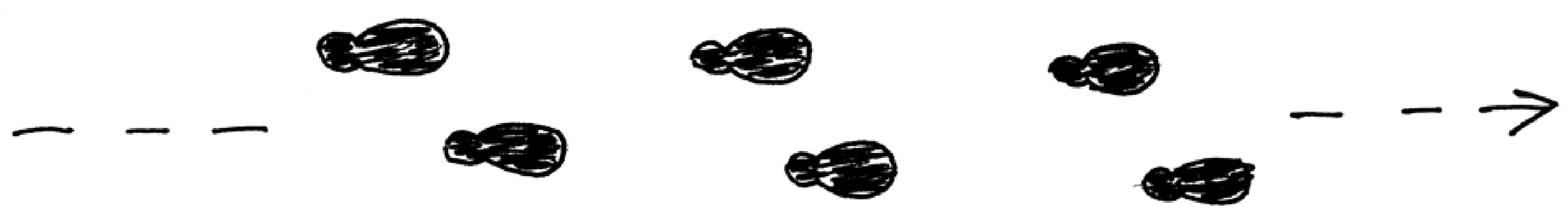
Let’s Build A Simple Interpreter. Part 12. (https://ruslanspivak.com/lsbasi-part12/)

Date Thu, December 01, 2016

“Be not afraid of going slowly; be afraid only of standing still.” – Chinese proverb.

Hello, and welcome back!

Today we are going to take a few more baby steps and learn how to parse Pascal procedure declarations.



What is a ***procedure declaration***? A ***procedure declaration*** is a language construct that defines an identifier (a procedure name) and associates it with a block of Pascal code.

Before we dive in, a few words about Pascal procedures and their declarations:

- Pascal procedures don’t have return statements. They exit when they reach the end of their corresponding block.
- Pascal procedures can be nested within each other.
- For simplicity reasons, procedure declarations in this article won’t have any formal parameters. But, don’t worry, we’ll cover that later in the series.

This is our test program for today:

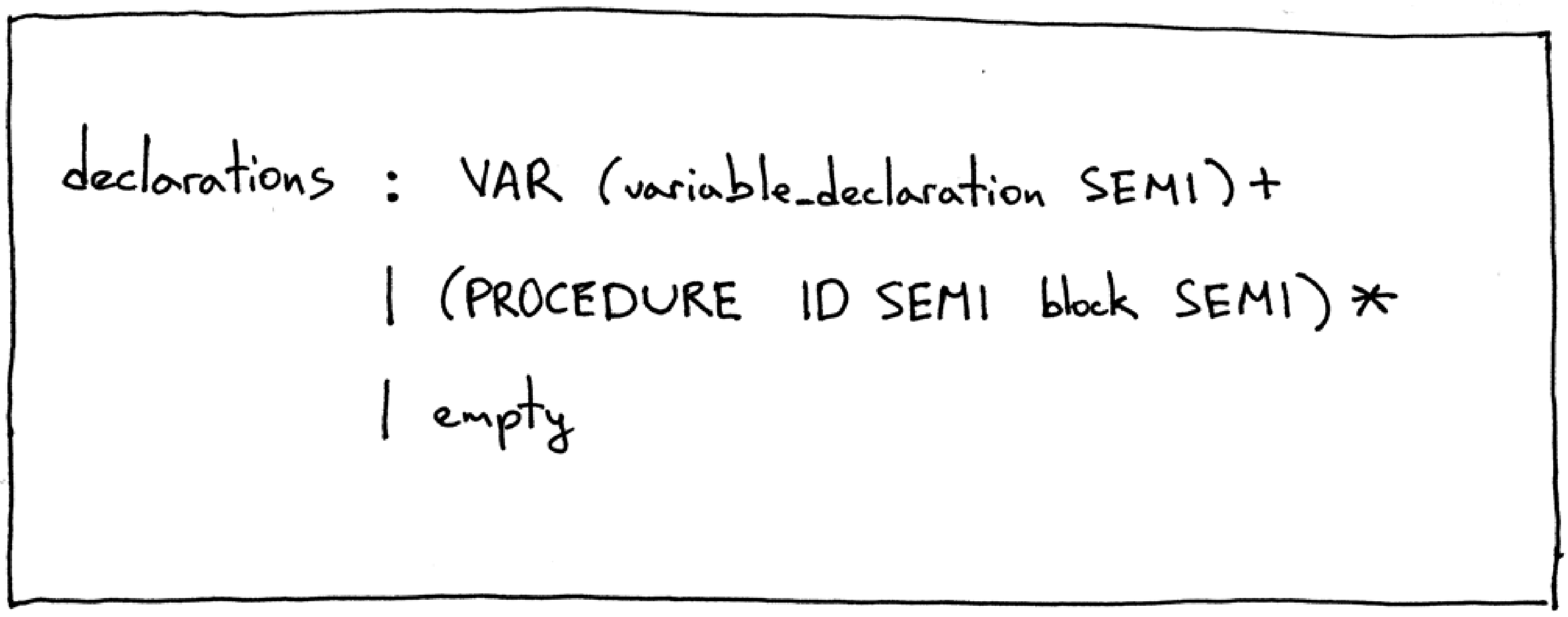
```
PROGRAM Part12;
VAR
   a : INTEGER;

PROCEDURE P1;
VAR
   a : REAL;
   k : INTEGER;

   PROCEDURE P2;
   VAR
      a, z : INTEGER;
   BEGIN {P2}
      z := 777;
   END;  {P2}

BEGIN {P1}

END;  {P1}

BEGIN {Part12}
   a := 10;
END.  {Part12}
```

