executed by a Lua interpreter, but by a Parrot interpreter. The Parrot interpreter can easily be embedded within any C program. This is done in almost the same way as when running Lua in an embedded environment. See [7] for more details on embedding a Parrot interpreter.

### 9.2.1 Running a Simple Program

A simple program is a program that consists of only one file. Running a simple program is easy. One can compile the Lua program with Pirate and invoke IMCC. Then, the program will run.

### 9.2.2 Running a Modular Program

When a program consists of multiple files, or modules, it is necessary to tell the 'main' module which other files it will need during run-time. For that purpose the function `dofile` can be used in Lua. This function was described in section 8.2. Then, in the file that is passed as an argument, the necessary statements are executed so the functions in that file can be used (particularly, the functions need to be assigned to some variables).

Imcc has an `.include` instruction for using multiple files. Actually, it is not really an instruction, but a *macro*. This is a directive to IMCC. However, just including a file randomly is not enough, moreover it will result in several errors or erroneous behaviour. Include directives must be handled in a specific way. The rest of this section will explain how a file can be included.

It is important to realize that a file that is to be included must be compiled with the **-c** option, otherwise the Lua run-time functions will be included and all kinds of errors will occur. The file that will be included must only contain function definitions, so all statements must be in some function body.

When the file in figure 9.1(a) is compiled with the **-c** option, then this will be translated to the instructions in figure 9.1(b).

```
function f()
    -- do something
end
```

(a) Lua source

```
I0 = addr f
P0 = new LuaFunction
P0 = I0
global "f" = P0

.sub f
# do something
ret
.end
```

(b) IMCC translation

Figure 9.1: Compiling with **compile only (-c)** option.

Now, if the file generated with the **-c** option in figure 9.1(b) is included, the function definition is appended to the main file, and the `.include` direction is replaced by the instructions in the included file that are **not** in a function body. For this to work correctly, all `.include` directives must be collected into a function that the main routine calls. So, the resulting main file will look something like this:

```
.sub _main
call include_files
# now we can invoke function f
P0 = find_global "f"
invoke
end
.end

.sub include_files
.include "my_lib.imc"
ret
.end
```

If this main file is fed to the assembler, then correct code is generated that will be executed the way it should be.

## 9.3 Running other languages

### 9.3.1 Calling conventions

As long as the modules written in other languages use the Parrot calling conventions, there is no problem using these modules (however, there are some other issues, described in 9.3.2). However, at this moment, no other language actually uses these conventions. This is because Parrot is a very new target, and the conventions have been subject to change for a while. However, they have been revised lately and it is very improbable they will change again.

At this moment, the `.arg` and `.param` instructions in IMCC translate to Parrot instructions that use the run-time stack. In the future, these instructions will be re-implemented in the IMC compiler, so that they will translate according to the Parrot calling conventions. Pirate can be compiled so that it generates the IMC `.arg` instructions that push any arguments onto the run-time stack, or it can be compiled so the real Parrot calling conventions are used. When Pirate is compiled so that it generates the IMC instructions, then it should be possible to communicate with other languages, which all make use of the IMC instructions. On the other hand, when it is necessary to generate code to adhere to the real calling conventions, then Pirate may be compiled to do so.

### 9.3.2 Exchanging data

Until now, there was only a focus on the *calling conventions* when executing modules written in other languages. However, there is another very important issue that must be addressed. This issue is data representation. Perl has its own Perl data types implemented as PMCs. Lua has its own data types too. The reason for a language to have its own data types is that each language may behave in its own way, or rather its data types behave in a particular way.

Some examples of this different behaviour follow.

1. All Lua data types have a so-called tag, but data types from other languages do not.

2. What happens when a LuaNumber is passed as an argument to a Perl subroutine? A LuaNumber is almost the same as a PerlNum, but it has a tag. However, it is easy to extract the number from the PMC in a Perl subroutine.

3. What happens when a LuaTable is passed as an argument to a Perl subroutine? This is more troublesome, because Perl has no notion of LuaTables. So, this will give trouble.

4. Lua functions pass arguments by value, Perl subroutines pass them by reference. As long as the Perl subroutine does not change the passed-in PMC, there is no problem. However, when it does change the value, this will affect the original argument, that may still be in use after the function call.

5. The other language is statically typed, so it may use the built-in data types, such as integer. If a PMC is passed in, this will yield an error, because the PMC will not fit if an integer is expected (or any other built-in type for that matter). Lua interacting with a statically typed language is not possible at all.

From these examples it becomes clear that inter-language communication with Lua is not *that* easy. The programmer has to realize the effects it has on its program. However, when the programmer takes care of these issues, then inter-language communication can be done.

On the other hand, a language such as Python has a behaviour similar to Perl, so when that language is targeted to Parrot, inter-language communication is easier.

# Chapter 10

# Conclusion

This chapter provides a summary of the work presented in the previous chapters. Writing a compiler is a lot of work, although the present tools and targets make it somewhat easier. Generators for scanners and parsers take much work away from a compiler writer. On the other hand, targeting an intermediate language such as IMC takes away much difficult tasks of writing a compiler back-end. These tasks include writing a code generator using real registers (as opposed to using infinite symbolic registers) and a code optimizer. Nevertheless, writing a compiler is still much work. So, although Pirate is operational, it is not finished completely.

