

# 编译原理学习（一）LSBASI

## 参考资料

- 自己动手写编译器：<https://pandolia.net/tinyc/index.html>
- github源码地址：<https://github.com/ZoEplA/LBASI>

## 总体架构

- 词法分析
- 语法分析
- 语义分析和中间代码生成
- 优化
- 目标代码生成

## 词法分析

词法分析的任务是输入源程序，对构成源程序的字符串进行扫描和分解，识别出一个单词符号，如基本字（if, for, begin 等）、标识符、常数、运算符和界符（如“（”，“）”，“=”，“；”）等，将所识别出的单词用统一长度的标准形式（也称内部码）来表示，以便于后继语法工作的进行。因此，词法分析工作是将源程序中的字符串变换成单词符号流的过程，词法分析所遵循的是语言的构词规则。

可以使用Lex做词法分析，参考资料：

- <https://blog.newnius.com/write-a-lexical-analyzer-using-lex.html>
- 

## 语法分析

语法分析的任务是在词法分析的基础上，根据语言的语法规则(文法规则)把单词符号流分解成各类语法单位(语法范畴)，如“短语”，“子句”，“句子（语句）”，“程序段”和“程序”。通过语法分析可以确定整个输入串是否构成一个语法上正确的“程序”。语法分析所遵循的是语言的语法规则，语法规则通常用上下文无关文法描述。

## 语义分析和中间代码生成

语义分析和中间代码生成的任务是对各类不同语法范畴按语言的语义进行初步翻译，包含两个方面的工作：一是对每种语法范畴进行静态语义检查，如变量是否定义，类型是否正确等；二是在语义检查正确的情况下进行中间代码的翻译。把语法范畴翻译成中间代码所遵循的是语言的语义规则，常见的中间代码有四元式、三元式、间接三元式和逆波兰记号。

## 优化

优化的任务主要是对前阶段产生的中间代码进行等价变换或改造（另一种优化是针对目标机即对目标代码进行优化），以期获得更为高效（节省时间和空间）的目标代码。常用的优化措施有删除冗余运算、删除无用赋值、合并已知量、循环优化等。优化所遵循的原则是程序的等价变换规则。

## 目标代码生成

目标代码生成的任务是把中间代码（或者经过优化处理之后）变换成特定机器上的机器语言程序或汇编语言程序，实现最终的翻译工作。

### part1

该部分实现了个位数相加的计算器操作

- 标记符（token）
- 词法分析器（lexical analyzer）

标记符：

标记符是指对象，它具有类型和值属性。例如，字符串“3”，标记符号的类型是整型（INTEGER），对应的值是整数3。

词法分析器：

将输入字符串分割为标记符的过程称为文法分析。所以，解释器的第一步工作是读取输入的字符序列，并将其转换为标记符流。解释器中处理该部分工作的部分称为文法分析器，或叫词法分析器。简而言之，你或许听到过该部件的其他命名，如 **scanner**（扫描器）或 **tokenizer**（标记符生成器）。这指的都是一回事：解释器或编译器的该部件将输入的字符序列转变成标记符号流。

### part2

该部分实现了多位整数相加减的操作

- 词素（lexemes）：一个词素是形成一个标记的一系列字符。
- 解析（parsing）：从标记流中发现结构的过程，或者说从标记流中识别语句的过程，称为解析（parsing）
- 解析器（parsers）：解释器或者编译器中承担该任务的部分称为解析器（parser）
- 

### part3

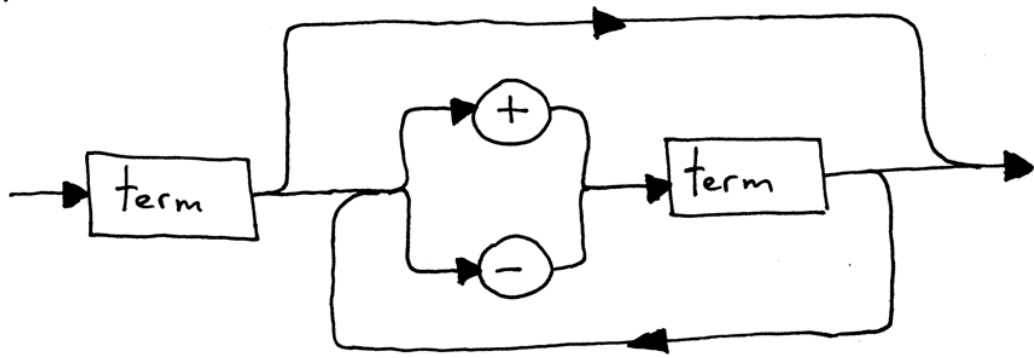
该部分实现了多位数多项的相加减操作。例如“7 - 3 + 2 - 1”

- 语法图
- 

语法图：

什么是语法图？一个语法图是一种编程语言的语法规则的图示。基本上，语法图可以直观地显示编程语言中允许哪些语句，哪些语句不允许。

expr



语法图有两个主要用途：

- 它们以图形方式表示编程语言的规范（语法）。
- 它们可用于帮助您编写解析器 - 您可以通过遵循简单的规则将图表映射到代码。

识别标记流中的短语的过程称为**解析**。执行该作业的解释器或编译器的一部分称为**解析器**。解析也称为**语法分析**，则解析器也适当被称为**语法分析器**。

上面语法图的主要实现部分：

```
def expr(self):
    # set current token to the first token taken from the input
    self.current_token = self.get_next_token()

    self.term()
    while self.current_token.type in (PLUS, MINUS):
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            self.term()
        elif token.type == MINUS:
            self.eat(MINUS)
            self.term()
```

## part4

**上下文无关文法**（简称语法）或BNF（Backus-Naur形式）是指另一个广泛使用的符号来指定编程语言的语法。

BNF是现在几乎每一位新编程语言书籍的作者都使用巴科斯范式来定义编程语言的语法规则。

巴科斯范式的内容大概如下：

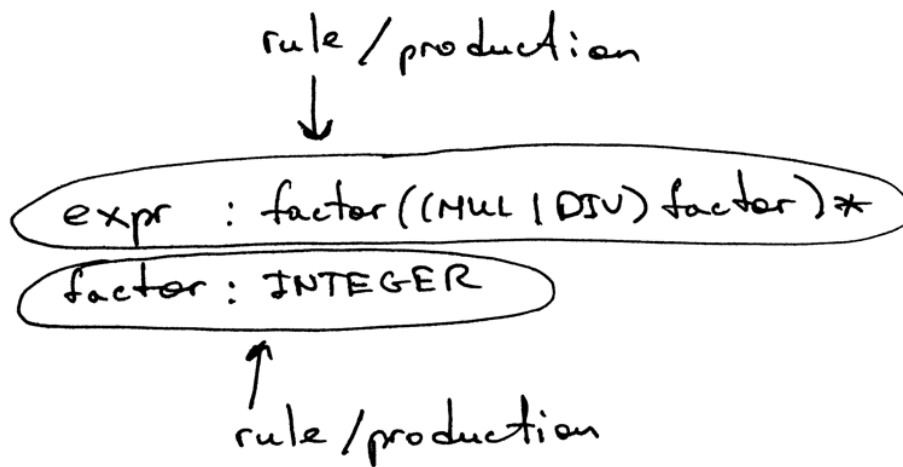
- 在双引号中的字("word")代表着这些字符本身。而double\_quote用来代表双引号。
- 在双引号外的字（有可能有下划线）代表着语法部分。
- 尖括号(< >)内包含的为必选项。
- 方括号([])内包含的为可选项。
- 大括号({})内包含的为可重复0至无数次的项。
- 括号()表示分组的意思

- 竖线(|)表示在其左右两边任选一项，相当于“OR”的意思。
- ::= 是“被定义为”的意思。

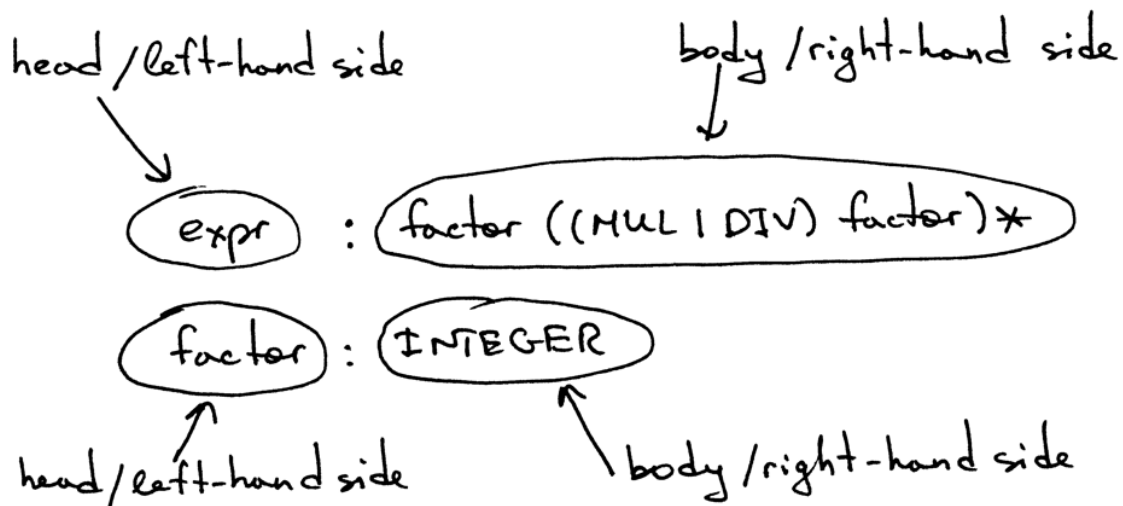
使用语法的几个原因：

1. 语法以简洁的方式指定编程语言的语法。与语法图不同，语法非常紧凑。您将在以后的文章中看到我越来越多地使用语法。
2. 语法可以作为很好的文档。
3. 即使您从头开始手动编写解析器，语法也是一个很好的起点。通常，您可以通过遵循**一组简单的规则**将语法转换为代码。
4. 有一组工具，称为**解析器生成器**，它接受语法作为输入，并根据该语法自动为您生成解析器。我将在本系列的后面讨论这些工具。

