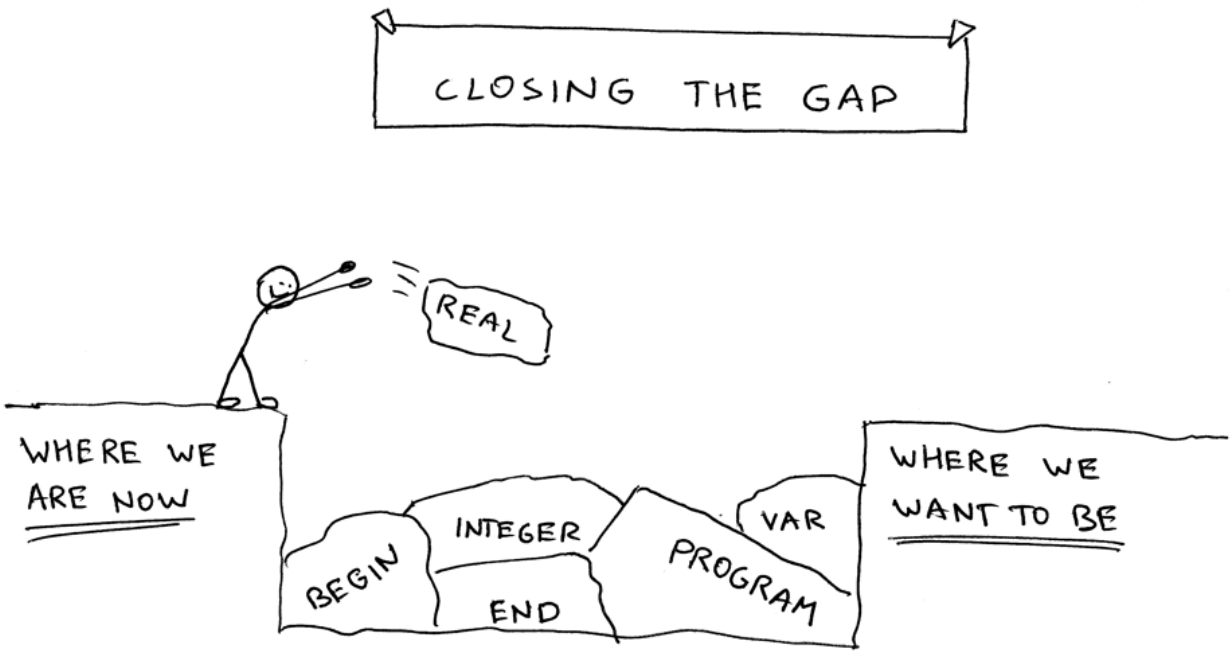
让我们构建一个简单的解释器。第 10 部分。 (https://ruslanspivak.com/lsbasi-part10/)

日期 2016 年 8 月 4 日，星期四

今天，我们将继续缩小我们现在所处的位置与我们想要成为的位置之间的差距：一个用于 Pascal 编程语言子集的全功能解释器 (/lsbasi-part1/)。



在本文中，我们将更新我们的解释器来解析和解释我们第一个完整的 Pascal 程序。该程序也可以由Free Pascal 编译器fpc (http://www.freepascal.org/)编译 (http://www.freepascal.org/)。

这是程序本身：

```
程序 第 10 部分;
VAR
   数     : 整数;
   a 、 b 、 c 、 x  :  整数;
   y           :  真实;

BEGIN  {Part10}
   BEGIN
      number  :=  2 ;
      一个 : = 数字;
      b  :=  10  *  a  +  10  * 数字 DIV  4 ;
      c  :=  a  -  -  b
   END ;
   x  :=  11 ;
   y  :=  20  /  7  +  3.14 ;
   { writeln('a = ', a); }
   { writeln('b = ', b); }
   { writeln('c = ', c); }
   { writeln('number = ', number); }
   { writeln('x = ', x); }
   { writeln('y = ', y); }
END 。  {Part10}
```

在我们开始深入研究之前，先从GitHub 上 (https://github.com/rspivak/lsbasi/blob/master/part10/python/spi.py)下载解释器的源代码 (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)和上面 (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)的 Pascal 源代码 (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)，然后在命令行上尝试一下：

```
$ python spi.py part10.pas
 a  =  2
b  =  25
c  =  27
数字 =  2
x  =  11
y  =  5 .99714285714
```