As you can see above, we have defined two procedures (*P1* and *P2*) and *P2* is nested within *P1*. In the code above, I used comments with a procedure’s name to clearly indicate where the body of every procedure begins and where it ends.

Our objective for today is pretty clear: learn how to parse a code like that.

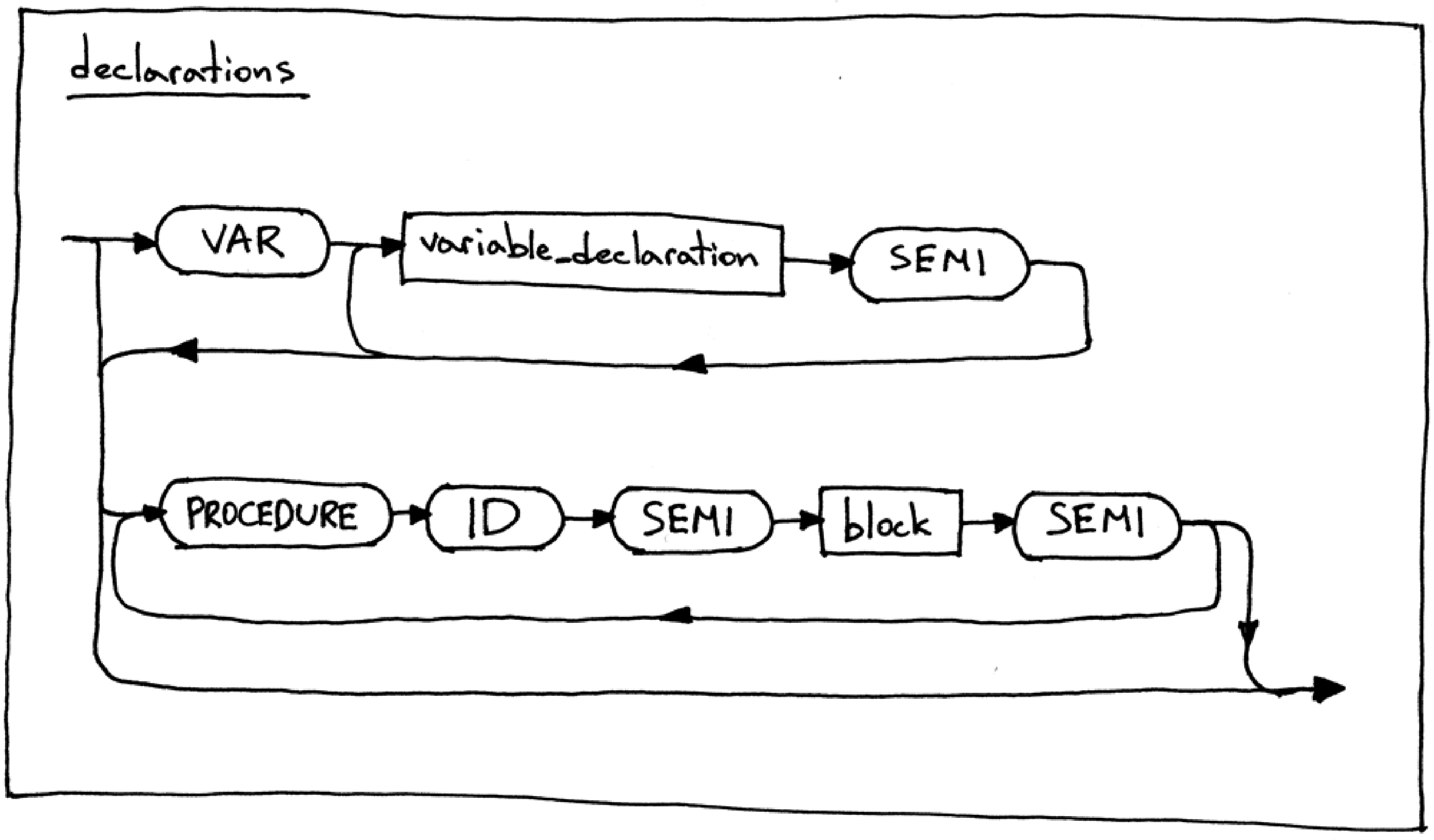
First, we need to make some changes to our grammar to add procedure declarations. Well, let’s just do that!