Section 10.1 describes the design of Pirate. After that, section 10.2 describes how Pirate was implemented. Finally, section 10.3 provides a summary of issues that should be dealt with in the future. The first part of the section desribes the current shortcomings of Pirate. The second part describes possible improvements that can be implemented.

## 10.1 Design

Pirate was designed to be a compiler for Lua that generates instructions for the Parrot virtual machine. Moreover, it should be possible to create a program consisting of multiple modules, each written in a different language.

Having different languages communicate is nothing new, but implementing it in such a way the programmer does not have to take care of it, is a bit harder. There are two important issues that must be addressed in order to have modules written in different languages to cooperate. These are *calling conventions* and *data representation*.

Calling conventions are important because they prescribe a standard way of storing data when passing arguments. Although it is not hard to adhere to some set of calling conventions, they must be taken into account when instructions are generated. As long as no arguments are passed to a function written in a different language, there is not really a problem. However, when arguments do get passed around, then data representation will play a role.

Data representation is important because each language may have special features. Lua has its table structure for storing data, but Perl for example, uses arrays and hash tables. It is possible to share primary data types, such as integers and the like, among different languages. Sharing more complex data structures such as hash tables, is much harder. No further research was done to actually share more complex data structures among different languages.

Pirate was designed to be a modular compiler. So, my goal was to design the compiler as clean as possible. Firstly, because a compiler is a complex program and dividing the task into several subtasks makes it easier. Secondly, a modular program can be adapted more easily. Because Parrot is still under development, and will be for some time, it is desirable to be able to adapt the compiler if necessary.

## 10.2 Implementation

The compiler consists of four phases. The first phase does lexical and syntactical analysis. Lexical analysis is done by the lexer. This verifies the structure of names of identifiers, numbers and recognizes keywords. The syntactical structure of a Lua program is verified by a parser. The parser and lexer are operating interleaved. This means that whenever the parser needs a new token, it calls the lexer. The lexer returns a code representing the type of the new token. During parsing, an Abstract Syntax Tree (AST) is built that represents the source program.

The lexer was built using a scanner generator. While writing a lexer by hand is quite trivial, a generated lexer is easier to adapt. Likewise, the parser was built using a parser generator, so minimal work had to be done to create it. For creating the AST and its operations the TreeCC program was used. This program greatly simplifies management of the source code for the tree.

The second phase does scope checking. This phase traverses the AST and verifies all identifiers for their scope.

After scope checking, either some errors were found, in which case the compiler terminates, or the program was found to be correct so the third phase can start: generating code. During this phase, the AST is traversed once more, and for each tree node, some IMC instructions are emitted that represent the meaning of that node. All generated instructions are organized in lists.

When the code generator is done, the last phase is entered. The last phase is the code optimizer that iterates over the lists of instructions. At the moment of writing the optimizer does not do anything useful, but in the future it will try to recognize small sequences of instructions that it can replace by faster or shorter sequences of code.

It must be noted that the implementation of Pirate is not finished. There is still a lot of work to be done. However, it is a working system, and some of the design goals are established. Some features needed for Pirate are not implemented on Parrot, yet. On the other hand, Parrot is already a functioning virtual machine, with great potential.

## 10.3 Where to go from here

This section describes some potential future work that can be done for Pirate. Firstly, the current shortcomings and errors in the current implementation of Pirate will be discussed (10.3.1). Secondly, some suggestions for improvements are done in 10.3.2. Finally, the future of Lua for Parrot is discussed in 10.3.3.

### 10.3.1 Known Issues

This section discusses some errors and shortcomings that are currently present in Pirate.

- **Testing the lexer** The lexer has not been tested completely. Scientific notation for numbers and nested quoted strings should be tested.
- **Parrot Magic Cookies** Finish implementing all Lua PMCs. Not all Lua PMCs are fully implemented yet.
- **Tag methods** Implementing tag methods as events, or maybe as exception handlers, if possible. At the moment Parrot does not have events and exceptions implemented. Depending on how events will be defined, events may be an efficient and clean way to implement tag methods for Lua. Whenever something non-standard is done, like adding two tables, an event is created, and a tag method is called. On the other hand, if the event system in Parrot is not suitable for tag methods, then another option is to implement tag methods as exception handlers. Then, each tag methods is used as an exception handler. Parrot will have built-in support for both events and exceptions (see 1.4.5).
- **Translating assignment statements** The current implementation of assignment statements should be revised. At this moment, all expressions are evaluated and stored in registers and, in case when a function call is present on the right-hand side, saved on the run-time stack. However, this solution is not as clean as I would like it. One possibility is to store all expressions in some sort of list, then in one way or the other put all variables in another list, and then do some sort of list assignment. The specific details must be investigated, though.
- **Translating For statement for Tables** Lua has a special for statement for iterating over a table. However, at this moment there is no iterator PMC available. When such a PMC exists, then implementing this type of for statement is straightforward.
- **Handling lists of variable length** This issue is important for assignment statements and for function calls. In assignment statements, the

number of expressions does not necessarily have to be as long as the number of variables. In function calls, the number of actual parameters does not have to correspond with the number of formal parameters. At this moment, both cases have not been implemented. Some research must be done on how to this in the most efficient way. The list structure mentioned in the previous item may be used for these cases.