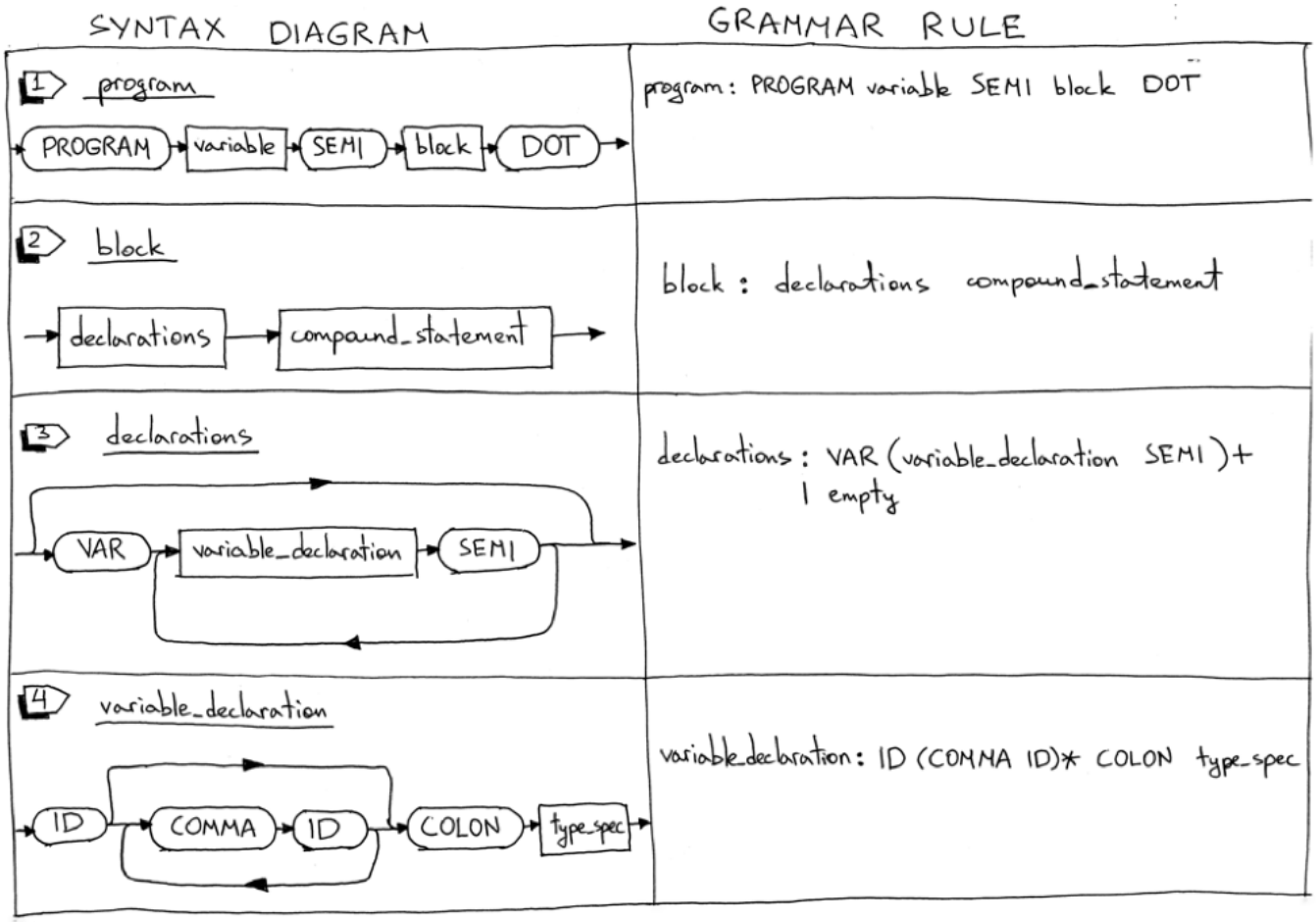
如果我删除part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)文件中writeln语句周围的注释，使用fpc (http://www.freepascal.org/)编译源代码，然后运行生成的可执行文件，这就是我在笔记本电脑上得到的：
(https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)
(http://www.freepascal.org/)