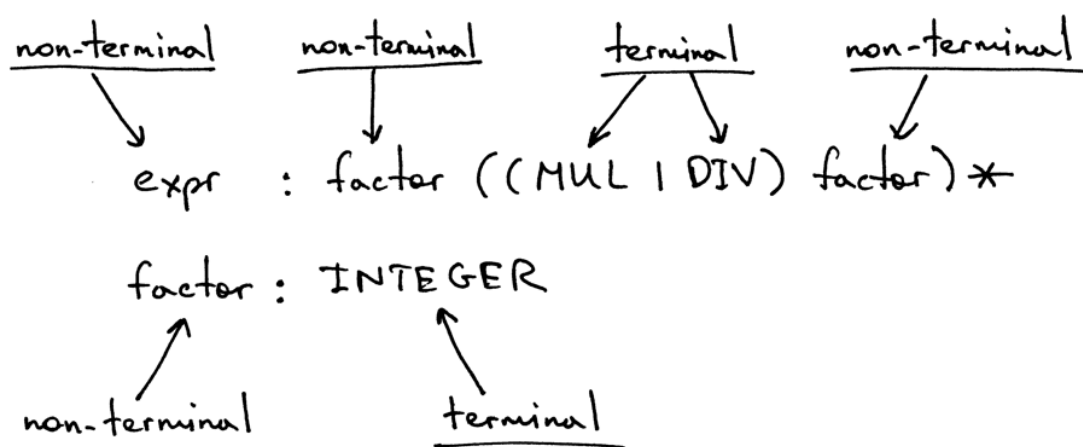
语法由一系列规则(也称为productions)组成，我们的语法有两个规则：



一系列规则包括非终端，称为制作production的头部或冒号左侧，以及一系列终端和/或非终端，称为生产的正文或右侧：



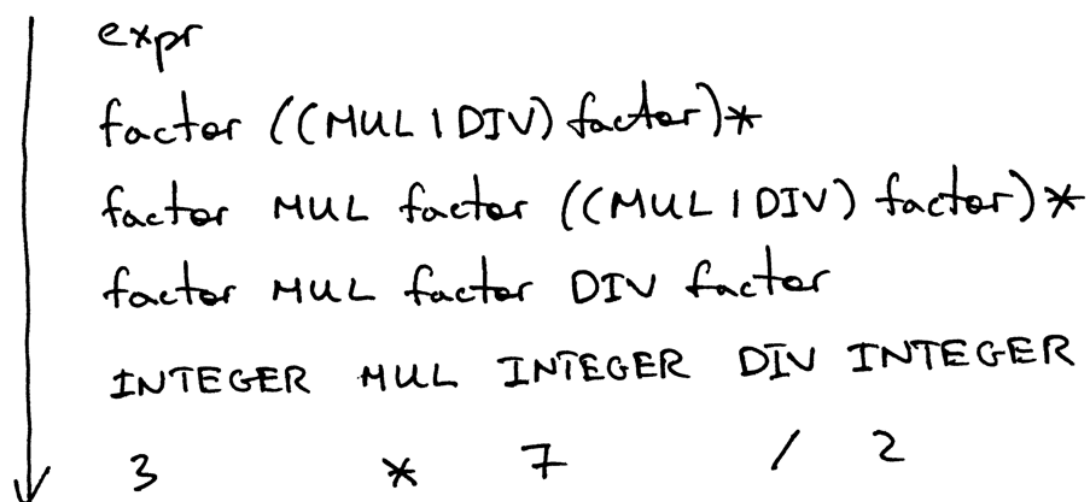
在我上面显示的语法中，像MUL，DIV和INTEGER这样的标记被称为终端，而像expr和factor这样的变量被称为非终端。非终端通常由一系列终端和/或非终端组成：



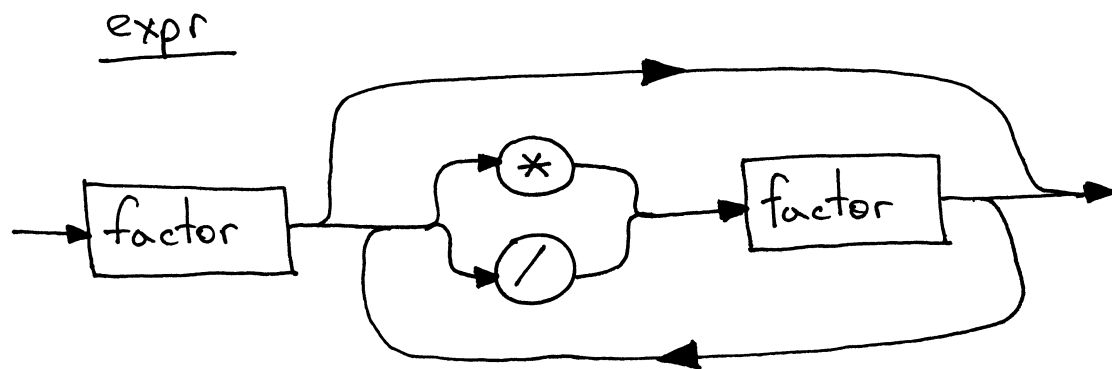
语法中使用的符号及其含义：

- | - 备择方案。条形表示“或”。因此 ( MUL | DIV ) 表示MUL或DIV。
- ( ... ) - 打开和关闭括号表示如 ( MUL | DIV ) 中的终端和/或非终端的分组。
- ( ... ) \* - 将组内的内容匹配零次或多次。

例子：语法派生表达式  $3 * 7 / 2$



乘法和除法的语法图：



|   |   |
|---|---|
| ① $\text{expr} : \text{factor} ((\text{MUL}   \text{DIV}) \text{factor})^*$ | <pre>def expr(self):     self.factor()     ...</pre>  |
| ② $(\text{MUL}   \text{DIV})$   | <pre>token = self.current_token if token.type == MUL:     ... elif token.type == DIV:     ...</pre> |
| ③ $((\text{MUL}   \text{DIV}) \text{factor})^*$                             | <pre>while self.current_token.type in (MUL, DIV):     ...</pre>                                     |
| ④ $\text{INTEGER}$  | <pre>self.eat(INTEGER)</pre>  |

该部分代码实现所做的工作：

- 将词法分析器与解释器分离
- 添加乘法和除法，实现了两项的加减乘除
- 多项的乘除**未实现先乘除后加减**

## part5

这部分主要解决多项的加减乘除计算器实现，主要需要解决乘除优先级比加减高的问题，优先级表如下：

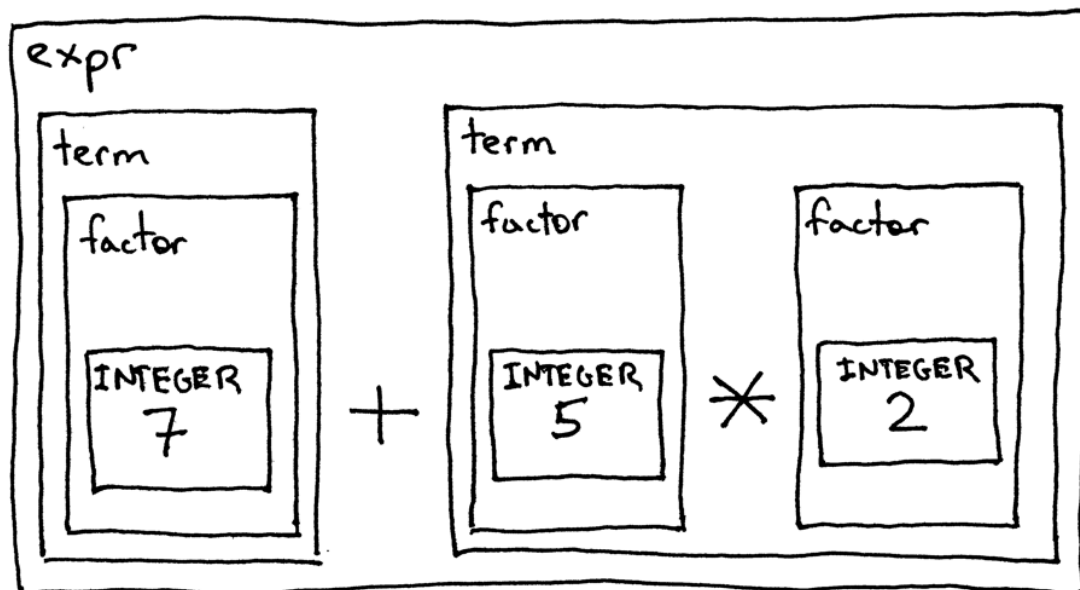
(因子规则)

higher  
precedence



| precedence level | associativity | operators |
|------------------|---------------|-----------|
| 2                | left          | +, -      |
| 1                | left          | *, /      |

