Writing an Interpreter in Object Pascal:

Part 3: Library Support

Herbert M. Sauro
Seattle, WA

Ambrosius Publishing

First Edition, version 1.0
Published by Ambrosius Publishing and Future Skill Software
books.analogmachine.org

Typeset using LaTeX 2_ε, TikZ, PGFPlots, WinEdt,
and 11pt Math Time Professional 2 Fonts

pgf version is: 3.1.9a

pgfplots version is 1.18.1

Limit of Liability/Disclaimer of Warranty: While the author has used his best efforts in preparing this book, he makes no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. The advice and strategies contained herein may not be suitable for your situation. Neither the author nor publisher shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages. No part of this book may be reproduced by any means without written permission of the author.

ISBN 10
ISBN 13:

Printed in the United States of America.

Mosaic image modified from Daniel Steger's Tikz image (http://www.texample.net/tikz/examples/mosaic-from-pompeii/

Contents

Preface

This is Part 3 of a series I am writing on how to write interpreters in Object Pascal. Part 1, published in early 2019, described the construction of a tokenizer and syntax checker for a simple procedural language I named after my dog, Rhodus. Part 1 ended with a simple calculator application and a full language recognizer. In Part 2, a virtual machine was built that could run Rhodus programs. In the version described in Part 2, bytecode was generated directly as a Rhodus program was parsed. In Part 3 we separate parsing from code generation via an abstract syntax tree. With the completion of Part 3, we will have a serviceable interpreter that supports user-defined functions, built-in functions, strings, and list support together with a range of looping constructs. The biggest outwards change is support for modules which can be imported.

There are many people and organizations whom I should thank, but foremost must be my infinitely patient wife, Holly, and my two boys Theodore and Tyler, who have put up with the many hours I have spent working alone. I want to thank Holly, in particular, for helping me edit the text. Naturally, I am responsible for the remaining errors, or as a contributor (Marc Claesen) to StackOverflow once humorously remarked, ‘Making the manuscript error-free is left as an exercise for the reader.’.

Many thanks to the authors of the TEX system, MikTeX (2.9), TikZ (3.1.9a), PGFPlots (1.18.1), WinEdt (10.2), and Affinity Designer (1.8), for making available such amazing tools for technical authors. It is these tools that make it possible for individuals like myself to publish. Also, not forgetting companies such as Createspace/KDP, and Ingramspark that make independent publishing possible at affordable prices. Finally, I should thank Michael Corral (http://www.mecmath.net/) and Mike Hucka (www.sbml.org), whose LATEX work inspired some of the styles I used in the text.

All code can be obtained from: https://github.com/penavon/BookPart2

Decemeber 2021
Seattle, WA

HERBERT M. SAURO

Source Code

The source code can be obtained from GitHub at:

https://github.com/penavon/BookPart3

1

Introduction to Version III

1.1 Introduction

Welcome to Part 3 of the book series on writing an interpreter using Object Pascal. In Part 1 we looked at how to tokenize source code into tokens, how to parse those tokens for syntactical structure, and how to build a simple evaluator of infix expressions. Along the way, we introduced unit testing via DUnitX, and we covered some of the essential concepts in parser theory.

In Part 2, we developed a virtual machine and emitted code from the parsed Rhodus scripts. We also talked a lot about memory management. At the end of Part 2 we had a serviceable interpreter that supported strings, lists and user functions.

What are we going to do in Part 3? Part 3 will focus on some major internal changes especially how we emit bytecode, how we can support external and builtin modules, and improve error handling at the parsing stage. We will also extend the language to support arrays, that is homogenous arrays of data and finally produce an embeddable version. For strings, lists, and arrays we also have a basic object model so that strings, lists, and arrays are genuine objects in the sense they have data and associated methods. To give you a flavour of the outward changes here are some examples of scripts in version 3 that illustrates some of the new features:

```
import math
```

```
x = math.sin (1.2)

a = "abcdefg"
length = a.len()

a = array ([[1,2],[3,4]])
nRows = a.len (0)
```

A significant change in the syntax has also been made in verison 3. After much deliberation I decided to use from now on square brackets to define lists, for example:

```
alist = [[1,2],[3,4]]
```

I did this because I realized that arrays didn't need their own literal syntax and I could reuse the list syntax. At that point I thought it better to use the Python syntax for lists which uses square brackets. The original intent was that arrays would look like MATLAB arrays which does use square brackets but I realized that this is an awkward syntax for specifying multidimensional arrays.

A new global method called `array` can be used to create arrays. e.g `a = array ([1,2,3])` It also means we can do tricks like:

```
>> a = array ([[0]*3]*3)
>> println (a)
>> a
[   0.0000,     0.0000,     0.0000,
    0.0000,     0.0000,     0.0000,
    0.0000,     0.0000,     0.0000]
```

1.2 Components of an Interpreter

Let us remind ourselves again what the various components are that make up an interpreter. At the most basic level, an interpreter will read a script of instructions and execute them. There are various ways this can be accomplished; for example, the interpreter can go line by line executing the instruction on each line. Early versions of BASIC used this technique. The disadvantage is that in a loop, one will be repeatedly decoding the instructions as there is no record of what instructions were interpreted in the past. The advantage is that such interpreters are relatively easy to write. More advanced interpreters, also developed early on in the history of software, convert a programming language into lower level code that is more easily executed. The advantage is that the original source code only needs to be decoded once; after that, the application executes the simpler code. The simpler code is often called intermediate code or virtual machine code. The application that executes the intermediate code is called the **virtual machine**. The conversion of the high-level source code into intermediate code is called compilation. Figure 1.2 shows a high-level view of these stages.

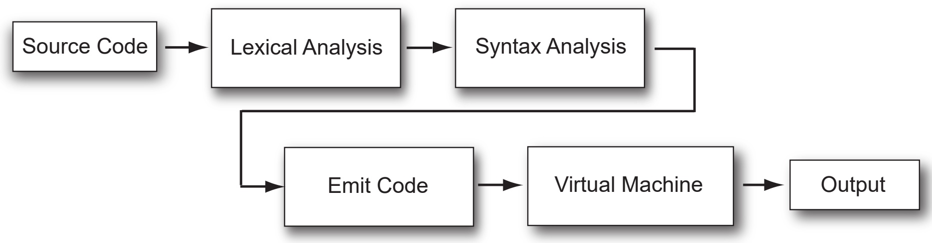

Figure 1.1 Simplified flow from source code to output in version 2 of the Rhodus interpreter.

In version 3 we are going to expand this picture and insert another stage between syntax analysis and code generation. This is the so-called abstract syntax tree, Figure 1.2, or AST. We'll have an entire chapter devoted to this topic. This division means that the first parsing stage only deals with syntax, not necessarily meaning. For example one might type `a = b`, this is syntactically correct but we don't know whether `b` has been declared previously or not. Such questions are considered by the second stage when we build the AST.

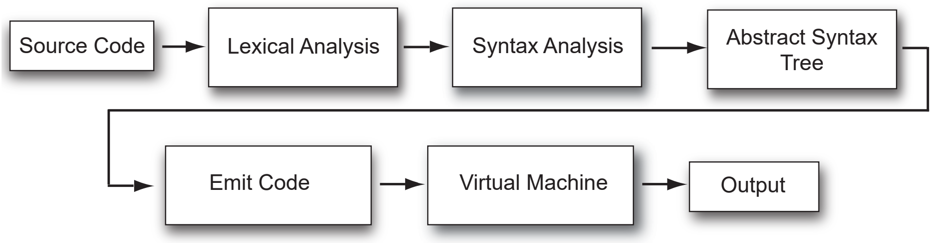

Figure 1.2 Simplified flow from source code to output in version 3 of the Rhodus interpreter incorporating the abstract syntax tree stage.

1.3 Language Changes

There are quite a few visible changes that the user will see in version 3. These include support for modules, making user functions first class objects and support for arrays.

1.3.1 Modules

Any self-respecting computer language needs to be able to reuse existing code. For example no one should attempt to write their own sine or cosine functions, or string routines to search and manipulate strings. Such routine activities are generally provided in the form of external libraries that a programmer can use.

