

Writing an Interpreter in Object Pascal:

Part 3: Library Support

Herbert M. Sauro
Seattle, WA



Ambrosius Publishing

Copyright ©2018-2021 Herbert M. Sauro. All rights reserved.

First Edition, version 1.0

Published by Ambrosius Publishing and Future Skill Software

books.analogmachine.org

Typeset using L^AT_EX 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	v
1 Introduction to Version III	3
1.1 Introduction	3
1.2 Components of an Interpreter	4
1.3 Language Changes	5
1.3.1 Modules	5
1.3.2 User Functions	9
1.3.3 Lists and Strings as Objects	10
1.3.4 Homogeneous Arrays	11
1.3.5 Built-in Libraries for lists, strings, arrays and matrices	13
1.4 Error Handling	14
1.5 Grammar Specification for Version 3	14
1.6 Useful Reading	16
2 Internal Changes	19
2.1 Introduction	19
2.2 Updates to the Byte Codes	21
2.3 Examples of Byte Code Programs	22
2.4 Compiling Code	35
3 Syntax Analysis	39
3.1 Introduction	39
3.2 Syntax Parsing	43
4 Abstract Syntax Tree	49
4.1 Introduction	49

5	Array Support	55
5.1	Introduction	55
5.2	Implementing Arrays	57
5.3	Matrices	62
6	Embedding Rhodus	65
6.1	Introduction	65
6.2	Simple Embedding	65
6.3	Rhodus Sharable Library	70
7	Optimizations	75
7.1	Introduction	75
8	Surprise Chapter	77
8.1	Introduction	77
9	Reference to Libraries	79
9.1	Introduction	79
9.2	Lists Module and List Object Methods	79
9.3	Strings Module and String Object Methods	81
9.4	Arrays Module and Array Object Methods	83
9.5	Math Library: math	83
9.6	Random Module: random	85
	Appendix	87
A	List of Virtual Machine Opcodes	87
	History	97
	Index	99

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

All source code is licensed under the open source license Apache 2.0.

<http://www.apache.org/licenses/LICENSE-2.0>

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 version 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.

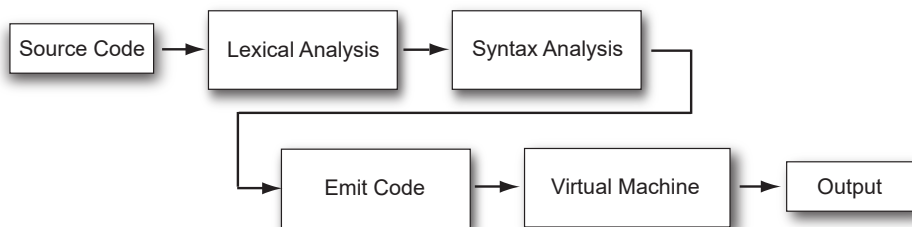


Figure 1.1 Simplified flow from source code to output in version 2 of the Rhodus interpreter.

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.

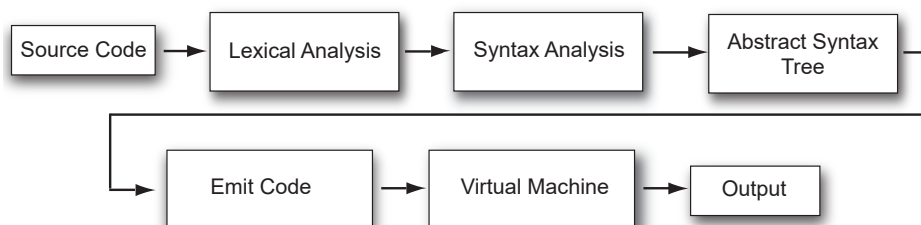


Figure 1.2 Simplified flow from source code to output in version 3 of the Rhodus interpreter incorporating the abstract syntax tree stage.

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 can also get help using the help command, for example:

```
help(math.min)  
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.toUpperCase().toLowerCase().toUpperCase().toLowerCase().toUpperCase()
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 curly 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 curly 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 `mat`. The contents of each of these can be obtained using `dir()`, for example:

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

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

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

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

```
>> import mat
>> // Generate a 3 by 3 matrix with random entries, 0 to 1
>> m = mat.rand(3,3)
>> minv = mat.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 DOWNT0 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".toUpperCase().toLowerCase().toUpperCase().toLowerCase()
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. 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.

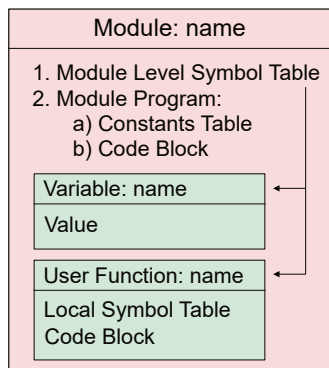


Figure 2.1 Internal structure of a module.

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 but without a constants table (They use the module level constants table). 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 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 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 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 reasonably fast to locate a given symbol.

The user function symbol tables are different. Symbols in a user function are stored using 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.

The class below shows the module structure in code:

```
// Base module class.
TModule = class (TObject)
  name : string;
  moduleProgram : TProgram; // Contains code block and constants table
  symbolTable : TSymbolTable;
  helpStr : string;

  function  getSize : integer;
  function  find (name : string) : TSymbol;
  procedure clearCode;
```

```
    constructor Create (name : string);  
    destructor Destroy; override;  
end;
```

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:

```
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 builtin 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 byte-code 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 purpose 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 b`. 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. The other opcode we've seen before is `pushi` which pushes a literal integer onto the stack.

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 move data between the stack and the symbol table. For example, arithmetic operations 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

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

operations, in particular OR, AND and NOT. Given that we need to deal with numbers as well 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

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

to the `push True` instruction, and we start again. Luckily the `until` statement expression is `True`, so that it will not jump back and get us 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

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 non-empty while loop

False from while False do onto the stack. The next instruction, jmpIfFalse, is used to check for the result of the 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.

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 is 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 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 is `True`, then the code continues in the body of the `if` statement. In line 4, you'll see an unconditional `jmp` instructions. This is a 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. This is not a very efficient approach. For example, 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. From a practical point of view this isn't the way to implement a switch statement but its advantage is that it's not difficult to implement. A better approach is to use a jump table. We'll consider this in a later chapter.

Considering 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 a final section to the code 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.

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

Indexing Support

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

```
>> 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 is what happens to 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.

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
```

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.

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, Figure ???. 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.

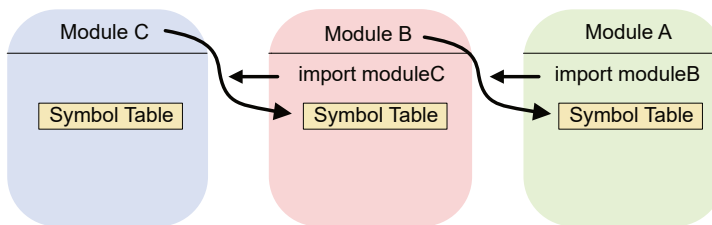


Figure 2.2 Modules importing other modules. Module references go into the symbol table.

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 `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 qualify 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 its 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 a reference to 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 the 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 actual 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 finds 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 code 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 off one item from the stack (it doesn't care what kind of object it is), creates the list which is then stored 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 pushes, 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 it emits a `lvecIdx`, and whenever it comes across a `()` it emits a `call`. This means we can create 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 variable `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 if we try this: `a[0]` giving the message:

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

In conclusion there are not that many specialist opcodes. The majority are related to arithmetic or boolean operations. The full list of bytecodes is given in an appendix.

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. 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 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 longer need it, the memory must be released. This is straightforward and involves traversing the tree and freeing the nodes and leaves one at a time.

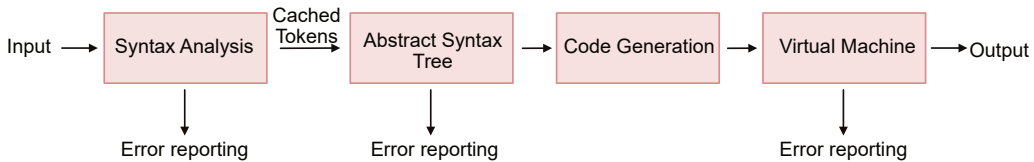


Figure 2.3 Stages in Version 3. The syntax analysis stage will cache the tokens it gets from the lexer which will be reused by the abstract syntax tree stage.

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, if there is a syntax error? We are now in 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 when an error occurred I would populate the AST at that point with error nodes. This meant I no longer had an incomplete tree and instead had error nodes which I could deal with later. However, this involved multiple checks at every stage and the code became 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 without 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 should exit 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 mode if the parser comes across something it doesn't understand it will try to look ahead to find something it does understand, and then continue from there. This allows the 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 the token stream.

The AST construction code is kept in `uConstructAST.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 the topic 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  
    nextToken  
  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. Here is a grammar rule that is left-recursive:

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

If we used this we’d end up in an infinite loop since this would be translated into the following code:

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

We covered this problem 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 than the one we have and it takes a little study to figure out the portion that parses 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
```

Here 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.

