

Let’s Build A Simple Interpreter. Part 9.

(<https://ruslanspivak.com/lbasi-part9/>)

Date

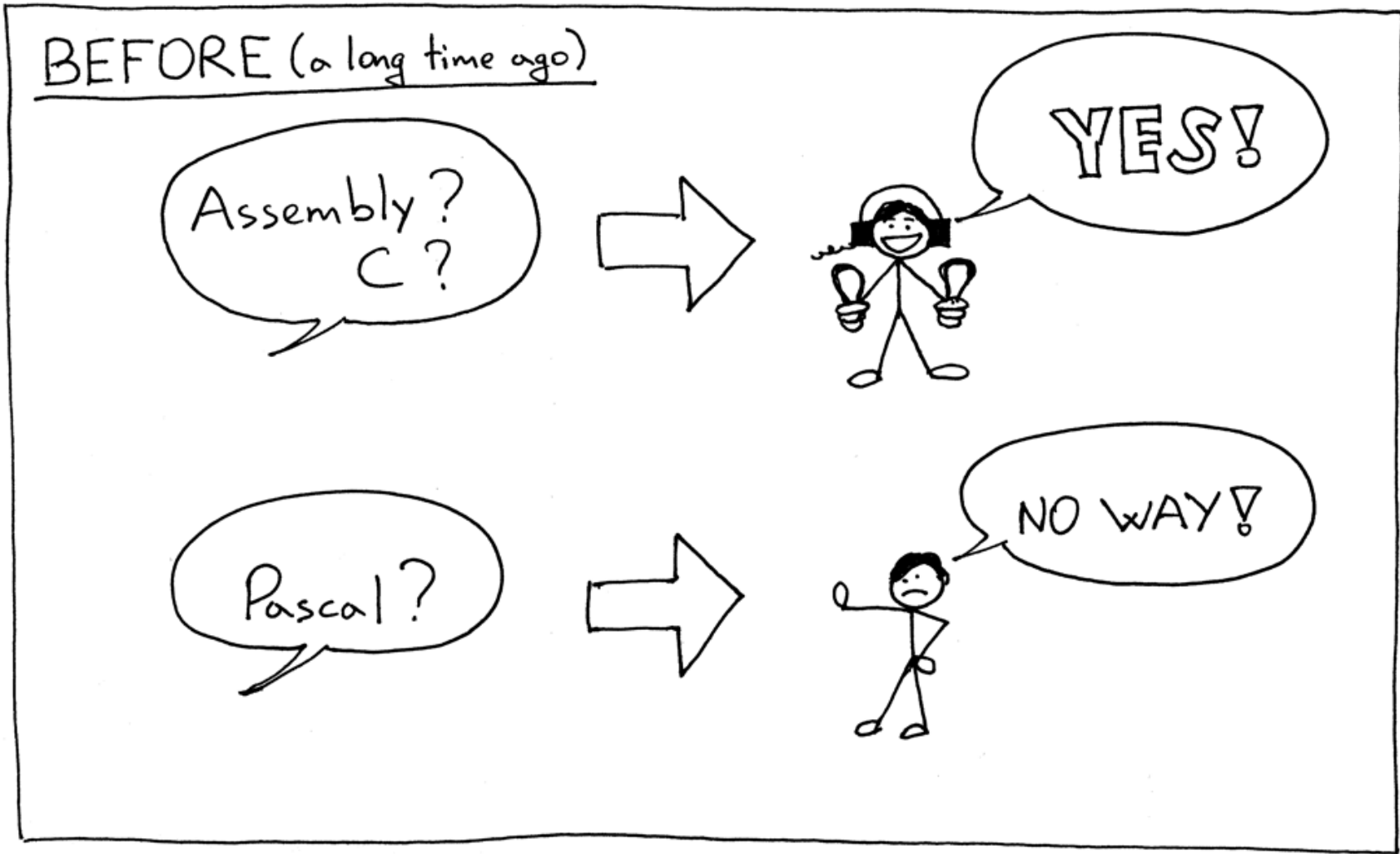
 📅 Sun, May 01, 2016

I remember when I was in university (a long time ago) and learning systems programming, I believed that the only “real” languages were Assembly and C. And Pascal was – how to put it nicely – a very high-level language used by application developers who didn’t want to know what was going on under the hood.

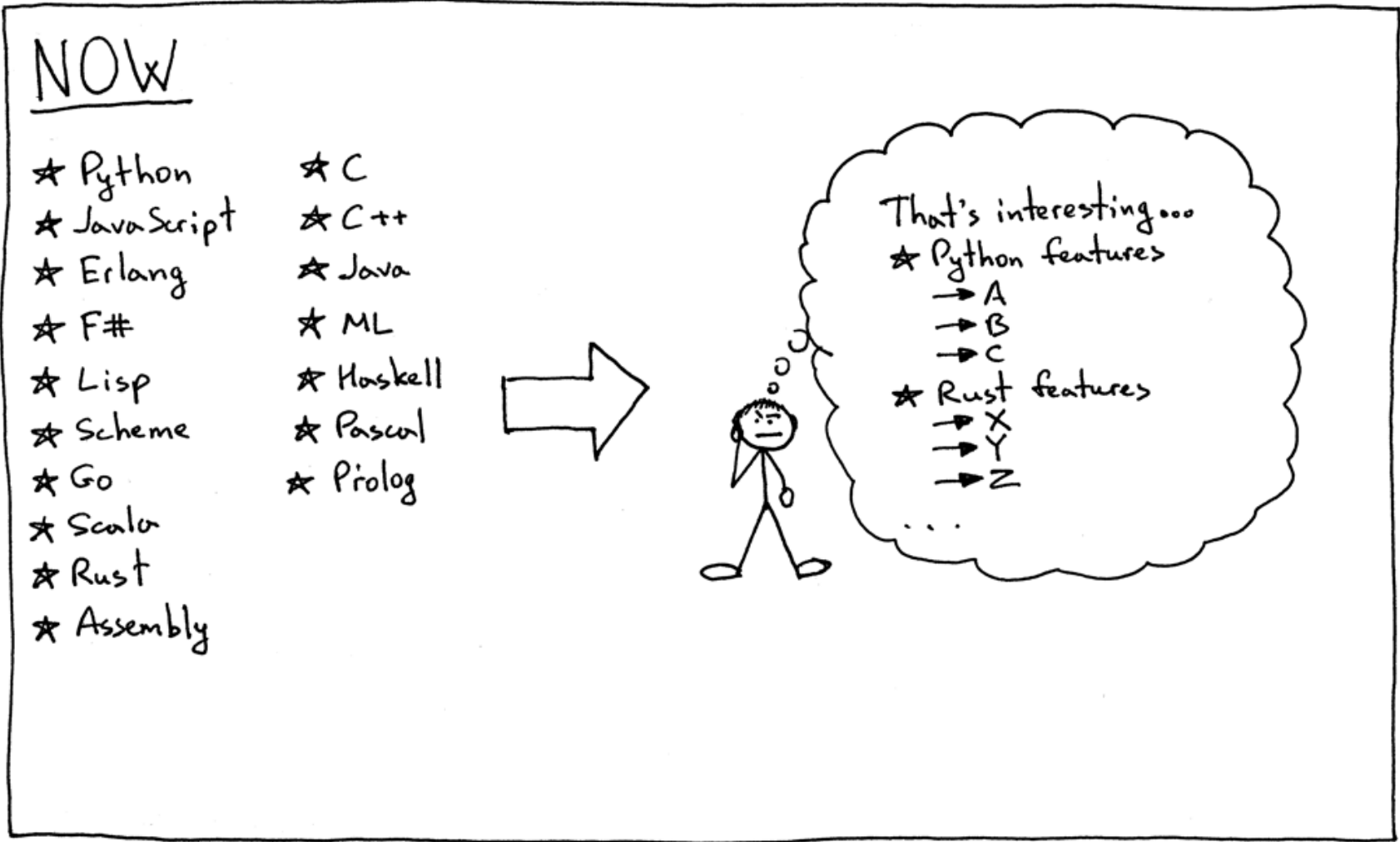
Little did I know back then that I would be writing almost everything in Python (and love every bit of it) to pay my bills and that I would also be writing an interpreter and compiler for Pascal for the reasons I stated in the very first article of the series ([/lbasi-part1/](#)).

