

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 Homogeneous Arrays	10
1.4 Error Handling	12
1.5 Grammar Specification for Version 3	12
1.6 Useful Reading	15
1.6.1 Introductory Books	15
1.6.2 More Advanced Books	16
1.6.3 Source Code	16
History	17
Index	19

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>

February 2020
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 continually and 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 was 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.

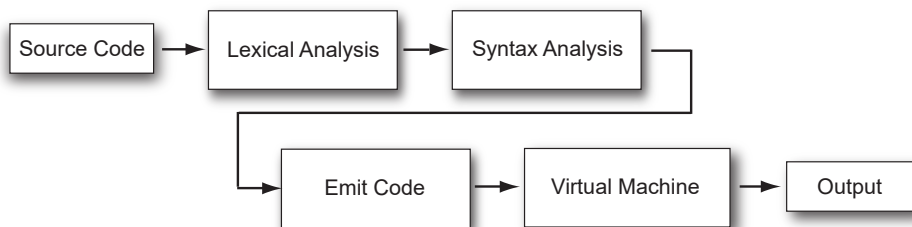


Figure 1.1 Simplified flow from source code to output in version 2 of the Rhodus interpreter.

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.

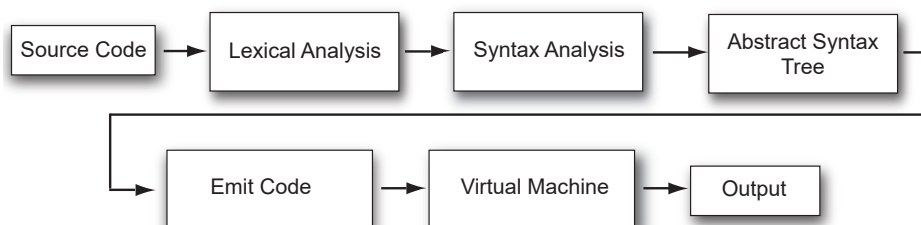


Figure 1.2 Simplified flow from source code to output in version 3 of the Rhodus interpreter incorporating the abstract syntax tree stage.

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 more commonly used `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 builtin 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 start up module.

With version 3, nine modules provided. The currently loaded modules can be listed by using the `modules` method:

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

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 `_main_` by using `dir()`, for example:

```
>> main().dir()
{"dir","main","readString","dis","asc","stackInfo","strings","getAttr","os",
"readNumber","modules","int","assertTrueEx","symbols","mem",
"assertFalseEx","chr","type","lists"}
```

What it lists in this case are the modules currently accessible from `_main_` which is the same list when we just typed `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()
{"time","os",,"dir","file","astring","strings","lists","math","random","sys"}
```

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, eg `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 there are a couple of issues that needed to be 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.sin)
Returns the minimum of two numbers: math.min (3, 5)
```

1.3.2 User Functions

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

```
x = math.sin
```

Note that function brackets are not included. What this does is create a new copy of `math.sin` and assign it to the variable `x`. Yes you heard right, it makes a copy. This is due to the way garbage collection is handled in the current versions of Rhodus. Other languages, such as Python, use reference counting. 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 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 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 from 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. The `array` 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 empty array and by default, entries are set to zero. `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 those 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]
```

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
```

or

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

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

```
>>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.

Indexing arrays is done in the same way you index lists. Note that 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.

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".toUpper().toLowerCase().toUpperCase().toLowerCase()
abc
```

A grammar that will satisfy such flexibility is given by:

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

Something similar is used in the Python grammar specification (<https://docs.python.org/3/reference/grammar.html>). For example, `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 straight forward even though perhaps initially, it looks a little scary. The big problem with it is that it's not friendly for our recursive decent parser. The grammar shown above is what's called left-recursive (see Part 1). We can see this because the `E` symbol, in three cases, is the first symbol in the production. For example:

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

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

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

it's also impossible to decide which one to pick unless we lookahead further into the token stream to identify the `'a'` and `'b'` but we don't want to do that. Without 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 in a right-recursive rule

which can be parsed using a single lookahead recursive decent parser. The method to do the transformation was described in Part 1 but here I will cheat by using a tool to do it for me. The site https://cyberzhg.github.io/toolbox/left_rec has an on-line tool to remove left-recursion. The tool has two panels, in the upper panel you paste your left-recursive grammar, hit the convert button and your new well-behaved grammar will appear in the bottom panel.

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

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

After conversion it looked like:

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

Notice how all the terminals, a, b, c and d have been moved to the front. This grammar can be recognized with a single token look-ahead, that is it's LL(1) friendly, which is what we're after. Notice also the empty option, that is E' can be a, b, c or none of them. d has been moved to it's own production and represents factor in the grammar we had previously. If we translate the symbols into more meaningful words we get:

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

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

1.6 Useful Reading

1.6.1 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.

1.6.2 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.

1.6.3 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.



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 16

G

Go 16

Gravity 16

W

wren 16