Here is the updated *declarations* grammar rule:


declarations : VAR (variable_declaration SEMI)+
| (PROCEDURE ID SEMI block SEMI)*
| empty

The procedure declaration sub-rule consists of the reserved keyword **PROCEDURE** followed by an identifier (a procedure name), followed by a semicolon, which in turn is followed by a *block* rule, which is terminated by a semicolon. Whoa! This is a case where I think the picture is actually worth however many words I just put in the previous sentence! :)

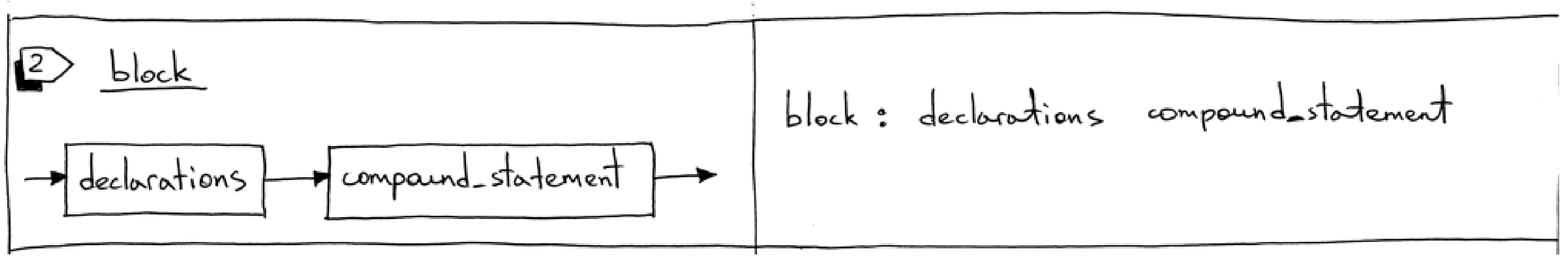
Here is the updated syntax diagram for the *declarations* rule:



From the grammar and the diagram above you can see that you can have as many procedure declarations on the same level as you want. For example, in the code snippet below we define two procedure declarations, *P1* and *P1A*, on the same level:

```
PROGRAM Test;
VAR
   a : INTEGER;

PROCEDURE P1;
BEGIN {P1}

END;  {P1}

PROCEDURE P1A;
BEGIN {P1A}

END;  {P1A}

BEGIN {Test}
   a := 10;
END.  {Test}
```

The diagram and the grammar rule above also indicate that procedure declarations can be nested because the *procedure declaration* sub-rule references the *block* rule which contains the *declarations* rule, which in turn contains the *procedure declaration* sub-rule. As a reminder, here is the syntax diagram and the grammar for the block rule from Part10 (/lsbasi-part10/):



Okay, now let's focus on the interpreter components that need to be updated to support procedure declarations:

Updating the Lexer

All we need to do is add a new token named **PROCEDURE**:

```
PROCEDURE = 'PROCEDURE'
```

And add *'PROCEDURE'* to the reserved keywords. Here is the complete mapping of reserved keywords to tokens:

```
RESERVED_KEYWORDS = {
    'PROGRAM': Token('PROGRAM', 'PROGRAM'),
    'VAR': Token('VAR', 'VAR'),
    'DIV': Token('INTEGER_DIV', 'DIV'),
    'INTEGER': Token('INTEGER', 'INTEGER'),
    'REAL': Token('REAL', 'REAL'),
    'BEGIN': Token('BEGIN', 'BEGIN'),
    'END': Token('END', 'END'),
    'PROCEDURE': Token('PROCEDURE', 'PROCEDURE'),
}
```

Updating the Parser

Here is a summary of the parser changes:

1. New *ProcedureDecl* AST node
2. Update to the parser's *declarations* method to support procedure declarations

Let's go over the changes.