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 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'
```

This 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, this 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
```

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 `primary` 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.

3.2 Syntax Parsing

We saw a lot of syntax parsing code in the previous parts of this book series. Here we will remind ourselves what such code would look like and also focus on the new grammar rules for primary non-terminals.

We'll first look at something we already seen and then move on to new language elements. The simplest non-trivial language element to look at is the `if` statement. The two grammar rules for this are shown below.

```

ifStatement     = IF expression THEN statementList ifEnd
ifEnd           = END
                | ELSE statementList END

```

The reason we have two rules is to take care of the optional `else` statement. This grammar is very easily expressed in Object Pascal as follows:

```

procedure TSyntaxParser.ifStatement;
begin
    expect(tIf);
    expression;
    expect(tThen);
    statementList;

    if tokenVector.token = tElse then
    begin
        nextToken;
        statementList;
    end;
    expect(tEnd);
end;

```

This is the actual the code in the interpreter, there is nothing else to add, it's that simple. Our error handling is a bit primitive at the moment and if there is any error, e.g expect (tThen) doesn't find the then keyword, then we drop out completely. Ideally we would issue the error message, then try to synchronize with a token we do understand and continue from there. Personally I'm not a fan of reams of errors coming out especially for an interpreter where one tends to work interactively. For a compiler, it makes more sense where one usually works in batch mode. However, interpreters tend to be used by people who aren't computer scientists but who are using it as a tool. A torrent of errors messages is more likely to confuse compared to having to deal with one problem at a time.

Switch Statement

Let's next look at the switch statement which we never had in part II. Switch statements can come in a variety of forms, but in general they can be thought of as glorified multiple if/then/else statements where depending on the value of the switch expression, a particular block within the switch statement is executed. For example:

```
switch index
  case 1 : a = 1
  case 2 : a = 2
  case 3 : b = math.sqr (9)
           a = 10 + b
else
  a = 99
end
```

If index has the value 2, then a = 2 will be executed. If no value matches then the else clause is executed. If there is no match and no else clause then nothing happens. Many languages permit the case values to be a range of values and some even allow case values to be strings. Some languages, such as C require a break statement at the end of each case otherwise the execution will 'fall through' to the next case statement. Object Pascal for example doesn't allow 'fallthrough' as do many other languages such as modern FORTRAN, Modula and many BASICs. We will do the same.

The grammar specification for the switch statement is shown below. Notice I'm using {} to indicate repetition. We're also indicating that the case specifier must be an integer. The switch expression can syntactically be any expression but at runtime it must resolve to an integer otherwise a runtime error is issued.

```
switchStatement  = SWITCH simpleExpression switchList END
switchList       = { CASE INTEGER ':' statementList } ELSE statementList
```

The Object Pascal that implements this grammar is shown below. The only thing to note is the while loop that will let us pick up multiple case options. We also check that the case option is an integer, and issue an error if it's not.

```

procedure TSyntaxParser.switchStatement;
begin
  expect(tSwitch);
  simpleExpression;

  while tokenVector.token = tCase do
    begin
      expect(tCase);

      if tokenVector.token <> tInteger then
        raise ESyntaxException.Create ('Expecting integer in case value',
          tokenVector.tokenRecord.lineNumber,
          tokenVector.tokenRecord.columnNumber);

      nextToken;
      expect(tColon);
      statementList;
      end;

    if tokenVector.token = tElse then
      begin
        nextToken;
        statementList;;
        end;
      expect(tEnd);
    end;
end;

```

One thing you should notice is how straight-forward the code is. The reason for this is that we're not building the AST at this point, just checking syntax. If we added the AST building code it would result in a lot more clutter; the main problem being unwinding the AST if we come across an error. I think keeping syntax analysis away from AST construction greatly improves maintainability. We'll return to this point in the next chapter.

Primary Rules

Finally let's look at the code that will parse the primary production rule. As a reminder here are the grammar rules again:

```

primary          = factor primaryPlus

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

primaryPlus      = primaryPeriod
                  | primaryFunction
                  | primaryIndex
                  | empty

```

The factor is given by:

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

This grammar can be used to parse complex expressions like `m.abc(1,2,3)[a+b]()`. This is a straightforward grammar to implement. For example, the rule:

`primary = factor primaryPlus,`
just translates to:

```
procedure TSyntaxParser.primary;
begin
  factor;
  primaryPlus;
end;
```

`primaryPlus` translates to the following.

```
procedure TSyntaxParser.primaryPlus;
begin
  case tokenVector.token of
    tPeriod :
      begin
        nextToken;
        expect (tIdentifier);
        primaryPlus;
      end;
    tLeftParenthesis: // '(' expression list ')'
      begin
        nextToken;
        parseFunctionCall;
        primaryPlus
      end;
    tLeftBracket: // '[' expression list ']'
      begin
        parseIndexedVariable;
        primaryPlus;
      end;
  end;
end;
```

We use a case statement check to figure out which option to pick. Notice that `primaryPlus` also has an empty option. The empty option is anything but the first three. In code, we just ignore it; and that's about it.

The take home message for this chapter, is that once you have a recursive descent friendly grammar in place its very straightforward to code. The formality of grammars makes it possible to write automated systems such as yacc or ANTLR that will generate all the syntax code for you. Unless you have a very weird syntax, writing a recursive parser is not that difficult. The one chief advantage of building your own recursive descent parser over automated systems like yacc is you have more control over the error handling.



4

Abstract Syntax Tree

4.1 Introduction

The biggest internal change in Part III is by far the introduction of an additional phase between syntax analysis and code generation. See Figure 1.2 from Chapter 1. In the new phase we construct the abstract syntax tree or AST for short.

The AST is so-called because it represents a very terse view of the source code. For example, bracketing is absent because the grouping of elements is implicit in the structure of the tree. But what is the purpose of the AST stage? There are several, the first is that we get a clean separation of syntax analysis from code generation and that the AST structure can be analysed for semantic correctness and even some basic code optimization can be carried out. For languages that are statically typed (such as Object Pascal or C), where variables have strict immutable types, semantic analysis can be very useful to eliminate errors.

In part II of this series we talked about tree structures and mentioned the different ways in which such a tree can be traversed. I am therefore going to assume that the reader knows something about tree structures but not necessarily a lot.

The source file that constructs the AST is called `uConstructAST.pas`. There is also a separate file called `uAST.pas` that contains all the support code to aid with construction and `uASTNodeTypes` which lists the types of node we can have.

The easiest example to consider is the binary operator such as `2 + 5`. A very simple parser for this statement would be:

```
procedure TConstructAST.parseBinaryOperator;
```

```

begin
  parseNumber;
  nextToken;
  if token = tPlus then
    begin
      nextToken;
      parseNumber;
    end
  else
    error
  end;
end;

```

We need to modify this code so that we can turn $2 + 3$ into a little tree. The first change is that the procedure must return the root to the mini-tree we construct. For that reason let's create a class called `TASTNode` that the function will return. We'll have more to say about this class later on. `parseNumber` should return a node that represents a number and we need to record the plus symbol we find. The method needs to return a tree that looks like the one shown in Figure 4.1. I've marked the plus node as the root of this tree.

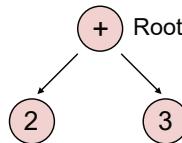


Figure 4.1 A little tree formed from $2 + 3$.

Let's update the function to collect any nodes we create:

```

function TConstructAST.parseBinaryOperator : TASTNode;
var : operand1, operand2 : TASTNode;
    operand : TASTNodeType;
begin
  operand1 := parseNumber;
  nextToken;
  if token = tPlus then
    begin
      operand := ntPlus;
      nextToken;
      operand2 := parseNumber;
    end
  else
    error
  end;
end;

```

We need a method that will knit the three components, `operand1`, `operand2` and `operator` into a little tree. Let's pretend we have a method that does that, called `TASTBinOp`. We might use it like this:


```
function TConstructAST.parseBinaryOperator : TASTNode;  
var : operand1, operand2 : TASTNode;  
    operator : TASTNodeType;  
begin  
    operand1 := parseNumber;  
    nextToken;  
    if token = tPlus then  
        begin  
            operator := ntPlus;  
            nextToken;  
            operand2 := parseNumber;  
        end  
    else  
        error  
    result := TASTBinOp.Create (operand1, operand2, operator);  
end;
```

You can probably guess from the syntax of `TASTBinOp` that the first two arguments are the leaves of tree and the third argument is the node type we use to make the root node.