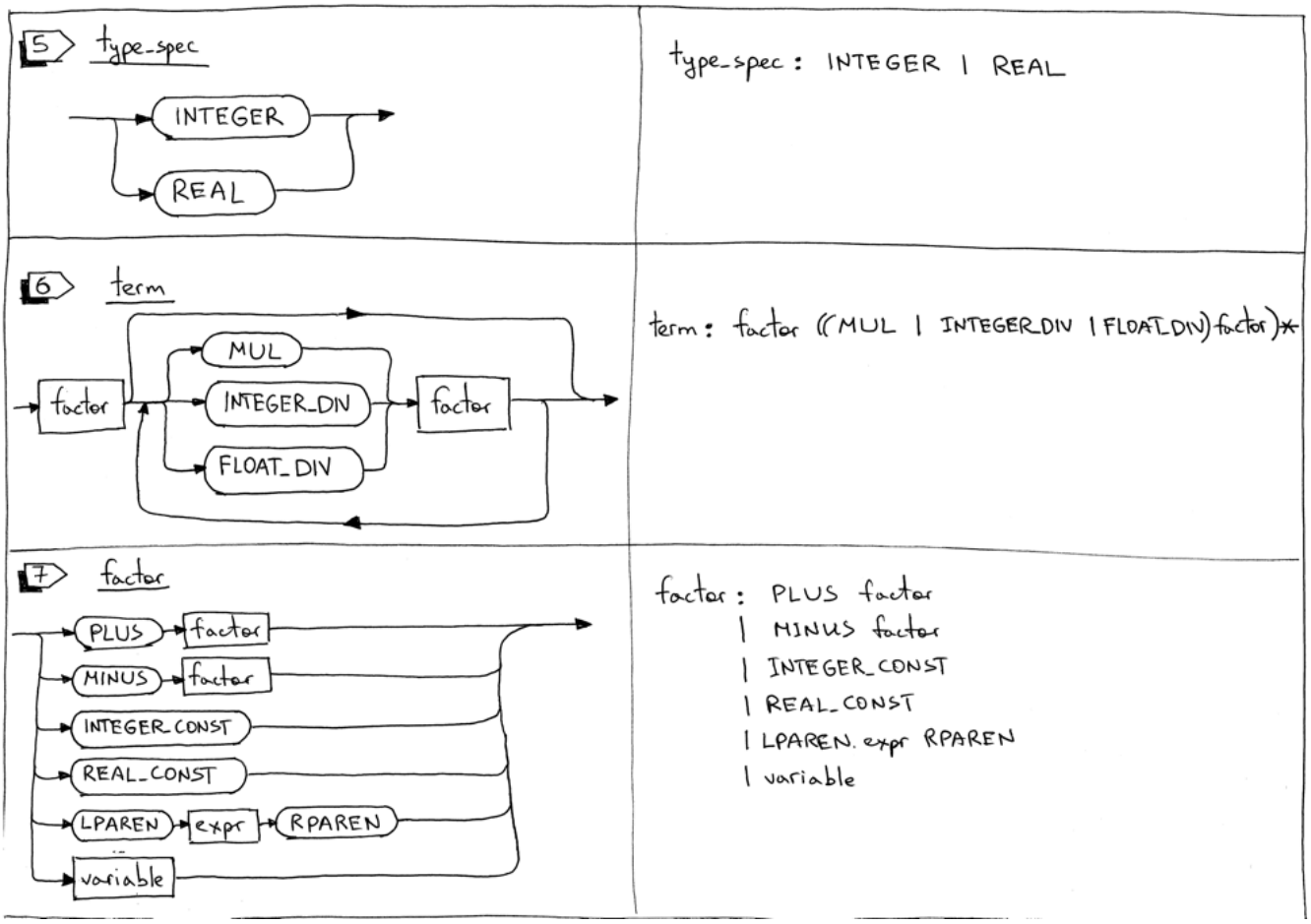
```
$ fpc part10.pas
$ ./part10
a  =  2
b  =  25
c  =  27
数字 =  2
x  =  11
y  =   5 .99714285714286E+000
```

好的，让我们看看今天要介绍的内容：

1. 我们将学习如何解析和解释 Pascal **PROGRAM** 标头
2. 我们将学习如何解析 Pascal 变量声明
3. 我们将更新我们的解释器以使用**DIV**关键字进行整数除法，使用正斜杠 / 进行浮点除法
4. 我们将添加对 Pascal 注释的支持

让我们先深入了解一下语法变化。今天我们将添加一些新规则并更新一些现有规则。





1. 该**程序**定义语法规则被更新，以包括**PROGRAM**保留关键字，节目名称，和一个块以点的端部。下面是一个完整的 Pascal 程序示例：

```
程序 第 10 部分；
开始
结束。
```

2. 该**块**规则结合了声明，排除和compound_statement规则。在本系列的后面添加过程声明时，我们还将使用该规则。下面是一个块的例子：

```
VAR
   数 ： 整数；

开始
结束
```

这是另一个例子：

```
开始
结束
```

3. Pascal 声明有几个部分，每个部分都是可选的。在本文中，我们将只介绍变量声明部分。该**声明**规则有任何变量声明子规则或者它是空的。

4. Pascal 是一种静态类型语言，这意味着每个变量都需要一个明确指定其类型的变量声明。在 Pascal 中，变量必须在使用前声明。这是通过使用**VAR**保留关键字在程序变量声明部分中声明变量来实现的。您可以像这样定义变量：

```
VAR
   数     ：  整数；
   a 、 b 、 c 、 x  ：  整数；
   y          ：  真实；
```

5. 该**type_spec**规则是用于处理INTEGER和REAL类型和变量声明中使用。在下面的例子中

```
VAR
   a  ：  整数；
   b  : 真实的；
```

   变量“a”声明为INTEGER类型，变量“b”声明为REAL（float）类型。在本文中，我们不会强制执行类型检查，但我们将在本系列的后面添加类型检查。

6. 本**项**规则更新为使用**DIV**整数除法关键词和斜杠（/）浮法事业部。

   之前，使用正斜杠将 20 除以 7 会产生整数 2：

```
20 / 7 = 2
```

   现在，使用正斜杠将 20 除以 7 将产生REAL（浮点数） 2.85714285714 ：

```
20 / 7 = 2.85714285714
```

   从现在开始，要获得INTEGER而不是REAL，您需要使用**DIV** 关键字：

```
20 格 7 = 2
```

7. 该**系数**的规则进行更新处理整数和实（浮点）常数。我还删除了INTEGER子规则，因为常量将由**INTEGER_CONST**和**REAL_CONST**标记表示，而**INTEGER**标记将用于表示整数类型。在词法分析器会产生下面的例子INTEGER_CONST令牌20和7以及REAL_CONST令牌3.14：

```
y  :=  20  /  7  +  3.14 ;
```