These days, I consider myself a programming languages enthusiast, and I’m fascinated by all languages and their unique features. Having said that, I have to note that I enjoy using certain languages way more than others. I am biased and I’ll be the first one to admit that. :)

This is me before:



And now:



Okay, let’s get down to business. Here is what you’re going to learn today:

- 1. How to parse and interpret a Pascal program definition.
- 2. How to parse and interpret compound statements.
- 3. How to parse and interpret assignment statements, including variables.
- 4. A bit about symbol tables and how to store and lookup variables.

I’ll use the following sample Pascal–like program to introduce new concepts:

```
BEGIN
  BEGIN
    number := 2;
    a := number;
    b := 10 * a + 10 * number / 4;
    c := a - - b
  END;
  x := 11;
END.
```

You could say that that’s quite a jump from the command line interpreter you wrote so far by following the previous articles in the series, but it’s a jump that I hope will bring excitement. It’s not “just” a calculator anymore, we’re getting serious here, Pascal serious. :)

Let’s dive in and look at syntax diagrams for new language constructs and their corresponding grammar rules.

On your marks: Ready. Set. Go!

SYNTAX DIAGRAM	GRAMMAR RULE
<div>1 <u>program</u> → compound_statement → DOT →</div>	program : compound_statement DOT
<div>2 <u>compound_statement</u> → BEGIN → statement_list → END →</div>	compound_statement : BEGIN statement_list END
<div>3 <u>statement_list</u> → statement → SEMI → statement →</div>	statement_list : statement statement SEMI statement_list
<div>4 <u>statement</u> → compound_statement → → assignment_statement → → empty →</div>	statement : compound_statement assignment_statement empty
<div>5 <u>assignment_statement</u> → variable → ASSIGN → expr →</div>	assignment_statement : variable ASSIGN expr
<div>6 <u>variable</u> → ID →</div>	variable : ID
<div>7 <u>empty</u> →</div>	empty :
<div>8 <u>factor</u> → PLUS → factor → → MINUS → factor → → INTEGER → → LPAREN → expr → RPAREN → → variable →</div>	factor : PLUS factor MINUS factor INTEGER LPAREN expr RPAREN variable

1. I'll start with describing what a Pascal *program* is. A Pascal **program** consists of a *compound statement* that ends with a dot. Here is an example of a program:

```
"BEGIN  END."
```

I have to note that this is not a complete program definition, and we'll extend it later in the series.

2. What is a *compound statement*? A **compound statement** is a block marked with BEGIN and END that can contain a list (possibly empty) of statements including other compound statements. Every statement inside the compound statement, except for the last one, must terminate with a semicolon. The last statement in the block may or may not have a terminating semicolon. Here are some examples of valid compound statements:

```
"BEGIN END"
"BEGIN a := 5; x := 11 END"
"BEGIN a := 5; x := 11; END"
"BEGIN BEGIN a := 5 END; x := 11 END"
```

3. A **statement list** is a list of zero or more statements inside a compound statement. See above for some examples.

4. A **statement** can be a *compound statement*, an *assignment statement*, or it can be an *empty* statement.

5. An **assignment statement** is a variable followed by an ASSIGN token (two characters, ':' and '=') followed by an expression.

```
"a := 11"
"b := a + 9 - 5 * 2"
```

6. A **variable** is an identifier. We'll use the ID token for variables. The value of the token will be a variable's name like 'a', 'number', and so on. In the following code block 'a' and 'b' are variables:

```
"BEGIN a := 11; b := a + 9 - 5 * 2 END"
```

7. An **empty** statement represents a grammar rule with no further productions. We use the *empty_statement* grammar rule to indicate the end of the *statement_list* in the parser and also to allow for empty compound statements as in 'BEGIN END'.

8. The **factor** rule is updated to handle variables.

Now let's take a look at our complete grammar:

```
program : compound_statement DOT

compound_statement : BEGIN statement_list END

statement_list : statement
                | statement SEMI statement_list

statement : compound_statement
          | assignment_statement
          | empty

assignment_statement : variable ASSIGN expr

empty :

expr: term ((PLUS | MINUS) term)*

term: factor ((MUL | DIV) factor)*

factor : PLUS factor
       | MINUS factor
       | INTEGER
       | LPAREN expr RPAREN
       | variable

variable: ID
```

You probably noticed that I didn't use the star '*' symbol in the *compound_statement* rule to represent zero or more repetitions, but instead explicitly specified the *statement_list* rule. This is another way to represent the 'zero or more' operation, and it will come in handy when we look at parser generators like PLY (<http://www.dabeaz.com/ply/>), later in the series. I also split the "(PLUS | MINUS) factor" sub-rule into two separate rules.