Many languages use a special keyword, `import`, to specify an external library. For example in Python, to use the math library we would use:

```
import math
```

In Object Pascal we would use the `uses` keyword, for example:

```
uses math;
```

In Rhodus we will use the `import` syntax. For example, let's say we create a really simple statistics module:

```
// Stats module
version = 1.0

// Compute the means of a list of numbers
function mean (values)
  sum = values.sum ()
  return sum/values.len ()
end
```

We will save this script to a file called `stats.rh`. We would then use this module as follows:

```
import stats

println ("Version number = ", stats.version)

values = [1,4,5,6,2,4,5,8,9]
answer = stats.mean (values)
println ("The mean is: ", answer)
```

The first thing to note is the dot notation for accessing items inside the module. This is a very common syntax used by many programming languages. We show two situations, accessing a variable called `version` and calling a user function called `mean`. Note that like any variable we can also assign a new value to `version`, that is:

```
stats.version = 2.0
```

At some point in the future we will allow users to define constants which can't be changed rather than use variables for important information.

This is the basic syntax and user experience for using modules. Rhodus 3 includes a series of built-in modules; two of these, strings and lists are loaded at startup. The others have to be imported using import. As an example, to find the square root of a value, first import the math module, for example:

```
import math
x = math.sqrt (25)
```

In version 3, Rhodus also looks for a startup file (startup.rh) in a directory called Modules. import statements can be put into this file if one would like certain modules to be always available in the main startup module.

With version 3, nine built-in modules are provided. The currently loaded modules can be listed by using the modules method:

```
>> modules()
["time","os","file","strings","lists","math","random","config","arrays"]
```

Note this list only includes the module names. All modules whether built-in or user defined also get a free method called dir(). This method will return a list of all variables and methods accessible in the module. For example, to find out what methods and variables are available in the math library we can call:

```
println (math.dir())
```

This will output the list:

```
["min","e","cos","dir","toRadians","tan","round","exp","toDegrees","abs","ceil",
"atan","sin","log","max","pi","ln","asin","sqrt","acos","floor"]
```

When the Rhodus engine starts up, it creates a module called _main_. This is what you interact with at the Rhodus console. Any new symbols you create, such as a = 5, will be stored in _main_. A method called main() can be used to return a reference to the main module. If you type main() you'll get:

```
main()
Module: _main_
```

Like any other module you can see what's inside the symbolTable kept in _main_ by using dir(), for example:

```
>> main().dir()
["array","a","dis","asc","stackInfo","help","assertTrueEx","assertFalseEx",
"lists","dir","sys","float","main","readString","strings","math","getAttr",
"os","readNumber","modules","int","symbols","mem","chr","type"]
```

What it lists in this case are the modules currently accessible from _main_. You get the same list if we just type dir() on its own. In fact, to access anything from _main_ you don't have to qualify the name with `_main_`. For example, to access `int` we don't need to type `main().int (4.5)`, just `int(4.5)` will do.

Let's assign a variable and ask for the list again:

```
>> astring = "This is a string"
>> main().dir()
["array","a","dis","asc","stackInfo","help","assertTrueEx","assertFalseEx",
"lists","astring","dir","sys","float","main","readString","strings","math",
"getAttr","os","readNumber","modules","int","symbols","mem","chr","type"]
```

You'll noticed that a new symbol is now in the main module, called `astring`. To summarise:

- `dir()` Every module gets a method `dir()` that lists all the variables and functions in a given module, e.g `math.dir()`
- All modules get a series of free methods e.g `int (3.4)`.
- Modules can be loaded using the `import` statement.

We will describe the build-in modules in more detail in a later chapter. Before we leave modules, however, there are a couple of issues that needed to considered. For example where does Rhodus look for modules when it imports them? What happens if a module is imported twice?

The first question, where to look, is easy. We'll use a similar mechanism to Python and have a list containing paths where Rhodus should look. We'll store this path in the built-in module `sys`. Since the path is a list we can add additional locations where Rhodus should look if we need to. Note that the path variable is locked meaning you can't change its value by assignment. To change the path you must use the append method that is part of the list object. For example to use `append` to add to the search path we would use:

```
println (os.path)
os.path.append ("c:\\myscripts")
```

At startup, two paths are included, the current working directory and a path to the Modules directory.

The second issue is what happens if you import a module twice? The simple answer is nothing. If you attempt to import a module with a name that is already present in the list of loaded modules, Rhodus will ignore the attempted import. This restriction may change in the future but for now, that's what happens.

Finally a brief comment on help. To get help at the console on any symbol in Rhodus, put a question mark in front of the symbol, for example:

```
?math.min
Returns the minimum of two numbers: math.min (3, 5)
```

Programmatically you call also get help using the help command, for example:

```
help(math.sin)
Returns the minimum of two numbers: math.min (3, 5)
```

1.3.2 User Functions

The other change to version 3 is that user functions or built-in functions are now first class objects. What this means is that the name of a function can be treated like any other variable. For example you can type:

```
x = math.sin
```

Note that function brackets are not included. What this does is create a new copy of `math.sin` and assign it to the variable `x`. Yes you heard right, it makes a copy. This is due to the way garbage collection is handled in the current versions of Rhodus. Other languages, such as Python, use reference counting where instead of a copy of a function being made, a reference to the function is assigned to a variable. We'll have more to say about this in a later chapter. From the user point of view, this shouldn't be of concern. However, you might not want to do something like the following because that will use up memory.

```
x = {}
for i = 0 to 1000000 do
    x.append (math.sin)
end
```

When we copy the function into another variable we can still call the copied function since they are now just like other variables. We can also pass functions as arguments to other functions. For example:

```
func = stats.mean
answer  func (values)

function fcn (method, values)
  result = method(values)
end

answer = fcn (stats.mean, values)
```

```
function callme()
    return "I was called"
```

```
end

x = callme
y = x()
println (y)
```

Here is another example:

```
function square(x)
    return x*x
end

function cube(x)
    return x*x*x
end

function compute (fcn, x)
    return fcn (x)
end

println (compute (square, 4))
println (compute (cube, 4))
```

1.3.3 Lists and Strings as Objects

There has been change to the object model for things like lists and strings. Both lists and strings now behave more like objects. In particular, they now have methods associated with them. For example, we can get the length of a string using the `len()` method:

```
>> s = "hello"
>> println (s.len())
5
```

Because they are more like genuine objects you can also do this:

```
>> x = "hello".len()
>> println (x)
5
```

Like modules, you can apply the `dir()` method, for example:

```
>> "hello".dir()
["len","find","toUpper","toLower","left","right","mid","trim","split","dir"]
```

Method objects are not like user defined functions in they cannot be copied. Help for a given object method can be obtained by simply typing the object method without the calling brackets. For example:

```
>> "a".left
Object Method: Returns the left n chars of a string: a.left (5)
```

If any operation returns an object then an object method can be applied to the result. This means you can do odd things like:

```
>> s = "hello"
>> s.toUpper().toLower().toUpper().toLower().toUpper()
HELLO
```

Note that the string stored in the variable `s`, is changed to `HELLO`.

Lists are also full objects so that they too have a suite of methods associated with them:

```
>> [1,2,3].dir()
["len","append","remove","sum","pop","max","min","dims","dir"]
```

The new array type has a similar method:

```
>> array([1,2,3]).dir()
["len","shape","ndim","sqr","add","sub","dir"]
```

1.3.4 Homogeneous Arrays

The other big change to version 3 is the introduction of arrays, that is structures that can hold homogeneous data. Arrays are used when we need faster access, often for applications that do numerical work where we're dealing with blocks of data. I went through a number of iterations on the handling of arrays and decided in the end to follow the example the numpy package that Python uses. Rather than coming up with an entirely new syntax for describing literal arrays, we repurpose the list syntax. To do this a new global level method is available called `array()`. The `array` method also features a new idea which is a variable number of arguments although its pretty limited at the moment. The `array` method can be used to do two different things. On the one hand it can be used to define an array of a given size, for example:

```
m = array (3,4)
```

The variable `m` will hold a 3 by 4 array where by default, entries are set to zero. Higher-dimensional arrays can also be specified, for example:

```
>>a = array (2,3,4)
```

This yields a 2 by 3 by 4 array, or 24 elements in total. By default, arrays hold double values, at a future date this will be extended.

array can also accept a list which the array method will convert into an array, for example

```
a = array ([[1,2,3], [4,5,6], [7,8,9]])
```

The above array yields a 3 by 3 array. Note, as mentioned before, in version 3, I decided to use square brackets for lists rather than curely brackets. I did this because I realized that arrays didn't need their own literal syntax and I could reuse the list syntax. At that point I thought it better to use the Python syntax for lists which uses square brackets. The original intent was that arrays would look like MATLAB arrays which does use square brackets but I realized that this is an awkward syntax for specifying multidimensional arrays beyond 2D. With the release of curely brackets I can now use {} for specifying maps.

