# Let’s Build A Simple Interpreter. Part 9. (https://ruslanspivak.com/lsbasi-part9/)

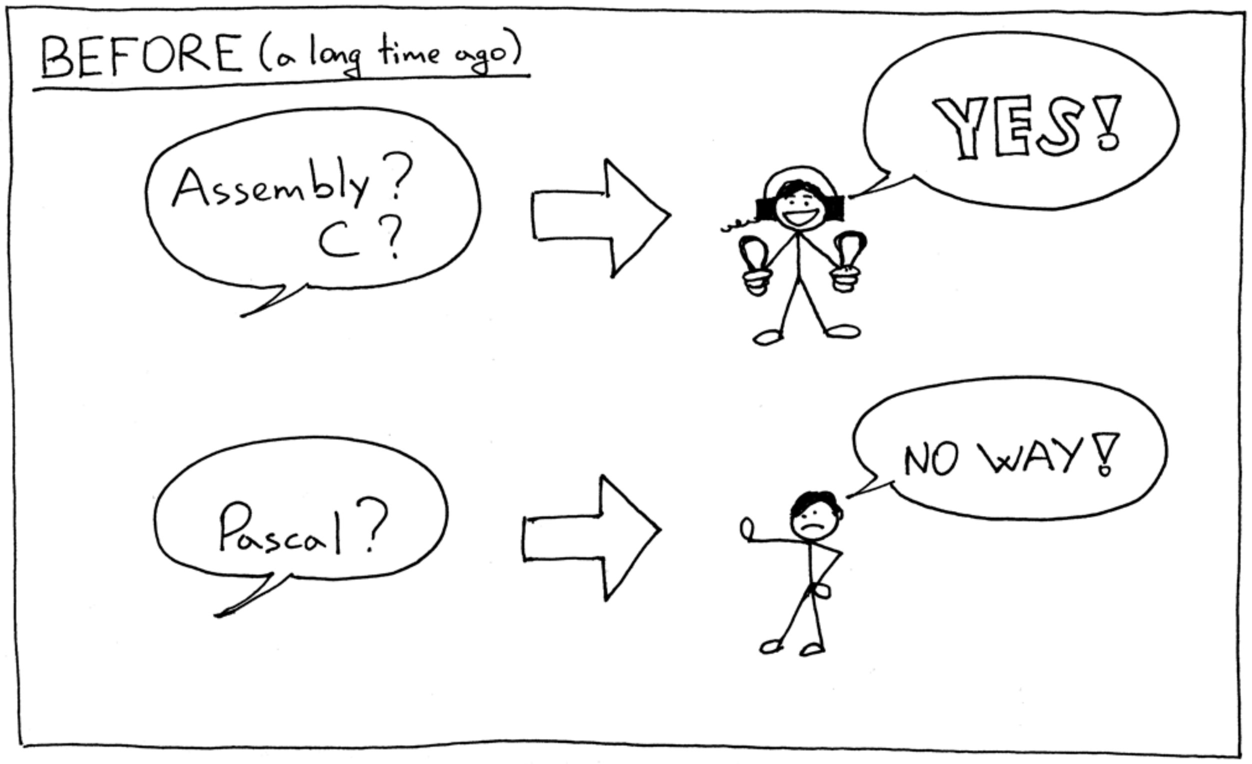
Date Sun, May 01, 2016

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.