In order to support the updated grammar, we need to make a number of changes to our lexer, parser, and interpreter. Let's go over those changes one by one.


Here is the summary of the changes in our lexer:

LEXER

1


New tokens

- ★ BEGIN
- ★ END
- ★ DOT
- ★ ID
- ★ ASSIGN
- ★ SEMI



2


New peek method

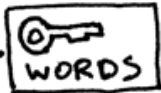


3

New _id method


- ★ Handles ID tokens
- ★ Handles reserved keywords





4

Changes to the get_next_token method



1. To support a Pascal program’s definition, compound statements, assignment statements, and variables, our lexer needs to return new tokens:

- BEGIN (to mark the beginning of a compound statement)
- END (to mark the end of the compound statement)
- DOT (a token for a dot character ‘.’ required by a Pascal program’s definition)
- ASSIGN (a token for a two character sequence ‘:=’). In Pascal, an assignment operator is different than in many other languages like C, Python, Java, Rust, or Go, where you would use single character ‘=’ to indicate assignment
- SEMI (a token for a semicolon character ‘;’ that is used to mark the end of a statement inside a compound statement)
- ID (A token for a valid identifier. Identifiers start with an alphabetical character followed by any number of alphanumerical characters)

2. Sometimes, in order to be able to differentiate between different tokens that start with the same character, (‘.’ vs ‘:=’ or ‘==’ vs ‘=>’) we need to peek into the input buffer without actually consuming the next character. For this particular purpose, I introduced a *peek* method that will help us tokenize assignment statements. The method is not strictly required, but I thought I would introduce it earlier in the series and it will also make the *get_next_token* method a bit cleaner. All it does is return the next character from the text buffer without incrementing the *self.pos* variable. Here is the method itself:

```
def peek(self):
    peek_pos = self.pos + 1
    if peek_pos > len(self.text) - 1:
        return None
    else:
        return self.text[peek_pos]
```

3. Because Pascal variables and reserved keywords are both identifiers, we will combine their handling into one method called *_id*. The way it works is that the lexer consumes a sequence of alphanumerical characters and then checks if the character sequence is a reserved word. If it is, it returns a pre-constructed token for that reserved keyword. And if it’s not a reserved keyword, it returns a new ID token whose value is the character string (lexeme). I bet at this point you think, “Gosh, just show me the code.” :) Here it is:

```
RESERVED_KEYWORDS = {
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
}

def _id(self):
    """Handle identifiers and reserved keywords"""
    result = ''
    while self.current_char is not None and self.current_char.isalnum():
        result += self.current_char
        self.advance()

    token = RESERVED_KEYWORDS.get(result, Token(ID, result))
    return token
```

4. And now let’s take a look at the changes in the main lexer method *get_next_token*:

```
def get_next_token(self):
    while self.current_char is not None:
        ...
        if self.current_char.isalpha():
            return self._id()

        if self.current_char == ':' and self.peek() == '=':
            self.advance()
            self.advance()
            return Token(ASSIGN, ':=')

        if self.current_char == ';':
            self.advance()
            return Token(SEMI, ';')

        if self.current_char == '.':
            self.advance()
            return Token(DOT, '.')

        ...
```

It’s time to see our shiny new lexer in all its glory and action. Download the source code from GitHub (<https://github.com/rspivak/lsbasi/blob/master/part9/python>) and launch your Python shell from the same directory where you saved the spi.py (<https://github.com/rspivak/lsbasi/blob/master/part9/python/spi.py>) file:

```
>>> from spi import Lexer
>>> lexer = Lexer('BEGIN a := 2; END.')
>>> lexer.get_next_token()
Token(BEGIN, 'BEGIN')
>>> lexer.get_next_token()
Token(ID, 'a')
>>> lexer.get_next_token()
Token(ASSIGN, ':=')
>>> lexer.get_next_token()
Token(INTEGER, 2)
>>> lexer.get_next_token()
Token(SEMI, ';')
>>> lexer.get_next_token()
Token(END, 'END')
>>> lexer.get_next_token()
Token(DOT, '.')
>>> lexer.get_next_token()
Token(EOF, None)
>>>
```

Moving on to parser changes.

Here is the summary of changes in our parser:

PARSER

➤ 1

New AST nodes

- ★ Compound
- ★ Assign
- ★ Var
- ★ NoOp

➤ 2

New methods

- ★ program
- ★ compound_statement
- ★ statement_list
- ★ statement
- ★ assignment_statement
- ★ variable
- ★ empty

➤ 3

Changes to the parse and factor methods.

Compound

Assign

NoOp

Var

Expr

NEW METHODS

changes ahead

1. Let's start with new AST nodes:

- *Compound* AST node represents a compound statement. It contains a list of statement nodes in its *children* variable.

```
class Compound(AST):
    """Represents a 'BEGIN ... END' block"""
    def __init__(self):
        self.children = []
```

- *Assign* AST node represents an assignment statement. Its *left* variable is for storing a *Var* node and its *right* variable is for storing a node returned by the *expr* parser method:

```
class Assign(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right
```

- *Var* AST node (you guessed it) represents a variable. The *self.value* holds the variable's name.

```
class Var(AST):
    """The Var node is constructed out of ID token."""
    def __init__(self, token):
        self.token = token
        self.value = token.value
```

- *NoOp* node is used to represent an *empty* statement. For example 'BEGIN END' is a valid compound statement that has no statements.

```
class NoOp(AST):
    pass
```

2. As you remember, each rule from the grammar has a corresponding method in our recursive-descent parser. This time we're adding seven new methods. These methods are responsible for parsing new language constructs and constructing new AST nodes. They are pretty straightforward:

<https://ruslanspivak.com/lsbasi-part9/>

7/17

```
def program(self):
    """program : compound_statement DOT"""
    node = self.compound_statement()
    self.eat(DOT)
    return node

def compound_statement(self):
    """
    compound_statement: BEGIN statement_list END
    """
    self.eat(BEGIN)
    nodes = self.statement_list()
    self.eat(END)

    root = Compound()
    for node in nodes:
        root.children.append(node)

    return root

def statement_list(self):
    """
    statement_list : statement
                    / statement SEMI statement_list
    """
    node = self.statement()

    results = [node]

    while self.current_token.type == SEMI:
        self.eat(SEMI)
        results.append(self.statement())

    if self.current_token.type == ID:
        self.error()

    return results

def statement(self):
    """
    statement : compound_statement
              / assignment_statement
              / empty
    """
    if self.current_token.type == BEGIN:
        node = self.compound_statement()
    elif self.current_token.type == ID:
        node = self.assignment_statement()
    else:
        node = self.empty()
    return node

def assignment_statement(self):
    """
    assignment_statement : variable ASSIGN expr
    """
    left = self.variable()
    token = self.current_token
    self.eat(ASSIGN)
    right = self.expr()
    node = Assign(left, token, right)
    return node

def variable(self):
    """
    variable : ID
    """
    node = Var(self.current_token)
    self.eat(ID)
    return node

def empty(self):
    """An empty production"""
    return NoOp()
```

3. We also need to update the existing *factor* method to parse variables:


```
def factor(self):
    """factor : PLUS factor
               / MINUS factor
               / INTEGER
               / LPAREN expr RPAREN
               / variable
    """
    token = self.current_token
    if token.type == PLUS:
        self.eat(PLUS)
        node = UnaryOp(token, self.factor())
        return node
    ...
    else:
        node = self.variable()
        return node
```

4. The parser’s *parse* method is updated to start the parsing process by parsing a program definition:

```
def parse(self):
    node = self.program()
    if self.current_token.type != EOF:
        self.error()

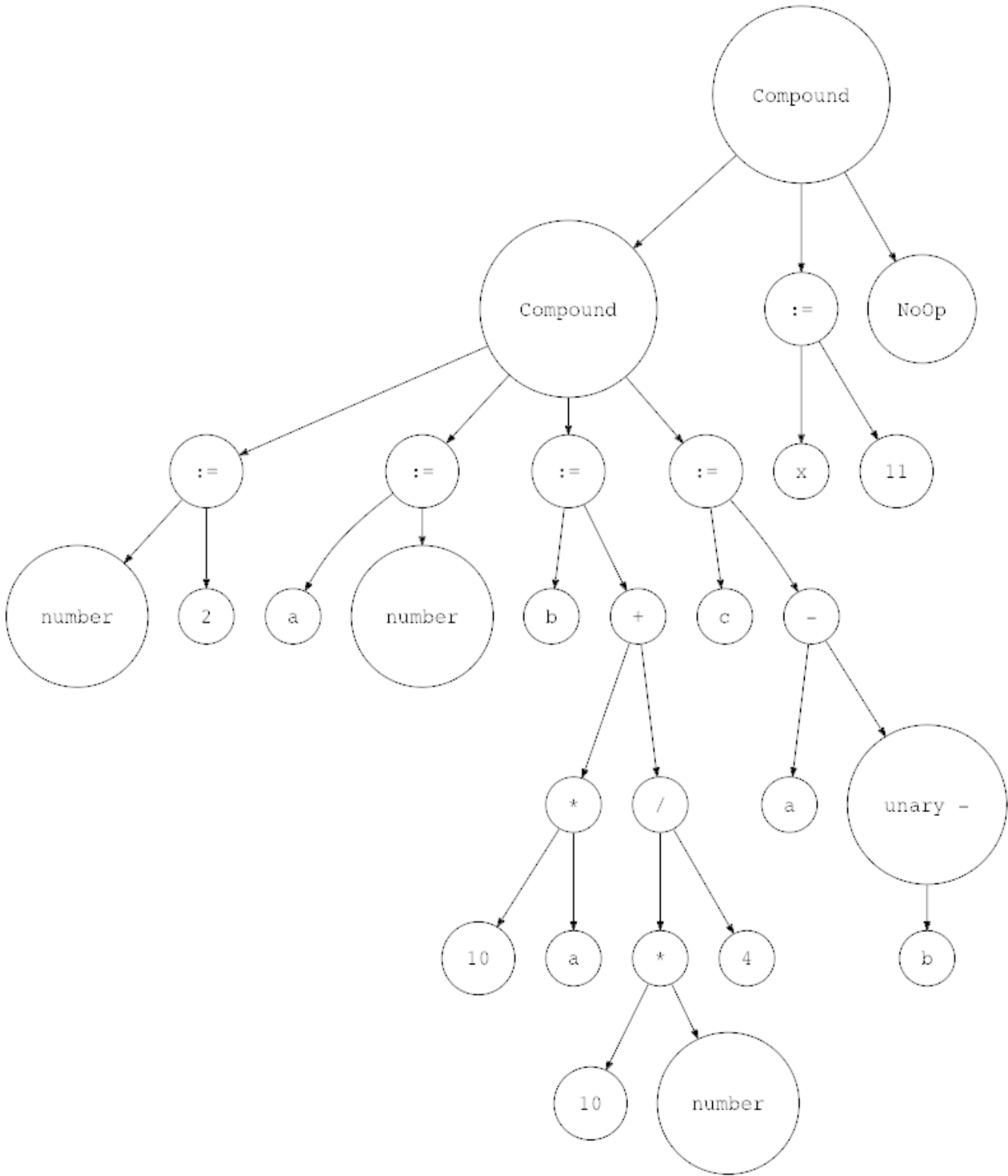
    return node
```

Here is our sample program again:

```
BEGIN
  BEGIN
    number := 2;
    a := number;
    b := 10 * a + 10 * number / 4;
    c := a - - b
  END;
  x := 11;
END.
```

Let’s visualize it with `genastdot.py` (<https://github.com/rspivak/lbasi/blob/master/part9/python/genastdot.py>) (For brevity, when displaying a *Var* node, it just shows the node’s variable name and when displaying an Assign node it shows ‘:=’ instead of showing ‘Assign’ text):

```
$ python genastdot.py assignments.txt > ast.dot && dot -Tpng -o ast.png ast.dot
```



And finally, here are the required interpreter changes:

INTERPRETER

➤ 1 New methods

- ★ visit_Compound
- ★ visit_Assign
- ★ visit_Var
- ★ visit_NoOp

➤ 2 Basic symbol table

GLOBAL_SCOPE

key	value
a	2
x	11
...	...