Like lists and strings, arrays are also treated as objects so that one can do the following:

```
>>println (a.shape())
[3,3]
```

Or even:

```
>>array([[1,2],[3,4]]).shape()
[3,3]
```

You can also get the individual dimensions using len where the argument of len is the nth dimension you are interested in:

```
>>println (a.len(0))
3
```

Indexing arrays is done in the same way you index lists. Like lists, indexing in arrays starts at zero, for example:

```
x = a[1,2]
```

Here is an example of iterating through each element in an array, a:

```
a = array([[1,2,3],[4,5,6],[6,7,8]])
for i = 0 to a.len(0) - 1 do
    for j = 0 to a.len(1) - 1 do
        print (a[i,j], " ")
    end
    println ()
end
```

Certain arithmetic operations are also possible with arrays, for example:

```
>>a = array([[1,2],[3,4]])
```

```
>>println (10 + a)
[[11,12],[13,14]]
```

or

```
>>a = array([[1,2],[3,4]])
>>b = array([[5,6],[7,8]])
>>println (a + b)
[[6,8],[10,12]]
```

In the above case, it should be clear that the dimensions of the two arrays must match in order for the sum to be computed.

The math operations can be divided into at least two groups, pair-wise element operations and matrix arithmetic as defined in linear algebra. Pair-wise operation is straightforward, and can be easily applied to arrays of any dimension. The one constraint is that the arrays should be the same shape. For example pair-wise multiplication requires both arrays to be exactly the same shape, multiplication is then performed between each corresponding entry.

Not all matrix operations are easily transferred to higher dimensions. For example, the inverse of a matrix A^{-1} is confined to two dimensional arrays. I am not personally aware that the inverse operation is defined for higher dimensions. Likewise many of the other more complex operations such as LU decomposition, QR factorisation, or finding eigenvalues is confined to two dimensional arrays. Finally matrix multiplication can be defined for higher dimensional arrays but it tends to be a rare requirement and its far more common to multiply two dimensional arrays.

There therefore appears to be a clear difference in the kinds of operations one is likely to do with an array compared to a matrix. As a result, all matrix related operations will be confined to two dimensional arrays, while more general operations will be applicable to any n-dimensional array, which would include pair-wise arithmetical operations.

We will have much more to say on this matter in a later chapter.

1.3.5 Built-in Libraries for lists, strings, arrays and matrices

A number of built-in libraries are provided to add additional support to lists, strings, arrays and 2D arrays we'll call matrices. These provide method that do not naturally belong to the objects themselves. The names for these built-in libraries are `strings`, `lists`, `arrays` and `matrix`. The contents of each of these can be obtained using `dir()`, for example:

```
>> import matrix
>> matrix.dir()
["ident","sub","dir","add","inv","rand","randi","mult"]
```

As with other module methods, help can be obtained using help:

```
>> import matrix
>> help(matrix.add)
Add two 2D matrices: m = matrix.add (m1, m2)
```

`inv` is the inverse matrix method. The following code shows an example of this in use:

```
>> import matrix
>> // Generate a 3 by 3 matrix with random entries, 0 to 1
>> m = matrix.rand(3,3)
>> minv = matrix.inv (m)
```

1.4 Error Handling

Another thing users should see in Rhodus 3 are better error messages during compilation. For example the following fragment now issues a more informative error message:

```
>>for i = 0 to 10 do println (i)
ERROR [line 1, column: 30] expecting key word: <end>
```

or this one:

```
>>for i = 0
ERROR [line 1, column: 10] expecting "to" or "downto" in for loop
```

or this one:

```
>>x+
ERROR [line 1, column: 2] expecting a literal value, an identifier or
 an opening '('. Instead I found "+"
```

1.5 Grammar Specification for Version 3

There have been some small but critical changes to the language grammar since Version 2. The changes revolve around the primary production rules. Below is the specification for Rhodus 3:

```
mainProgram          = statementList [ ';' ] endOfStream
statementList        = statement { [ ';' ] statement }

statement            = assignment | forStatement | ifStatement
                     | whileStatement | repeatStatement
                     | returnStatment | breakStatement
                     | switchStatement | importStatement
                     | function | expression | endOfStream

list                 = '{' [ expressionList ] '}'
expressionList       = expression { ',' expression }
assignment           = variable '=' expression
function             = FUNCTION identifier '(' [ argumentList ] ')' functionBody
functionBody         = statementList END
argumentList         = argument { ',' argument }
argument             = identifier | REF variable
returnStatement      = RETURN expression
breakStatement       = BREAK

relationOpExpression = simpleExpression
                     | simpleExpression relationalOp simpleExpression

expression           = relationalExpression
                     | relationalExpression BooleanOp relationalExpression

simpleExpression     = term { addingOp term }
term                 = power { multiplyOp power }
power                = { '+' | '-'} primary [ '^' power ]

factor               = '(' expression ')'
                     | identifier
                     | integer
                     | float
                     | string
                     | NOT expression
                     | TRUE
                     | FALSE
                     | list

// The follow five rules were derived using left-recursion,
// and are new to version 3. See text for details

primary              =  factor primaryPlus

primaryPeriod        = '.' identifier primaryPlus
primaryFunction      = ( exp ) primaryPlus
primaryIndex         = [ exp ] primaryPlus

primaryPlus          = primaryPeriod
                     | primaryFunction
                     | primaryIndex
                     | empty

addingOp             = '+' | '-'
multiplyOp           = '*' | '/' | MOD | DIV
relationalOp         = '==' | '!=' | '<' | '<=' | '>=' | '>'
BooleanOp            = OR | AND | XOR
whileStatement       = WHILE expression DO statementList END
```

```
repeatStatement      = REPEAT statementList UNTIL expression
forStatement         = FOR identifier = forList DO statementList END
forList              = value TO value | value DOWNTO value
ifStatement          = IF expression THEN statementList ifEnd
ifEnd                = END | ELSE statementList END
switchStatement      = SWITCH simpleExpression switchList END
switchList           = { CASE INTEGER ':' statementList } ELSE statementList
importStatement      = IMPORT fileName
```

In the grammar, everything in square brackets is optional, and the vertical line represents 'or'. There have been some important changes to the grammar since Part 2. In Rhodus 3 you can type the following:

```
a = m.func ("abc")[5](math.pi)
```

In words, this reads, in the module, string or list, `m`, a function called `func` is called with one string argument, this returns a list which we index at position 5 which in turn returns another function which we call with argument `math.pi`. Admittedly contrived but we should be able to deal with such code. Or what about something like:

```
>>"abc".toUpper().toLower().toUpper().toLower()
abc
```

We will describe how to deal with such expressions in Chapter 3.

1.6 Useful Reading

Introductory Books

1. Ball, Thorsten. Writing A Compiler In Go. Thorsten Ball, 2018.

2. Kernighan, Brian W.; Pike, Rob (1984). The Unix Programming Environment. Prentice-Hall. ISBN 0-13-937681-X.

3. Nisan, Noam, and Shimon Schocken. The elements of computing systems: building a modern computer from first principles. MIT press, 2005.

4. Parr, Terence. Language implementation patterns: create your own domain-specific and general programming languages. Pragmatic Bookshelf, 2009.

5. Robert Nystrom. Crafting Interpreters, genever benning, 2021. ISBN 978-0-9905829-3-9.

More Advanced Books

1. Jim Smith, Ravi Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kauffmann, June 2005

2. Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques and Tools (also known as The Red Dragon Book), 1986.

Source Code

1. Mak, Ronald. Writing compilers and interpreters: an applied approach/by Ronald Mark. 1991

Note, this is the first edition, 1991. The code is in C, which I found to be understandable. The later editions that use C++ are not as clear. The issue I found is that the object orientated approach that's used tends to obscure the design principles of the interpreter and requires much study to decipher, The C version is much more straightforward.