- **Implementing the optimizer** At this moment, the optimizer is not implemented. Although the necessary code to iterate over the generated instructions is present, no actual optimizations are done in the optimizer.
- **Implementing dostring/dofile functions** At the moment of writing, nothing is done for the dostring and dofile functions (see section 8.2). Parrot has some special support for implementing them, so it should be possible.

### 10.3.2 Improving the compiler

There are a number of points I would like to improve.

The following are suggestions for improving the compiler.

- **Better string management by using a central symbol table.** Now, string management is implemented in an *ad-hoc* way. It would be better if each string was stored only once. The functions creating strings should use a symbol table to check if such a string was already created. If it was, the string is looked up, and a pointer should be returned. If it was not, the string may be created and inserted into the symbol table.
- **Better memory management by using a built-in manager.** Memory management is now done separately for each module. That is, all data structures have a `create` and a `destroy` function for allocating and freeing memory, respectively. These are in fact wrapper functions for allocating and deallocating memory, respectively. Because some data structures are used in combination with others, deallocating memory may be dangerous, because it can affect other data structures. This is especially true for memory allocated for strings. A better way would be to implement a module that keeps track of all allocated memory, so no memory leaks would occur.
- **Better error handling in the parser.** Recovering from parse errors can be done using the constructs described in 3.4. However, this should be done with great care, because it may introduce errors in the grammar.
- **Reconstructing the code generator.** The code generator is quite complex, because it uses a lot of flags during traversing the tree. This is to discover the context of the nodes to be able to generate correct code. However, this solution is not as clean as I would like it. Another option is to transform the AST in its whole, so more types of nodes are introduced.

That way, it would be possible, for example, to make a distinction between a *name* node on the right-hand side of an assignment, and a *name* node on the left-hand side.

- **Implement a special `argument` pmc.** This PMC should prevent the run-time checks that are done in functions (see 6.4.2). When assigning a LuaTable to an `argument` PMC, only the reference should be copied. If, however, another Lua data type is assigned to this `argument`, then a 'deeper' copy should be made. When this functionality is put into such an `argument` PMC, then no more explicit run-time checks have to be done. This may improve the efficiency.

### 10.3.3 Lua 5.0

At the moment, a new version of Lua is being developed. This will be version 5.0. It is a completely revised version, in which some features of version 4 will be left out, others are being replaced with complete new implementations and some features are completely new. The next list enumerates the most important differences.

- **coroutines**
- **meta-tables as a replacement for tag methods**
- **no more upvalues**
- **small syntactic changes**

If a Lua compiler for version 5.0 has to be written, it is most probable that it must be completely rewritten. Of course, some modules, such as the symbol table, may be used, but the parser and code generator must be rewritten. This is because of the syntactic changes in the language. The AST will have a different structure too, so the operations that do a tree-traversal must be rewritten.

# Appendix A

# Lexical and Syntactical Structure of Lua

## A.1 Lexical Conventions for Lua

This section describes the lexical conventions for Lua as they can be found in [12].

### A.1.1 Conventions for naming Identifiers

Identifiers may be any string of letters, digits and underscores, not beginning with a digit. The definition of letter depends on the current locale. Any character considered alphabetic by the current locale may be used as a letter.

### A.1.2 Lua Reserved Words

Reserved words may not be used as identifiers. However, they *can* be used as literal keys for indexing tables when an index notation is used. So, the following is legal:

`t["and"]` but this is not: `t.and`

The following are reserved words.

```
and     break  do        else   elseif
end     for    function  if     in
local   nil    not       or     repeat
return  then   until     while
```

### A.1.3 Other Tokens

```
~=   <=   >=   <   >   ==   =   +   -   *   /   %
(    )    {    }   [   ]    ;   ,   .   ..  ...
```

### A.1.4 Strings

Strings may be delimited in several ways:

- Using single quotes: 'Hello, World!'
- Using double quotes: "Hello, World!"
- Using double brackets: [[Hello, World!]]

Strings delimited by double brackets may run for multiple lines and may contain nested pairs of double brackets.

### A.1.5 Numbers

Numbers may be written with an optional decimal part and an optional decimal exponent. Some examples are listed below.

- 123
- 123.456
- 123.45e-6
- 0.123E2

### A.1.6 Comments

Comments start with a double hyphen (- -) and run until the end of the line. The first line is skipped if it starts with a #.