Welcome, Visitors! 🎉

To interpret new AST nodes, we need to add corresponding visitor methods to the interpreter. There are four new visitor methods:

- visit_Compound
- visit_Assign
- visit_Var
- visit_NoOp

<https://ruslanspivak.com/lsbasi-part9/>

10/17

Compound and *NoOp* visitor methods are pretty straightforward. The *visit_Compound* method iterates over its children and visits each one in turn, and the *visit_NoOp* method does nothing.

```
def visit_Compound(self, node):
    for child in node.children:
        self.visit(child)

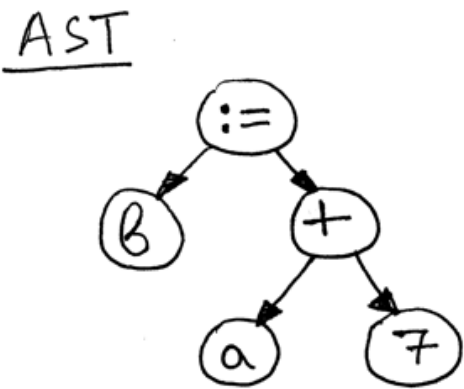
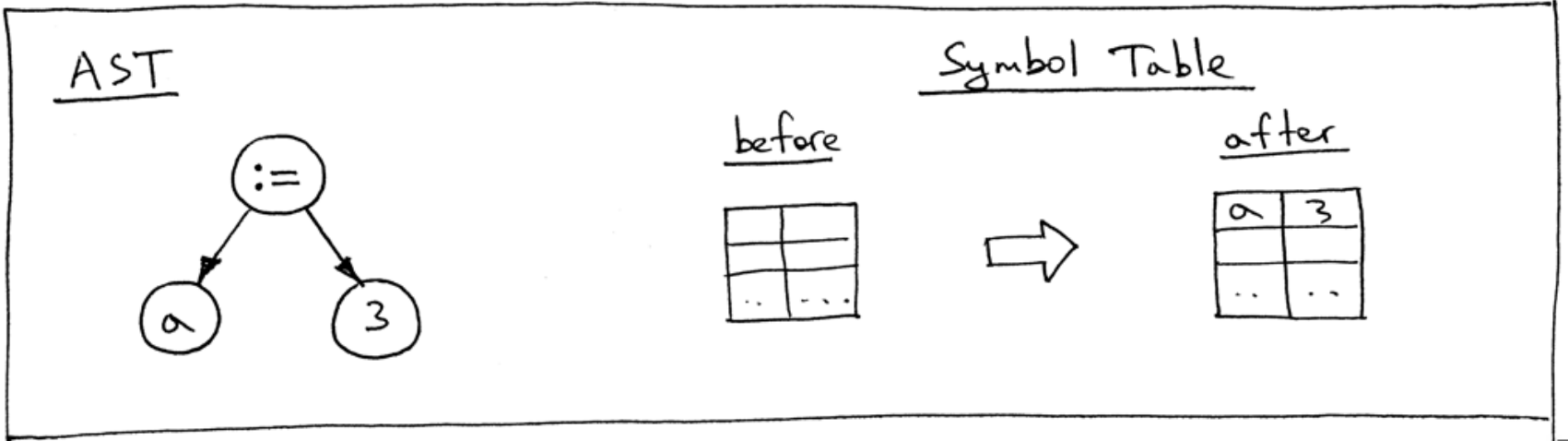
def visit_NoOp(self, node):
    pass
```

The *Assign* and *Var* visitor methods deserve a closer examination.

When we assign a value to a variable, we need to store that value somewhere for when we need it later, and that's exactly what the *visit_Assign* method does:

```
def visit_Assign(self, node):
    var_name = node.left.value
    self.GLOBAL_SCOPE[var_name] = self.visit(node.right)
```

The method stores a key–value pair (a variable name and a value associated with the variable) in a *symbol table* GLOBAL_SCOPE. What is a *symbol table*? A *symbol table* is an abstract data type (ADT) for tracking various symbols in source code. The only symbol category we have right now is variables and we use the Python dictionary to implement the symbol table ADT. For now I'll just say that the way the symbol table is used in this article is pretty “hacky”: it's not a separate class with special methods but a simple Python dictionary and it also does double duty as a memory space. In future articles, I will be talking about symbol tables in much greater detail, and together we'll also remove all the hacks. Let's take a look at an AST for the statement “a := 3;” and the symbol table before and after the *visit_Assign* method does its job:

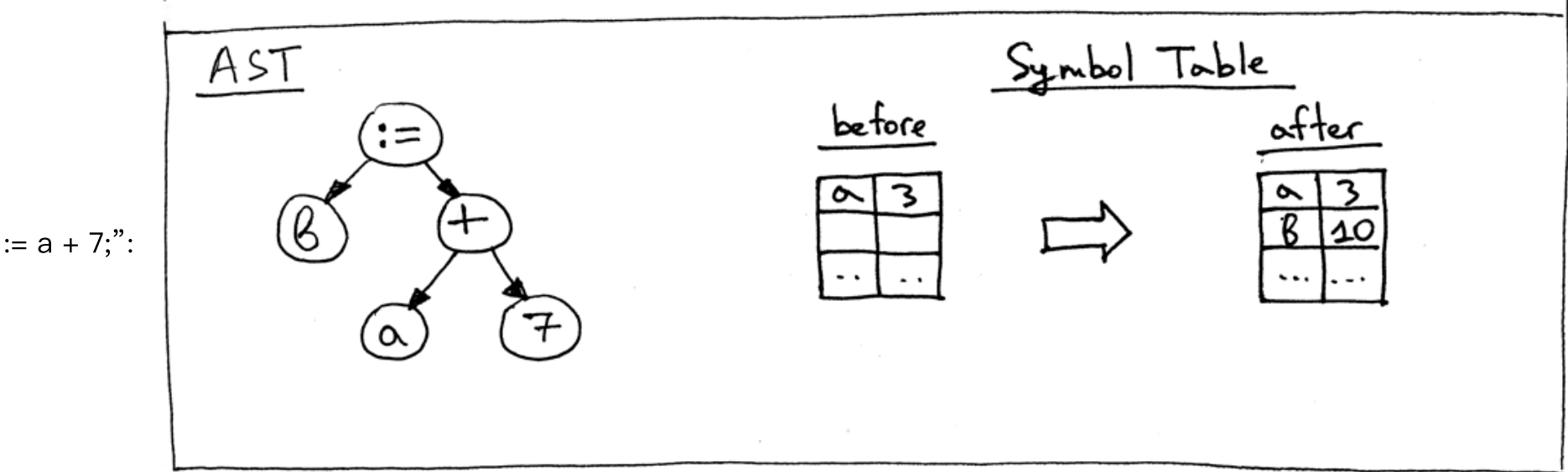


Now let's take a look at an AST for the statement “b := a + 7;”