$\text{expr} : \text{term} (\text{PLUS} | \text{MINUS}) \text{term} ) *$   
 $\text{term} : \text{factor} ((\text{MUL} | \text{DIV}) \text{factor}) *$   
 $\text{factor} : \text{INTEGER}$



- 解决加减乘除优先级问题
- 解决第一项为乘除问题，满足加减乘除多项运算(eg :  $3 * 4 + 6 + 8 / 4$ )

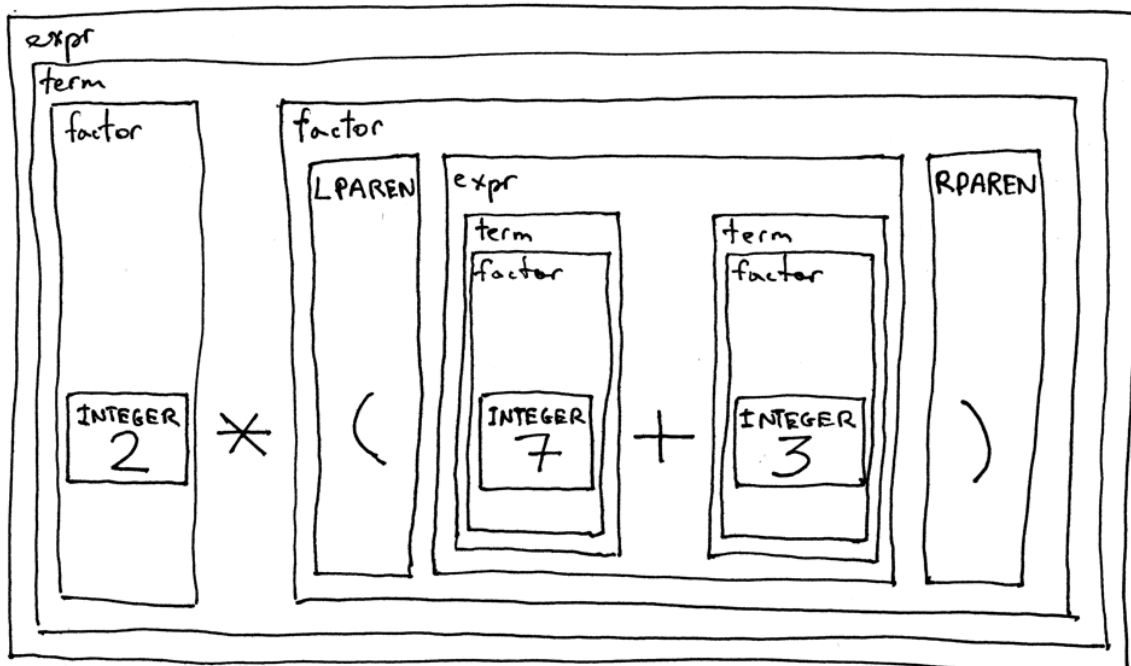
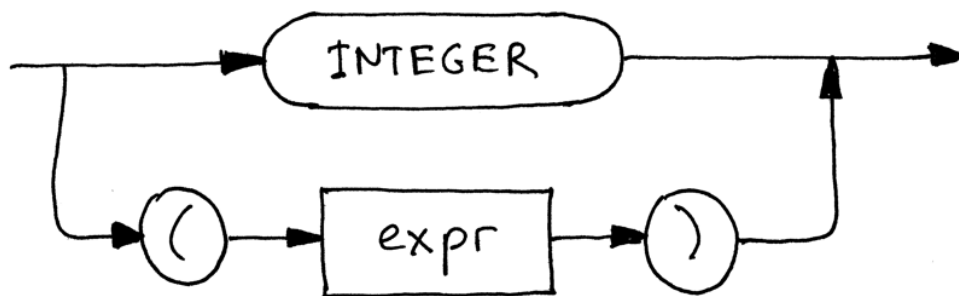
## part6

本部分目标是要添加括号这个优先级，能计算有括号的表达式。例如： $7 + 3 * (10 / (12 / (3 + 1) - 1))$

更新一些图，包括语法图等等：

$\text{expr} : \text{term}((\text{PLUS} | \text{MINUS}) \text{term}) *$   
 $\text{term} : \text{factor}((\text{MUL} | \text{DIV}) \text{factor}) *$   
 $\text{factor} : \text{INTEGER} | \text{LPAREN expr RPAREN}$

factor



主要操作：

- 在因子规则中修改 ( factor )
- 添加左右括号标记符

```

def factor(self):
    """Return an INTEGER token value."""
    token = self.current_token
    if token.type == INTEGER:

```



```

self.next(INTEGER)
return token.value
elif token.type == LPAREN:
self.next(LPAREN)
result = self.expr()
self.next(RPAREN)
return result

```

## part7

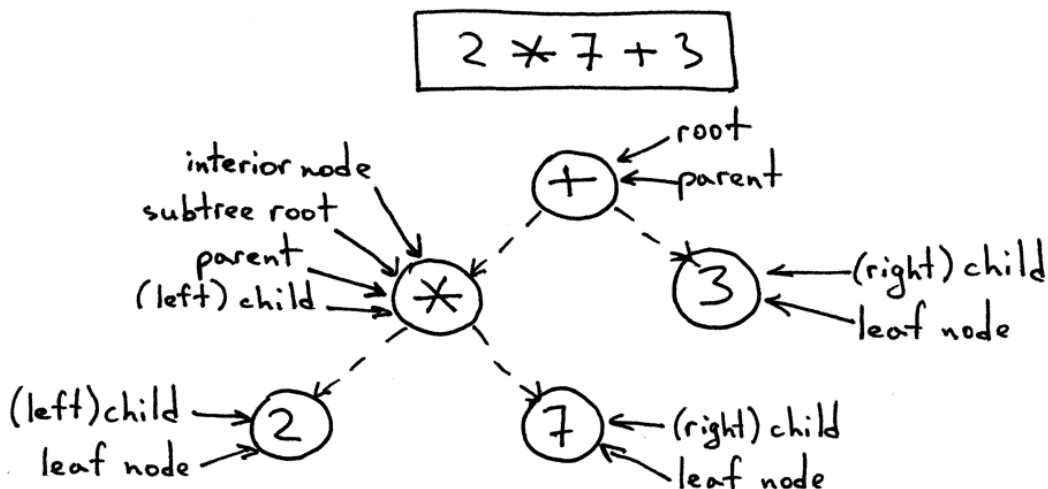
这一部分就是相当于一个进阶，引入AST作为解析器，并将其与解释器分离。

到目前为止，我们将解释器和解析器代码混合在一起，解析器会在解析器识别某个语言构造（如加法，减法，乘法或除法）后立即计算表达式。这种解释器称为语法导向解释器。它们通常只对输入进行一次传递，适用于基本语言应用程序。为了分析更复杂的Pascal编程语言结构，我们需要构建一个中间表示（IR）。我们的解析器将负责构建IR，我们的解释器将使用它来解释表示为IR的输入。

**事实证明，树是一种非常适合IR的数据结构**

让我们快速谈谈树术语。

- 一个树是一种数据结构，由组织成一个层次中的一个或多个节点。
  - 树有一个根，它是顶部节点。
  - 除根之外的所有节点都具有唯一的父节点。
  - 下图中标有\*的节点是父节点。标有2和7的节点是它的孩子；孩子们从左到右排序。
  - 没有子节点的节点称为叶子节点。
  - 具有一个或多个子节点且不是根节点的节点称为内部节点。
  - 孩子也可以组成完整的子树。在下图中，+节点的左子节点（标记为\*）是一个包含自己子节点的完整子树。
  - 在计算机科学中，我们将树木倒置，从顶部的根节点开始向下，树枝向下生长。
- 这是表达式  $2 * 7 + 3$  的树，并附有解释：