## A.2 Syntactical Structure of Lua

This section contains the grammar for Lua that was used for Pirate. It is based on a grammar in [9].

| | | |
|---:|:---:|:---|
| *chunk* | → | *block* |
| *block* | → | *stmt_list end* |
| *stmt_list* | → | $\epsilon$ |
| | \| | *stmt_list stmt opt_semicolon* |
| *end* | → | $\epsilon$ |
| | \| | **return** *opt_expr_list opt_semicolon* |
| | \| | **break** *opt_semicolon* |
| *opt_semicolon* | → | $\epsilon$ |
| | \| | ’;’ |
| *stmt* | → | *call* |
| | \| | *var_list* ’=’ *expr_list* |
| | \| | **do** *block* **end** |
| | \| | **while** *expr* **do** *block* **end** |
| | \| | **repeat** block **until** expr |
| | \| | **if** *expr* **then** *block opt_elseif_list opt_else* **end** |
| | \| | **for** for_stmt |
| | \| | **function** *function_name* ’(’ *param_list* ’)’ *block* **end** |
| | \| | **local** *name_list initializer* |
| *name_list* | → | *name* |
| | \| | *name_list* **,** *name* |
| *initializer* | → | $\epsilon$ |
| | \| | ’=’ *expr_list* |
| *param_list* | → | $\epsilon$ |
| | \| | ’**...**’ |
| | \| | *name_list* |
| | \| | *name_list* ’,”**...**’ |
| *var_list* | → | *var* |
| | \| | *var_list* ’,’ *var* |
| *opt_elseif_list* | → | $\epsilon$ |
| | \| | *opt_elseif_list* **elseif** *expr* **then** *block* |
| *opt_else* | → | $\epsilon$ |
| | \| | **else** *block* |
| *for_stmt* | → | *name* ’=’ *expr* ’,’ *expr opt_comma_expr* **do** *block* **end** |
| | \| | *name* ’,’ *name* **in** *expr* **do** *block* **end** |
| *opt_comma_expr* | → | $\epsilon$ |
| | \| | ’,’ *expr* |
| *function_name* | → | *name dotted_key_list opt_colon_key* |
| *dotted_key_list* | → | $\epsilon$ |
| | \| | *dotted_key_list* ’**.**’ *key* |

$$
\begin{array}{rcl}
\mathit{key} & \rightarrow & \mathit{name} \\
 & | & \mathbf{and} \mid \mathbf{do} \mid \mathbf{end} \mid \mathbf{else} \mid \mathbf{elseif} \mid \mathbf{break} \mid \mathbf{for} \mid \mathbf{function} \\
 & | & \mathbf{if} \mid \mathbf{in} \mid \mathbf{local} \mid \mathbf{nil} \mid \mathbf{not} \mid \mathbf{or} \mid \mathbf{return} \mid \mathbf{repeat} \\
 & | & \mathbf{then} \mid \mathbf{until} \mid \mathbf{while} \\
\mathit{opt\_colon\_key} & \rightarrow & \epsilon \\
 & | & \text{’}\mathbf{:}\text{’}\ \mathit{key} \\
\mathit{expr\_list} & \rightarrow & \mathit{expr} \\
 & | & \mathit{expr\_list}\ \text{’,’}\ \mathit{expr} \\
\mathit{opt\_expr\_list} & \rightarrow & \epsilon \\
 & | & \mathit{expr\_list} \\
\mathit{expr} & \rightarrow & \mathit{primary} \\
 & | & \mathit{var} \\
 & | & \mathit{call} \\
 & | & \mathbf{not}\ \mathit{expr} \\
 & | & \mathbf{minus}\ \mathit{expr} \\
 & | & \mathit{expr\ operator\ expr} \\
\mathit{operator} & \rightarrow & \mathbf{and} \mid \mathbf{or} \mid \mathbf{lt} \mid \mathbf{gt} \mid \mathbf{le} \mid \mathbf{ge} \mid \mathbf{ne} \mid \mathbf{eq} \mid \mathbf{conc} \\
 & | & \mathbf{plus} \mid \mathbf{minus} \mid \mathbf{multiply} \mid \mathbf{power} \\
\mathit{primary} & \rightarrow & \mathbf{nil} \mid \mathit{literal} \mid \mathit{number} \mid \text{’\%’}\mathit{name} \\
 & | & \mathbf{function}\ \text{’(’}\ \mathit{param\_list}\ \text{’)’}\ \mathit{block}\ \mathbf{end} \mid \mathit{table\_cons} \mid \text{’(’}\ \mathit{expr}\ \text{’)’} \\
\mathit{var} & \rightarrow & \mathit{name} \mid \mathit{primary\ index} \mid \mathit{var\ index} \mid \mathit{call\ index} \\
\mathit{index} & \rightarrow & \text{’[’}\ \mathit{expr}\ \text{’]’} \mid \text{’}\mathbf{.}\text{’}\ \mathit{key} \\
\mathit{call} & \rightarrow & \mathit{primary\ key\_and\_args} \mid \mathit{var\ key\_and\_args} \mid \mathit{call\ key\_and\_args} \\
\mathit{key\_and\_args} & \rightarrow & \mathit{args} \mid \text{’}\mathbf{:}\text{’}\ \mathit{key\ args} \\
\mathit{args} & \rightarrow & \text{’(’}\ \mathit{opt\_expr\_list}\ \text{’)’} \mid \mathit{literal} \mid \mathit{table\_cons} \\
\mathit{table\_cons} & \rightarrow & \text{’}\{\text{’}\ \mathit{table\_contents}\ \text{’}\}\text{’} \\
\mathit{table\_contents} & \rightarrow & \epsilon \\
 & | & \mathit{expr\_field\_list\ opt\_sc\_mf\_list} \\
 & | & \mathit{mapping\_field\_list\ opt\_comma\ opt\_sc\_ef\_list} \\
 & | & \text{’;’}\ \mathit{expr\_field\_list} \\
 & | & \text{’;’}\ \mathit{mapping\_field\_list\ opt\_comma} \\
 & | & \text{’;’} \\
\mathit{expr\_field\_list} & \rightarrow & \mathit{expr\_list\ opt\_comma} \\
\mathit{opt\_sc\_ef\_list} & \rightarrow & \epsilon \\
 & | & \text{’;’} \\
 & | & \text{’;’}\ \mathit{expr\_field\_list} \\
\mathit{opt\_sc\_mf\_list} & \rightarrow & \epsilon \\
 & | & \text{’;’} \\
 & | & \text{’;’}\ \mathit{mapping\_field\_list\ opt\_comma} \\
\mathit{mapping\_field\_list} & \rightarrow & \mathit{mapping\_field} \\
 & | & \mathit{mapping\_field\_list}\ \text{’,’}\ \mathit{mapping\_field} \\
\mathit{mapping\_field} & \rightarrow & \text{’[’ expr ’]’}\ \text{’=’}\ \mathit{expr} \\
 & | & \mathit{key}\ \text{’=’}\ \mathit{expr} \\
\mathit{opt\_comma} & \rightarrow & \epsilon \\
 & | & \text{’,’}
\end{array}
$$