As you can see, the right–hand side of the assignment statement – “a + 7” – references the variable ‘a’, so before we can evaluate the expression “a + 7” we need to find out what the value of ‘a’ is and that's the responsibility of the *visit_Var* method:

```
def visit_Var(self, node):
    var_name = node.value
    val = self.GLOBAL_SCOPE.get(var_name)
    if val is None:
        raise NameError(repr(var_name))
    else:
        return val
```

When the method visits a *Var* node as in the above AST picture, it first gets the variable's name and then uses that name as a key into the *GLOBAL_SCOPE* dictionary to get the variable's value. If it can find the value, it returns it, if not – it raises a *NameError* exception. Here are the contents of the symbol table before evaluating the assignment statement “b



These are all the changes that we need to do today to make our interpreter tick. At the end of the main program, we simply print the contents of the symbol table GLOBAL_SCOPE to standard output.

Let’s take our updated interpreter for a drive both from a Python interactive shell and from the command line. Make sure that you downloaded both the source code for the interpreter and the assignments.txt (<https://github.com/rspivak/lsbasi/blob/master/part9/python/assignments.txt>) file before testing:

Launch your Python shell:

```
$ python
>>> from spi import Lexer, Parser, Interpreter
>>> text = """\
... BEGIN
...
...     BEGIN
...         number := 2;
...         a := number;
...         b := 10 * a + 10 * number / 4;
...         c := a - - b
...     END;
...
...     x := 11;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> interpreter = Interpreter(parser)
>>> interpreter.interpret()
>>> print(interpreter.GLOBAL_SCOPE)
{'a': 2, 'x': 11, 'c': 27, 'b': 25, 'number': 2}
```

And from the command line, using a source file as input to our interpreter:

```
$ python spi.py assignments.txt
{'a': 2, 'x': 11, 'c': 27, 'b': 25, 'number': 2}
```

If you haven’t tried it yet, try it now and see for yourself that the interpreter is doing its job properly.

Let’s sum up what you had to do to extend the Pascal interpreter in this article:

1. Add new rules to the grammar
2. Add new tokens and supporting methods to the lexer and update the `get_next_token` method
3. Add new AST nodes to the parser for new language constructs
4. Add new methods corresponding to the new grammar rules to our recursive–descent parser and update any existing methods, if necessary (*factor* method, I’m looking at you. :)
5. Add new visitor methods to the interpreter
6. Add a dictionary for storing variables and for looking them up

In this part I had to introduce a number of “hacks” that we’ll remove as we move forward with the series:

HACKS

1

Incomplete program definition

2

Variables have no declared types

3


No type checking

4

A basic symbol table that also does double duty as a memory space

5

Using the character `'/'` for integer division



1. The *program* grammar rule is incomplete. We’ll extend it later with additional elements.

2. Pascal is a statically typed language, and you must declare a variable and its type before using it. But, as you saw, that was not the case in this article.

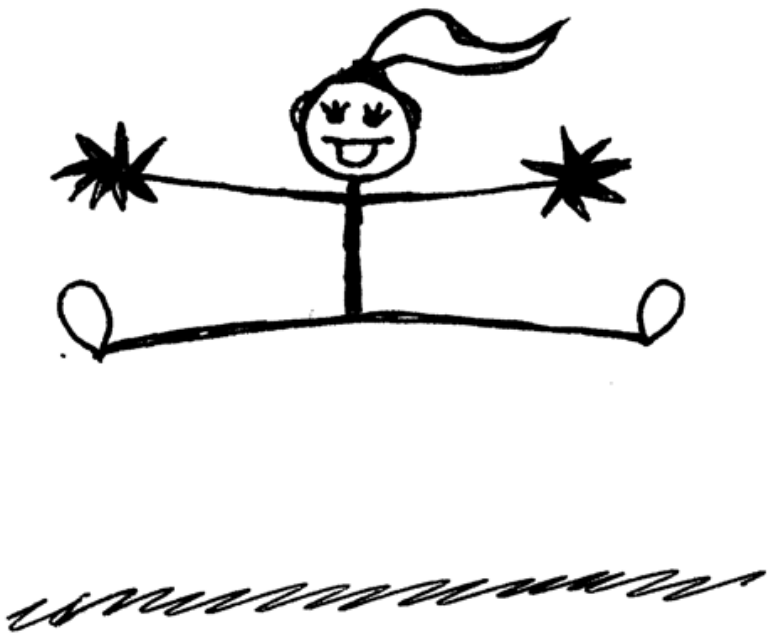
3. No type checking so far. It’s not a big deal at this point, but I just wanted to mention it explicitly. Once we add more types to our interpreter we’ll need to report an error when you try to add a string and an integer, for example.

4. A symbol table in this part is a simple Python dictionary that does double duty as a memory space. Worry not: symbol tables are such an important topic that I’ll have several articles dedicated just to them. And memory space (runtime management) is a topic of its own.

5. In our simple calculator from previous articles, we used a forward slash character `'/'` for denoting integer division. In Pascal, though, you have to use a keyword *div* to specify integer division (See Exercise 1).

6. There is also one hack that I introduced on purpose so that you could fix it in Exercise 2: in Pascal all reserved keywords and identifiers are case insensitive, but the interpreter in this article treats them as case sensitive.