1. The *ProcedureDecl* AST node represents a *procedure declaration*. The class constructor takes as parameters the name of the procedure and the AST node of the block of code that the procedure's name refers to.

```
class ProcedureDecl(AST):
    def __init__(self, proc_name, block_node):
        self.proc_name = proc_name
        self.block_node = block_node
```

2. Here is the updated *declarations* method of the *Parser* class

```
def declarations(self):
    """declarations : VAR (variable_declaration SEMI)+
                    | (PROCEDURE ID SEMI block SEMI)*
                    | empty
    """
    declarations = []

    if self.current_token.type == VAR:
        self.eat(VAR)
        while self.current_token.type == ID:
            var_decl = self.variable_declaration()
            declarations.extend(var_decl)
            self.eat(SEMI)

    while self.current_token.type == PROCEDURE:
        self.eat(PROCEDURE)
        proc_name = self.current_token.value
        self.eat(ID)
        self.eat(SEMI)
        block_node = self.block()
        proc_decl = ProcedureDecl(proc_name, block_node)
        declarations.append(proc_decl)
        self.eat(SEMI)

    return declarations
```

Hopefully, the code above is pretty self-explanatory. It follows the grammar/syntax diagram for procedure declarations that you've seen earlier in the article.

Updating the SymbolTable builder

Because we’re not ready yet to handle nested procedure scopes, we’ll simply add an empty *visit_ProcedureDecl* method to the *SymbolTreeBuilder* AST visitor class. We’ll fill it out in the next article.

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

Updating the Interpreter

We also need to add an empty *visit_ProcedureDecl* method to the *Interpreter* class, which will cause our interpreter to silently ignore all our procedure declarations.

So far, so good.

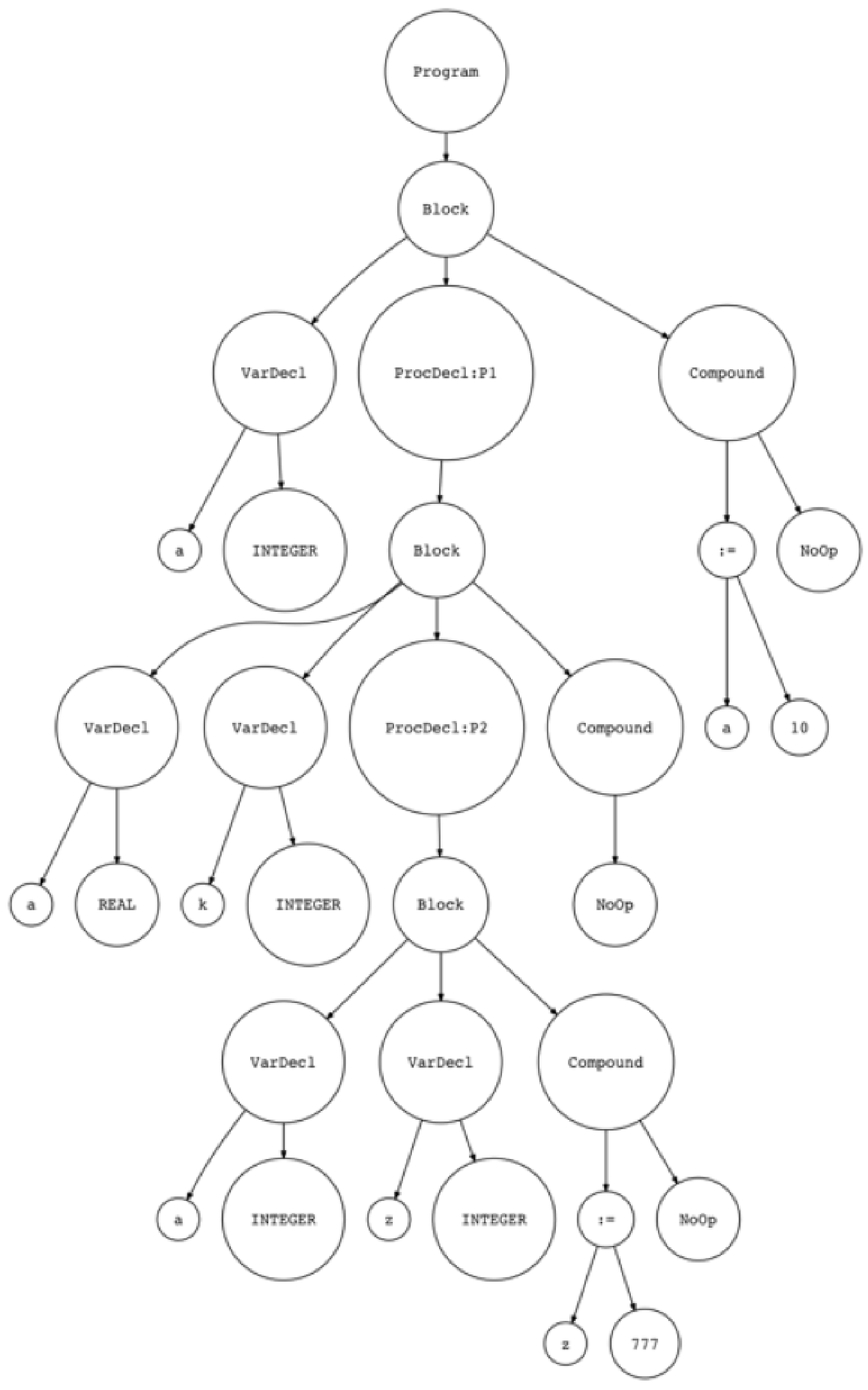
Now that we’ve made all the necessary changes, let’s see what the *Abstract Syntax Tree* looks like with the new *ProcedureDecl* nodes.