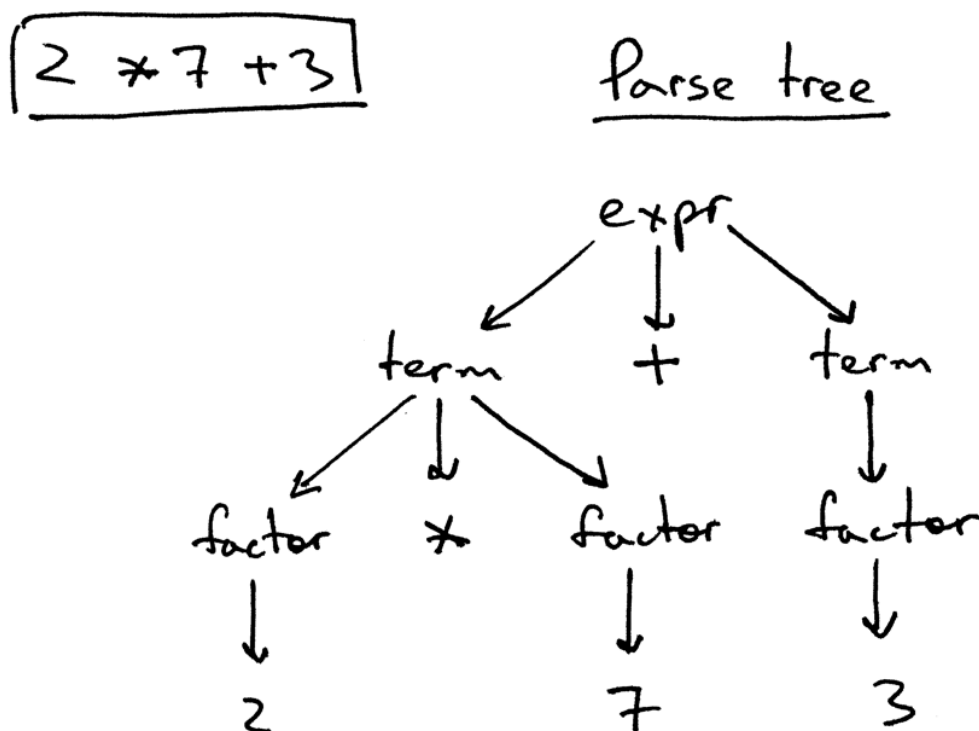
我们将在整个系列中使用的IR称为**抽象语法树**（AST）。但在我们深入研究AST之前，让我们简单地谈谈**解析树**。虽然我们不会为解释器和编译器使用解析树，但它们可以帮助您了解解析器如何通

过可视觉解析器的执行跟踪来解释输入。我们还将它们与AST进行比较，以了解为什么AST比解析树更适合中间表示。

那么，什么是解析树呢？一个解析树（有时称为具体语法树）是表示根据我们的语法定义语言结构的句法结构树。它基本上显示了解析器如何识别语言结构，换句话说，它显示了语法的起始符号如何在编程语言中派生出某个字符串。

解析器的调用堆栈隐式表示一个解析树，它在解析器尝试识别某个语言构造时会自动在内存中构建。

让我们看看表达式 $2 * 7 + 3$ 的解析树：



我们将在整个系列中使用的IR称为抽象语法树（AST）。

在了解AST之前我们先来看一看解析树，为什么AST比解析树更适合做IR。

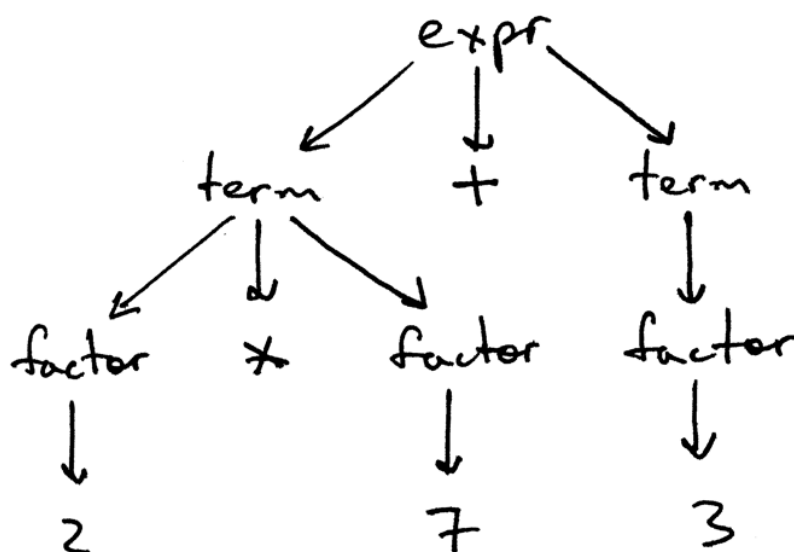
一个解析树（有时称为具体语法树）是表示根据我们的语法定义语言结构的句法结构树。它基本上显示了解析器如何识别语言结构，换句话说，它显示了语法的起始符号如何在编程语言中派生出某个字符串。

解析器的调用堆栈隐式表示一个解析树，它在解析器尝试识别某个语言构造时会自动在内存中构建。

让我们看看表达式 $2 * 7 + 3$ 的解析树：

2 \* 7 + 3

parse tree



从上图可以看到：

- 解析树记录解析器应用于识别输入的一系列规则。
- 解析树的根用语法起始符号标记。
- 每个内部节点代表一个非终端，即它代表语法规则应用程序，如我们的情况下的expr，term或factor。
- 每个叶节点代表一个令牌。

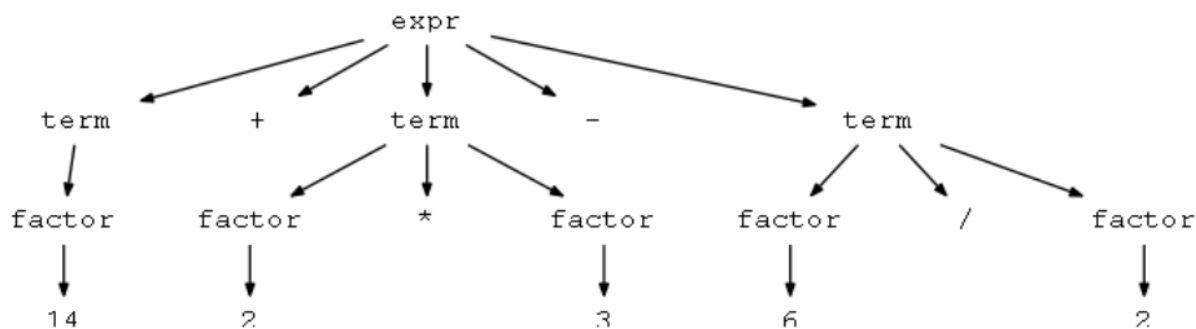
**例子(实现生成一个树)：**

`*"14 + 2 3 - 6 / 2"`

```
digraph astgraph {
    node [shape=None, fontsize=12, fontname="Courier", height=.1];
    ranksep=.3;
    edge [arrowsize=.5]

    node1 [label="expr"]
    node2 [label="term"]
    node1 --> node2
    node3 [label="+"]
    node1 --> node3
    node4 [label="term"]
    node1 --> node4
    .....
    node18 [label="2"]
    node13 --> node18
}
```

可视化：`dot -Tpng -o [parsetree].png(outputfile) [name].dot(inputfile)`

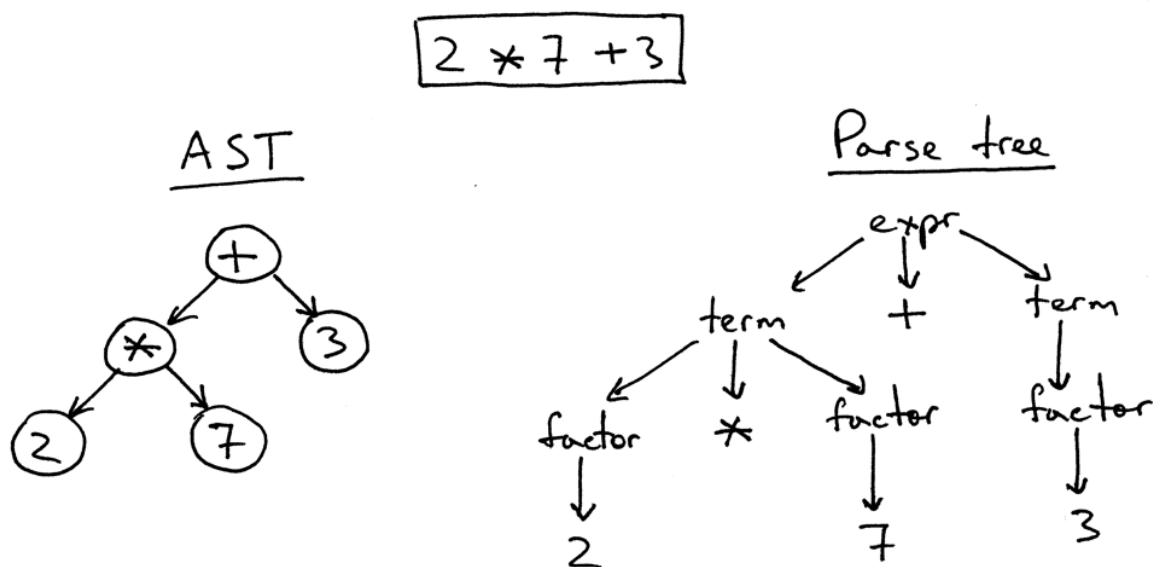


简单谈一下实现：

- 首先要在每次next的时候都去把当前节点的子节点（整数）（每次调用的时候都在factor和term之后，此时factor和term已经重置好当前子节点了，只需要往上面加东西就行了）
- 分别在factor和term的因子上加上对应的子节点
- 使用bfs遍历的时候从左往右把每个子节点的children等依次打印并清楚其对应指向就行。