One problem we need to deal with is what to do with errors?. For example, what if there is no second operand? The problem is that we've already constructed the first leaf, `operand1`. An easy way out would be to just raise an exception and get out quick. The problem with that strategy is we'll end up with memory leaks since we're leaving behind tree components we no longer need. We have to somehow gracefully exit if there is an error. I came up with a number of possible solutions to this:

- Create two parsers, one parser only checks for syntax errors and generates no output other than success or failure and in the case of a failure the line number and nature of the error. If the first parse is successful we run the second parser which does create the AST. Since we're guaranteed there won't any errors, there won't be any chance to create a partial tree that needs cleaning up.
- Build the AST using a data structure, perhaps a table, that doesn't require any memory to be allocated as the parser runs. If there is syntax error we can get out and free the table.
- Monitor for errors as we build the tree, take care to free any partially constructed tree components in the event of an error. When an error occurs, clean up any partial tree components and return a special error node that includes information on the nature of the error including any line number information. When we come to generate the bytecode from the AST, we would need to watch out for any error nodes and respond with the error message kept in the error node.

All three solutions are viable. The first one results in a two pass system that could slow things down. The second seems a bit complicated as we'd have to use indices to represent

links between nodes and tree traversals won't be as simple to implement. In the third solution, we clean up any dangling tree components if an error occurs and insert a special error node into the tree to indicate a problem. How might this translate into parsing `2 + 3` example? I am going to assume that `parseNumber` and `expect` will return an AST error node in the event of an error. We can check for this error node as we do the parsing. The listing below shows what I mean.

```
function TConstructAST.parseBinaryOperator : TASTNode;
var : operand1, operand2, operator : TASTNode;
begin
  operand1 := parseNumber;
  if operand1 = error then
    exit (operand1)

  operator := expect(plus)
  if operator = error then
    begin
      operand1.free;
      exit (operator)
    end;

  operand2 := expect(number)
  if operand2 = error then
    begin
      operand1.free;
      operator.free
      exit (operand2)
    end

  // No errors, so lets build the minitree and return it.
  result := TASTBinOp.Create (operand1, operand2, operator);
end;
```

If there is an error, `parseBinaryOperator` returns an AST node that represents an error. Let me show you what the `TASTError` node might look like:

```
TASTErrorNode = class (TASTNode)
  lineNumber, columnNumber : integer;
  errorMsg : string;
  constructor Create (errMsg : string; lineNumber, columnNumber : integer);
end;
```

Notice that the error node will store the current line number and column number in the text that is being parsed. This is important, because once the tree is built there won't be another opportunity to collect this information. The error node also has an `errorMsg` field that we use to store a suitable message for the user.

Let's look at another example, this time parsing the repeat/until construct. The grammar for this is simple:

```
repeatStatement = REPEAT statementList UNTIL expression
```

The AST that we'll get from this is equally simple, its just like the $2 + 3$ minitree, except the root is labelled `repeat` and the leaves are `statementList` and `condition`.

(repeat) -> (statementList) and (condition)

In the following code I've left out the break statements to make it easier to understand. Also there is no error checking at the moment. Hopefully the code that is left is self-explanatory. We first expect the `repeat` keyword, then a list of statements, followed by the `until` keyword, and finally the until condition. With that we construct the minitree, using `TASTRepeat` and splice together the `condition` and `statementList` and return the repeat node to the caller. One thing you may have realized is that the left-hand side of each grammar statement will have its own AST node. This is so that during compilation we can recognise particular nodes.

```
function TConstructAST.parseRepeatStatement: TASTNode;
begin
  expect(tRepeat);
  listOfStatements := statementList;
  expect(tUntil);
  condition := expression;

  result := TASTRepeat.Create(listOfStatements, condition);
end;
```

You might start to get the feeling that we will have quite a few different nodes representing different aspects of the Rhodus language, and you'll be right. `TASTNode` will be the parent node for a wide range of descendent nodes.

Let's now add the error detection. For the first line, `expect (tRepeat)`, there won't be an error because in order to have got here we must already have seen the keyword `repeat`. However, errors can occur when we parse the `statementList`, `condition` and the `tUntil`. In particular if there is an error at the condition state we must make sure to free up the partial tree we get from `statementList`. You'll noticed I also check for an error from `expect (tUntil)`. This is in case we read the `endOfStream` token. The method `freeAST` is part of the `AST` class and can be used to free any complete or partially constructed tree.

```
function TConstructAST.parseRepeatStatement: TASTNode;
var node : TASTNode;
begin
  expect(tRepeat);
  listOfStatements := statementList;
  if listOfStatements = error then
    exit (listofStatements);

  node := expect(tUntil);
  if node == error then
```

```

begin
  listOfStatements.freeAST;
  exit (node);
end;

condition := expression;
if condition = error then
  begin
    listOfStatements.freeAST;
    exit (condition)
  end;

  result := TASTRepeat.Create(listOfStatements, condition);
end;

```

You can imagine having to do this at every stage in the syntax analysis. I did in fact implement it this way at one point but realised that maintaining the code could be problematic. It was too easy to forget to free something and cause a memory leak and it took some time to fully debug the code. In the end I decided this was not the way to do it and ultimately I discarded the work and reverted to the first solution which involved doing two passes. One pass to check syntax and a second to construct the AST.

```

// Base AST Node
TASTNode = class (TObject)
  nodeType : TASTNodeType;

  procedure freeAST;
  constructor Create(nodeType: TASTNodeType);
  destructor Destroy; override;
end;

```



5

Array Support

5.1 Introduction

In most programming languages, the word array usually refers to a data structure that stores identically sized elements in a sequential and continuous block of memory. Lists on the other hand are used to store non-identically sized elements that might be located anywhere in memory. Arrays are therefore referred to as homogenous collections whereas lists are heterogeneous collections. The advantage of arrays is that they can be used to process information very efficiently. Arrays are often used to store numerical data which require efficient numerical processing. The simplest array is a one dimensional collection of numbers, such as:

```
[1.3, 5.6, 7.8, 4.5, 2.3, 1.2]
```

This array has six elements. Since the data in an array is guaranteed to be laid out sequentially in memory, accessing individual elements can be efficiently done by using indexing to the appropriate element. Object Pascal has a variety of ways for declaring arrays. For example an array with a fixed number of elements (a static array) can be declared using:

```
TMyArray = array[1..100] of double;
```

It is also possible to declare so-called dynamic arrays which are allocated a size at runtime, and makes them more flexible:

```
var
```

```
myArray : array of double;  
setLength (myArray, 100);
```

Arrays need not be only one-dimensional but in principle can be any dimension. A common array structure used in mathematics is the two-dimensional matrix:

$$\begin{bmatrix} 3.4 & 6.7 & 8.9 \\ 1.2 & 4.5 & 3.1 \\ 5.6 & 7.7 & 2.5 \end{bmatrix}$$

Such matrices have very widespread applications in science, engineering, statistics and machine learning. Matrices can easily be represented using 2D arrays.

Rhodus Array Syntax

We've already encountered the basic syntax used with arrays. Rhodus follows to some degree the model used by Python. In particular we repurpose the list syntax to define literal arrays and use a global method, `array` to convert lists into the array data model. For example:

```
>> a = array([[1,2],[3,4]])  
>> println (a)  
[ 1, 4;  
 9, 16]  
>> type (a)  
array
```

The `array` method also can be used to specify the size of a new array, for example:

```
>> a = array(3,4)  
>> b = array(5,5,5)
```

The first array, `a` is a 3 by 4 array while the second array, `b`, is a three-dimensional array of size 5 in each dimension. By default all elements in a new array are initialized to zero. For now arrays will only be able to contain floating point values.

Like lists, arrays can be indexed using the usual indexing syntax, for example:

```
>> a = array(5,5,5)  
>> a[1,1,2]  
0.0  
>> a[1,1,2] = 3.14  
>> a[1,1,2]  
3.14
```

Like lists, indexing starts at zero.

Rhodus has two variants of arrays. The first is an n-dimensional array and the second, derived from arrays, is the 2-dimensional matrix. The two types only differ in what operations can be applied to them. These operations are governed by two built-in libraries, `arrays` and `mat`. Arithmetic operations on arrays are element-wise whereas the operations provided by the matrix library correspond to the classic matrix operations found in linear algebra. For example, multiplying two arrays together is done by multiplying each corresponding element to form a new array of products, for example:

```
>> import arrays
>> a = array([[1,2],[3,4]])
>> println (arrays.mult (a, a))
[ 1, 4;
 9, 16]
```

In contrast, multiplication using the `mat` library yields a different result:

```
>> import mat
>> a = array([[1,2],[3,4]])
>> println (mat.mult (a, a))
[ 7, 10;
 15 22]
```

We will cover more of this topic shortly.