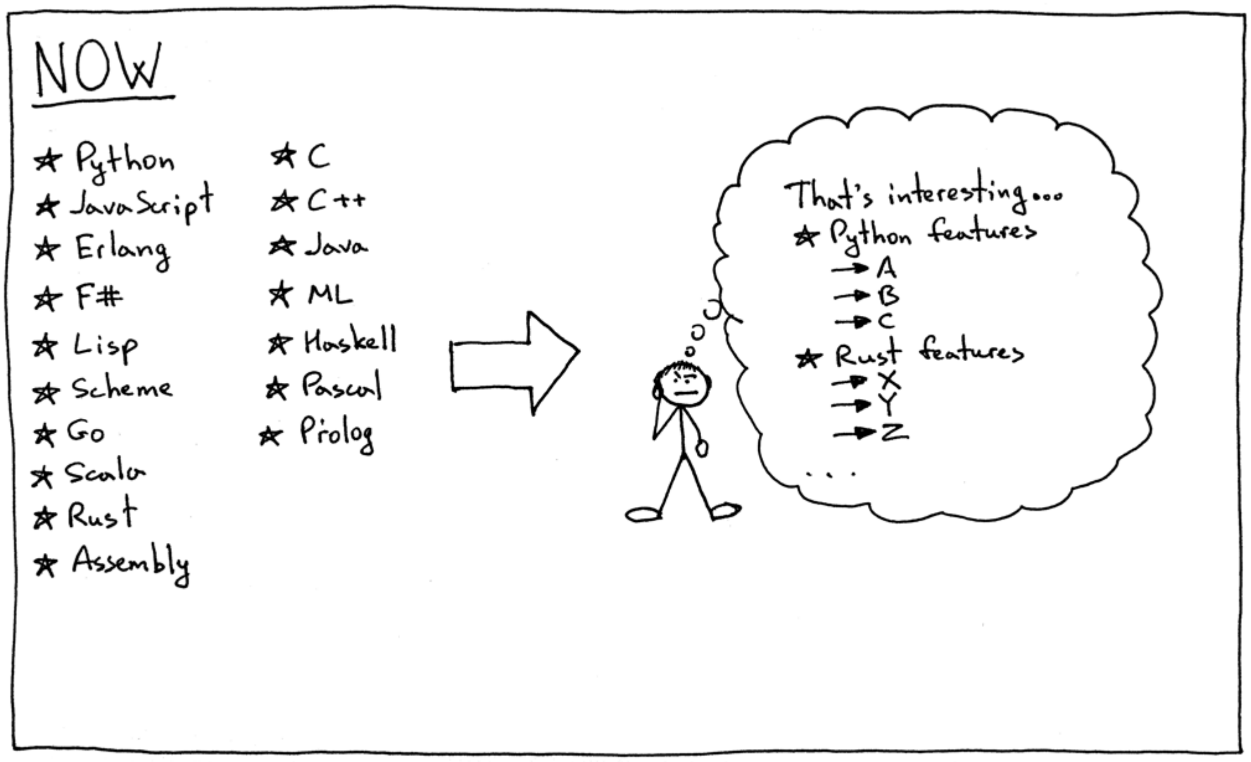
Little did I know back then that I would be writing almost everything in Python (and love every bit of it) to pay my bills and that I would also be writing an interpreter and compiler for Pascal for the reasons I stated in the very first article of the series (/lsbasi-part1/).

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.

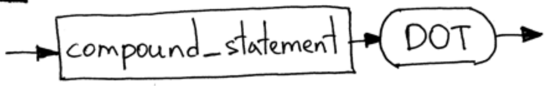
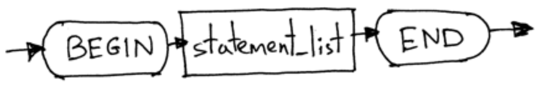
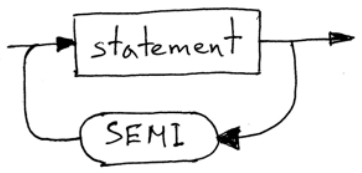
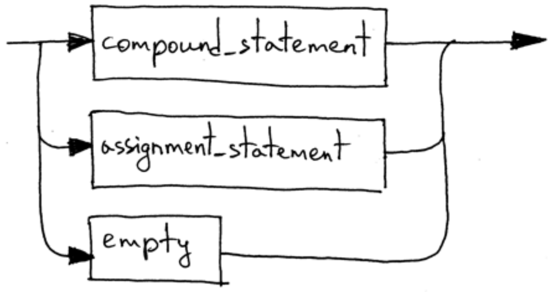
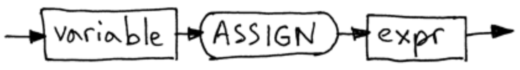
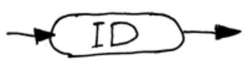
I'll use the following sample Pascal-like program to introduce new concepts:

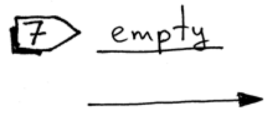
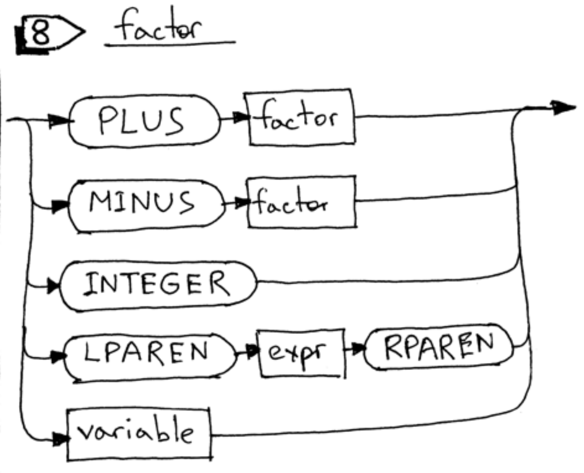
```
BEGIN
    BEGIN
        number := 2;
        a := number;
        b := 10 * a + 10 * number / 4;
        c := a - - b
    END;
    x := 11;
END.
```

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 |
|---|---|
| 1 program  | program : compound_statement DOT |
| 2 compound_statement  | compound_statement : BEGIN statement_list END |
| 3 statement_list  | statement_list : statement<br>\| statement SEMI statement_list |
| 4 statement  | statement : compound_statement<br>\| assignment_statement<br>\| empty |
| 5 assignment_statement  | assignment_statement : variable ASSIGN expr |
| 6 variable  | variable : ID |

| | |
|---|---|
|  | empty : |
|  8<br>factor<br>PLUS<br>factor<br>MINUS<br>factor<br>INTEGER<br>LPAREN<br>expr<br>RPAREN<br>variable | factor : PLUS factor<br>\| MINUS factor<br>\| INTEGER<br>\| LPAREN expr RPAREN<br>\| 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.

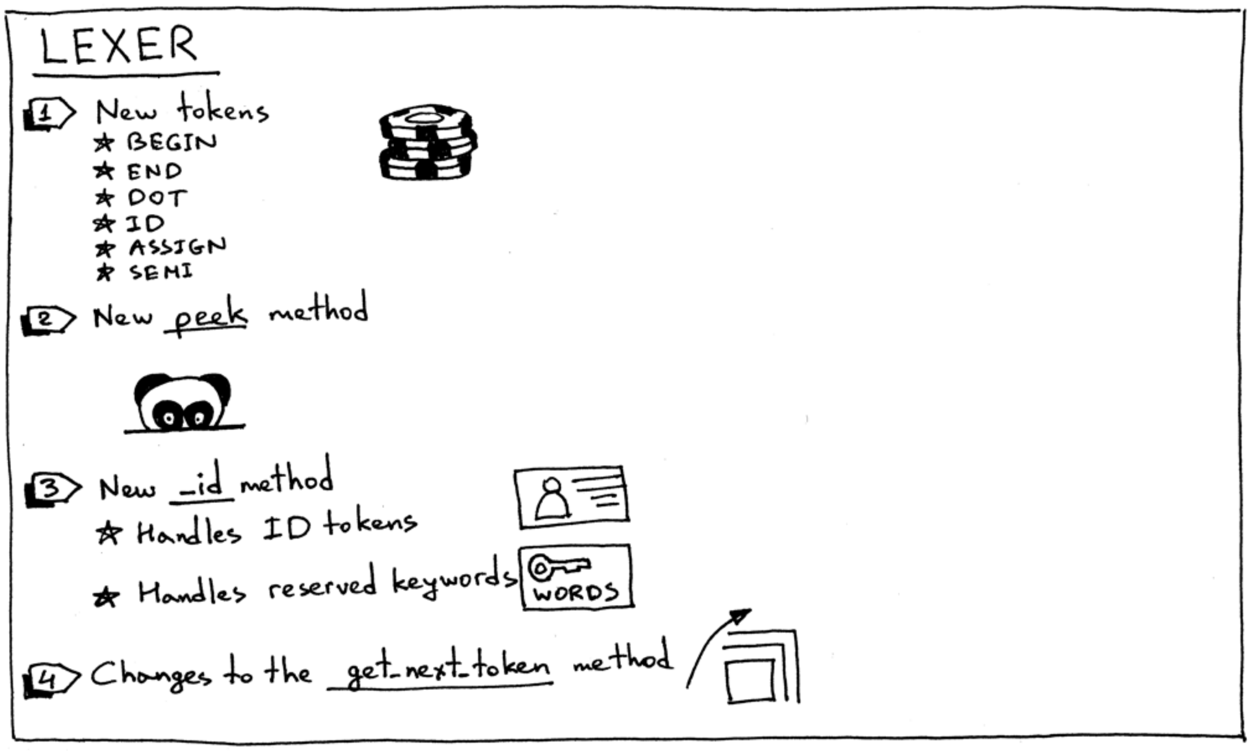
Now let's take a look at our complete grammar:

```
    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:



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:

```python
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:

```python
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*:

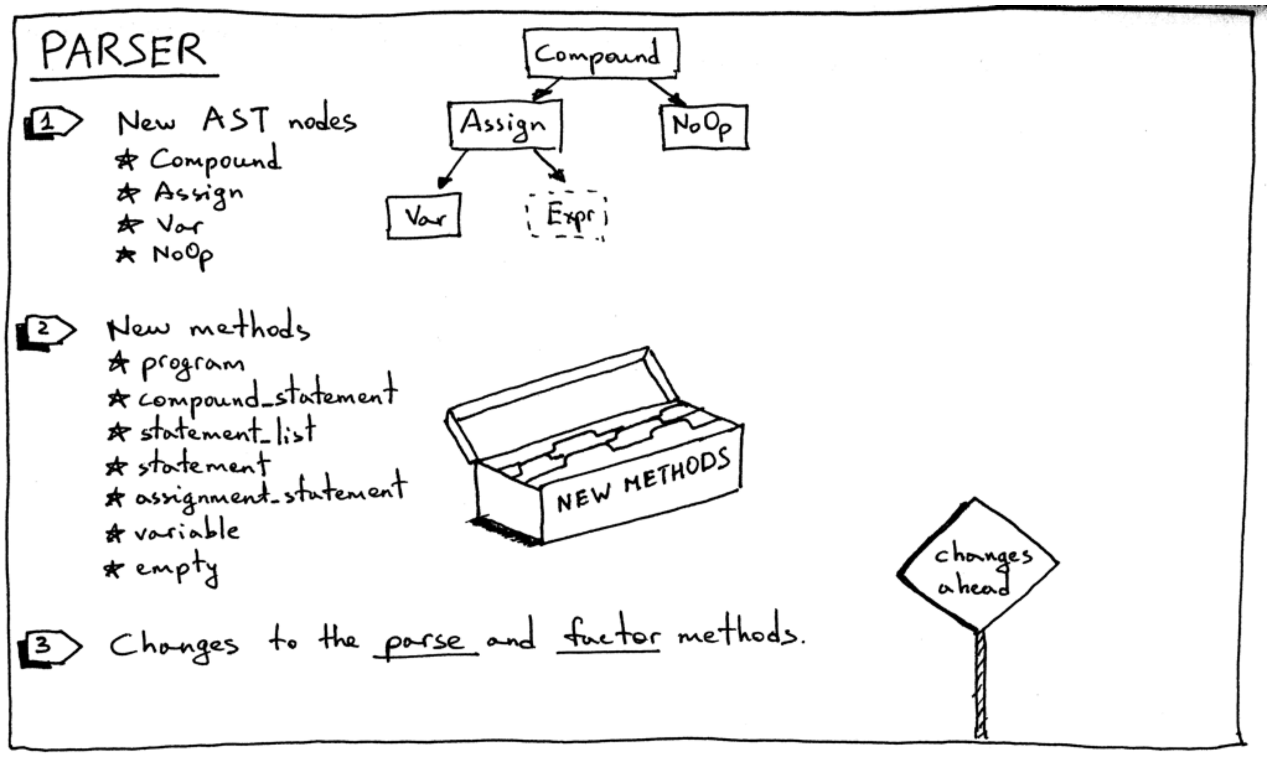
```python
def get_next_token(self):
    while self.current_char is not None:
        ...
        if self.current_char.isalpha():
            return self._id()