To keep you fit, here are new exercises for you:



1. Pascal variables and reserved keywords are case insensitive, unlike in many other programming languages, so *BEGIN*, *begin*, and *BeGiN* they all refer to the same reserved keyword. Update the interpreter so that variables and reserved keywords are case insensitive. Use the following program to test it:

```
BEGIN

    BEGIN
        number := 2;
        a := NumBer;
        B := 10 * a + 10 * NUMBER / 4;
        c := a - - b
    end;

    x := 11;
END.
```

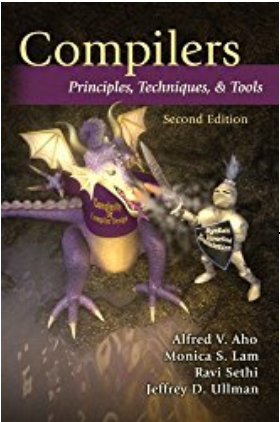
- I mentioned in the “hacks” section before that our interpreter is using the forward slash character ‘/’ to denote integer division, but instead it should be using Pascal’s reserved keyword *div* for integer division. Update the interpreter to use the *div* keyword for integer division, thus eliminating one of the hacks.
- Update the interpreter so that variables could also start with an underscore as in ‘_num := 5’.

That’s all for today. Stay tuned and see you soon.

Here is a list of books I recommend that will help you in your study of interpreters and compilers:

- Language Implementation Patterns: Create Your Own Domain–Specific and General Programming Languages (Pragmatic Programmers) (http://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=MP4DCXDV6DJMEJBL)

(http://www.amazon.com/gp/product/B00QMJQHYG/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=B00QMJQHYG&linkCode=as2&tag=russblo0b-20&linkId=I53DN2FPOSCOLBXA)
- Compilers: Principles, Techniques, and Tools (2nd Edition) (http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=GOEGDQG4HIHU56FQ)



(http://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=MD7L2CQHFXDYKOG6)

By the way, I’m writing a book “**Let’s Build A Web Server: First Steps**” that explains how to write a basic web server from scratch. You can get a feel for the book [here](https://ruslanspivak.com/lsbaws-part1/) (<https://ruslanspivak.com/lsbaws-part1/>), [here](https://ruslanspivak.com/lsbaws-part2/) (<https://ruslanspivak.com/lsbaws-part2/>), and [here](https://ruslanspivak.com/lsbaws-part3/) (<https://ruslanspivak.com/lsbaws-part3/>). Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- [Let’s Build A Simple Interpreter. Part 1. \(/lsbasi-part1/\)](#)
- [Let’s Build A Simple Interpreter. Part 2. \(/lsbasi-part2/\)](#)
- [Let’s Build A Simple Interpreter. Part 3. \(/lsbasi-part3/\)](#)
- [Let’s Build A Simple Interpreter. Part 4. \(/lsbasi-part4/\)](#)
- [Let’s Build A Simple Interpreter. Part 5. \(/lsbasi-part5/\)](#)

- Let's Build A Simple Interpreter. Part 6. (/lsbasi-part6/)
- Let's Build A Simple Interpreter. Part 7. (/lsbasi-part7/)
- Let's Build A Simple Interpreter. Part 8. (/lsbasi-part8/)
- Let's Build A Simple Interpreter. Part 9. (/lsbasi-part9/)
- Let's Build A Simple Interpreter. Part 10. (/lsbasi-part10/)
- Let's Build A Simple Interpreter. Part 11. (/lsbasi-part11/)
- Let's Build A Simple Interpreter. Part 12. (/lsbasi-part12/)
- Let's Build A Simple Interpreter. Part 13. (/lsbasi-part13/)
- Let's Build A Simple Interpreter. Part 14. (/lsbasi-part14/)

Comments

35 Comments

Ruslan's Blog

1 Login

Recommend 6

Share

Sort by Best

Join the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS ?



Name

Diego Marcia · 2 years ago

WHOA.
Took me one week, but I menaged to reproduce all 9 parts... And now I can't help but beg you:
Part ten, part ten, part ten, part ten, please, please, please, please, please, please PLEEEEEEEEEAAAAASEEEEEEE

1 ^ | v · Reply · Share ›

rspivak **Mod** ➔ Diego Marcia · 2 years ago

Way to go, Diego. Thanks for reading and actually implementing the interpreter! You're super fast. :) Part 10 is already in the making, should be out in the next two weeks.

4 ^ | v · Reply · Share ›

Andy · 2 years ago

I love this series, thank you so much!

1 ^ | v · Reply · Share ›

rspivak **Mod** ➔ Andy · 2 years ago

Thank you!

^ | v · Reply · Share ›

Jonathan · 2 years ago

YEEEEAAAAaaaah, finally! <3 Thank you

1 ^ | v · Reply · Share ›

rspivak **Mod** ➔ Jonathan · 2 years ago

You're welcome. :)

^ | v · Reply · Share ›

lonelywaiting · 5 days ago

thanks very much~

^ | v · Reply · Share ›

Christian Alexander (TheNewCom) · 9 months ago

In python 3.6, pylint is telling me that the Interpreter class has no member called GLOBAL_SCOPE.How do you get this?

^ | v · Reply · Share ›

evanxg852000 · a year ago

I wish i had this wonderful resource when I started learning about compilers. Still this is really a nice and important article series. I really like the way this is structured. at the end of each part you have something working. this helps keep motivation. I believe this is the best part of this series. Thanks for this series. Any plan on OS dev or Database engines implementation ?

^ | v · Reply · Share ›

Raju Choudhari · a year ago

Thanks. Very useful.

^ | v · Reply · Share ›

ER · 2 years ago

I would love you as my teacher

^ | v · Reply · Share ›

Humoyun Ahmedov · 2 years ago

Awesome, maybe more than awesome :)

^ | v · Reply · Share ›

Hilde · 2 years ago

Thank you so much for these series! You made it very clear and understandable for me. I'm enjoying programming as a hobby and also a small part of my scientific work, but I have no formal education in it. I managed to do your tasks, and found out the recursive implementation of the nested expression before you explained it. I was so proud for trying ;)

from, I am going to interpret not a programming language, but something that might resemble one. The one in that it has tags with names and various values. Your tutorial did not get me there, but you gave me a solid foundation to work on. Thank you!

^ | v

• Reply • Share ›



Sniper435 • 2 years ago

Finding the series really useful and informative, keep going over parts to try and make sure I fully understand them.

could I ask how you're doing with the next part and also whether you'll touch on how to implement an interpreter in a language that uses strict data types as python seems to make this "easy" for you.

^ | v

• Reply • Share ›



Sniper435 ➔ Sniper435 • 2 years ago

FYI, as I've been going through I've been implementing every step of the series in another language to solidify my understanding, one part that really interests me how how to implement functions - I hope this comes up soon.