5.2 Implementing Arrays

How do we implement array support? In considering this question, the main point to keep in mind is that access to arrays are meant to be fast and operations on arrays should be as efficient as possible. This requirement will dictate how an array, and in particular a multidimensional arrays is stored. To keep things simple, let's first consider arrays up to two dimensions. Since we don't know how big arrays will be we need to use dynamic arrays at the Object Pascal level. Secondly, our arrays will be objects that can be garbage collected, this means an array must be derived from the same parent as strings and lists. We begin with a simple array object class:

```
TArrayObject = class (TRhodusObject)
end;
```

The parent `TRhodusObject` class includes a number of fields:

```
blockType : TBlockType;
objectType : TSymbolElementType;
methodList : TMethodList;
```

The `blockType` is for the garbage collector to know what to do with the object during garbage collection. The `objectType` just tells us what kind of object it is, string, list, user function or array. Finally, the `methodList` is a new field for version 3 of Rhodus and points to a list of methods that can be applied to the object. This is what lets us do things like:

```
a = "String"
l = a.len()

println ("How long am I".len())
```

The first thing to add to the `TArrayObject` is the field that will hold the array data. For now, we will only support arrays that hold floating point numbers.

```
T1DArray = array of double;
TArrayObject = class (TRhodusObject)
  data : T1DArray;
end;
```

You may be thinking, ok this will store a 1D array, but what about a 2D array? The best way to handle n-dimensional arrays is to map them into a 1-dimensional array. If we didn't do this we'd have to have special cases for all the possible dimensions, e.g `T1DArray`, `T2DArray`, `T3DArray`, etc. I don't think we want to do that.

Figure 5.1 shows how we can map a one-dimensional array to an n-dimensions. All we do is slice the 1D-array into segments that represent the rows, in this case a 2D-array. We can slice at many times as we like to model n-dimensional arrays.

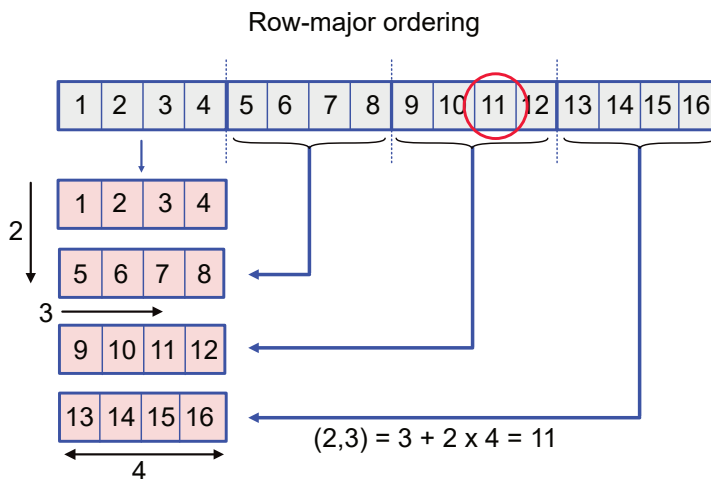


Figure 5.1 Using a 1D-array to represent a 2D-array.

With a bit of simple arithmetic we can convert any 2D-index, such as (i, j) , into a single index along the 1D-array. To do this we must know the intended column width of the 2D-

array. In the example the column width is 4. By convention, in a 2D index such as (i, j) , the i represents the row number and j the column number. Let's say we had a coordinate of $(2, 3)$, that represents the second row and third column. One thing we have to be careful of is what is the index of the first element in the array? As with lists, we will always index from zero. this means that the coordinate $(2, 3)$ actually means the 3rd row and 4th column. Given a coordinate (i, j) , the index along the 1D-array that corresponds to this coordinate will be:

$$\text{index} = j + (i \times \text{width})$$

If we plug $(2, 3)$ into this formula we get: $3 + 2 \cdot 4 = 11$. Hence we would pull the value out of index 11 in the 1D-array. If you look at Figure 5.1 you can confirm this by eye. We can rewrite the above formula in a more general way:

$$\text{index} = i \cdot d_j + j$$

where d_j is the width or dimension (number of elements) in the j th direction. This approach can be extended to any number of dimensions. For example, for a 3D array, with coordinate, i, j, k , the formula for computing the index in a linear array can be extended to give:

$$\text{index} = i \cdot d_j \cdot d_k + j \cdot d_k + k$$

where d_j is the dimension of the j th coordinate, and d_k the dimension of the k th coordinate. By induction the formula for a 4D array with coordinates i, j, k, l , would be:

$$\text{index} = i \cdot d_j \cdot d_k \cdot d_l + j \cdot d_k \cdot d_l + k \cdot d_l + l$$

This can be turned into a function for computing the index of any array with any dimension as follows¹:

```
// Given the dimension of the array in the array dimensions
// and the coordinates in the array index, this routine
// will return the index assuming the array is stored
// in a 1D block of memory.
function getIndex (const dimensions, index: array of Integer): integer;
var i: Integer;
begin
  result := idx[0];
  for i := 1 to high(dimensions) do
    result := result*dimensions[i] + index[i];
  end;
```

For example, if we have a 6 dimensional array of dimensions 5, 5, 5, 5, 5, 5, which is 15,625 elements, the index at coordinate 1, 2, 3, 3, 0, 4 will be position 4829 in the 1D storage array.

We can create two helper methods for getting and setting a value in an n-dimensional array. The methods will take a dynamic array containing the indices to use.

¹Modified from Heffernan's answer in <https://stackoverflow.com/questions/28569850/>

```

function TArrayObject.getValue (idx : array of integer) : double;
var index : integer;
begin
  for var i := 0 to length (dim) - 1 do
    if idx[i] >= dim[i] then
      raise ERuntimeExpection.Create(outOfRangeMsg);

  index := getIndex (dim, idx);
  result := data[index];
end;

```

For example in the `getValue` method we can use the new open array syntax introduced in XE7.

```
value := getValue ([1,3,2])
```

Likewise we have a `setValue` method that also takes an dynamic array argument for specifying the indices.

```

procedure TArrayObject.setValue (idx : array of integer; value : double);
var index : integer;
    i : integer;
begin
  for i := 0 to length (dim) - 1 do
    if idx[i] >= dim[i] then
      raise ERuntimeExpection.Create(outOfRangeMsg);

  index := getIndex (dim, idx);
  data[index] := value;
end;

```

For example:

```
setValue ([6,2,7], value)
```

In both methods we need to check for index out of bounds errors. The bounds will be stored in a object variable called `dim`. We can expand the `TArrayObject` to:

```

TArrayObject = class (TRhodusObject)
private
public
  data : T1DArray;
  dim  : TIndexArray; // Of type array of integer;

  function      getValue (idx : array of integer) : double;
  procedure     setValue (idx : array of integer; value : double);

  function      clone : TArrayObject;

```

```

constructor  Create; overload;
constructor  Create (dim : TIndexArray); overload;
destructor   Destroy; override;

```

We'll add a bunch of extra methods. One in particular is the ability to clone an array. This is implemented in the clone method:

```

function TArrayObject.clone : TArrayObject;
begin
    result := TArrayObject.Create (dim);
    result.data := copy (self.data);
end;

```

There are also a bunch of other methods to make like easier, as well as some arithmetic functions such as add, subtract, and multiply. These are all pair-wise operations. That is addition of two arrays is achieved by summing up pair-wise values in each array. For this work, both arrays must have the same dimensions. For a 2 by 2 array addition is defined as:

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} a_1 + b_1 & a_2 + b_2 \\ a_3 + b_3 & a_4 + b_4 \end{bmatrix}$$

This idea can be extended to n-dimensional arrays. Subtraction is defined in a similar way except we take the difference between each pair. When we come to multiplication we are confronted with two common multiplication approaches. This includes the simple pair-wise multiplication and, arguably, the more common matrix multiplication. For arrays we will provide pair-wise multiplication. This will be specified using the usual multiply operator '*'.