2. Wren: `https://github.com/wren-lang/wren`.

Of the open source interpreters on GitHub, I found this to be the easiest to read. It's written by Bob Nystrom in C, the same person who is writing the web book: Crafting Interpreters `https://craftinginterpreters.com/`.

3. Gravity: Another open source interpreter worth looking at is Gravity (`https://github.com/marcobambini/gravity`). Gravity, like Wren, is also written in C.

4. If you prefer Go, then the source code to look at is the interpreter written by Thorsten Ball (see book reference above).

There are umpteen BASIC interpreters and other languages that can be studied.

2

Internal Changes

2.1 Introduction

Before we embark on building the new parser, we should first talk about the internal structure of the new modules.

In order to deal with modules, allowing user functions to be first class and having a very basic object model for lists and strings, quite a few internal changes were made to the code compared to version 2. In the new version the basic unit is the module. The structure of a module is shown in Figure 2.1. Every module has a name which can be assigned by the user. The only modules that have fixed names is the main module, called `_main_`, and a series of built-in modules such as `math, os`, etc.

The internal structure of a module is given in Figure 2.1. The three important components are the code block, the constant table and the symbol table. User functions have a similar structure. The code block is where bytecode is stored, the constant table is where literal constants such as doubles, lists and strings are stored, and finally the symbol table which stores any variables that the user might have introduced, including user functions.

When you start up the Rhodus repl (i.e the interactive console), Rhodus first creates the main module, `_main_` then waits for the user to type in code. When the user enters code, its gets compiled into the main module code block. Control is then handed over to the virtual machine to actually run the code. Once that is complete, control returns to wait for the user to type in more code and so the cycle repeats.

The constant factor in all this are the symbol and constant values tables, more precisely the module level symbol and constants table which we can see in Figure 2.1. Whenever

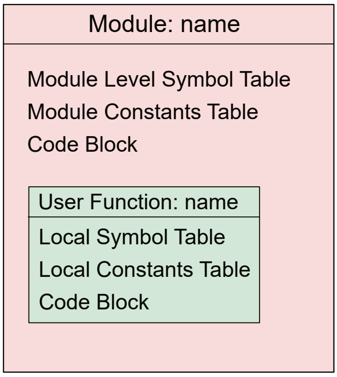

Figure 2.1 Internal structure of a module.

we type something like `a = 1.2`, its first gets converted into code, the code is run which causes the symbol `a` to appear in the symbol table along with the value `1.2` in the constants value table. In essence, what code does is manipulate the symbol table. This makes the symbol table a very important structure in an interpreter.

The final thing left to mention in Figure 2.1 are the user functions. The figure only shows one but there could be many of these in a single module. User functions are almost little modules within the big module. They have their own symbol table, constant table and code block. The big difference is how the user function stores its local symbols and the fact that a user function can accept inputs and of course return values. The module level and user function symbol tables operate in quite different ways. The module level symbol table retrieves symbols by name. Thus with the code `a = 1.2`, a new symbol is created with the name `a` and the number `1.2`. If we ever want to reference that symbol we locate it using its name `a`. In Rhodus we store the module level symbol tables using a dictionary which makes it reasonable fast to locate a given symbol.

The user function symbol tables are different. Symbols in a user function are stored by way of an integer index that references locations on the runtime stack. One would imagine that this would make accessing symbols in a user function faster compared but in practice it appears not to be the case, suggesting that the dictionary lookup at the module level is quite fast.

2.2 Updates to the Byte Codes

There are some updates to the byte codes to accommodate the new changes. The load and store opCodes are changed and expanded to those shown in Table 2.1. One pair, the load and save symbols, are used when the symbols are in the currently running module. The second pair, the load and save attr opCodes are used when a module needs to access symbols in a different module. For example, the following code:

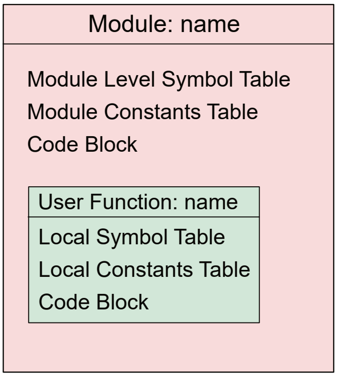

Figure 2.2 Simplified flow from source code to output in version 2 of the Rhodus interpreter.

```
a = b
```

will first use `load` to push the value of `b` obtained from the current module symbol table onto the stack and then use `save` to store the value that's on the stack to the symbol `a`, also in the symbol table of the current module. In contrast, for the code:

```
a = m1.b
```

will first use `load` to push the value of `m1` onto the stack. `loadAttr` will then be used to pop off the value of `m1` and load the value of `b` in module `m1` onto the stack. `saveSymbol` will store the value on the stack to the symbol `a` that resides in the symbol table of the current module. The main difference is that the secondary opCodes expect the stack to hold a reference to a module, where as the symbol opCodes will use the current module.

The reason for having two types of load and save was to try to improve the performance of code accessing symbols within the module the code resides. To be completely symmetric we could have insisted that any code accessing symbols in the module it resides in, should also push a reference of the module onto the stack and then used the attr opCodes to access the symbol. However this would have slowed down the load and store operation within a module. Instead, before we run a code block we pass a reference to the module the code belongs to so that `load` and `save` know what module the current module is.

The only other new opCode is `importModule` and this is called when the compiler encounters the Rhodus code `import moduleName`. The other minor change is that the `bultin` OpCode has been merged with the `call` opCode. Overall there haven't been many changes at all to the set of OpCodes we use in the virtual machine.

OpCode	Description
`load`	Load symbol in the current module onto the stack.
`save`	Store the current item on the stack to a symbol in the current module.
`loadAttr`	Pop a module reference from the stack and load value in that module onto the stack.
`saveAttr`	Pop a module reference from the stack and store the value to the symbol in that module.

Table 2.1 New load and store opCodes

2.3 Examples of Byte Code Programs

I think the best way to describe the operation of the interpreter is to look at some bytecode programs. The first thing to note is that the Rhodus virtual machine is a stack based machine. That is all operations occur around a stack. For example, to add two numbers, 2 + 3, we first push the operands 2 and 3 onto the stack, then apply the addition operator and push the result back onto the stack. What's left on the stack is the sum. The stack is used as a temporary place to hold data to which operations are applied. As simple as it is, the stack turns out to be an incredible versatile structure. So versatile that we can build a general purchase computer around one. Interestingly, real hardware based computers such as the ARM or Intel microprocessors are not stack based but register based. The reason for the dominance of register based machines is a combination of historical accident and better performance.

Let's start with some simple examples such as assigning a value to a variable. The following Table 2.2, shows four such examples. One key instruction is `store a`. This takes whatever is on the stack and saves it to a variable called `a`. The other key instruction is `load a`. This pushes onto the stack whatever value is stored in variable `b`. The variables `a` and `b` along with the values associated with them are kept in the symbol table. We already see at this simple stage, an active interaction between the virtual machine and the symbol table. Most operations in the virtual machine either involve the stack or the symbol table with data moving back and forth between the two as a program executes. Some instructions will manipulate the stack while others will involve moving data between the stack and the symbol table. For example, arithmetic operations will only involve operations on the stack. A few instructions just manipulate the symbol table, for example `inc` and `dec`.

With just three bytecodes, `push`, `load`, `store` and the arithmetic operations such as `add` and `mult` we can do a lot. However, computer programs need to do one other thing to make them really useful, that is conditional branching. For that, we have a number additional bytecodes such as `isGt`, and `jmpIfTrue`. Add to that we also need some Boolean operations, in particular `OR`, `AND` and `NOT`. Given that we need to deal with numbers as well

Code	Bytecode	Code	Bytecode

`a = 2`	`0 pushi 2`
	`1 store a`
		`a = 2 + 3`	`0 pushi 2`
			`1 pushi 3`
			`2 add`
			`3 store a`

`a = 2 + 3`	`0 pushi 2`
	`1 pushi 3`
	`2 add`
	`3 store a`
		`b = a * 2`	`0 pushi 2`
			`1 load a`
			`2 mult`
			`3 store a`

Table 2.2 Code generated simple expressions

.

as boolean values, the stack needs to be able to handle different data types. That is, not only numbers, but also data such as True and False values. We could of course model True and False using the numbers 1 and 0. By convention, 0 is often considered False and 1, True. Using such numbers is in fact what the computer does at the hardware level. But being human we like to deal with higher level concepts and since Object Pascal has the notion of True and False we might as well use these rather than just 0 and 1.