# Appendix B

# Details for a Recursive Descent Parser

This appendix gives a more detailed description of how a recursive descent parser can be implemented. The grammar that is used here, is the same as in figure 3.1, and is repeated here for ease of reading.

| | |
|---|---|
| *program* | → `begin` *statement* `end` |
| *statement* | → *printstatement* `\|` *assignstatement* |
| *printstatement* | → `print (` *string* `)` |
| *assignstatement* | → *id* `=` *expression* |
| *expression* | → *number operator number* |
| *operator* | → `+` |

For each nonterminal, there should be a parse subroutine.

```
void parse_program();
void parse_statement();
void parse_printstatement();
void parse_assignstatement();
void parse_expression();
void parse_operator();
void parse_id();
void parse_string();
```

Note that there are parse functions for *id* and *string.* These tokens represent terminals, but are no terminals themselves, that is, the strings ”id” and ”string” are not returned by the lexer. The syntactic structure of these tokens is handled by the lexer. The parse functions for *id* and *string* handle the actual values.

Often, there is some subroutine to accept keywords and other nonterminals that just have to be read, but can be thrown away afterwards:

```
void accept(Terminal_token t) {
    if(current_token == t) {
       current_token = next_token();
    }
    else {
       syntax_error(t);
    }
}
```

Parsing starts with invoking `parse_program`. Then, whenever a production rule is applied, the subroutine for that production rule is called.

```
void parse_program() {
    accept("begin"); /* eat the "begin" symbol */
    parse_statement(); /* parse the statement */
    accept("end"); /* eat the "end" symbol */
}

void parse_statement() {
    switch(current_token) {
       case PRINT: /* "print" */
          parse_printstatement();
          break;
       case ID: /* some identifier */
          parse_assignstatement();
          break;
       default:
          syntax_error(current_token);
    }
}
```

The rest of this parser is similar to the code above. Note that a lot of details are left out. These parsing functions do not create an AST, neither do they generate code. They are just an example of how the parsing itself can be implemented.

# Appendix C

# Parrot Calling Conventions

This appendix describes the Parrot Calling Conventions as they are defined in [5]. Appendix C.1 describes the conventions that are used when calling functions. Conventions concerning returning value from functions are described in appendix C.2.

## C.1 Calling Conventions

The caller is responsible for preserving any environment it is interested in keeping. For that purpose, the `saveall` and `restoreall` instructions should be used. The following registers are used for calling a function:

- **P0** holds the object that represents the function.
- **P1** holds the context object.
- **P2** holds the object the function was called on (as a method call). Not used in Pirate.
- **P3** holds an array with overflow parameters. These are parameters that would not fit in a register.
- **S0** holds the fully qualified name of the function that is called.
- **I0** is *true* if the function was called with prototyped parameters. That is, the types of the parameters are available. In Lua, this is never the case.
- **I1** holds the number of parameters in the overflow array.
- **I2** holds the number of parameters in PMC registers.
- **I3** holds the return type or number of expected return values.
- **I4** holds the hash value of the function name, or 0 if there is no hash value.

- **P5-P15** hold the first 11 PMC parameters.

If there are more than 11 parameters, then P3 will hold an array that keeps all overflow parameters. These parameters will be placed starting at position 0, so parameter #12 will be placed at position 0.