现在，我们来谈谈抽象语法树（AST）。这是我们在整个系列的其余部分中大量使用的中间表示（IR）。它是我们的解释器和未来编译器项目的核心数据结构之一。

通过表达式  $2 * 7 + 3$  的AST和解析树的对比来开始我们的讨论：



同样的实现一遍生成AST树：

```

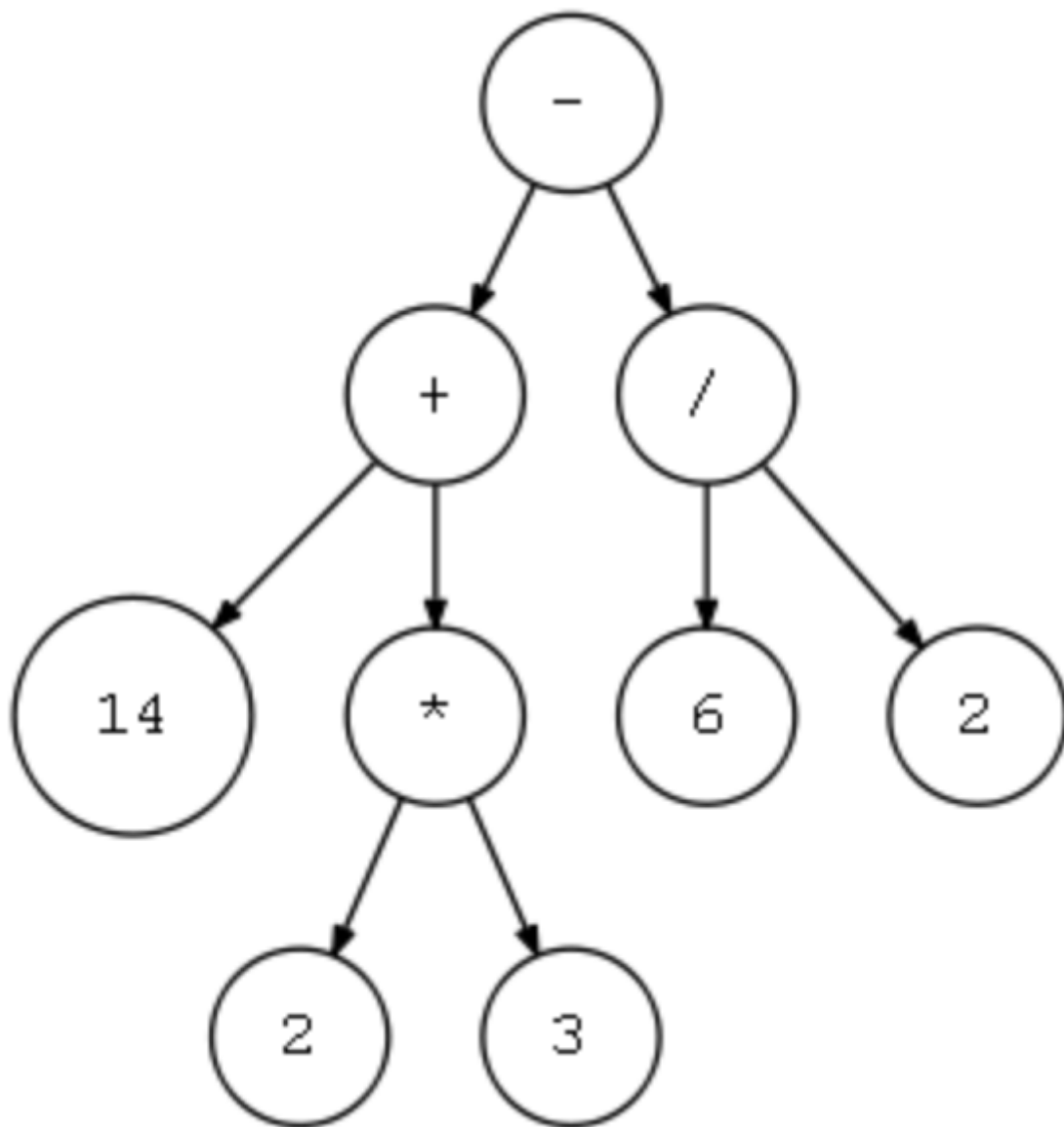
digraph astgraph {
    node [shape=circle, fontsize=12, fontname="Courier", height=.1];
    ranksep=.3;
    edge [arrowsize=.5]

    node1 [label="-"]
    node2 [label="+"]
    node3 [label="14"]
    node4 [label="*"]
    node5 [label="2"]
  
```

```

node6 [label="3"]
node4 -> node5
node4 -> node6
node2 -> node3
node2 -> node4
node7 [label="/"]
node8 [label="6"]
node9 [label="2"]
node7 -> node8
node7 -> node9
node1 -> node2
node1 -> node7
}

```



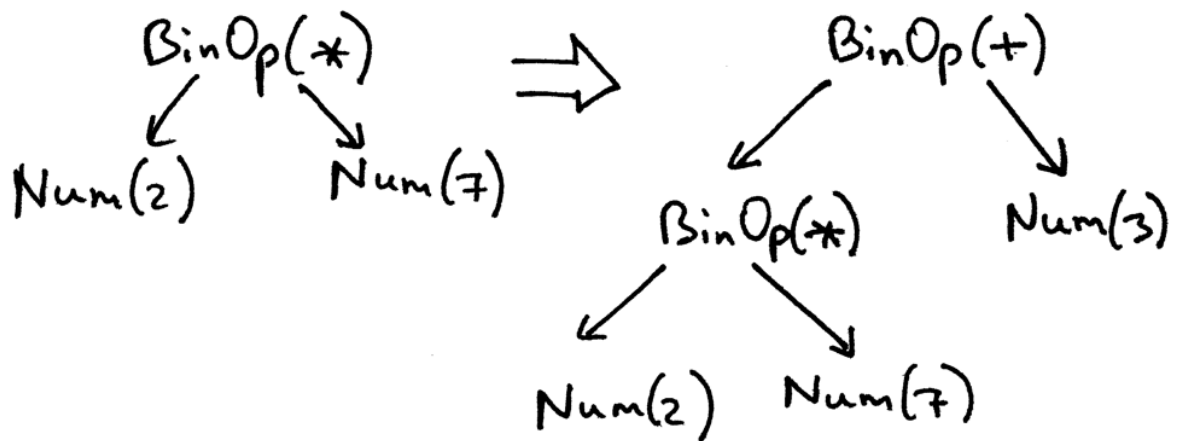
重点类代码展示：

```

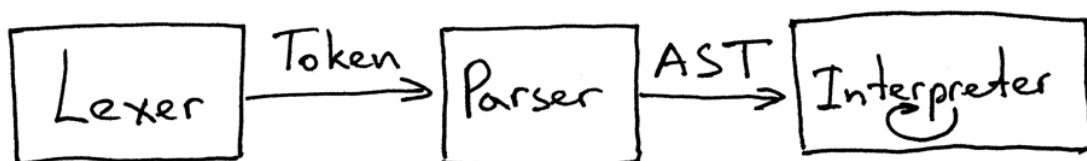
class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = rig

```

$$2 * 7 + 3$$



学习到这，我们可以清楚的发现我们的“编译器”可以看作三大模块：词法分析器、解析器和解释器；输入一个表达式，利用词法分析器(也可以说是扫描器)扫描得到每一个不同类别的记号（这里有整数、加减乘除和左右括号），利用解析器生成AST，最后使用解释器对AST操作得到最后结果。



## part8

这部分主要是解决添加负数(添加一元运算符)

$$5 - - - 2 = 5 - (-(-(-2))) = 5 - (-(-2)) = 5 - 2 = 3$$

Diagram illustrating the evaluation of the expression  $5 - - - 2$  (5 minus minus minus 2). The expression is shown with arrows pointing to the operators and the final result:

- $5$  is the first operand.
- $-$  is a **binary minus (subtraction)**.
- $-$  is a **unary minus (negation)**.
- $-$  is a **unary minus (negation)**.
- $2$  is the second operand.