这是我们今天的完整语法：

```
程序：PROGRAM 变量 SEMI 块 DOT

块：声明复合语句

声明：VAR（variable_declaration SEMI）+
             | 空的

variable_declaration : ID (COMMA ID)* COLON type_spec

type_spec : 整数 | 真实的

Compound_statement : BEGIN statement_list END

statement_list : 语句
               | statement SEMI statement_list

语句：复合语句
          | 赋值语句
          | 空的

assignment_statement : 变量 ASSIGN expr

空的 :

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

术语：因子（（MUL | INTEGER_DIV | FLOAT_DIV）因子）*

因素：加因素
       | 减因子
       | INTEGER_CONST
       | REAL_CONST
       | LPAREN expr RPAREN
       | 多变的

变量：ID
```

在本文的其余部分，我们将进行与上次相同的练习：

1. 更新词法分析器
2. 更新解析器
3. 更新解释器

更新词法分析器

以下是词法分析器更改的摘要：

1. 新代币
2. 新的和更新的保留关键字
3. 处理 Pascal 注释的新skip_comment方法

4. 重命名整数方法并对方法本身进行一些更改
5. 更新get_next_token方法以返回新令牌

让我们深入研究上面提到的变化：

1. 为了处理程序头、变量声明、整数和浮点常量以及整数和浮点除法，我们需要添加一些新标记——其中一些是保留关键字——我们还需要更新INTEGER标记的含义来表示整数类型而不是整数常量。以下是新令牌和更新令牌的完整列表：
   - PROGRAM（保留关键字）
   - VAR（保留关键字）
   - 冒号 (:)
   - 逗号 (,)
   - 整数（我们将其更改为整数类型而不是整数常量，如 3 或 5）
   - REAL（Pascal REAL 类型）
   - INTEGER_CONST（例如，3 或 5）
   - REAL_CONST（例如 3.14 等）
   - INTEGER_DIV 用于整数除法（**DIV**保留关键字）
   - FLOAT_DIV 用于浮点除法（正斜杠 / ）
2. 这是保留关键字到标记的完整映射：

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

3. 我们添加了skip_comment方法来处理Pascal 注释。该方法非常基本，它所做的就是丢弃所有字符，直到找到右花括号：

```
def skip_comment ( self ):
    while self 。current_char != '}' :
        self 。提前（）
    自我。Advance () # 右花括号
```

4. 我们将整数方法重命名为数字方法。它可以处理整数常量和浮点常量，如 3 和 3.14：

```
def  number ( self ):
    """返回从输入中消耗的（多位）整数或浮点数。"""
    result  =  ''
    while  self . current_char 是 不 无 和 自我。current_char 。isdigit ():
        结果 +=  self 。current_char
        自我。提前()

    如果 自. current_char  ==  '.' :
        结果 += 自我。current_char
        自我。提前()

        而 （
            自我。current_char 是 不 无 和
            自我。current_char 。ISDIGIT （）
        ）:
            结果 + = 自我。current_char
            自我。提前()

        token  =  Token ( 'REAL_CONST' ,  float ( result ))
    else :
        token  =  Token ( 'INTEGER_CONST' ,  int ( result ))

    返回 令牌
```

5. 我们还更新了get_next_token方法以返回新令牌：

```
def  get_next_token ( self ):
    而 self 。current_char 是 不 无：
        ...
        如果 自我。current_char  ==  '{' :
            self 。提前（）
            自我。skip_comment ()
            continue
        ...
        如果 self 。current_char 。isdigit ():
            返回 self 。数()

        如果 自. current_char  ==  ':' :
            self 。提前()
            返回 令牌( COLON ,  ':' )

        如果 自. current_char  ==  ',' :
            self 。提前（）
            返回 令牌（逗号， “” ）
        ......
        如果 自我。current_char  ==  '/' :
            self . 提前()
            返回 令牌( FLOAT_DIV ,  '/' )
        ...
```

更新解析器

现在到解析器更改。

以下是更改的摘要：

1. 新的AST节点：Program、Block、VarDecl、Type
2. 新语法规则对应的新方法：block、declarations、variable_declaration和type_spec。
3. 对现有解析器方法的更新：program、term和factor

让我们一一回顾一下变化：

1. 我们将首先从新的AST节点开始。有四个新节点：
   - 该计划 AST节点代表一个程序，将是我们的根节点

```
class  Program ( AST ):
    def  __init__ ( self ,  name ,  block ):
        self 。姓名 = 姓名
        自我。块 = 块
```