Here is our Pascal program again (you can download it directly from GitHub (https://github.com/rspivak/lsbasi/blob/master/part12/python/part12.pas)):

```
PROGRAM Part12;
VAR
   a : INTEGER;

PROCEDURE P1;
VAR
   a : REAL;
   k : INTEGER;

   PROCEDURE P2;
   VAR
      a, z : INTEGER;
   BEGIN {P2}
      z := 777;
   END;  {P2}

BEGIN {P1}

END;  {P1}

BEGIN {Part12}
   a := 10;
END.  {Part12}
```

Let’s generate an AST and visualize it with the genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part12/python/genastdot.py) utility:

```
$ python genastdot.py part12.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



In the picture above you can see two *ProcedureDecl* nodes: *ProcDecl:P1* and *ProcDecl:P2* that correspond to procedures *P1* and *P2*. Mission accomplished. :)

As a last item for today, let's quickly check that our updated interpreter works as before when a Pascal program has procedure declarations in it. Download the interpreter (https://github.com/rspivak/lsbasi/blob/master/part12/python/spi.py) and the test program (https://github.com/rspivak/lsbasi/blob/master/part12/python/part12.pas) if you haven't done so yet, and run it on the command line. Your output should look similar to this:

```
$ python spi.py part12.pas
Define: INTEGER
Define: REAL
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: a

Symbol Table contents:
Symbols: [INTEGER, REAL, <a:INTEGER>]

Run-time GLOBAL_MEMORY contents:
a = 10
```

Okay, with all that knowledge and experience under our belt, we're ready to tackle the topic of nested scopes that we need to understand in order to be able to analyze nested procedures and prepare ourselves to handle procedure and function calls. And that's exactly what we are going to do in the next article: dive deep into nested scopes. So don't forget to bring your swimming gear next time! Stay tuned and see you soon!

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

comments powered by Disqus (http://disqus.com)

Social

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

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

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

Popular posts

Let's Build A Web Server. Part 1. (https://ruslanspivak.com/lsbaws-part1/)

Let's Build A Simple Interpreter. Part 1. (https://ruslanspivak.com/lsbasi-part1/)

Let's Build A Web Server. Part 2. (https://ruslanspivak.com/lsbaws-part2/)

Let's Build A Web Server. Part 3. (https://ruslanspivak.com/lsbaws-part3/)

Let's Build A Simple Interpreter. Part 2. (https://ruslanspivak.com/lsbasi-part2/)

Disclaimer

Some of the links on this site have my Amazon referral id, which provides me with a small commission for each sale. Thank you for your support.

Back to top