With this set of bytecodes we could probably do almost everything. All higher level constructs in a computer language can be described using only these bytecodes. The only thing that perhaps can't be done is implementing subroutines, but perhaps with enough imagination one could even do that. However, rather than being so sparse with our list of bytecodes, most stack based virtual machines supplement the list with a range of other instructions such as calling subroutines and in particular indexing operations. For now lets see we what we can do with the minimal set of bytecodes. In part II of the series there was some discussion of how loops and conditional statements were dealt with we didn't discuss the bytecode that could represent such structures. this is what we'll do here.

Repeat/Until

Let's first consider the repeat/until loop as its probably the simplest. We'll really make it simple by just considering the statement: `repeat until True`, that is no content to the loop itself. The bytecode for this is surprisingly simple, just two instructions, Table 2.3.

The first instruction `push True` pushes True onto the stack. The second instruction is a relative jump instruction. It pops the value off the stack and checks to see if the value is False. If False it jumps -1 instructions. The minus means it jumps back which takes it back to the `push True` instruction, and we start again. Luckily the until statement expression is

Code	Bytecode
repeat	0 push True
until True	1 jmpIfFalse -1

Table 2.3 Code generated for an empty `repeat` loop

.

Code	Bytecode
repeat	0 push 7
a = 7	1 store a
until True	2 push True
	3 jmpIfFalse -3

Table 2.4 Code generated for a non-empty `repeat` loop

.

`True`, so that it will not jump back and essentially get use out of the loop. If we'd set the expression to `False` we would have gone into an infinite loop.

What happens if we put a simple statement inside the repeat loop, such as `a = 7`? The result is shown in Table 2.4. The number of instructions expands to four. The extra two instructions are in front of the `push True`. The `jmpIfFalse` has been modified to jump back three steps corresponding to the start of the `a = 7` statement.

While Loop

Next in line is the while loop. Let's first look at an empty while loop, such as `while False do end`.

It doesn't take many instructions to implement a while loop, just three. We start by pushing False from `while False do` onto the stack. The next instruction, `jmpIfFalse`, is used

Code	Bytecode
while False do	0 push False
end	1 jmpIfFalse 2
	2 jmp -2

Table 2.5 Code generated for an empty `while` loop

.

Code	Bytecode
`while a > 5 do`	`0 load a`
`  a = a - 1`	`1 pushi 5`
`end`	`2 isGt`
	`3 jmpIfFalse 6`
	`4 load a`
	`5 pushi 1`
	`6 sub`
	`7 store a`
	`8 jmp -8`

Table 2.6 Code generated for a `while` loop

.

to check for the result of while expression. If false it jumps forward two instructions. If it does this its jumps right out of the while loop. In the process it jumps over the last instruction which is an unconditional `jmp` which jumps us two instructions back to the beginning and so we start again. The initial instruction that pushes `False` onto the stack would in practice be the result of evaluating a while expression such as `a > 4` which would evaluate to `True` or `False`. If we add actual statements inside the while loop, these will generate bytecode between line 0 and line 1 with a corresponding change to the jump distances. As an example, Table 2.6, shows a while loop with an assignment in the body of the loop. The first three instructions concern themselves with evaluating `a > 5`. After than its the same as before except we have a different body statement: `a = a - 1`.

For Loop

A much more complicated loop to model in bytecode is the `for` loop. To keep things simple lets again consider an empty loop first such as `for i = 1 to 5 do end`. This is shown in Table 2.7.

The `for` loop requires nine instructions to implement which will make the `for` loop slow to execute. This is a situation where adding specialist bytecode might improve performance. For example, we could introduce a special `doFor` bytecode that would handle some of the incrementing and testing. For now we'll look at the code above and at a later time consider a more efficient looping structure.

The first thing the code does, Table 2.7 is initialize the loop variable `i`. It never executes this again. The loop proper stats at line 2 where it loads the value of the loop variable and the upper limit value, 5. It compares the two using `isGT`, standing for isGreaterThan, and pushes the boolean result onto the stack. In line 5 we use a `jmpIfTrue` to pop off the boolean result and if True we jump forward three instructions which essentially jumps us completely out of the loop. However, if the result if False we move to the next instruction

Code	Bytecode
`for i = 1 to 5 do`	`0 pushi 1`
`end`	`1 store i`
	`2 load i`
	`3 pushi 5`
	`4 isGt`
	`5 jmpIfTrue 3`
	`6 inc i, 1`
	`7 jmp -5`

Table 2.7 Code generated for a `for` loop

.

Code	Bytecode
`if True then`	`0 push True`
`  a = 14`	`1 jmpIfFalse 3`
`end`	`2 pushi 14`
	`3 store a`

Table 2.8 Code generated for a simple `if` statement without the `else` clause

.

which increments the `i` variable by one. Note that the `inc` instruction modifies the variable in the symbol table and neither pops or pushes anything to and from the stack. Finally, on line 7 we jump back five instructions to line 2 and start the process again. One obvious place we could optimize the code is in line 4 and 5. We could easily replace these two instructions with a single `jmpIfGt`. We'll consider such optimizations in a later chapter.

If Statement

The `if` statement is reasonably straightforward. Without the `else` clause, Table 2.8, the code initially pushes the result of the `if` evaluation onto the stack. If the value is false we jump three instructions on which gets us out the `if` statement. If not, we continue, which results in the body of the statement being executed.

When we add the `else` clause, Table 2.9, the initial jump after the `if` test, is to the start of the `else` code. If the `if` statement if `True`, then the code continues in the body of the `if` statement. In line 4, you'll see an unconditional `jmp` instructions. This is is jump over the code for the `else` clause and out of the `if` statement completely.

Code	Bytecode
`if True then`	`0 push True`
`  a = 14`	`1 jmpIfFalse 4`
`else`	`2 pushi 14`
`  a = 26`	`3 store a`
`end`	`4 jmp 3`
	`5 pushi 26`
	`6 store a`

Table 2.9 Code generated for a simple `if` statement with an `else` clause

.

Switch Statement

By far the most complex construct to consider is the switch statement which involves thirteen core instructions for a switch with two case options, Table 2.10. Two new instructions have been introduced to deal with the switch statement, `dup` and `popdup`. `dup` duplicates the current top entry on the stack. For example, if the top of the stack has an integer value 3, after dup we will have two stack entries with value 3. The second new bytecode, `popdup` just pops the stack entry. This is to remove the duplicated stack entry that was introduced by `dup`. We could have just used `pop` but I used `popdup` to help remind me what was going on in the code. The switch statement itself is implemented as a series of tests to compare the switch value with each case value. One could be more sophisticated by creating a jump table of sorts which would be more efficient. Imagine you have 100 case statements, and the switch value is 100, the current method would have to do 99 comparisons before it reached 100. Clearly not the most efficient way to do things but it will do for now. In the example in Table 2.10 you'll see two case options and the corresponding two `jmpIftrue` instructions

If we add an else clause to the switch construct, Table 2.11 we see in the code below a final section that supports `else` where we set `a = 45`.

In summary, the minimal number of bytecodes we introduced previously together with the addition of one or two others is sufficient to implement any looping syntax you'd care to invent. For example, it wouldn't be hard to put together bytecode to mimic loops like `a = 10; repeat a = a - 1 while a > 5`. There are also many variants on the for loop we could also implement.

Indexing Support

Arrays and lists are always an important part of any programming language. The most common operation is indexing. For example if we had a list such as:

Code	Bytecode
`switch 1`	`0 pushi 1`
`  case 1 :   a = 14`	`1 dup`
`  case 2 :   a = 23`	`2 pushi 1`
`end`	`3 isEq`
	`4 jmpIfTrue 6`
	`5 dup`
	`6 pushi 2`
	`7 isEq`
	`8 jmpIfTrue 5`
	`9 jmp 7`
	`10 pushi 14`
	`11 store a`
	`12 jmp 4`
	`13 pushi 23`
	`14 store a`
	`15 jmp 1`
	`16 popDup`

Table 2.10 Code generated for a `switch` statement without an `else` clause

.