        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:



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:

```
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:

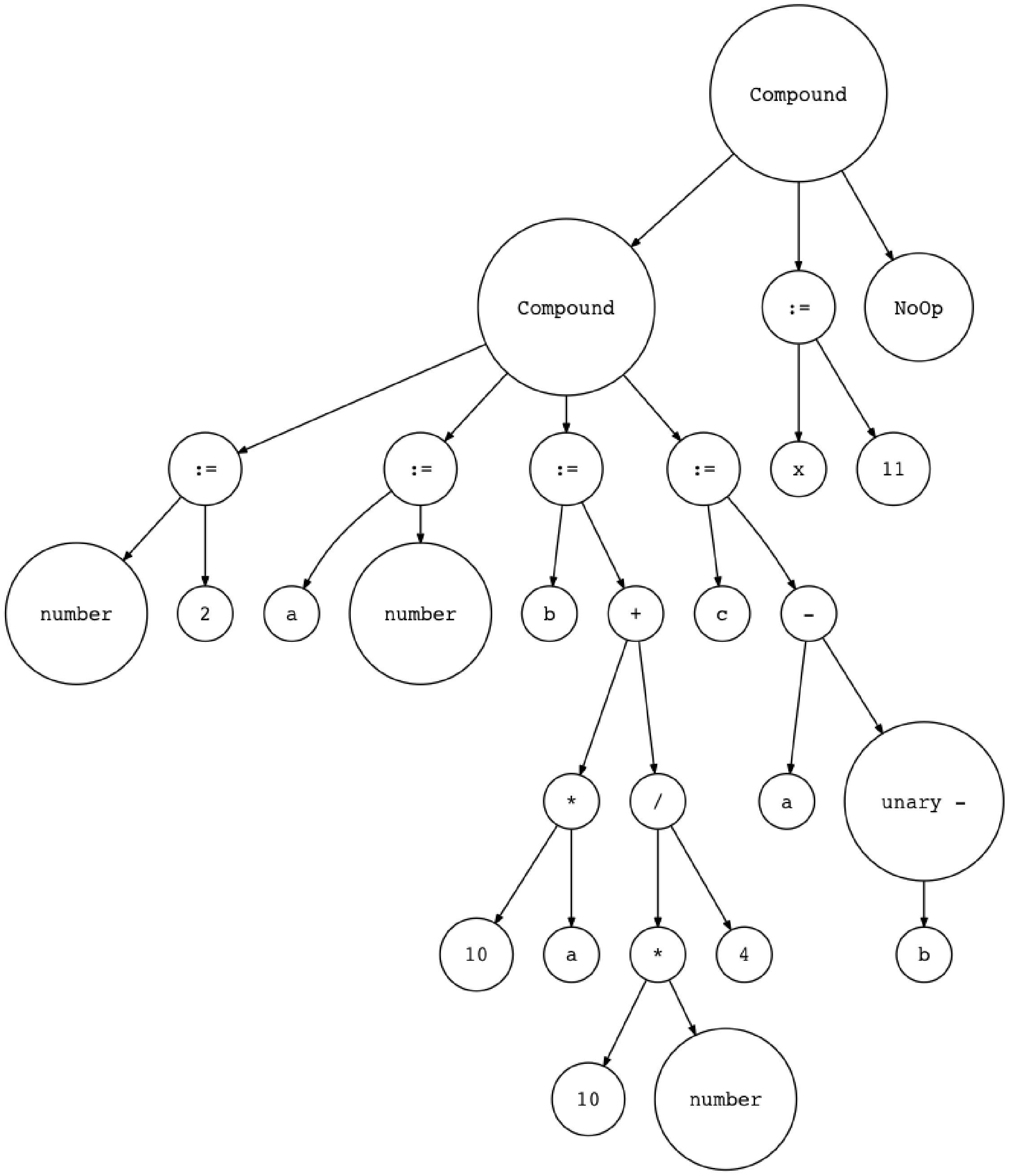
```
def parse(self):
    node = self.program()
    if self.current_token.type != EOF:
        self.error()