^ | v

• Reply • Share ›



rspivak **Mod** ➔ Sniper435 • 2 years ago

Thanks for reading and yes, procedures and functions are coming soon.

^ | v

• Reply • Share ›



Matthew Tolman • 2 years ago

Thanks for this. Looking forward to the rest of the series! Don't stop now!

^ | v

• Reply • Share ›



Daniel Kurashige-Gollub • 2 years ago

Thanks for this blog post series about Interpreter/Compiler implementation. I am enjoying it a great deal and am following it along in the D programming language, implementing a very rudimentary BASIC dialect (so no Pascal for me; reason for this is, I want to implement something simpler than Pascal that I can use to teach my daughters programming when they are a bit older). Please continue your series, as it is pretty entertaining and very informative. Thank you.

^ | v

• Reply • Share ›



gabodev • 2 years ago

It is helping to improve the implementation of the interpreter of my project :)

<https://github.com/centauri...>

Thank you so much!

^ | v

• Reply • Share ›



Venkatesh Pitta • 2 years ago

Excellent work and writing. Thank you. I just finished one reading of all the 9 articles in two evenings. Starting to reread. Repetition :). One thing that stands out to me is, why use type for a field name? Because type(node) of Python is used as well. Perhaps a bit verbose, and a better descriptive name token_type could be used. I will send you a PR on GitHub. Cheers!

^ | v

• Reply • Share ›



Mars Cheng • 2 years ago

I'm studying interpreter recently and really like this series. Would you mind I translate your articles to Traditional Chinese? It would help others a lot. :-)

^ | v

• Reply • Share ›



rspivak **Mod** ➔ Mars Cheng • 2 years ago

Go ahead and good luck! :)

^ | v

• Reply • Share ›



Connor Stack • 2 years ago

Just finished part 9! As an extra challenge I also made error messages include line numbers. Can't wait for part 10 :D

^ | v

• Reply • Share ›



Dimas Salazar • 2 years ago

Excelent articles Ruslan. It helps me a lot to understand some things about interpreters and to learn some concepts for some things I do instinctively. I can't wait for part 10 and beyond.

How many parts are you planning for this serie.

I will implement this in other languages to really learn it all.

Thanks!!!

(Pascal was my first programming language)

^ | v

• Reply • Share ›



Ike S. Ma • 2 years ago

Thanks soooo much! These tutorials are so awesome! They are way too addictive and I end up finishing the entire series (1-9) in two days! I've been implementing along the series and I can hardly feel my wrists now....

I audited the compiler course on coursera last summer for fun (but did not attempt any of the program assignments). I ended up just knowing lots of terms and sort of lost in the theory... By following along the tutorials so far, I have better grasp and understanding the what and why of the theory. So many things I thought I knew before had been relearned!

It's also mind-blowing to see all the stuff I learnt at various places come together here! Visitor pattern from design pattern, tree traversal from data structure, object-oriented design from a C++ course, and context free grammar from automaton course etc. Just WOW! Lots of things I didn't quite understand before make so much sense now!

I love how the theories have been brought into practice with beautiful python code! I am fairly new to python, and I have picked quite a few python hacks and implementation tricks from these tutorials. Thanks so much for your time writing up these awesome tutorials!

How many more tutorials are we gonna have for the interpreter topic? Any spoilers what's gonna come up next? Are we gonna touch on implementations such as control flow, functions, pointers, class, garbage collection, and so forth? Anyways, I can't wait to see whatever comes up next !!!

^ | v

• Reply • Share ›



Alexander Zhuravlev • 2 years ago

To be more honest, we need to add a little bit to our grammar, here it is:


```
program : compound_statement DOT EOF
```

I am looking forward to the next part.

^ | v · Reply · Share ›



tuvi · 2 years ago

it is an awesome article...Thank you so much..

^ | v · Reply · Share ›



Alexander Zhuravlev · 2 years ago

Thanks for the series, Ruslan. By following it I write my own implementation of the interpreter in Java. And I have a question, why do we need this 'if' statement in statement_list?

```
if self.current_token.type == ID:
    self.error()
```

It seems redundant to me, because we already have eat(END) on the above level.

^ | v · Reply · Share ›



rspivak **Mod** ➔ Alexander Zhuravlev · 2 years ago

Thanks, Alexander. You're right, it is redundant in this case. I'll update the code to reflect that.

^ | v · Reply · Share ›



thulani mtetwa ➔ Alexander Zhuravlev · 2 years ago

Do you mind sharing the code. I am finding it hard to follow since I do not really understand Python

^ | v · Reply · Share ›



Alexander Zhuravlev ➔ thulani mtetwa · 2 years ago

You are welcome!

<https://github.com/zelark/spi>

^ | v · Reply · Share ›



thulani mtetwa ➔ Alexander Zhuravlev · 2 years ago

Thanks Alexandr. You just made a ton of things easier

^ | v · Reply · Share ›



Michael Karotsieris · 2 years ago

Awesome material ! Thank you! :D

^ | v · Reply · Share ›



PubSubPenguin · 2 years ago

I'm trying to follow this guide in rust. And up untill now I could always look to your github if I got stuck. But you've stoped updating the rust example code and I've gotten stuck with the symbol table.

Would you consider updating it again? I would appreciate it a lot if you would! Thank you for the helpful guide.

^ | v · Reply · Share ›



rspivak **Mod** ➔ PubSubPenguin · 2 years ago

You're welcome and thank you for reading. I can't promise you the date, but I will add Rust code eventually.

^ | v · Reply · Share ›

ALSO ON RUSLAN'S BLOG

Let's Build A Web Server. Part 1.

78 comments · 3 years ago

ChinaMoe — how a great post!

Let's Build A Simple Interpreter. Part 4.

8 comments · 3 years ago

Daniel Coronado — Good!!

Let's Build A Simple Interpreter. Part 6.

12 comments · 3 years ago

Ahmed Hamdan — I already made my own version in java and it's working

Let's Build A Simple Interpreter. Part 1.

43 comments · 3 years ago

Jack Crenshaw — I enjoyed reading your tutorial, but I do sorta wish you'd referenced my own tutorial, "Let's Build a Compiler,"<http://compilers.iecc.com/c...>which I started ...

🏠 Social

github (<https://github.com/rspivak/>)

twitter (<https://twitter.com/alienoid>)

linkedin (<https://linkedin.com/in/ruslanspivak/>)