   - 该座 AST节点保存的声明和复合语句：

```
class  Block ( AST ):
    def  __init__ ( self ,  declarations ,  Compound_statement ):
        self 。声明 = 声明
        self 。复合语句 = 复合语句
```

   - 所述VarDecl AST节点代表一个变量声明。它包含一个变量节点和一个类型节点：

```
类 VarDecl ( AST ):
    def  __init__ ( self ,  var_node ,  type_node ):
        self 。var_node  =  var_node
        self 。type_node  =  type_node
```

   - 该类型 AST节点表示的变量类型（INTEGER或REAL）：

```
类 类型( AST ):
    def  __init__ ( self ,  token ):
        self 。令牌 = 令牌
        自我。价值 = 令牌。价值
```

2. 您可能还记得，语法中的每个规则在我们的递归下降解析器中都有一个对应的方法。今天我们添加了四个新方法：block、declarations、variable_declaration和type_spec。这些方法负责解析新的语言结构和构建新的AST 节点：

```
def  block ( self ):
    """block : 声明复合语句"""
    Declaration_nodes  =  self . 声明（）
    复合 语句 节点=自我。Compound_statement ()
    node  =  Block ( declaration_nodes ,  Compound_statement_node )
    返回 节点

def 声明( self ):
    """declarations : VAR (variable_declaration SEMI)+
                    | 空的
    """
    声明 =  []
    if  self . current_token 。类型 ==  VAR :
        自我。吃（VAR ）
        而 自我。current_token 。类型 ==  ID :
            var_decl  =  self 。variable_declaration ()
            声明。扩展( var_decl)
            自我。吃（半）

    返回 声明

def  variable_declaration ( self ):
    """variable_declaration : ID (COMMA ID)* COLON type_spec"""
    var_nodes  =  [ Var ( self . current_token )]   # 第一个 ID
    self . 吃（身份证）

    而 自我。current_token 。类型 == 逗号：
        自我。吃（逗号）
        var_nodes 。追加（瓦尔（自我。current_token ））
        的自我。吃（身份证）

    自我。吃（冒号）

    type_node  =  self 。type_spec ()
    var_declarations  =  [
        VarDecl ( var_node ,  type_node )
        for  var_node  in  var_nodes
    ]
    return  var_declarations

def  type_spec ( self ):
    """type_spec : INTEGER
                 | REAL
    """
    token  =  self . current_token
    如果是 self 。current_token 。类型 == 整数：
        自我。吃（整数）
    其他：
        自我。吃（真实）
    节点 = 类型（令牌）
    返回 节点
```

3. 我们还需要更新program、term和factor方法以适应我们的语法变化：

```
def  program ( self ):
    """program : PROGRAM variable SEMI block DOT"""
    self . 吃（程序）
    var_node  =  self 。变量()
    prog_name  =  var_node 。重视
    自我。吃（半）
    block_node  =  self 。block ()
    program_node  =  Program ( prog_name ,  block_node )
    self 。吃（DOT ）
    返回 程序节点

def  term ( self ):
    """term : factor ((MUL | INTEGER_DIV | FLOAT_DIV) factor)*"""
    node  =  self . 因子()

    而 自我。current_token 。键入 在 （MUL ,  INTEGER_DIV ,  FLOAT_DIV ）：
        令牌 = 自我。current_token
        如果是 token 。类型 ==  MUL :
            self 。吃( MUL )
        elif 令牌。type  ==  INTEGER_DIV :
            self 。吃( INTEGER_DIV )
        elif 令牌。类型 ==  FLOAT_DIV:
            自己。吃（FLOAT_DIV ）

        node  =  BinOp ( left = node ,  op = token ,  right = self . factor ())

    返回 节点

def  factor ( self ):
    """factor : PLUS factor
              | MINUS factor
              | INTEGER_CONST
              | REAL_CONST
              | LPAREN expr RPAREN
              | variable
    """
    token  =  self . current_token
    如果是 token 。类型 ==  PLUS :
        自我。吃（PLUS ）
        节点 =  UnaryOp （令牌,  自。因子（））
        返回 节点
    的elif 令牌。类型 == 减号:
        自我。吃（MINUS ）
        节点 =  UnaryOp （令牌,  自。因子（））
        返回 节点
    的elif 令牌。type  ==  INTEGER_CONST :
        self 。吃（INTEGER_CONST ）
        返回 Num （令牌）
    elif 令牌。类型 ==  REAL_CONST :
        self 。吃( REAL_CONST )
        返回 Num ( token )
```

```
    elif 令牌。类型 ==  LPAREN :
        self 。吃（LPAREN ）
        节点 = 自我。expr ()
        自我。吃（RPAREN ）
        返回 节点
    else :
        node  =  self 。变量()
        返回 节点
```