    return node
```

Here is our sample program again:

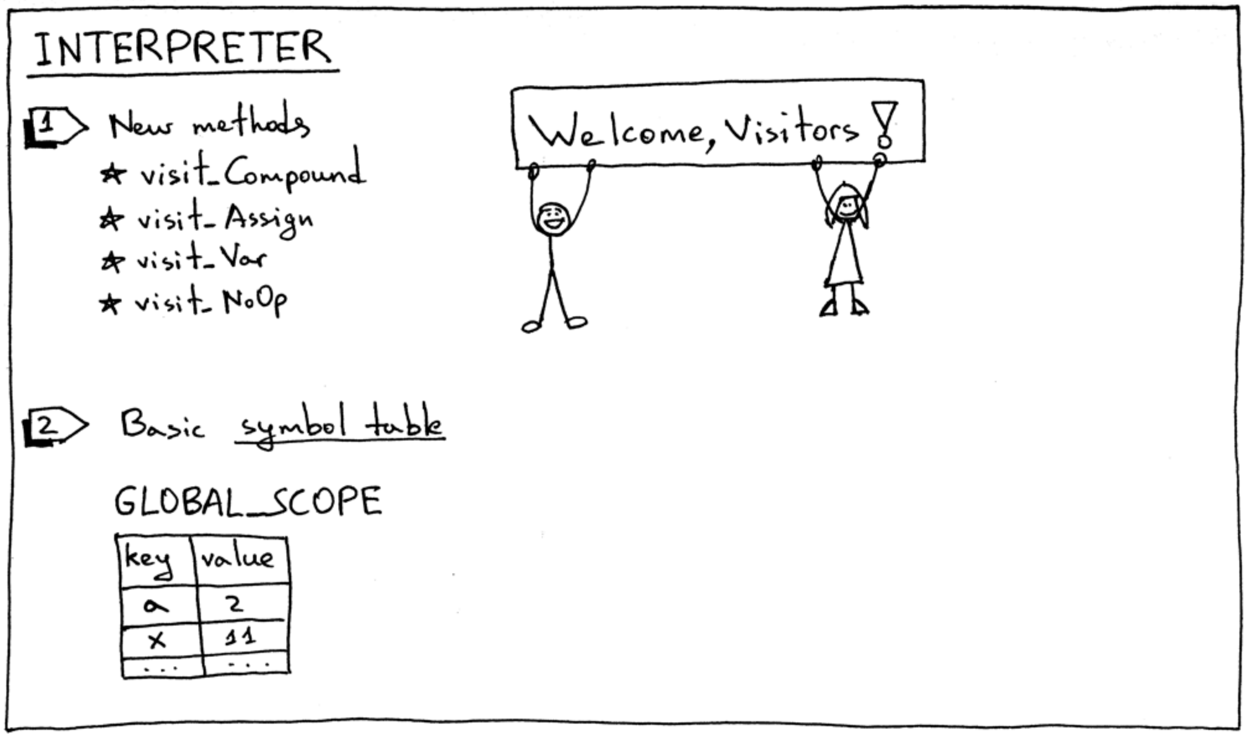
```
BEGIN
    BEGIN
        number := 2;
        a := number;
        b := 10 * a + 10 * number / 4;
        c := a - - b
    END;
    x := 11;
END.
```

Let’s visualize it with genastdot.py (https://github.com/rspivak/lsbasi/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
```