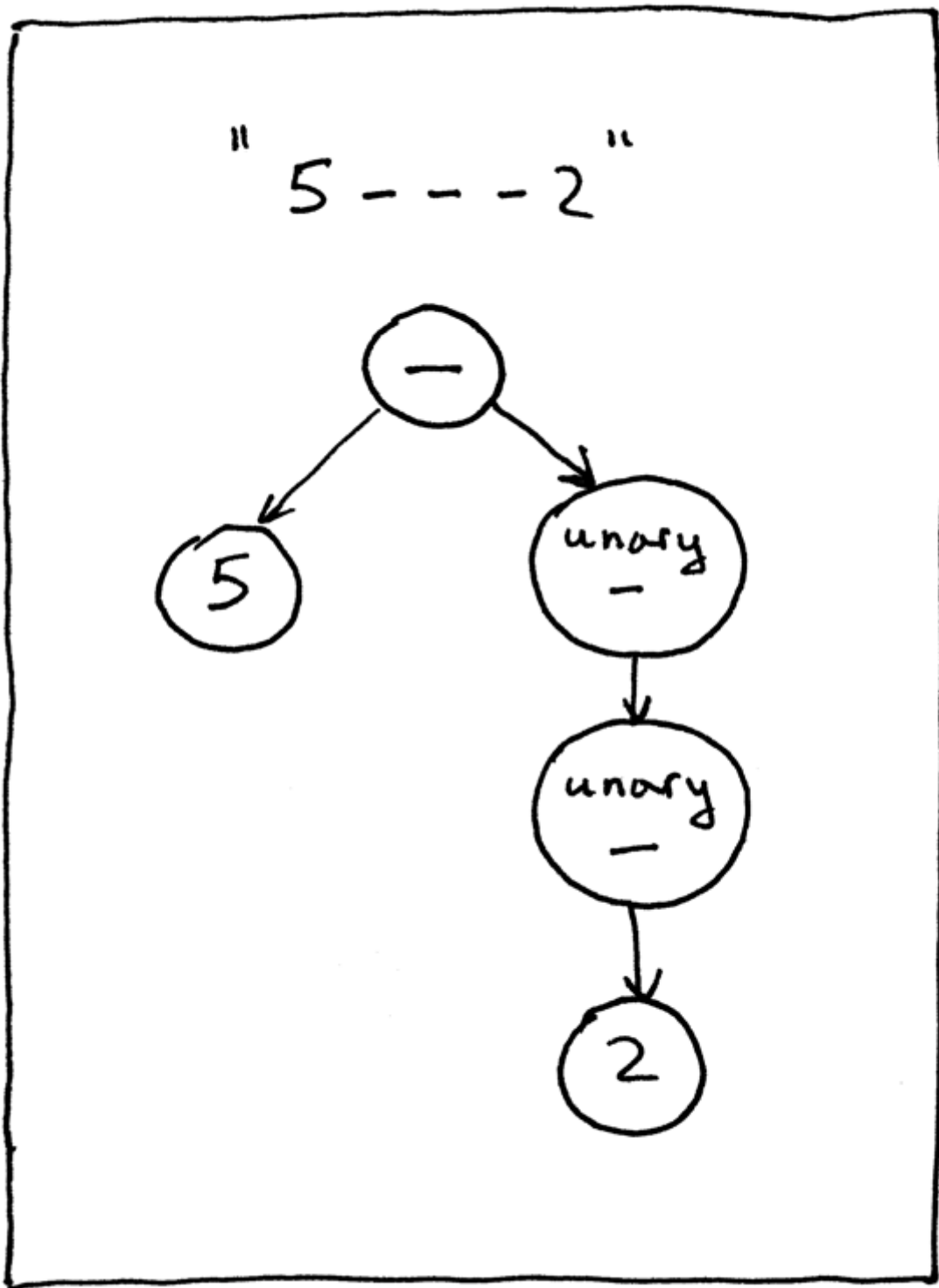
The evaluation steps are shown as:  $5 - (-(-(-2))) = 5 - (-(-2)) = 5 - 2 = 3$ .

主要修改部分：

```
class UnaryOp(AST):
    def __init__(self, op, expr):
        self.token = self.op = op
        self.expr = expr

-----
----

    def factor(self):
        """factor : (PLUS | MINUS) factor | INTEGER | LPAREN expr
        RPAREN"""
        token = self.current_token
        if token.type == PLUS:
            self.next(PLUS)
            node = UnaryOp(token, self.factor())
            return node
        elif token.type == MINUS:
            self.next(MINUS)
            node = UnaryOp(token, self.factor())
            return node
        elif token.type == INTEGER:
            self.next(INTEGER)
            return Num(token)
        elif token.type == LPAREN:
            self.next(LPAREN)
            node = self.expr()
            self.next(RPAREN)
            return node
```



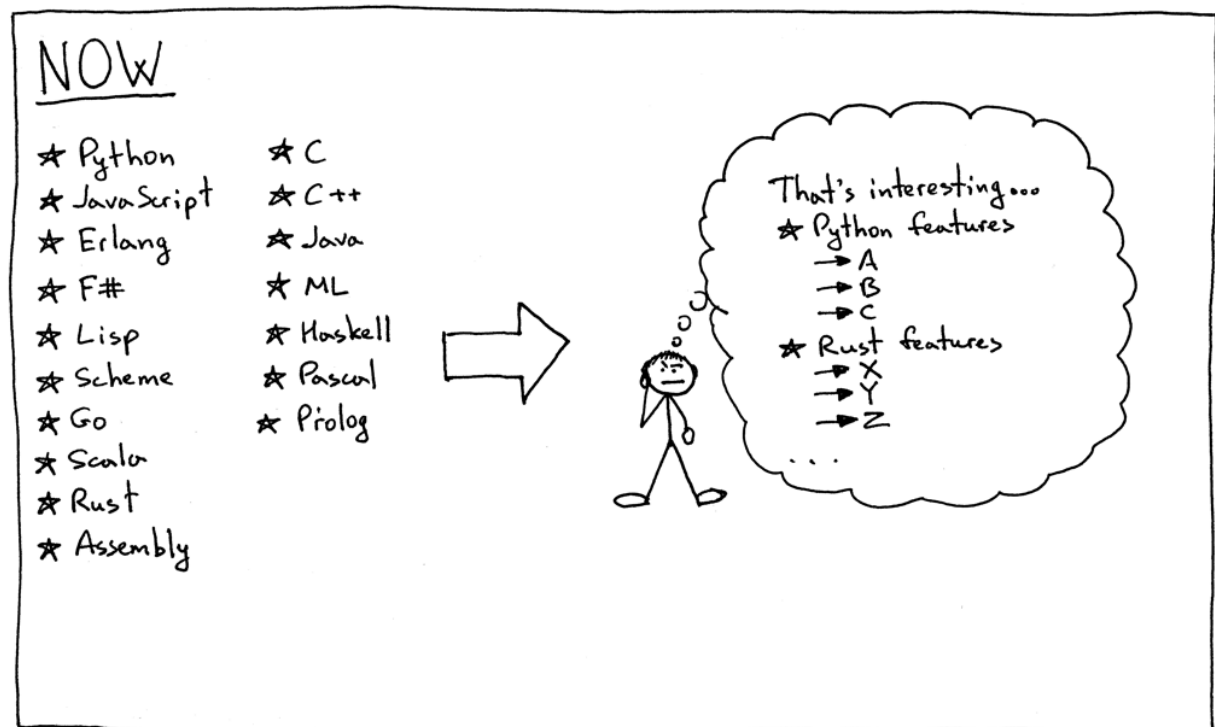
## part9

从这一部分开始，我们面对的不仅仅是一个计算器，还需要去解析Pascal类程序。这个部分与前面将是一个比较大的跳跃。

以下是这部分的内容：

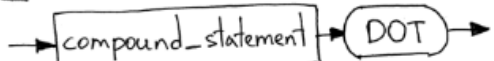
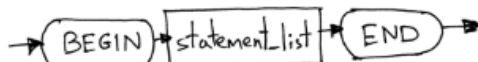
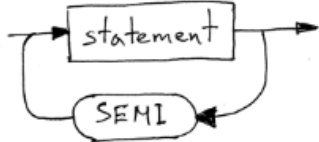
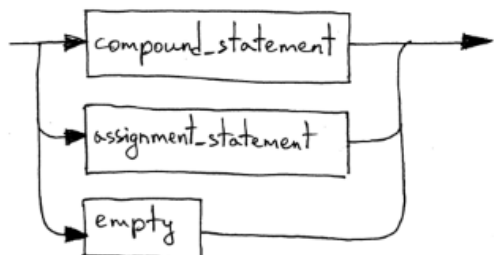
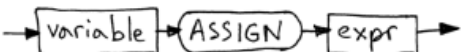