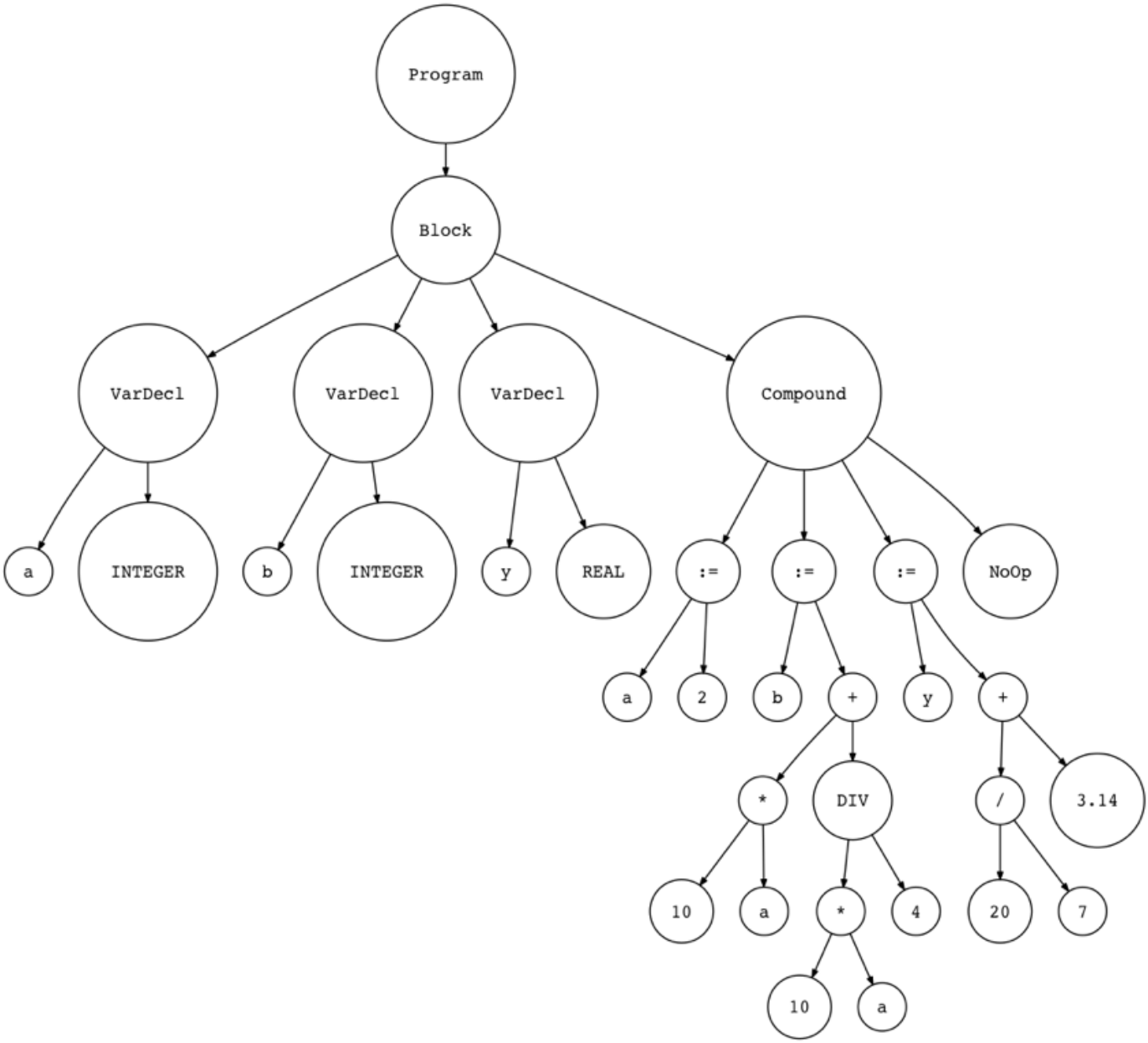
现在，让我们看看带有新节点的**抽象语法树**是什么样子。这是一个小的工作 Pascal 程序：

```
程序第 10 部分；
VAR
   a ，  b  ：  整数；
   y     ：  真实；

开始 {Part10AST}
   a  :=  2 ;
   b  :=  10  *  a  +  10  *  a  DIV  4 ;
   y  :=  20  /  7  +  3.14 ;
结束。  {Part10AST}
```

让我们生成一个AST并使用genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part10/python/genastdot.py)对其进行可视化：

```
$ python genastdot.py part10ast.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



在图片中您可以看到我们添加的新节点。

更新解释器

我们完成了词法分析器和解析器的更改。剩下的就是向我们的Interpreter类添加新的访问者方法。将有四种新方法可以访问我们的新节点：

- 访问_程序
- 访问_阻止
- 访问_VarDecl
- 访问类型

他们很直接。您还可以看到解释器对VarDecl和Type 节点不执行任何操作：

```
def visit_Program(self, node):
    self.visit(node.block)

def visit_Block(self, node):
    for declaration in node.declarations:
        self.visit(declaration)
    self.visit(node.compound_statement)

def visit_VarDecl(self, node):
    # do nothing
    pass

def visit_Type(self, node):
    # do nothing
    pass
```

我们还需要更新visit_BinOp方法以正确解释整数和浮点除法：