Compound
Compound
:=
NoOp
:=
:=
:=
:=
x
11
number
2
a
number
b
+
c
-
*
/
a
unary -
10
a
*
4
b
10
number

And finally, here are the required interpreter changes:



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

*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
```

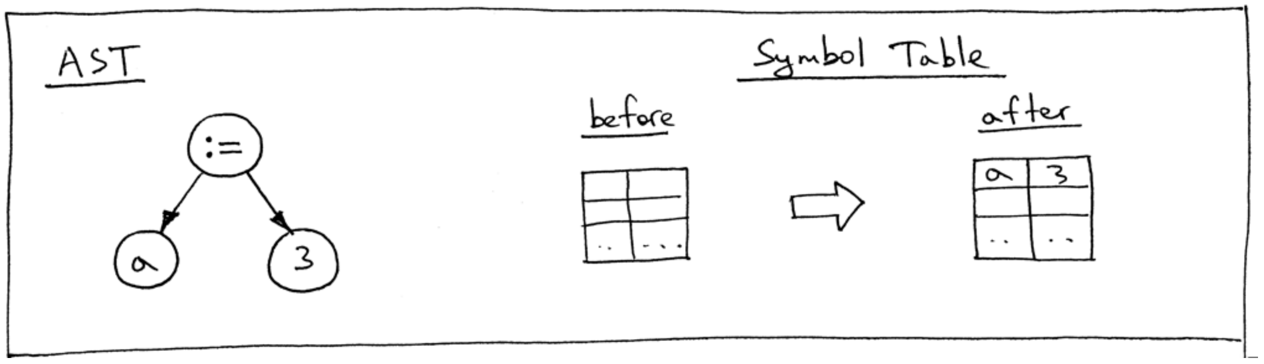
The *Assign* and *Var* visitor methods deserve a closer examination.

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:

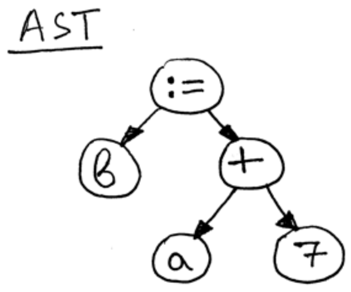
```
def visit_Assign(self, node):
    var_name = node.left.value
    self.GLOBAL_SCOPE[var_name] = self.visit(node.right)
```

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:



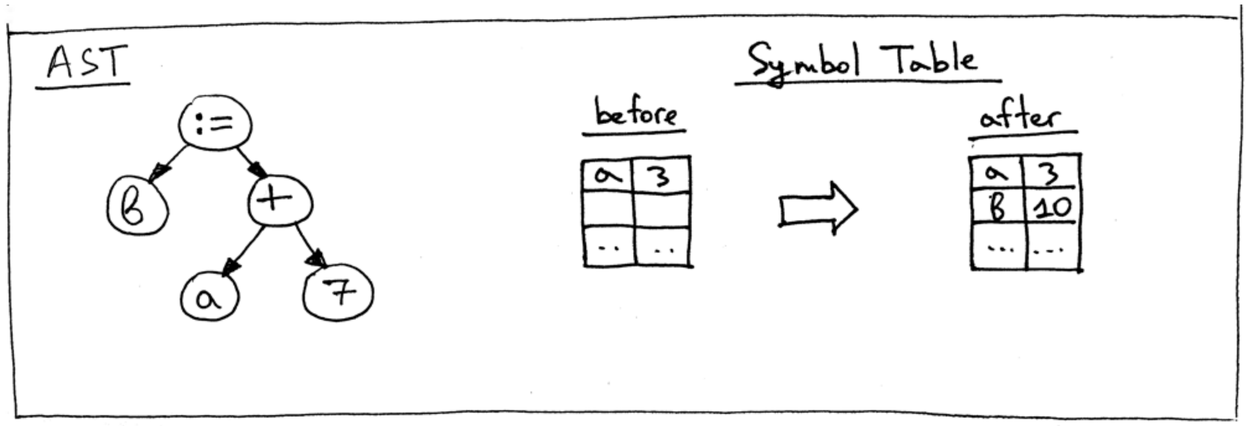
Now let's take a look at an AST for the statement "b := a + 7;"



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 := a + 7;”:



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:

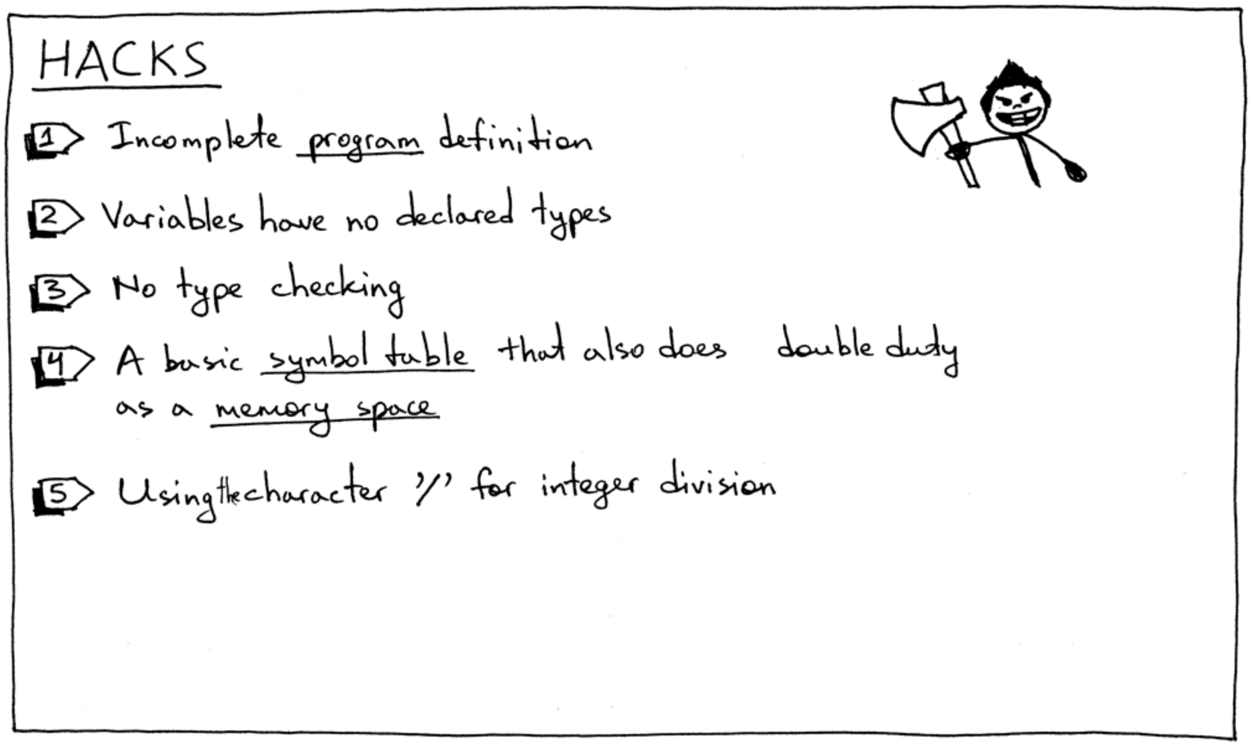
```
$ python spi.py assignments.txt
{'a': 2, 'x': 11, 'c': 27, 'b': 25, 'number': 2}
```

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.
```

2. 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.
3. 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:

1. 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)
2. 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)

**All articles in this series:**

## Comments