- 如何解析和解释Pascal程序定义。
- 如何解析和解释复合语句。
- 如何解析和解释赋值语句，包括变量。
- 关于符号表以及如何存储和查找变量。

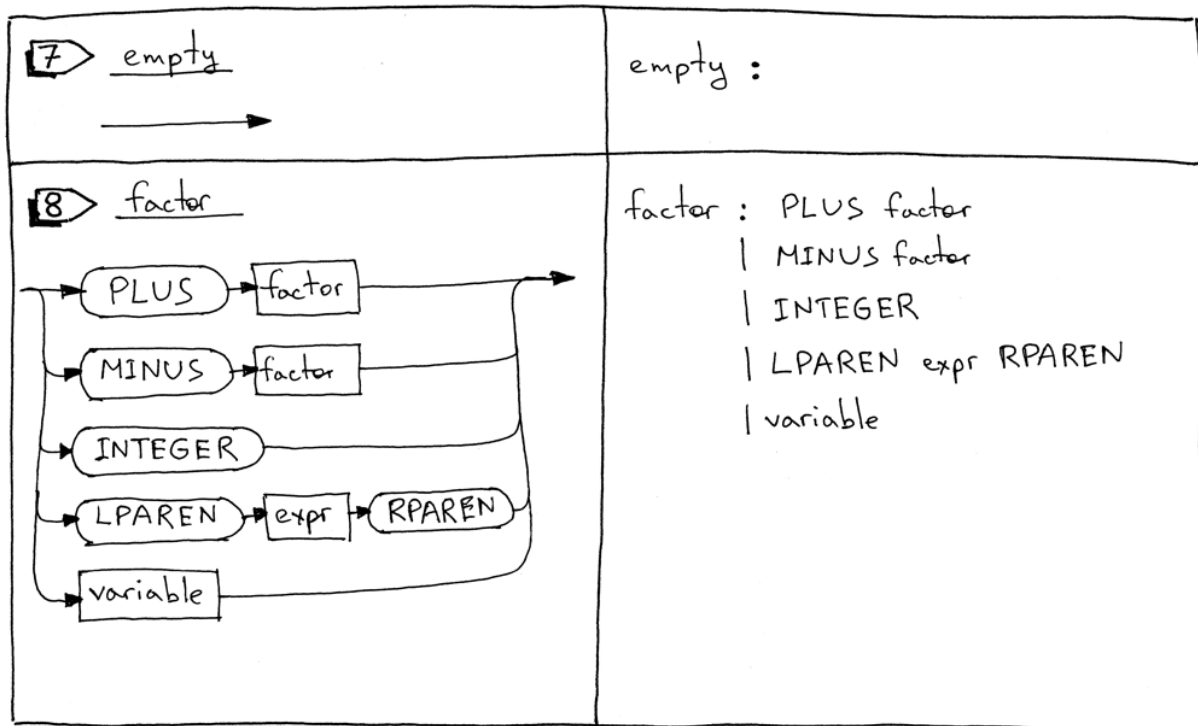




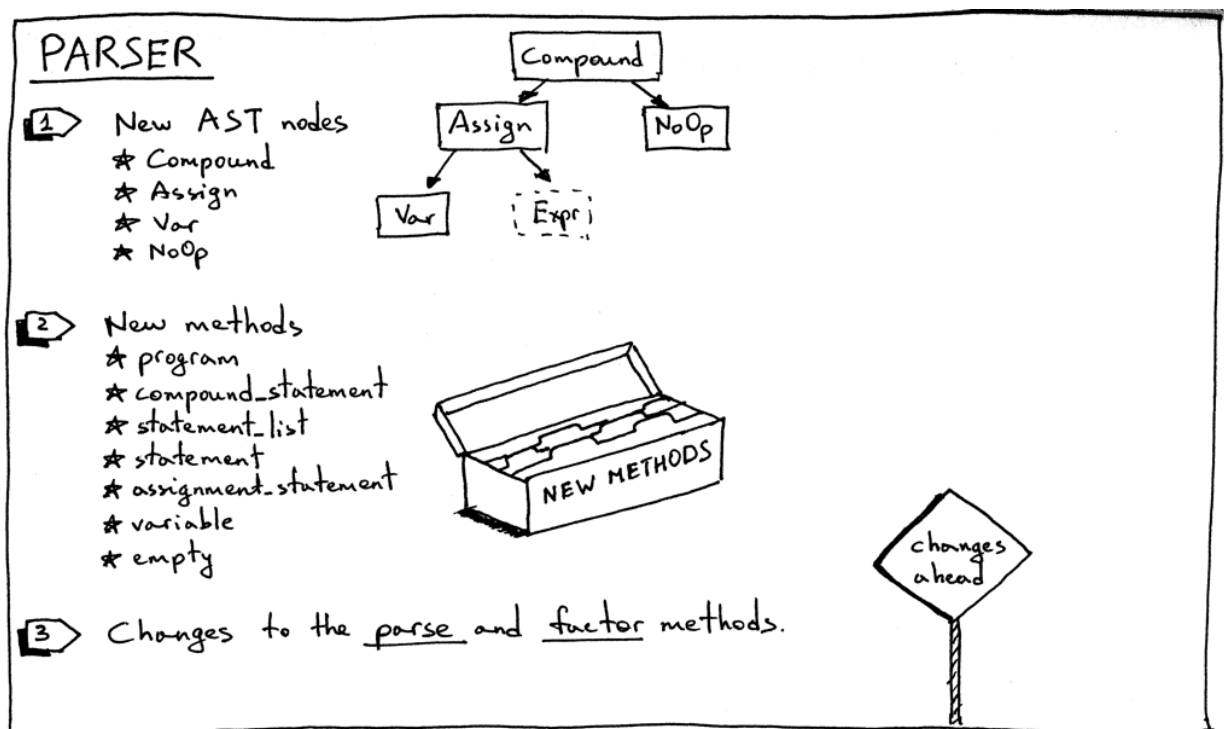
主要对标记进行的修改如下：

- BEGIN ( 标记复合语句的开始 )
- END ( 标记复合语句的结尾 )
- DOT ( Pascal程序定义所需的点字符'.'的标记 )
- ASSIGN ( 两个字符序列的标记':=' )。在Pascal中，赋值运算符与许多其他语言 ( 如C , Python , Java , Rust或Go ) 不同，在这些语言中，您将使用单个字符“=”来表示赋值
- SEMI ( 分号字符的标记';'用于标记复合语句中语句的结尾 )
- ID ( 有效标识符的标记。标识符以字母字符开头，后跟任意数量的字母数字字符 )

| SYNTAX DIAGRAM  | GRAMMAR RULE   |
|---|--|
| <p>1 <u>program</u></p>  <pre>graph LR; A[compound_statement] --&gt; B((DOT));</pre>   | <p>program : compound_statement DOT</p>                                      |
| <p>2 <u>compound_statement</u></p>  <pre>graph LR; A((BEGIN)) --&gt; B[statement_list]; B --&gt; C((END));</pre>   | <p>compound_statement : BEGIN statement_list END</p>                         |
| <p>3 <u>statement_list</u></p>  <pre>graph LR; A[statement] --&gt; B((SEMI)); B --&gt; A;</pre>  | <p>statement_list : statement<br/>  statement SEMI statement_list</p>        |
| <p>4 <u>statement</u></p>  <pre>graph LR; A[compound_statement] --&gt; J(( )); B[assignment_statement] --&gt; J; C[empty] --&gt; J; J --&gt; Exit(( ));</pre> | <p>statement : compound_statement<br/>  assignment_statement<br/>  empty</p> |
| <p>5 <u>assignment_statement</u></p>  <pre>graph LR; A[variable] --&gt; B((ASSIGN)); B --&gt; C[expr];</pre>   | <p>assignment_statement : variable ASSIGN expr</p>                           |
| <p>6 <u>variable</u></p>  <pre>graph LR; A((ID)) --&gt; Exit(( ));</pre>   | <p>variable : ID</p>   |



关于解析器的更改摘要：



实例：

pascal实例：

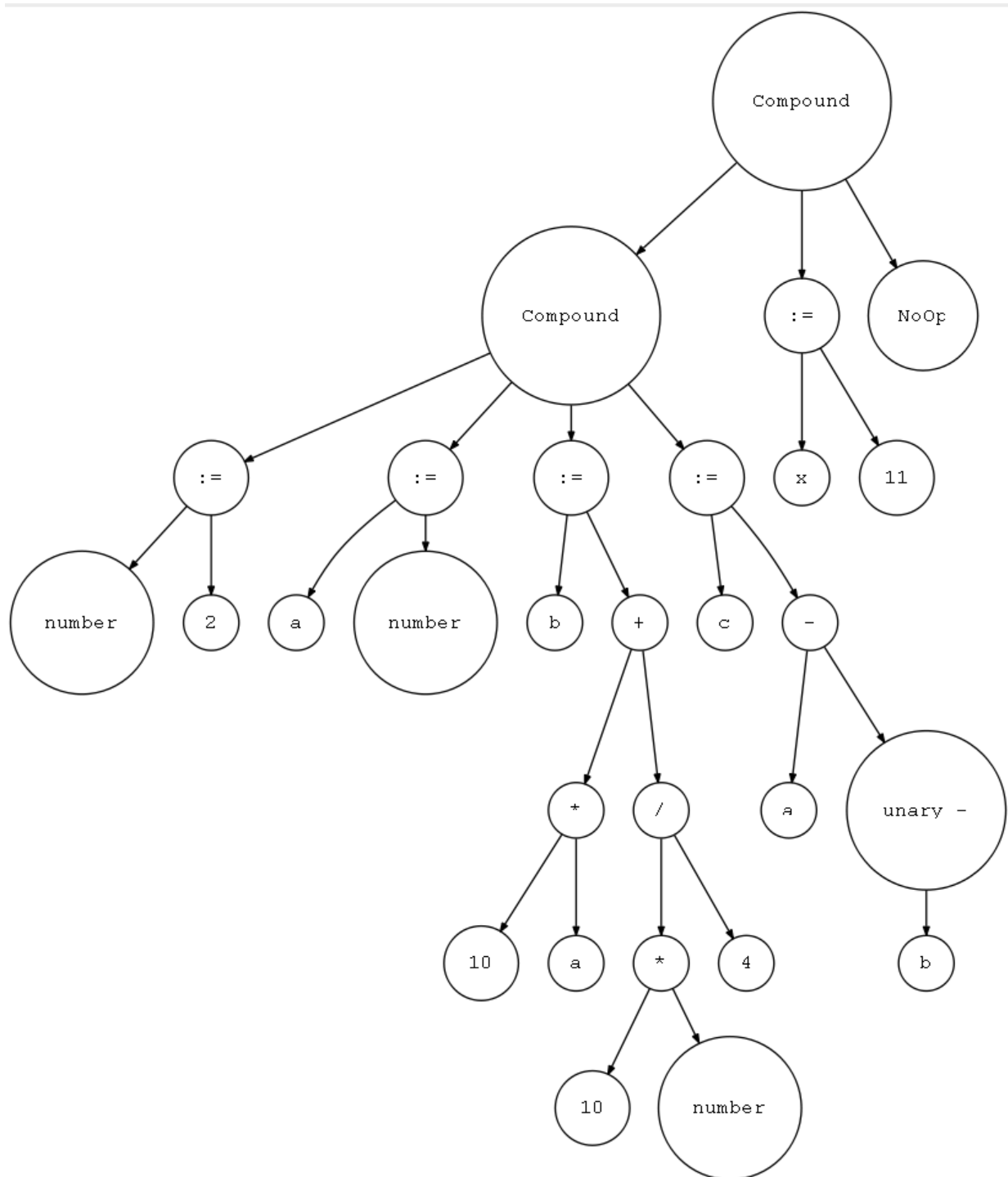
```

BEGIN

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