For 2D-arrays, i.e matrices, we need a way to specify matrix multiplication. Python uses '@', so we might as well use that too. We'll have to update the lexical scanner, syntax analysis and AST constructions but that is simple to do. For example, let's compare pair-wise and matrix multiplication:

```

>> a = array([[1,2],[3,4]])
>> println (a*a)
[[ 1.0000,  4.0000]],
 [  9.0000, 16.0000]]
>> println (a@a)
[[ 7.0000, 10.0000]],
 [15.0000, 22.0000]]

```

Multiply arrays

[[0]*4]*3 [[0,0,0,0],[0,0,0,0],[0,0,0,0]]

or

```
>>println (a.sqr())
```

This squares each element in the array `a`. n -dimensional arrays can be created using the `array` method:

There are at least two classes of operations we can apply to arrays, these include manipulating the contents of an array, for example extracting rows and columns, or adding rows and columns, and secondly carrying out arithmetic on arrays. Let's first focus on arithmetic operations.

5.3 Matrices

There isn't much to say about matrices other than they are strictly 2-dimensional arrays. There is nothing intrinsically different between an array and a matrix other than its dimensions. The key difference is that matrix operations are different from the equivalent array operations. This is particularly the case for multiplication which is dot-product based for matrices. This is handled by the `mat` library which also offers other classical matrix operations such as computing the inverse and determinant.

All operations on arrays can also be applied to matrices, but not vice versa, since matrices are strictly 2-dimensional.

As mention in the last section, matrix multiplication comes in two forms, pair-wise and dot product. Pair-wise multiplication can be easily extended to arrays of any dimension. In Rhodus there are two ways to specify pair-wise multiplication, using the normal multiplication operator, `*`:

```
>> a = array([[1,2],[3,4]])
>> b = array([[5,6],[7,8]])
>> x = a*b
>>println (x)
[[ 5.0000, 12.0000]],
[ 21.0000, 32.0000]]
```

Matrix multiplication, as defined in linear algebra, is the generated by taking the dot-product of corresponding rows and columns. Unlike array multiplication, where the two arrays must be the same size, in matrix multiplication this need not be the case. Moreover, matrix multiplication generally applies to 2-dimensional matrices. When applying matrix multiplication, the number of columns of the first matrix must equal the number of rows in the second matrix. The outer rows and columns can be of any size. In Rhodus there are two ways to specify matrix multiplication, either using the `mult` method from the `matrix` library or using the special matrix multiplication operator symbol `@`. For example:

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

```
>> x = a@b
>> x = mat.mult (a, b)
>>println (x)
[[ 19.0000,   22.0000]],
[  43.0000,   50.0000]]
>>
```

Application

[m1, m2]

combine (m1, m2)

[m1; m2]

mat.appendRow ([m1, m2]) mat.appendCol ([m1, m2])

m1.appendRow (m2) m1.appendCol (m2)

m1.hstack (m2) m1.vstack (m2)



6

Embedding Rhodus

6.1 Introduction

So far we'll seen Rhodus being used from a dedicated console terminal. What if someone wanted to use Rhodus from their own application? This short chapter will discuss that topic, that is embedding Rhodus in a host application.

There are various levels to which embedding can be done. The simplest, which we will discuss here, is to allow a host application to pass Rhodus scripts to the interpreter and retrieve the output. This is a fairly shallow form of embedding but the easiest to implement.

A more deeper embedding would allow a hosting application much more access to the internals of Rhodus and vice versa. Of particular interest would be the capacity to add extensions to Rhodus written in the hosting language. For example, if the hosting language were C, it might be useful to write performance critical computations (for example fast fourier transforms) in C and let Rhodus call out to those functions.

6.2 Simple Embedding

The simplest form of embedding is to provide the hosing application a series of API calls to allow the application to pass Rhodus scripts to the interpreter and retrieve the results. The API for this can be quite simple and in fact can be accomplished in just three API methods. Using C syntax, the three methods are given by:

```
intPtr_t rhodus_initialise (rhodusConfig *config);
```

```
int      rhodus_run (intPtr_t handle, char* sourceCode);
int      rhodus_terminate (intPtr_t handle);
```

`intPtr_t` is a C type introduced with C99 that is equivalent to the Delphi type `NativeInt`. Both types are signed integers but their size is equal to the size of a pointer on the current platform, be it 32 or 64-bit.

In Object Pascal format, the API methods are given by:

```
function rhodus_initialise (var config : TRhodusConfig) : NativeInt;
function rhodus_run (handle : NativeInt; sourceCode : AnsiString) : integer;
function rhodus_terminate (handle : NativeInt) : integer;
```

In order that Rhodus can be called from any potential programming language, the interface API must be C compatible. For example, the lowest common denominator for handling strings is the humble `char*`, that is a pointer to a null terminated array of byte characters (not unicode).

The three methods should be fairly easy to understand. We need to first initialise Rhodus using `rhodus_initialise`, then call the Rhodus interpreter with a string containing the source code and finally terminate the session with `rhodus_terminate`.

The initialization call will return a handle to the Rhodus interpreter. Internally this is a pointer to a Rhodus `TRhodus` object. This handle must be used in subsequent calls to run and terminate. Multiple instances of Rhodus can be created in this way, one just needs to keep track of the handles. `rhodus_terminate` will free up any memory used by the Rhodus interpreter.

An important question is how to deal with output from the interpreter including error messages and output generated by the `print` statements? Print output is host specific, e.g is it a terminal console or a GUI application? For this reason, we need a more generic way to handle such output. The easiest way to deal with output is to pass the Rhodus interpreter a callback. Internally we already do this, when the interpreter calls `print`, it in turn calls a callback that will do the actual printing. The other thing to keep in mind is we can't just ask the host application to display a list or array since the host has no idea what the data types are. Instead, we will convert all output into a string form. That is we will represent whatever data type was asked to be output as a string. Displaying strings at the host is very generic and should be possible by almost any host.

Let's declare two callback signatures:

```
void rhodus_print (char* astr);
void rhodus_println (char* astr);
```

One is for when `print` is called and another for `println` is called. We need some way to tell Rhodus about these. A convenient way to do this is to define a struct (i.e a record) that will include function pointers to the print methods. We'll define this, together with the

function pointer declarations, using a C struct as:

```
void (*print)(char*);
void (*println)(char*);

typedef struct rhodus_config {
    print printptr;
    println printlnptr;
} rhodus_config_t;
```

```
void rhodus_print (char* astr);
void rhodus_println (char* astr);
```

We can create this struct and populate it with the function pointers. We then pass it to `rhodus_initialize`. For example:

```
void rhodus_print (char* msg)
{
    printf ("%s", msg);
}

void rhodus_println (char* msg)
{
    printf ("%s\n", msg);
}

void main() {
    rhodus_config_t *pConfig;
    intPtr_t handle;

    pConfig->print = rhodus_print;
    pConfig->println = rhodus_println;

    handle = rhodus_initialize (&pConfig);
}
```

We can use this to execute some code by using the following to lines:

```
rhodus_run (handle, "a = 4; b = 6; println (a + b)");
rhodus_terminate (handle);
```

That deals without output from the `print` functions, but what about error handling? What if there is a compiler or runtime error? You may have noticed that the method `rhodus_run` returns an integer. The value of this integer determines success or failure. If `rhodus_run` returns a negative integer, we have an error condition, otherwise the call was successful. This might seem a bit primitive compared to modern programming techniques such as exception handling. The reason we can't use exception handling is that exceptions are

very programming language specific and there is no way a hosting application would even understand how to deal with it. Most likely, if we were to raise an exception in Object Pascal inside a C host, the C host would likely crash. In general, exceptions should never attempt to propagate out of the Rhodus library when being used by a host. The safest way to deal with errors is by passing back error codes. We should therefore modify the above call to:

```
if (rhodus_run (handle, "a = 4; b = 6; println (a + b)") < 0)
    printf ("Error while calling rhodus interpreter\n");

rhodus_terminate (handle);
```

How do we find out what kind of error it was? For this we need to add a forth API method called `rhodus_getLastError()`. If there was an error, a call to this method will return a pointer to a struct (record) that contains the error message. The reason for using a struct is that in future we can expand the type of error information returned. The signature for this method is currently:

```
typedef struct rhodus_error {
    int errorCode;
    char* errorMsg;
} rhodus_error_t;
```

The `errorCode` is not used at the moment. The signature for the `rhodus_getLastError()` method is therefore:

```
rhodus_error_t* rhodus_getLastError();
```

We might use it this way:

```
if (rhodus_run (handle, "a = 4; b = 6; println (a + b)") < 0) {
    printf ("Error while calling rhodus interpreter:\n");
    printf ("\s\n", rhodus_getLastError().errorMsg);
}

rhodus_terminate (handle);
```

In Object Pascal the error record is defined as follows:

```
TRhodusError = record
    errorCode : integer;
    errorMsg : PAnsiChar;
end;
PRhodusError = ^TRhodusError;
```

We next come to one thorny little issue which is to do with memory ownership. The prob-

lem arises because the host application and the Rhodus library, operate completely separately when it comes to memory management. It's not possible for the host, perhaps a C based host, to free memory allocated by the Rhodus interpreter and vice versa. The key area of concern is string handling, particularly when returning string data back from the Rhodus library, for example `rhodus_getLastError`. The string that this method returns was created within the Rhodus library and therefore the library owns the memory that was allocated for this error string. Under no circumstances must the host attempt to free this memory. If the host is C based, it might be tempting to free the string after use, don't, it's not the host's memory to free. If it does attempt to free the string, the likely result is a crash.