```
def visit_BinOp(self, node):
    if node.op.type == PLUS:
        return self.visit(node.left) + self.visit(node.right)
    elif node.op.type == MINUS:
        return self.visit(node.left) - self.visit(node.right)
    elif node.op.type == MUL:
        return self.visit(node.left) * self.visit(node.right)
    elif node.op.type == INTEGER_DIV:
        return self.visit(node.left) // self.visit(node.right)
    elif node.op.type == FLOAT_DIV:
        return float(self.visit(node.left)) / float(self.visit(node.right))
```

让我们总结一下我们在本文中扩展Pascal解释器所要做的：

- 向语法添加新规则并更新一些现有规则
- 向词法分析器添加新的标记和支持方法，更新和修改一些现有方法
- 为新的语言结构向解析器添加新的AST节点
- 将与新语法规则相对应的新方法添加到我们的递归下降解析器中并更新一些现有方法
- 向解释器添加新的访问者方法并更新一个现有的访问者方法

由于我们的更改，我们还摆脱了我在第 9 部分中 (/lsbasi-part9/)介绍的一些技巧，即：

- 我们的解释器现在可以处理**PROGRAM** 标头
- 现在可以使用**VAR** 关键字声明**变量**
- 该**DIV**关键字被用于整数除法和斜杠使用/浮法除法

如果您还没有这样做，那么作为练习，无需查看源代码，重新实现本文中的解释器，并使用part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)作为您的测试输入文件。

这就是今天的全部内容。在下一篇文章中，我将更详细地讨论符号表管理。请继续关注，我们很快就会见到你！

如果您想在收件箱中获取我的最新文章，请在下方输入您的电子邮件地址，然后单击“获取更新”！

输入您的名字 *

输入您最好的电子邮件 *

获取更新！

本系列所有文章：

- 让我们构建一个简单的解释器。第1部分。 (/lsbasi-part1/)
- 让我们构建一个简单的解释器。第2部分。 (/lsbasi-part2/)
- 让我们构建一个简单的解释器。第 3 部分。 (/lsbasi-part3/)
- 让我们构建一个简单的解释器。第 4 部分。 (/lsbasi-part4/)
- 让我们构建一个简单的解释器。第 5 部分。 (/lsbasi-part5/)
- 让我们构建一个简单的解释器。第 6 部分。 (/lsbasi-part6/)
- 让我们构建一个简单的解释器。第 7 部分：抽象语法树 (/lsbasi-part7/)
- 让我们构建一个简单的解释器。第 8 部分。 (/lsbasi-part8/)
- 让我们构建一个简单的解释器。第 9 部分。 (/lsbasi-part9/)
- 让我们构建一个简单的解释器。第 10 部分。 (/lsbasi-part10/)
- 让我们构建一个简单的解释器。第 11 部分。 (/lsbasi-part11/)
- 让我们构建一个简单的解释器。第 12 部分。 (/lsbasi-part12/)
- 让我们构建一个简单的解释器。第 13 部分：语义分析 (/lsbasi-part13/)
- 让我们构建一个简单的解释器。第 14 部分：嵌套作用域和源到源编译器 (/lsbasi-part14/)
- 让我们构建一个简单的解释器。第 15 部分。 (/lsbasi-part15/)

- 让我们构建一个简单的解释器。第 16 部分：识别过程调用 (/lsbasi-part16/)
- 让我们构建一个简单的解释器。第 17 部分：调用堆栈和激活记录 (/lsbasi-part17/)
- 让我们构建一个简单的解释器。第 18 部分：执行过程调用 (/lsbasi-part18/)
- 让我们构建一个简单的解释器。第 19 部分：嵌套过程调用 (/lsbasi-part19/)

注释

ALSO ON **RUSLAN'S BLOG**

Let's Build A Simple Interpreter. Part 16: …

2 years ago • 7 comments

Learning is like rowing upstream: not to advance is to drop back. — Chinese …

Let's Build A Simple Interpreter. Part 9.

5 years ago • 42 comments

I remember when I was in university (a long time ago) and learning systems …

Let's Build A Web Server. Part 1.

7 years ago • 88 comments

Out for a walk one day, a woman came across a construction site and saw …

Let's B Interpr

4 years a

Anything worth ov doing a

Maddy • a year ago

I am 18 and I'm like which part of the internet have I just discovered ? I am completely blown away by this series. I have been programming for 4 years (web + iOS) and partly python to basically have fun automation. That's what all the latest books and YouTube tutorials teach BUT THERE IS SUCH A LACK OF CONTENT ON A MORE FUNDAMENTAL LEVEL OF COMPUTERS - HOW COMPILERS AND INTERPRETERS ARE BUILT. This blog just completely clears that niche. I cannot express my gratitude to what Ruslan has done here. I hope to see more of this content on the blog in the future. Till then everybody Peace ! ✌ and stay healthy protect yourselves from 🦠 (Covid 19)

1 • Reply • Share ›

Owen • 3 years ago • edited

Great blog! Just noticed, where it says "Here is our complete grammar for today", the `type_spec` production should include `REAL`:

```
type_spec : INTEGER | REAL
```

1 ^ | v • Reply • Share ›

Mas Bagol • 5 years ago

Very well written and understandable as always for non English speaker like me. Especially the pictures XD. Waiting for scoped variable

1 ^ | v • Reply • Share ›

Jakob • 6 months ago

This blog is absolutely priceless! Thank you so much for sharing you knowledge!

^ | v • Reply • Share ›

Michael • 9 months ago

Hey Ruslan, long time reader slowly working my way through your accessible articles.

I have a technical question: How do you determine which parts of the language grammar become represented as nodes within the AST for the intermediate representation produced by the parser?

I have reviewed your previous articles looking for some hint but to no avail. I have yet to purchase a book on compiler/language theory but it is slowly coming to that. It seems to me that it is similar to the tokens which I have come to understand how those are chosen. However, there seems to be a bit more nuance to selecting and representing the nodes of the AST.

^ | v • Reply • Share ›

Daniel Zhu • a year ago • edited

Here's my spin on it with c++:
https://github.com/dantheking-crypto/MakeALanguage
It's not Pascal though, very similar. Check out the test.txt and README for more information. Also, for the latest update, always check develop, not master.

^ | v • Reply • Share ›

solstice333 • a year ago

Consider that the statement_list rule might make more sense as

statement_list: statement (SEMI statement)*

since the parser eats up `SEMI statement` iteratively. Alternatively, I've tried converting the parser logic to recurse into statement_list like how

statement_list: statement | statement SEMI statement_list

suggests. That's works fine as well.

^ | v • Reply • Share ›

James • 2 years ago

I have write it with TypeScript at https://github.com/shiftone...

^ | v • Reply • Share ›

James M. Lay • 4 years ago

Absolutely excellent. Thank you.

Reply • Share ›

Hugo Dufour • 5 years ago

This series is what got me into compilers, interpreters and broadly language theory!
Just one thing, I don't think I have used Pascal more than twice in my life but I do seem to remember that nested comments are allowed, I'm sure it must be easy to implement with a stack, but I was wondering if you would add that ;)
But honestly just a tiny minor detail x)
Love this series <3

Reply • Share ›

rspivak Mod ➚ Hugo Dufour • 5 years ago

Thanks, Hugo!
Here is some information about nested comments in Pascal:
http://www.freepascal.org/d...
I may add them down the road or I might leave it as an exercise. I haven't decided yet. :)

1 Reply • Share ›

Alex Teodor • 5 years ago

Very nice series. I'd suggest, but maybe it's just me, that you optionally 'import readline', which allows user input from raw_input() to present a command history, much like the shell. Thus, when being asked for input, one has but to hit the 'up' key to get her or his previous input command.

Reply • Share ›

Armleo • 5 years ago

Good Job!

Reply • Share ›

Luke Scalf • 5 years ago

Loving this series! Looking forward to the next article!

Reply • Share ›

elinx • 5 years ago

Thank you very much for this awesome tutorial and look forward for you next article.

Reply • Share ›

rspivak Mod ➚ elinx • 5 years ago

You're welcome.

Reply • Share ›

Randy • 5 years ago

Your tutorials are very concise and easy to follow. You are very talent in writing. Look forward to your book.

Reply • Share ›

Aleksandr Zhuravlev • 5 years ago

Thank you, Ruslan for the great stuff. I've just updated my Java version of SPI.

^ | v • Reply • Share ›

Tomatosoup • 5 years ago

These are reaaally great tutorials, waiting impatiently for the next part ;)

^ | v • Reply • Share ›

Leon Bauer • 5 years ago

Whohooooo! Thank you :)))

^ | v • Reply • Share ›

Hilde • 5 years ago

I'm looking forward to the next article! Thank you so much for this series. Is there a way to donate (without buying your recommended books)?

^ | v 1 • Reply • Share ›

JCFF ➚ Hilde • 4 years ago

社会的

- github (https://github.com/rspivak/)
- 推特 (https://twitter.com/rspivak)
- 链接 (https://linkedin.com/in/ruslanspivak/)

热门帖子

让我们构建一个 Web 服务器。第1部分。 (https://ruslanspivak.com/lsbaws-part1/)

让我们构建一个简单的解释器。第1部分。 (https://ruslanspivak.com/lsbasi-part1/)

让我们构建一个 Web 服务器。第2部分。 (https://ruslanspivak.com/lsbaws-part2/)

让我们构建一个 Web 服务器。第 3 部分。 (https://ruslanspivak.com/lsbaws-part3/)

让我们构建一个简单的解释器。第2部分。 (https://ruslanspivak.com/lsbasi-part2/)

免责声明

这个网站上的一些链接有我的亚马逊推荐 ID，它为我提供了每次销售的小额佣金。感谢您的支持。

↑ 回到顶部