Code	Bytecode
`switch 1`	`0 pushi 1`
`  case 1 :   a = 14`	`1 dup`
`else`	`2 pushi 1`
`  a = 45`	`3 isEq`
`end`	`4 jmpIfTrue 2`
	`5 jmp 4`
	`6 pushi 14`
	`7 store a`
	`8 jmp 3`
	`9 pushi 45`
	`10 store a`
	`11 popDup`

Table 2.11 Code generated for a `switch` statement with an `else` clause

.

```
>> a = [1,2,3,4]
```

We would want to either access or set a particular element using this syntax:

```
>> a = [1,2,3,4]
>> b = a[1]
>> a[1] = 99
```

The same applies to arrays. We only have to introduce two new bytecodes to support this functionality, they are:

`lvecIdx` To access an element from an array or list

`svecIdx` To store a value to an index array or list.

These were introduced in part II of the series and were described there in some detail. As a reminder `lvecIdx` expect two items to be on the stack, the index followed by the object itself. In part II the object could be either a list or a string. In part III we extend this to include arrays. The store bytecode, `svecIdx` expects three items to be on the stack, the index, the object itself (array, list or string), and finally the value to store at the indexth position.

One thing we've not mentioned so far are user functions. Because user functions access variables differently, we need a parallel set of bytecodes for user functions. In this case we have `localLvecIdx` and `localSvecIdx`, the former for accessing and the later for storing. The same applies to the load and store bytecode we talked about in the last section. For user functions we'll need `loadLocal` and `storeLocal`. Semantically these parallel bytecodes do the same thing but under the hood, one set accesses variables by name and the other, the local ones, access variables by index. This is because all variables (other than global) that are part of a user function are stored on the stack and can therefore be indexed directly via the stack.

One last bytecode we need for lists, is `createList`, this was also discussed in part II and is used to construct a list at runtime. All the elements of the list are expected to be on the stack. The number of items in the list is stored in the `createList` operand field. Let's see some examples of bytecode that uses lists.

Creating a list:

```
a = [1,2,3]
```

This statement would generate the following bytecode:

```
  0  pushi 1
  1  pushi 2
  2  pushi 3
  3  createList 3
  4  store a
```

Things to note, the `createList` bytecode also includes the number of items, in this case 3. The items are expected to be on the stack so that `createList` can just pop them off. One small issue, the items when popped come off in reverse order compared to the original Rhodus statement, where the order was `[1,2,3]`. You'll notice in the bytecode that 3 is popped of first so that the order `createList` gets is `3,2,1`. This isn't a real problem and we just have to make sure that the popped items get put into the right locations in the new list. This is most easily done with a `for/downto` loop.

For a nested list, such as:

```
a = [[1,2],[3,4]]
```

we'd have the following bytecode program:

```
  0  pushi 1
  1  pushi 2
  2  createList 2
  3  pushi 3
  4  pushi 4
  5  createList 2
  6  createList 2
  7  store a
```

The way this is done, means we can have lists nested to any depth. First lines 1, 2 and 3 create the sublist `[1,2]`, note that when `createList` execute it leaves the new list on the stack. Next we see the sublist `[3,4]` being made, again, the `createList` leaves the sublist on the stack. At this point we have two sublists on the stack. When we get to the final `createList` instruction, it pops the two sublists from the stack and puts them into a new list, forming `[[1,2],[3,4]]`, again it leaves this list on the stack but the final instruction, `store a`, pops the list and stores it in `a`.

If you've ever disassembled Python bytecode in relation to lists, you'll see it uses the same approach to handle lists. Out of interest here is the python bytecode for the statement: `a = [[1,2],[3,4]]`:

```
// Python Bytecode
  0 LOAD_CONST     0 (1)
  1 LOAD_CONST     1 (2)
  2 BUILD_LIST     2
  3 LOAD_CONST     2 (3)
  4 LOAD_CONST     3 (4)
  5 BUILD_LIST     2
  6 BUILD_LIST     2
  7 STORE_NAME     0 (a)
```

The only difference is in the names of the bytecodes, `LOAD_CONST` for `pushi`, `BUILD_LIST` for `createList` and `STORE_NAME` for `store`. I should mention that the Rhodus bytecode wasn't modelled on Python's bytecode but similar solutions popup repeatedly in different interpreters.

User Functions

With the bytecodes so far, we can deal with loops, conditionals, indexing and local variables in user functions. User functions have very modest requires for bytecode support. In fact user functions only need two bytecodes:

```
call
ret
```

User functions were discussed in detail in part II but its worth summarising some elements of their implementation. `call` is used to call a user function and expects the user function object to be on the stack followed by any user function arguments it needs. `call` also includes the number of expected arguments in the bytecode operand, so that something like `call 2`, means that this call was called with 2 arguments. The function object itself stores the number of argument it actually expects. Since we have the actual and expected number of arguments we can check if there are enough arguments in the first place to satisfy the function object and secondly it allows us to implement a simple form of variable argument support.

If we used the `array` method to create an array from a list, such as:

```
a = array([[1,2],[3,4]])
```

we'd have the following bytecode program:

```
  0  load array
  1  pushi 1
  2  pushi 2
  3  createList 2
  4  pushi 3
  5  pushi 4
  6  createList 2
  7  createList 2
  8  call
  9  store a
```

Module Support

The final topic to discuss is module support. Modules include user defined modules or the built-in modules. When the console starts up a special module is created, called `_main_`. All interaction at the console is with this module. When a new module is loaded, its name is loaded into the _main_ symbolTable. If the new module itself loads in another module, the name of the second module will be added to the new module's symbol table. Any number of modules can be loaded this way. The virtual machine also has the notion of a current module. Whenever code is executed by the virtual machine it is run within the context of the module it exists in. This gives the code access to the symbol tables of the module it resides in.

ADD FIGURE OF MODULES

User defined modules are in practice just another file containing Rhodus code. The key difference is the ability to import modules so that all the variables and user functions are put into their own user space determined by the name of the module file. Access at the Rhodus language level is achieved using the period (or full stop) notation. For example, if we create a file called, bankDetails.rh that contains the variable, bankBalance, we would import the file using import:

```
import bankDetails
```

After that we can access the bankBalance using the period syntax as follows:

```
bankDetails.bankBalance
```

This is a convenient way to package up user functions and other information into their own name space. For example, there may be another module file called myMoney that also uses a variable called bankBalance. We can safely use both variables bankBalance because we can quality each one with the module its associated with.

Rhodus also has a set of built-in modules which behave in the same way as user modules. For example we can import the math module:

```
import math
```

then access its information, for example, the value of π:

```
a = math.pi
```

The question here is what extra bytecodes do we need to support this kind of feature? It turns out to be quite simple and in fact we only need three additional bytecodes:

```
import
loadAttr
storeAttr
```

The import bytecode is used, unsurprisingly, by the import statement. It's used to make sure that whatever is imported gets it own module and that any code in the module is compiled into bytecode. It also adds the name of the module to the currently accessible symbol table. Thus using import from the console will insert the module into the _main_ symbolTable.

The two other instructions, loadAttr and storeAttr, are used to deal with the period syntax. You may be asking why not reuse store and load? The load instruction takes a single operand, the name of the symbol and attempts to access that symbol in the symbolTable that belongs to current module. When using something like bankDetails.bankBalance, we will have the module object, bankDetails on the stack and loadAttr is expected to pop that value off the stack and use that to reference the symbol bankBalance. Let's look at some examples.

For the statement:

```
a = math.pi
```

we would generate the following bytecode:

```
  0  load math
  1  loadAttr pi
  2  store a
```

The actually access to pi involves two instructions, push the math object onto the stack using load and calling loadAttr. loadAttr expects the stack to hold the module object. It will pop off the module object and use that module to access the symbol pi. Once it accesses the symbol pi it pushes whatever value it find onto the stack. store a just pops off the value on the stack and stores it to symbol a, this time however in the current module.