Likewise, when the host passes the script string using `rhodus_run`, that memory is owned by the host and the Rhodus library should not attempt to manipulate it. Instead the Rhodus library makes a copy and works with the copy where it has full ownership of the allotted memory.

The second issue to be aware of is that the Rhodus interpreter deals with Object Pascal strings, that is unicode, referenced counted strings. If your host is a C application, it will have no idea what this string type is. Instead, we again revert to the lowest common denominator by using `char*` based strings. Luckily, Delphi makes it very easy to move back and forth between `string` and `char*`.

The final method I'd like to add to the API at this stage is a method we can use to get configuration information from the Rhodus interpreter. The signature for this method is given by:

```
rhodus_settings_t* rhodus_getSettings (int handle);
```

That is `rhodus_getSettings` returns a pointer to a `rhodus_settings_t` struct:

```
typedef struct rhodus_settings {
    char* version;
} rhodus_settings_t;
```

At the moment the only information available is the version for the current Rhodus interpreter. In C you'd use it like:

```
typedef struct rhodus_settings {
    char* version;
} rhodus_settings_t;

rhodus_settings_t* pRhodusSettings;

pRhodusSettings = rhodus_getSettings (handle);
printf ("version = %s\n", pRhodusSettings->version);
```

Last but not least, what is the Rhodus library itself? Because the library is written in Object Pascal, the only viable option is to distribute the library as a sharable library, i.e `dll`

(Windows), dylib (Mac), or .so library (Linux). We can't distribute the library as a static library because the object format that Delphi uses isn't compatible with other compilers.

6.3 Rhodus Sharable Library

Let's now look at the Rhodus library itself.

First of all the library will be in the form of a dll. We must therefore create a dll project in the Delphi IDE. For those of you have never created a dll before this is quite straightforward. To start a new dll project, go to the File menu in the Delphi IDE and select New. In New you should see the option 'Dynamic Library'. If not, select 'Other' and look for the option there. Upon selection, you'll get a library file as shown below:

```
library Project1;

uses
  System.SysUtils,
  System.Classes;

{$R *.res}

begin
end.
```

Note that I've removed a bunch of commenting that Delphi includes to save space. The only thing we need to do with this file is to specify what methods we would like to export from the dll. To do this we just add the keyword `exports` and the list of methods to export. For example:

```
library librhodus;

uses
  System.SysUtils,
  System.Classes;

{$R *.res}

exports
  rhodus_initialize,
  rhodus_terminate,
  rhodus_run,
  rhodus_getLastError,
  rhodus_getSettings;

begin
end.
```

I've also renamed the project to `librhodus`. This is all we have to do with the project file. The rest of the work involves implementing the methods we intend to export. We will implement these methods in a separate unit, which we will call `uEmbeddAPI.pas`. The easiest method to implement is `rhodus_terminate` method:

```
function rhodus_terminate (handle : NativeInt) : integer; stdcall;
var rhodus : TRhodus;
begin
  try
    rhodus := TRhodus (handle);
    rhodus.free;
  except
    on e: Exception do
      begin
        lastErrorStr := e.Message;
        result := -1;
      end;
    end;
  end;
end;
```

The method signature uses `NativeInt` which not everyone may be familiar with. We have to keep in mind that the library might be used as a 32-bit or 64-bit library. The argument called `handle` is a reference to a Rhodus object, i.e a pointer. On a 32-bit platform that size is 32-bit, whereas on a 64-bit platform the size is 64-bit. The argument type therefore needs to adapt to the particular platform. One way to do that is to use `NativeInt` as the type because this integer is 32-bit on a 32-bit platform and 64-bit on a 64-bit platform.

The other major issue is if the caller passes in a `handle` that doesn't represent a reference to a Rhodus object. That has the potential to crash the system. There are a number of ways around this. One of the most robust methods is to maintain a dictionary of handles where the key is the ordinal value of the dictionary item. In this case the call doesn't pass a reference but an integer number. If that number isn't found in the dictionary, then it is an error, otherwise it uses the corresponding entry it finds as the Rhodus reference. This approach is much more important if we were building a python interface to `librhodus`. Because python is user interactive, the user could literally pass anything as the `handle`. In this situation it is essential we insulate the Rhodus library from rogue handles and some kind of dictionary mechanism on the library side would be useful in this case. Of course we would also most likely create a Python class that would insulate a user from calling the library methods directly so that each instance of a Rhodus interpreter would be a separate instantiation of that class.

For now we will just wrap the code in an exception handler. I'm assuming that a call from a host application to the library will have a higher level of trust than a user typing python code. That means triggering the exception will be rare event and if it does happen its probably due to some other more serious issues. Using an exception handle to hide bad inputs to the library call is not the best way to do it and I will likely return to this another

time and implement a dictionary at the library end to completely insulate a user from having to deal with references at all.

The next method to look at is `rhodus_initialize`.

```
function rhodus_initialize (var config : TRhodusConfig) : NativeInt; stdcall
var rhodus : TRhodus;
begin
  try
    rhodus := TRhodus.Create;
    result := NativeInt (rhodus);

    rhodus.setPrintCallBack(config.printPtr);
    rhodus.setPrintLnCallBack(config.printlnPtr);
  except
    on e: Exception do
      begin
        lastErrorStr := e.Message;
        result := -1;
      end;
    end;
  end;
end;
```

The method returns a reference to a Rhodus object, hence we return a `NativeInt` for reasons given previously. If you're calling this from C then I recommend declaring the method at the C end using `IntPtr_t` as its properties are the same as `NativeInt`. We also wrap the code in an exception handler to ensure any errors are caught and are handled gracefully by returning a -1 to the caller. At this point you might be thinking why not return a `nil` if a problem arises? We could and it is something that one often sees if a method returns a pointer but there has been an error. The reason why I'm using -1 instead is simply to remain consistent with the other methods that do return a -1 in the event of an error condition. This is a personal preference. It would be trivial to change it so that it returns `nil` instead if there is a problem.

As well as returning an error condition, we also extract the exception message and store in a global record variable `lastError` which we can retrieve afterwards using `rhodus_getLastError`.

The other thing we do is copy the print function pointers from the config struct to Rhodus.

The most complicated method is `rhodus_run`. This method must compile the code and run it. In each case we need to check for errors. In the event of an error we return -1 and copy the error message to the global variable `lastError` which can be retrieved by `rhodus_getLastError`.

We could make the running of scripts more sophisticated. For example, we could split the call into two, one call to compile and another to run. This would allow scripts to be precompiled and would make running scripts faster especially if they were called often.,

```

function rhodus_run (handle : NativeInt; code : AnsiString) : integer; stdcall;
var rhodus : TRhodus;
    syntaxError : TSyntaxError;
    compilerError : TCompilerError;
begin
    result := 0;
    rhodus := TRhodus (handle);
    if rhodus.compileToAST (code, syntaxError) then
        begin
            if rhodus.generateByteCode (True, compilerError) then
                rhodus.runCode (mainModule, True)
            else
                begin
                    lastErrorStr := 'ERROR ' + '[line ' + inttostr (compilerError.lineNumber)
                        + ', column: ' + inttostr (compilerError.columnNumber)
                        + ']' + compilerError.errorMsg;
                    result := -1;
                end;
            end
        else
            begin
                lastErrorStr := 'ERROR ' + '[line ' + inttostr (syntaxError.lineNumber)
                    + ', column: ' + inttostr (syntaxError.columnNumber)
                    + ']' + syntaxError.errorMsg;
                result := -1;
            end;
        end;
    end;
end;

```

Last but not least, we must decide on the calling convention, at least for the 32-bit platform. For the 64-bit platform there is only one calling convention so we needn't worry about that platform. For the 32-bit platform there are a number of options, most notably, *cdecl* and *stdcall*. C generally follows the *cdecl* convention whereas the windows API uses *stdcall*. For our purpose it doesn't really matter which one we choose. What is important is that we make it clear which one we're using and that the caller must comply. If I'm writing a pure C application dll that will be called from C then I tend to use *cdecl*. For everything else one can use *stdcall* or *cdecl*. We'll use *stdcall*. In any event, we must decorate every exported method on the library side and every imported method on the host side with *stdcall*. For example:

```

function rhodus_run (handle : NativeInt; code : AnsiString) : integer; stdcall;

```



7

Optimizations

7.1 Introduction

for loop: combine isGt and jmpIfTrue with jmpIfGt

Watch for downto which uses isLt.