## C.2 Returning Conventions

These registers are used when returning from a function:

- **I0** is *true* if the function is prototyped (always false in Lua).
- **I1** holds the number of return values in integer registers. Always 0 in Lua.
- **I2** holds the number of return values in string registers. Always 0 in Lua.
- **I3** holds the number of return values in PMC registers.
- **I4** holds the number of return values in numeric registers. Always 0 in Lua.
- **P3** holds the overflow return values in an array PMC.
- **P5-P15** hold the first 11 PMC return values.

If there are more than 11 return values, then P3 will hold an array that keeps all overflow return values. These return values will be placed starting at position 0, so return value #12 will be placed at position 0.

# Appendix D

# Internal Data Structures

## D.1 The Compiler State Structure

Pirate has several variables that are used by more than one modules. These variables are collected into one structure named *Compiler_State*. The layout of this structure is:

```
struct compiler_state {
    int errors;                 counter for number of errors
    int warnings;               counter for number of warnings
    BOOL[a] verbose;            run-time option
    BOOL runIMCC;               run-time option
    BOOL compileonly;           run-time option
    int linebuffersize;         size of buffer to store a source code line
    long linenumber;            current line number
    char *linebuffer;           buffer to store current source code line
    int tokenpos;               column position in current source code line
    int num_ids;                number of identifiers returned by lexer
    compiler_phase phase;       current phase of the compiler
    char *currentsourcefile;    name of the source file
    char *currentdestfile;      name of the destination file
    FILE *fp;                   destination file
    char *compilername;         name of the compiler executable
    ASTnode *tree;              the AST representing the source program
    Symbol_Table *table;        symbol table for the scope checker
    Function *rootfunc;         list of functions (containing instructions)
    Function *currentfunc;      current function, for fast adding
    Symbol_Table *export;       table containing exported functions
```

[a]BOOL is implemented as an enumeration of two values: 0 representing false and 1 representing true.

```
};
```

## D.2 The Symbol Table

This section describes the symbol table data structure. Only the important functions are discussed. Statistical and debug functions, such as `print_table` are not discussed.

### D.2.1 Interface of the Symbol Table

`void enter(Symbol_Table *t, char *n, ASTnode *node, BOOL is_upvalue);`

Enter a new symbol for name `n` into symbol table `t`. When the symbol table is used in the scope checker, then the current AST node is passed as the third parameter.

`Symbol *lookup(Symbol_Table *t, char *n);`

Find the symbol with name `n` in symbol table `t`. If there are multiple entries for a particular symbol, then the entry that was entered *last* is returned.

`void open_scope(Symbol_Table *t);`

Open a new scope in symbol table `t`.

`void close_scope(Compiler_State *s, Symbol_Table *t)`

Close the current scope of symbol table `t`. All symbols of the current scope are removed from the table. When a symbol in the current scope was never used (looked up), then a warning is emitted, and the number of warnings in the compiler state `s` is incremented.

`Symbol_Table *create_table(size_t s);`

Function to create a symbol table of size `s`. A reference to the table is returned.

`void destroy_table(Symbol_Table *t);`

Function to release all memory of symbol table `t`.

`int get_current_scope(Symbol_Table *t);`

Return the current scope of symbol table `t`.

`int get_table_size(Symbol_Table *t);`

Return the number of buckets of symbol table `t`.

#### Accessor functions for the Symbol Structure

`int get_scope(Symbol *symbol);`

Return the scope of symbol `symbol`.

`char *get_name(Symbol *symbol);`

Return the name of symbol `symbol`.

`BOOL is_upvalue(Symbol *symbol);`

Return *true* if symbol `symbol` is an upvalue.

`void set_used(Symbol *symbol);`

Set a flag of symbol `symbol`, indicating this symbol is used.

### D.2.2 Implementation of the Symbol Table

The symbol table is a structure with only a few fields, the most important being a pointer to an array. This array holds pointers to lists. Each list contains symbols that have the same hash value.

Collisions that occur when multiple symbols get the same hash value are resolved by linking these symbols into a list. Figure D.1 shows how this is done.

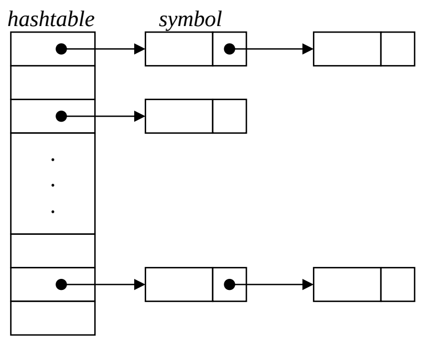


Figure D.1: Implementation of the symbol table.

A symbol is an object with the following layout:

```
struct symbol
{
    char *name;
    int scope;
    BOOL is_used;
    union
    {
       ASTnode *node;
       BOOL is_upvalue;
    }a;
    Symbol *next;
};
```

This is a data structure that keeps the name of the symbol, along with some other fields. One of those is a number that stores the scope of the symbol. This field is used in the scope checking phase. The symbol also keeps a flag indicating if it was ever looked up in the symbol table. If it is never looked up, then the symbol is not used. In this case, a warning is generated indicating that the symbol was declared but never used. Note that an identifier is only entered when it was declared local, or as a parameter.