For a user defined module, let's say called lib, the code is exactly the same. For example, assume the module lib contains a variable a. If we were to access the variable using:

```
x = lib.a
```

the bytecode would be:

```
  0  load lib
  1  loadAttr lib
  2  store a
```

which is exactly the same the import math example. If we wanted to store a new value into lib.a, for example:

```
lib.a = 4.5
```

we'd generate the following bytecode:

```
  0  pushd 4.5
  1  load lib
  2  storeAttr a
```

We push 4.5 onto the stack, then push the module object for `lib`. Finally we use `storeAttr` which pops the module object, and stores whatever it find next in the stack (4.5), to the symbol `a` in module `lib`.

More Complex Expressions

We now have everything we need to express any Rhodus program we might write. Let's looks at some more complex examples. The fist example shows us copying the `math.sin` function into a list, then calling the first function via indexing and the function call syntax:

```
a = [math.sin]
x = a[0](3.14)
```

this gets turned into the following byte code. I've split the code in two, one for each line:

```
  // a = [math.sin]
  0  load math
  1  loadAttr sin
  2  createList 1
  3  store a

  // x = a[0](3.14)
  0  load a
  1  pushi 0
  2  lvecIdx
  3  pushd 3.14
  4  call
  5  store x
```

Knowing what we already know the code shouldn't be difficult to understand. The first part loads the math module and pushes the function object associated with `sin` onto the stack. `createList` then pops of one item from the stack (it doesn't care what kind of object it is), creates the list which is then stores to a variable `a` via `store`. The second half does the indexing and function calling. It first loads the list which is stored in `a` onto the stack. It then pushing what will be the index, `0` onto the stack. The first big event is the `lvecIdx` instruction. This pops off the index and the list object, and pushes whatever it finds at `[0]` onto the stack. Finally, `call` is executed which takes the current value on the stack (which will be the function object) and executes it. Lastly we store whatever the `call` left on the stack into symbol `a`.

What's happening here is that whenever the compiler comes across an indexing operation is emits a `lvecIdx`, and whenever it comes across a `()` it emits a `call`. This means we can creates some crazy expressions such as the following:

```
x = modules[2].func(1,2)[2,2](True)
```

If the object stored at `[2,2]` is not a function then the call `(True)` will issue a runtime error. The same applies if we try to index a non-indexable object. For example, if we do the following:

```
>> a = 5
>> a()
```

Clearly `a` doesn't hold a function object we can call, the second line therefore results in:

```
ERROR: integer is not something that can be called as a function
```

The same happens is we try this: `a[0]` giving the message:

```
ERROR: integer variable is not indexable
```

The full list of bytecodes is given in an appendix but you'll notice that many of them are related to arithmetic or boolean operations and that there are only a few that do specialist things like calling functions etc.

2.4 Compiling Code

Now that we've seen examples of bytecode we would generate for various Rhodus constructs, how do we generate such bytecode sequences? This topic returns us to the parsing stage. Figure 1.2 shows a birds-eye view of the interpreter. Here I want to say a little bit about the syntax analysis and generation of the abstract syntax tree, Figure 2.3. The figure shows that syntax analysis and the abstract syntax tree phase have been separated into two blocks. It's possible to combine these into one block where syntax analysis and AST generation occur simultaneously and originally this is what I had. However I was confronted with a messy problem. An AST is as it says a tree structure, with nodes and leaves and a root. Each node and leaf requires an allocation of memory to store the details pertinent to the node or leaf. ASTs vary in size depending on the size of the source code we parse, hence the AST has to be dynamically built on a need basis. Traditionally this is done by requesting memory from the heap. Once we've built the AST and eventually no long need it the memory must be released. This is straightforward and involves traversing the tree and freeing the nodes and leaves.

A problem arises however if we do syntax analysis and AST construction at the same time. That is, as we recognize syntactically correct pieces of source code we allocate memory to create nodes and leaves. What however, there is a syntax error? We are now a position where we have a partially constructed tree with most likely incomplete nodes. In this situation we have to report the error to the user and dismantle the current state of the AST. The solution I came up with to dismantling the tree was to populate the AST with error nodes when a syntax error ever arose. However, this involved multiple checks

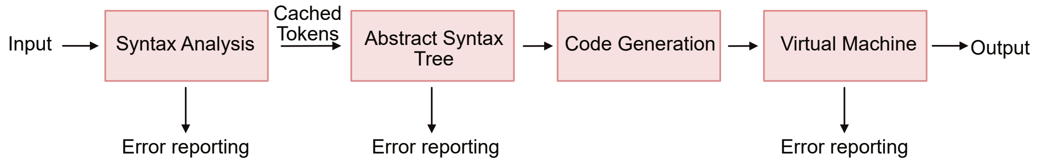

Figure 2.3 Stages in Version 3. The syntax analysis stage will cache the tokens it gets from the lexer which will be reused by the abstract syntax tree stage.

at every stage and the code because unnecessarily messy and likely difficult to maintain. The solution to this was to separate syntax analysis from AST construction. In the syntax stage, the only thing that is done is to check syntax, no memory allocation is involved and if an error is detected we can get out without having to worry about unwinding complex data structures. If the syntax analysis passes with out incident we then ask the AST phase to parse the source code again knowing that there will be no errors forthcoming. We don't have to worry about unwinding any data structures in the event of an error. Obviously there could be errors such as out of memory errors but if these happen, we're toast anyway, and at that point we would exist the interpreter completely.

Another advantage of separating syntax analysis from AST construction is that we can start to think about doing more sophisticated error handling at the syntax stage. At the moment, if we detect any error we just return immediately to the user. I'm not sure what this way of handling errors is called but we could call it the 'I give up' approach. An alternative, which sounds equally bad is called 'panic mode'. In panic model if the parser comes across something it doesn't understand it will try to lookahead to find something it does understand, and then continue from there. This allows to parser to report multiple errors if they exist in the source code, rather than a single error message as we do now. One disadvantage is that we run the risk of overwhelming the human operator with multiple, possibly conflicting, error messages all at once.

For now we'll use the 'I give up' strategy but we'll revisit panic mode at a later time since its a common way to handle syntax errors.

I want to return now to the split between syntax and AST construction. You may have wondered that since we're reading the source code twice, once for syntax and another for the AST, isn't this inefficient? I suppose it is and it concerned me enough to at least reduce its impact. You'll notice in Figure 2.3 that the arrow between the syntax block and AST block has the words 'Cached Tokens'. What this is meant to indicate is that when we do the syntax analysis we will cache the tokens we've read. To do this we don't have to change the lexical scanner it self, `uScanner.pas`. Instead, in the syntax analysis phase all requests to the scanner method `nextToken` are routed through a single `nextToken` method in the syntax analysis code, `uSyntaxAnalysis.pas`. This method looks like:

```
procedure TSyntaxParser.nextToken;
```

```
begin
  sc.nextToken;
  tokenVector.append (sc.tokenElement);
end;
```

This method calls the scanner `nextToken()` method but also appends the new token to a token cache held in a variable called `tokenVector`. Other than this method, the syntax analysis has no other interaction with the lexical scanner. Instead it uses the `tokenVector`. At the end of the syntax analysis, `tokenVector` will contain a complete record of token stream. The AST construction code in kept in `uConstruutAST.pas` and the constructor for the AST construction object takes as an argument the tokenVector. When parsing the source code to build the AST, it will use the cached tokens in tokenVector rather than parse the source code from scratch again. Although I've not compared timings, this should improve performance.

One issue that some of you might be wondering is how much space will the token cache take up? Each token requires at minimum 32 byte. Any extra space will be required to store literal strings. Let's assume for argument sake that on average each token will occupy 40 bytes. The biggest file in the test suite (arith1), can be translated into 1390 token, that requires 56K of storage. By today's standards that's a very modest file size. Even if we had a file that translated into 10,000 tokens that's 400K. Given that this memory requirement is temporary, the memory burden doesn't seem significant.

In the remaining chapters we will cover in more detail, the various stages of the interpreter starting with syntax analysis.