Base switch statement on a Jump table?

8

Surprise Chapter

8.1 Introduction

This chapter will be available in the commercial version of the book.

9

Reference to Libraries

9.1 Introduction

This chapter is a reference chapter for the current set of built-in modules.

9.2 Lists Module and List Object Methods

The list module is called `lists`. If you type `lists` at the console you'll get:

```
>>lists
Module: lists List Module
```

We can use `dir()` to see the current set of supported methods in the list module:

```
>>lists.dir()
["rndu", "range", "dir", "rmdi"]
```

These methods are described below:

range(start, end, step): This method returns (end-start)/step values in a list between the values of start and end.

```
>> lists.range (0, 10, 2)
[0,2,4,6,8]
```

rndi(lower, upper, n): This method returns a list random integer such that $\text{lower} \leq n \leq \text{upper}$. The random numbers will include the upper and lower bounds.

```
>>println (lists.rndi (5, 10, 3))
[10,6,7]
```

rndu (n): This method returns n random floating point number in the range [0.0, 1.0).

```
>>println (lists.rndu (4))
[0.651631591841578,0.56693290756084,0.532966091763228,0.318821681430563]
```

List Object Methods

As well as methods from the list module, a list object itself has associated with it a number of methods. This list of methods can be obtained by using `dir()` on a list object. For example:

```
>> a = [1,2]
>> a.dir()
["len","find","toUpper","toLower","left","right","mid","trim","split","dir"]
```

min (): This returns the smallest value it can find in a list. The one restriction is the list must only contain numerical values. An error is reported if it is found to contain a string, another list etc.

```
>>a = [5,4,6,7,8]
>>a.min ()
5
```

max (): This returns the largest value it can find in a list. The one restriction is the list must only contain numerical values. An error is reported if it is found to contain a string, another list etc.

```
>>a = [5,4,6,7,8]
>>a.max ()
8
```

append (value): This method takes one argument, a value and will append the list. The method does not return anything but the associated list is modified in the process.

```
>>a = [5,4,6,7,8]
>>a.append (23)
>>println (a)
[,4,6,7,8,23]
```

Note that if you want to preserve the original list, then make a copy first as in:

```
>>a = [5,4,6,7,8]
>>b = a
>>a.append (23)
>>println (a)
[5,4,6,7,8,23]
>>println (b)
[5,4,6,7,8]
```

pop (): This removes the last item from the list and returns the item to the caller.

```
>>a = [5,4,6,7,8]
>>a.remove ()
>>println (a)
[5,4,6,7]
```

len (): This returns length of the list.

```
>>a = [5,4,6,[7,8], True]
>>println (a.len ())
5
```

insert(value, insertIndex): This is a method that will insert a new item before the indexth-position. Indexing starts at zero.

```
>>a = [5,4,6,[7,8]]
>>a.insert (True, 2)
>>a
[5,4,6,True,[7,8]]
```

9.3 Strings Module and String Object Methods

The string module is called `strings`. If you type `strings` at the console you'll get:

```
>>strings
Module: strings String Module
```

Use `dir()` we can see the current set of supported strings methods:

```
>>strings.dir()
["format","str","dir","val"]
```

format (str, format): This method will format a string according to a format specification. This follows the format specification used by Delphi, see <https://docwiki.>

embarcadero.com/Libraries/Sydney/en/System.SysUtils.Format.

```
>>>strings.format (1.23, "%4.1f")
1.2
```

val (value): This method will convert an integer or float into a string,

```
>>>strings.val (2.356)
2.356
```

str (str): This method will convert a string into an integer or float depending on the string.

```
>>>strings.str ("0.23")
0.23
```

String Object Methods

left (n): This method returns the n left-most characters of the string.

```
>>"12345".left (3)
123
```

right (n): This method returns the n right-most characters of the string.

```
>>"12345".right (3)
345
```

mid (n, length): This method returns a string of specified length, length, and starting point, n from the given string. Note that n indexes from zero.

```
>>"123456789".mid (3, 5)
45678
```

len (): This method returns a string of specified length, length, and starting point, n from the given string. Note that n indexes from zero.

```
>>"123456789".len ()
9
```

trim (): This method returns a string where any spaces on the left or right of the string have been removed.

```
>>" 123456789 6 ".trim ()
123456789 6
```


toUpper (): This method returns a string where any lowercase characters have been made uppercase.

```
>>"aBcD".toUpperCase ()
ABCD
```

toLowerCase (): This method returns a string where any uppercase characters have been made lowercase case.

```
>>"aBcD".toLowerCase ()
abcd
```

var.find (substr): This method finds a substring in string. It returns -1 if it fails or the index position if it succeeds. Note indexing starts at zero.

```
>>"ABCDEFGH".find ("CD")
2
```

var.split (character): This method will split the string argument at a given character into a list of strings. Note the split character is not included in the split strings.

```
>>>"AB CD DE".split (" ")
["AB", "CD", "DE"]
```

9.4 Arrays Module and Array Object Methods

9.5 Math Library: math

The math module is called math. If you type math at the console you'll get:

```
>>math
Module: math Math Module
```

Using `dir()` we can see the current set of supported strings methods:

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

Some of these should be self-explanatory such as `sqrt` for computing the square root, and `sin`, `cos` and `tan` for computing the elementary trigonometric functions (arguments should be in radians). Almost all of them appear with the same name in other programming languages

The `acos`, `asin` and `atan` are the inverse trigonometric functions.

`toDegree` and `toRadians` will convert radians and degrees to degrees and radians respectively.

```
>>math.toDegrees (math.pi/2)
90
>>math.toRadians (180)
3.14159265358979
```

`log` computes the logarithm to the based 10 and `ln` the log to the base e . `exp` will compute the exponential e^x .

The two constants `pi` and `e` return π and Napier's constant, e respectively.

`min` will return the minimum of two numbers and `max` the maximum of two numbers.

```
>>math.min (4, 2)
2
>>math.max (7, 3.4)
7
```

`abs` will return the absolute value.

```
>>math.abs (-5.6)
5.6
```

`round` will returns the value rounded to the nearest integer number.

```
>>math.round (-5.6)
6
>>math.round (-5.1)
5
>>math.round (5.6)
6
>>math.round (5.1)
5
```

`floor` will round down the nearest integer. That means -3.4 goes to -3 and 3.4 goes to 2 .

```
>>math.floor (-5.1)
6
>>math.floor (5.9)
5
```

`ceil`, short for ceiling, will round up to the nearest integer. That means -3.9 goes to -3 and 3.4 goes to 3 .

```
>>math.ceil (-5.1)
```

```
5
>>math.ceil (5.9)
6
```

9.6 Random Module: random

The random module is called random and provides access to random number generation. If you type random at the console you'll get:

```
>>random
Module: random Random Module
```

Use dir() we can see the current set of supported strings methods:

```
["randint", "gauss", "seed", "dir", "random"]
```

random (): Returns a pseudo-random floating point number between 0 and 1 (but not including 1). In this version the random number generator uses the Delphi random function which itself uses a simple linear congruential generator, that is a recursive equation of the form: $X_{x+1} = (aX_n + c) \bmod m$. According to Wikipedia (https://en.wikipedia.org/wiki/Linear_congruential_generator), Delphi random uses a modulus, m of 2^{23} , a multiplier a of 134775813 and an increment c of 1.

```
>>random.random ()
0.49627447151579
```

randint (number): Returns an integer pseudo-random number between 0 and *number* – 1

```
>>random.randint (5)
2
```

gauss (mean, stdev): This method returns a pseudo-random number drawn from a normal distribution with mean *mean* and standard deviation, *stdev*.

```
>>random.gauss (1, 0.4)
0.532361312210798
```

seed (number): Use this method to set the seed for the random number generator. This allows stochastic studies to be exactly repeatable. A convenient 'random' seed can be provided by time.getTickCount().

```
>>random.randint (5)
```

2





List of Virtual Machine Opcodes

Abbreviation: TOS = Top Of Stack. The virtual machine stack grows upwards, therefore if the top of the stack is TOS, the next entry in the stack will be TOS-1.

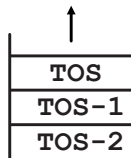


Figure A.1 Operand Stack: The stack grows upwards, TOS = Top Of Stack.