All data structures in the symbol table are hidden from the client programmer. The only way to access data members of the data structures is through accessor methods. This is done to separate the interface from the implementation. In this way, it is possible to change the implementation without having to change the modules that actually use the symbol table.

## D.3 The Symbol Table Iterator

### D.3.1 The Iterator Interface

The iterator structure allows for iterating over all entries in a symbol table. The interface functions of the iterator are listed below.

`Iterator *create_iterator(Symbol_Table *table);`

Create an iterator object for symbol table `table`. If there are no symbols in `table`, then no iterator is created and `NULL` is returned. If the symbol table is changed, that is, a symbol is entered or removed *after* the iterator object was created, then the results of the iterator are unpredictable. The client programmer should make sure this does not happen.

`void destroy_iterator(Iterator *i);`

Destroy iterator `i`. All memory used by this object is released.

`Symbol *next(Iterator *i);`

Return the next symbol object in the symbol table of iterator `i`. If there is no next symbol, then `NULL` is returned.

`Symbol *get_current(Iterator *i);`

Return the current symbol of iterator `i`.

`void reset_iterator(Iterator *i);`

Reset iterator `i` to the first symbol in the symbol table. This function cannot fail, because there *must* be at least one symbol in the table. Otherwise, the iterator could not have been created in the first place.

## D.3.2 Implementation of the Iterator

The iterator structure and its operations belong to the same module as the symbol table. This way, the iterator can access all fields of the symbol table directly, without using inefficient function calls. This does mean, however, that the current implementation of the iterator cannot be used anymore if the implementation of the symbol table changes. In that case, the iterator implementation should be changed, too.

The iterator structure is defined as below.

```
struct iterator
{
    int currentkey;
    Symbol *currentsymbol;
    Symbol_Table *table;
};
```

The field `currentkey` is the index of the array that represents the hash table in the symbol table. The `currentsymbol` field points to the current symbol, or, if the iterator has iterated beyond the last symbol, is NULL. The last field, `table`, is the table on which the iterator operates.

## D.4 The Stack

### D.4.1 Interface to the Stack

`Stack *create_stack(size_t s);`

Create a stack with size `s`.

`void destroy_stack(Stack *s);`

Destroy stack `s`. All memory held by this object is released.

`void push(Stack *s, void *object);`

Push the item referenced by `object` onto stack `s`.

`void *pop(Stack *s);`

Pop the topmost item off of stack `s`. This item is returned.

`void *top(Stack *s);`

Get the topmost item of stack `s`.

`void *index_stack(Stack *s, int o);`

Get the item at offset `o` of stack `s`. Offset 0 will return the topmost item, offset 1 will return the item 1 position under the topmost item, offset 2 will return the item 2 positions under the topmost item, et cetera. If the offset is too high, so that the resulting index would be below the stack base, then NULL is returned. If the offset is negative, again NULL is returned.

`void clear_stack(Stack *s);`

Clear stack `s` effectively by setting the stack pointer to 0. The items that are on the stack are *not* removed, but there is no way to get to these anymore.

`int get_stacksize(Stack *s);`

Return the size of the stack. This is the sum of the number of items and the number of free slots. Note that the stack size may change, if necessary.

`int get_stackitemcount(Stack *s);`

Return the number of items currently on stack `s`.

`void set_stackname(Stack *s, char *name);`

Set the `name` field of stack `s` to `name`.

### D.4.2 Implementation of the Stack

The stack data structure is hidden from the client programmer. This means that all fields of the stack can only be accessed through accessor functions. Each of these functions takes a pointer to a stack as the first parameter.

The stack module is designed to be generic. This means that it should be able to store any kind of data. This is accomplished by declaring the data as a pointer of type `void`.

The stack data structure has four fields. Below is the definition of the stack data structure.

```
struct stack
{
    void **stack;
    char *name;
    int stacksize;
    int num_elements;
};
```

The first is a pointer to the data area that keeps the elements. This is a pointer to pointers, because the stack only stores pointers to elements. The second is a pointer to a string holding the name of the stack. This is to allow for easy debugging. The third field keeps the actual space available for storing elements. The last element stores the actual number of elements that is stored. This field is also used as a stack pointer.

The actual data is stored in a dynamically allocated array of pointers. This array is allocated when the stack is created. Now, the most important operations are described below.

When an element is pushed onto the stack, the saved pointer must be cast to a pointer of type `void`. Then, the element is saved at the position that is pointed to by the current value of `num_elements`. Then this value is incremented. Before the actual storing of the new element, the number of elements is compared to the stack size. If they are equal, an internal function is called to increase the size of the stack. When this function returns, the size is twice its original size.

The `top` operation returns a pointer to the topmost item on the stack. The `pop` does the same, but the item is removed from the stack. If a `top` or `pop` operation is done on an empty stack, NULL is returned. Before using the returned pointer, it should be cast to the appropriate type.

Unconventional in this stack implementation is the `index_stack` operation. This operation returns a pointer to the element at a specified offset from the stack top.