3

Syntax Analysis

3.1 Introduction

We discussed syntax analysis quite a bit in part II especially with respect to recursive descent parsers. Let's brief review them here. A recursive descent is a top-down parser where a method is created for each grammar rule. An example of simple grammar rule would be:

```
expr = number '+' number
```

Given such a grammar rule we need to determine whether a given sentence is consistent with that grammar. In the case of our simple grammar, the following would be legal sentences:

```
expr = 1 + 2
expr = 89 + 43
```

Sentences that don't match the grammar would include:

```
expr = 1 +
expr = + 3
expr = 4 5
```

With a recursive descent parser we'd take each grammar rule and convert it into a method. For our simple grammar we'd write a method like:

```
procedure expr;
begin
  parseNumber;
  expect ('+');
  parseNumber
end
```

A real grammar would have many such rules, with some rules being recursive, for example:

```
stmt =   name '=' expr
expr =   expr '+' expr
       | number
```

where the ‘|’ character translates to ‘or’. The expr rule would be translated to the following method

```
procedure expr;
begin
  if token = number then
     begin
     nextToken;
     exit
     end
  else
     begin
     expr;
     expect ('+');
     expr
     end;
end
```

Notice that the method calls itself because that’s what the grammar requires. However, whenever there is recursion there always has to be a way get out of the recursion, in this case by detecting a number.

One of the key features of a recursive descent parer is that it must be possible to decide which grammar rule to use based solely on the current token in hand. Left-recursive grammars do not satisfy this, for example here is a grammar rule that is left-recursive

```
expr =  expr '+' expr
```

we would end up in an infinite loop since this would be translated into:

```
procedure expr;
begin
  expr;
  expect ('+');
  expr;
```

```
end
```

We covered this aspect and more in part I of the series. The point I wish to make here is that our grammar must be free of left-recursive rules. In the first chapter we described one problematic area in our grammar for Rhodus and how we removed the left-recursion. One of the requirements was that we should be able to parse something like:

```
a = m.func ("abc")[5](math.pi)
```

This reads: call a function in a module, that returns a list which we index to get another function which we also call with an argument that accesses a variable in another module. One could imagine all sorts of convolutions. Whatever grammar we design, it must be able to accept such sentences as valid. Similar constructs can be found in Python, so why not look at the Python grammar specification for clues. You'll find the latest Python grammar specification at https://docs.python.org/3/reference/grammar.html. The Python grammar is more complicated that the one we have and it takes a little study to figure out the portion that parsers expressions. If you look for the primary grammar rule you'll see part of what we need. With that in mind I came up with the following grammar that satisfies our needs and is modelled on the Python grammar. It is:

```
E       = E '.' identifier
        | E '()'
        | E '[]'
        | factor
factor  = identifier
        | Number
```

Sure are some simple examples of the expressions that are legal in this grammar. `a()`, would satisfy `E '()'` where `E` would then be replaced by the `identifier` in `factor`. For a more complicated expression such as: `a.b[]()` we would use the second subrule `E '()'`, then substitute the `E` for `E '[]'`, followed by another substitution of `E` with `E '.'identifier`. Finally the last `E` would be replaced by `factor`. Since we resolved to all terminals `a.b[]()` is a legal sentence.

Unfortunately this grammar is left-recursive, where the `E` symbol is the first symbol on the right-side of the rule in three locations. This will cause a number of problems. Assuming that we're not detecting a factor, its impossible to say which of the three subrules we should pick and even if we did pick one, we end up in an infinite loop, repeatedly picking `E`. As it stands this is impossible to parse with a single lookahead recursive descent parser. This problem was resolved by using an online tool called left_rec that remove the left-recursion, giving on instead the following grammar, which we can parse with a single lookahead recursive descent parser.

For example, the sentence `a()`, would satisfy `E '()'` where `E` would then be replaced by the identifier in factor. For a more complicated expression such as: a.b[]() we would use the second subrule E '()', then substitute the E for E '[]', followed by another substitution of E with E '.'identifier. Finally the last E would be replaced by factor. Since we resolved to all terminals a.b[]() is a legal sentence.

If you exercise this grammar by doing more examples you'll realize it's quite straightforward even though perhaps initially, it looks a little scary. The big problem with it is that it's not friendly for our recursive descent parser. The grammar shown above is what's called left-recursive (see Part 1). We can see this because the E symbol, in three cases, is the first symbol in the production. to illustrate the problem consider the following:

If you exercise this grammar by doing more examples you'll realize it's quite straight forward even though perhaps initially, it looks a little scary. The big problem with it is that it's not friendly for our recursive decent parser. The grammar shown above is what's called left-recursive (see Part 1). We can see this because the E symbol, in three cases, is the first symbol in the production. As a reminder here is a simple grammar that is left-recursive:

```
E = E 'a'
```

is left-recursive because the first symbol on the right of the equals sign is E and E is not a terminal. If you think about it, applying this production rule, will result in a recursive loop, continually recognizing E. When a grammar is left-recursive we almost always have to lookahead more than one token in order to decide which production to use. However, we only do a single lookahead in Rhodus so that a left-recursive grammar is going to be trouble for us. Another example that shows the problem with left-recursions is that with multiple alternative options such as:

```
E   = E 'a'
    | E 'b'
```

it's also impossible to decide which one to pick unless we lookahead further into the token stream to identify the ’a’ and ’b’ but we don't want to do that. With out further lookahead, the parser will go into an infinite loop. The solution is to remove the left-recursion. What this essentially does is move the terminals into the front of the production and the offending left-recursive terms towards the end, resulting is a right-recursive rule which can be parsed using a single lookahead recursive decent parser. The method to do the transformation was described in Part 1 but here I will cheat by using a tool to do it for me. The site https://cyberzhg.github.io/toolbox/left_rec has an on-line tool to remove left-recursion. The tool has two panels, in the upper panel you paste your left-recursive grammar, hit the convert button and your new well-behaved grammar will appear in the bottom panel.

I took the following generic left-recursive grammar and entered it into the tool:

```
E = E a | E b | E c | d
```

After conversion it looked like:

```
E = E 'a'
```

This grammar is left-recursive because the first symbol on the right of the equals sign is E and E is not a terminal. If you think about it, applying this production rule, will result in a recursive loop, continually recognizing E. When a grammar is left-recursive we almost always have to lookahead more than one token in order to decide which production to use. However, we only do a single lookahead in Rhodus so that a left-recursive grammar is going to be trouble for us. Another example that shows the problem with left-recursions is what we have multiple alternative options such as:

```
E  = E 'a'
   | E 'b'
```

In this case it is impossible to decide which subrule to pick unless we lookahead further into the token stream to identify the 'a' and 'b' but we don't want to do that. With out further lookahead, the parser will go into an infinite loop.

Returning to the grammar we'd like to implement, assuming that we're not detecting a factor, its impossible to say which of the three subrules we should pick and even if we did pick one, we end up in an infinite loop, repeatedly picking E. As it stands the grammar we'd like to use cannot be implemented in a single lookahead recursive decent parser.

The solution is to remove the left-recursion. What this essentially does is move the terminals into the front of the production and the offending left-recursive terms towards the end, resulting is a right-recursive rule which can be parsed using a single lookahead recursive decent parser. The method to do the transformation was described in Part 1 but here I will cheat by using a tool to do it for me. The site https://cyberzhg.github.io/toolbox/left_rec has an on-line tool to remove left-recursion. The tool has two panels, in the upper panel you paste your left-recursive grammar, hit the convert button and your new well-behaved grammar will appear in the bottom panel.

I took the following generic left-recursive grammar and entered it into the tool:

```
E = E a | E b | E c | d
```

After conversion it looked like:

```
E   = d E'
E'  = a E'
    | b E'
    | c E'
    | empty
```

Notice how all the terminals, a, b, c and d have been moved to the front. This grammar can be recognized with a single token look-ahead, that is it's LL(1) friendly, which is what we're after. Notice also the empty option, that is E' can be a, b, c or none of them.

d has been moved to it's own production and represents factor in the grammar we had previously. If we translate the symbols into more meaningful words we get:

```
factor                   = Identifier
primary                  =  factor primaryPlus
primaryPlus              = '.' identifier primaryPlus
                         | '(' exp ')' primaryPlus
                         | '[' exp ']' primaryPlus
                         | empty
```

where d is factor, E' is primaryPlus and priamry is E. In the final grammar the individual options in the primaryPlus production are separated out for convenience into primaryPeriod, primaryIndex and primaryFunction respectively. factor we will include all the literals as well as expressions with parentheses and the not operator. We will use this grammar in Rhodus version 3.