Nop

No operation

Description: This instruction performs no operation and can be used as a place holder.

Arithmetic Operations

add

Add operation

Description: This instruction will carry out addition by popping two values from the stack, adding their values and pushing the result back onto the stack.

$TOS = TOS-1 + TOS$

sub

Subtract operation

Description: This instruction will carry out subtraction by popping two values from the stack, subtracting their values, and pushing the result back onto the stack. The order of operands is given by:

pop value₁, pop value₂, push (value₂ - value₁)

$TOS = TOS-1 - TOS$

mult

Multiplication operation

Description: This instruction will carry out multiplication by popping two values from the stack, multiplying their values, and pushing the result back onto the stack. The order of operands is given by:

$TOS = TOS-1 \times TOS$

div

Division operation

Description: This instruction will carry out division by popping two values from the stack, dividing their values, and pushing the result back onto the stack. The order of operands is given by:

pop value₁, pop value₂, push (value₂ / value₁)

$TOS = TOS-1 / TOS$

mod

Modulus operation

Description: This instruction will pop two items from the stack, compute the modulus then pushes the result back onto the stack

pop value₁, pop value₂, push (value₂ mod value₁)

$TOS = TOS-1 \text{ mod } TOS$

pow

Power operation

Description: This instruction will compute the power by popping two values from the stack, computing one to the power of the other, and pushing the result back onto the stack. The order of operands is given by:

pop value₁, pop value₂, push (value₂ ^ value₁)

TOS = POWER (TOS-1, TOS)

umi

Unary-minus operation

Description: This instruction will pop off an item from the stack, negate it then push the result back onto the stack.

TOS = -TOS

inc

Increment symbol

Description: This instruction will use the value of the index value to access a value in the symbol table and increment the value by one.

dec

Increment symbol

Description: This instruction will use the value of the index value to access a value in the symbol table and decrement the value by one.

localinc

Increment symbol

Description: This instruction will use the value of the index value to access a value in a user function and increment the value by one.

localdec

Increment symbol

Description: This instruction will use the value of the index value to access a value in a user function and decrement the value by one.

Boolean Operations

and

AND operation

Description: This instruction will pop two items off the stack, compute the Boolean AND operation, then push the result back onto the stack.

TOS = TOS-1 and TOS

or

OR operation

Description: This instruction will pop two items off the stack, compute the Boolean OR operation, then push the result back onto the stack.

$TOS = TOS-1 \text{ or } TOS$

not

NOT operation

Description: This instruction will pop off an item from the stack, compute the Boolean NOT operation, then push the result back onto the stack.

$TOS = \text{not } TOS$

xor

XOR operation

Description: This instruction will pop two items off the stack, compute the Boolean XOR operation, then push the result back onto the stack.

$TOS = TOS-1 \text{ xor } TOS$

Push and Pop Instructions

pushi

push integer:

pushi value

Description: This instruction will push the integer value of the argument in the bytecode onto the stack.

pushb

push Boolean

pushb value

Description: This instruction will push the Boolean value of the argument in the bytecode onto the stack.

pushd

push double value

pushd index

Description: This instruction will take the integer argument in the bytecode and use it to index a floating point value in the constant table, which it then pushes onto the stack.

pushs

push string

pushs index

Description: This instruction will take the integer argument in the bytecode and use it to index a string stored in the constant table, which it then pushes onto the stack.

pop

pop a value off the stack

Description: This instruction will pop one item off the stack and return the value to the caller.

Branching Instructions

jmp

Unconditional jmp to a new instruction

Description: This instruction will use the bytecode integer argument as a relative jump distance.

jmpIfTrue

Jmp if the value on the stack is true

Description: This instruction will pop one item of the stack. If the popped item is True, the instruction will do a relative jump based on the bytecode integer argument.

jmpIfFalse

Jmp if the value on the stack is false

Description: This instruction will pop one item of the stack. If the popped item is False, the instruction will do a relative jump based on the bytecode integer argument.

User and Built-in Function Calls

call

Call a user-defined function.

Description: This instruction will call a user-defined function. The stack will contain a reference to the function object followed by any optional arguments. When a call is made, the first item popped off the stack will be the user function object. The following items on the stack will be the arguments required by the user function. The first argument

on the stack is the last argument in the function call. Note that built-in functions are also routed through this call.

ret

Return from a function with a value

Description: Push a return value onto the stack and return from a call.

Symbol Table Instructions

store

Store a value to the symbol table

Description: Pop a value from the stack and copy the value to the symbol table. The index argument represents the index to the symbol table entry.

load

Load a value from the symbol table

Description: Load a value from the symbol table and push the value onto the stack. The index argument represents the index to the symbol table entry.

storeAttr

Store an attribute to the symbol table.

Description: `storeAttr` expects the first item on the stack to be the value to be stored, followed by a module reference. The operand for `loadAttr` will be a name of a symbol. For example in: `mymodule.a = 45` the stack will contain `mymodule`, `45` and the operand will be the string `a`. `storeAttr` only makes sense when storing values in a model. The followed expression for example wouldn't make sense: `"hello".len() = 56` in this context.

loadAttr

Load an attribute.

Description: Given a syntactic expression such `primary.secondary`, `loadAttr` will expect a reference to the primary to be on the stack. The operand for `loadAttr` will be a name of a symbol. For example `mymodule.a`, will result in `loadAttr` pushing the value of `mymodule` onto the stack. it will then attempt to local the symbol name in the module, and if found push its value on to the stack. A primary can be a variable name, module, string, list, array or function.

localStore

Store a value to the local storage area on the stack

Description: Pop a value from the stack and copy the value to local storage in a user function. The index argument represents the index to the local space in the user function.

localLoad

Load a value from the local storage area on the stack

Description: Load a value from the local storage in a user function and push the value onto the stack. The index argument represents the index to the local space in the user function.

Boolean Test Instructions

isEq

Test for Equality

Description: Pop two values from the stack, if they are equal in value, push True on the stack, else push False.

If TOS-1 = TOS then push true else push false

isGt

Test for greater than

Description: Pop two values from the stack, if one is greater than the other, push True on the stack, else push False.

If TOS-1 > TOS then push True else push False

isGte

Test for greater than or equal to

Description: Pop two values from the stack, if one is greater than or equal to the other, push True on the stack, else push False.

If TOS-1 >= TOS then push True else push False

isLt

Test for less than

Description: Pop two values from the stack, if one is less than the other, push True on the stack, else push False.

If TOS-1 < TOS then push True else push False

isLte

Test for less than or equal to

Description: Pop two values from the stack, if one is less than or equal to the other, push True on the stack, else push False.

If TOS-1 <= TOS then push True else push False

isNotEq

Test for not equal to

Description: Pop two values from the stack, if they are not equal to each other, push True on the stack, else push False.

If TOS-1 <> TOS then push True else push False

List Instructions

createList

Create a list: [1,2,3, etc]

Description: Use this instruction to create a list structure. The first item popped off the stack will be an integer representing the number of items on the stack to create the list.

The pseudo-code for this option is given below:

```
list = listObject
n = pop()
for i = 1 to n do
    list.append (pop())
end
```

lvecIdx

Load an indexed element of a list onto the stack

Description: Use this instruction to extract an element from a list, and push the value on to the stack. The first item on the stack will be the index, and the second item on the stack will be the list.

svecIdx

Store a value to an index element of a list

Description: Use this instruction to copy an item into a list. The first item on the stack will be the list, the second item the index into the list, and the third item the value to copy into the list.

localLvecIdx

Load an indexed element of a list onto the stack

Description: Use this instruction to extract an element from a list, and push the value on to the stack. The first item on the stack will be the list, and the second item on the stack the index into the list. Used to access lists that are defined locally in a user function.

localSvecIdx

Store a value to an index element of a list

Description: Use this instruction to copy an item into a list. The first item on the stack will be the list, the second item the index into the list, and the third item the value to copy into the list. Used to access lists that are defined locally in a user function.

Miscellaneous

halt

Halt the program

Description: Use this instruction to terminate the execution of a program.

print

Print a string

Description: Use this instruction to print one or more values on the stack. The instruction pops the first item from the stack, which will be the number of items to print. The method then issues a print event for each item out in turn (in reverse order). print events may be captured.

println

Print a string followed by a newline

Description: This is the same as print except right at the end it issues a newline character.

History

1. VERSION: 1.0

Date: 2020-3-1

Author(s): Herbert M. Sauro

Title: Writing an Interpreter in Object Pascal: Part 3

Modification(s): First Release to Printing Press

Index

B

BASIC 17

G

Go 17

Gravity 17

L

list of opcodes 87

W

wren 17