The `clear_stack` operation effectively removes all elements on the stack by setting the number of elements to 0.

When the stack is no longer needed, the stack should be destroyed. This is done by a call to `destroy_stack`.

# Appendix E

# Instruction Sets

This appendix describes the instructions that are generated by Pirate. Appendix E.1 describes the IMC instructions that are generated by Pirate. The IMC instructions can be found in [21], [16], [17], [18], [19] and [20]. Appendix E.2 gives a description of the Parrot instructions that are used in Pirate. The Parrot instructions are defined in [6].

## E.1 IMC Instructions

The following IMC tokens are used during code generation (in alphabetical order). Not all of them are real instructions, but are instructions to the IMC compiler. The instructions are written in context, so the complete syntax will be given.

| Instruction | Meaning |
|---|---|
| `.end` | indicates the end of a `.sub` |
| `.include "`*file.imc*`"` | include the contents of the file *file.imc* |
| `.local` *l* | declaration of a local variable *l* |
| `.result` *r* | restore a return value *r* from the run-time stack |
| `.return` *r* | save a return value *r* onto the run-time stack |
| `.sub` *f* | start a subroutine called *f* |
| `I0 = addr` *f* | get the address of *f* and store it in I0. |
| `call` *f* | call a function with name *f* |
| `endnamespace` *s* | indicates the end of namespace *s* |
| `P1 = clone P0` | make a clone of `P0` and store it in `P1` |
| `global "`*n*`" = P0` | store `P0` as a global with name *n* |
| `P0 = global "`*n*`"` | fetch a global with name *n* and put it in `P0` |
| `goto` *l* | branch to label *l* |
| `if I0 == 0,` *l* | if register `I0` equals 0, jump to label *l* |
| *l* `:` | a label called *l* |
| `namespace` *s* | indicates the start of a new namespace *s* |
| `P0 = new LuaNumber` | create a new LuaNumber PMC in register P0 |
| `ret` | return from a subroutine |
| `restoreall` | restore all 128 registers from the appropriate stacks |
| `saveall` | save all 128 registers onto the appropriate stacks |

## E.2 Parrot Instructions

This section gives an overview of Parrot instructions that are used in Pirate, and do not have an equivalent instruction in IMC.

| Instruction | Meaning |
|---|---|
| `find_lex P0, "`*id*`"` | find the lexical *id* and put it in P0 |
| `find_lex P0,` *n*`, "`*id*`"` | find the lexical *id* at depth *n* and put it in P0 |
| `invoke` | invoke the subroutine in P0 |
| `invoke P`*x* | invoke the subroutine in P$x$ |
| `new_pad` | push a new scratchpad onto the lexical stack |
| `pop_pad` | remove the topmost scratchpad |
| `restore P0` | restore the top of the run-time stack into P0 |
| `save P0` | save the PMC in P0 on the run-time stack |
| `store_lex "`*id*`", P0` | store the PMC in P0 as lexical *id* |
| `typeof I0, P0` | store the type of PMC in P0 in I0 |
| `typeof S0, P0` | store the type of PMC in P0 in S0 |

# Bibliography

[1] `http://www.gnu.org/software/flex`

[2] `http://www.gnu.org/software/bison/bison.html`

[3] `http://www.southern-storm.com.au/treecc.html`

[4] `http://www.parrotcode.org`

[5] `http://dev.perl.org/perl6/pdd/pdd03_calling_conventions.html`

[6] `http://dev.perl.org/perl6/pdd/pdd06_pasm.html`

[7] `http://www.parrotcode.org/docs/embed.pod.html`

[8] `http://dev.perl.org/perl6`

[9] `http://lua-users.org/lists/lua-l/2001-08/msg00191.html`

[10] Aho, A.V. et. al., *Compilers, principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1988.

[11] Fischer, C.N. et. al., *Crafting a compiler with C*, Benjamin/Cummings, Redwood City, CA, 1991.

[12] Ierusalimschy, R. et. al., *Reference Manual of the Programming Language Lua*, October 2000.

[13] Ierusalimschy, R., *Programming in Lua*, January 2003.

[14] Levine, J.R. et. al., *lex & yacc*, O'Reilly, Beijing, 1995.

[15] Pierce, C., *Perl 5 in 24 uur*, Academic Service, Schoonhoven, 2000.

[16] Tötsch, L., *IMCC - Calling Conventions*, Online, available in Parrot 0.0.10 distribution

[17] Tötsch, L., *IMCC - Documentation*, Online, available in Parrot 0.0.10 distribution

[18] Tötsch, L., *IMCC - Operation*, Online, available in Parrot 0.0.10 distribution

[19] Tötsch, L., *IMCC - Parsing*, Online, available in Parrot 0.0.10 distribution

[20] Tötsch, L., *IMCC - Running*, Online, available in Parrot 0.0.10 distribution

[21] Tötsch, L., *IMCC - Syntax*, Online, available in Parrot 0.0.10 distribution

[22] Watt, D.A. et. al, *Programming language processors in Java: compilers and interpreters*, Pearson Eduction, Harlow, England, 2000.