```
END;  
  
x := 11;  
END.
```

转换为AST图的例子



## part10

这部分我们需要解析的是以下几个方面：

- 我们将学习如何解析和解释Pascal PROGRAM 头
- 我们将学习如何解析Pascal变量声明(declarations)
- 我们将更新我们的解释器以使用DIV关键字进行整数除法和正斜杠/浮点除法

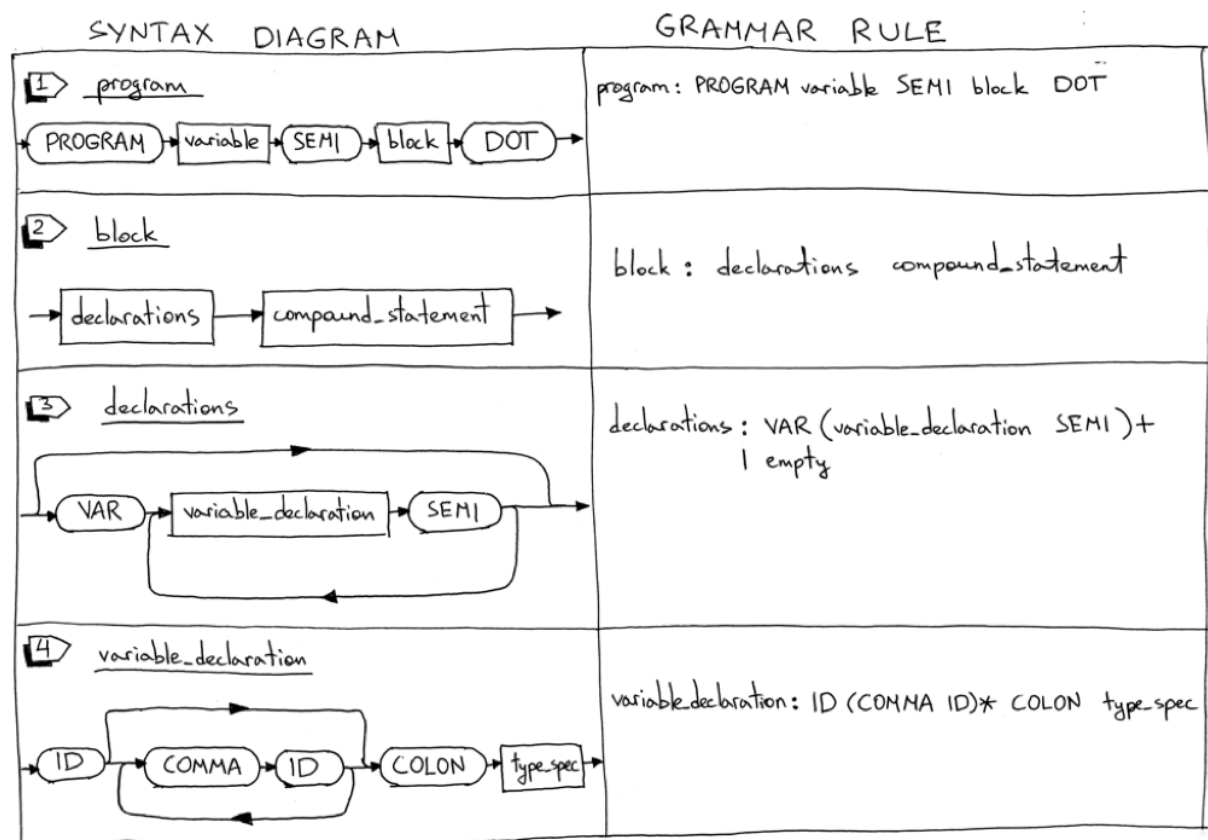
- 我们将添加对Pascal注释的支持

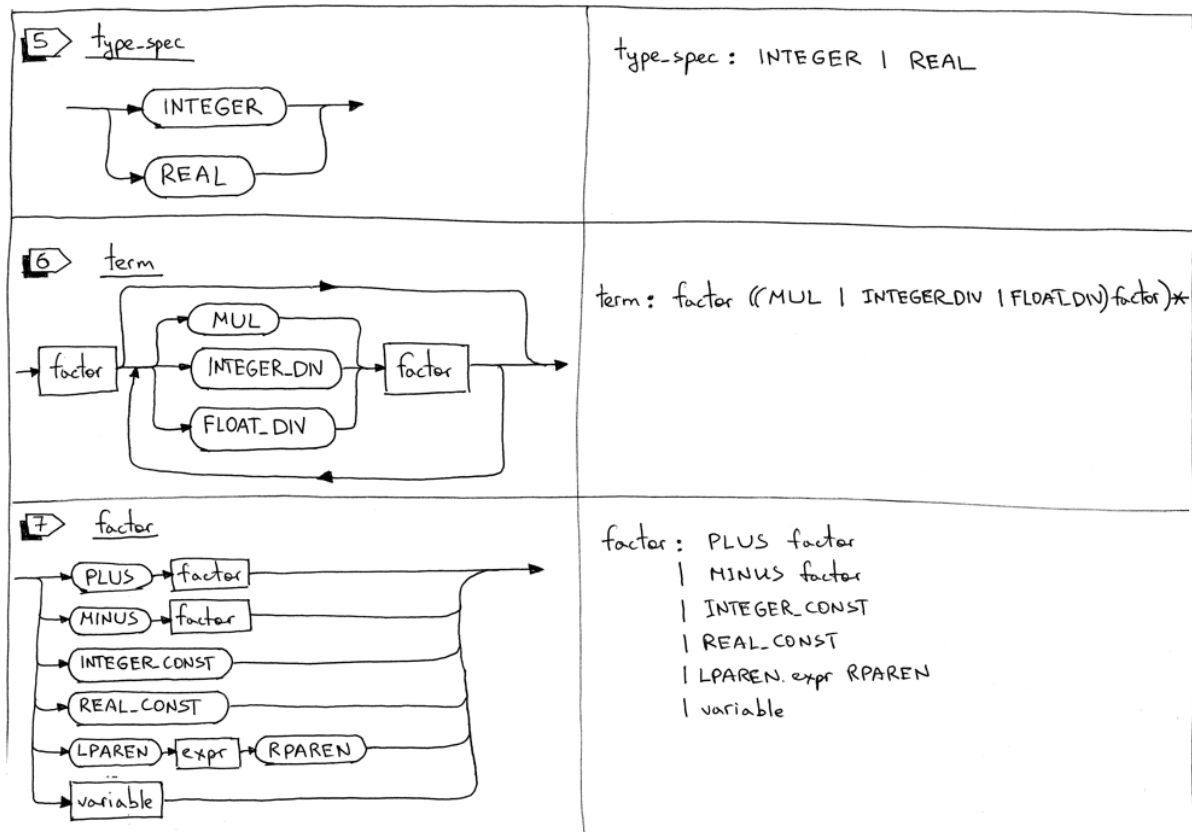
增加的一些关键字：

```

LPAREN      = 'LPAREN'  # (
RPAREN      = 'RPAREN'  # )
.....
SEMI        = 'SEMI'    # ;
DOT         = 'DOT'     # .
PROGRAM     = 'PROGRAM'
VAR         = 'VAR'
COLON       = 'COLON'   # :
COMMA       = 'COMMA'   # ,
EOF         = 'EOF'
  
```

首先需要做的是更新语法规则：





详细语法规则：

```

program : PROGRAM variable SEMI block DOT

block : declarations compound_statement

declarations : VAR (variable_declaration SEMI)+
              | empty

variable_declaration : ID (COMMA ID)* COLON type_spec

type_spec : INTEGER | REAL

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 | INTEGER_DIV | FLOAT_DIV) factor)*
  
```

```

    factor : PLUS factor
          | MINUS factor
          | INTEGER_CONST
          | REAL_CONST
          | LPAREN expr RPAREN
          | variable
    variable: ID

```

更新Lexer词法分析器：

- New tokens
- New and updated reserved keywords
- New skip\_comment method to handle Pascal comments
- Rename the integer method and make some changes to the method itself
- Update the get\_next\_token method to return new tokens

更改解析器：

- 新的AST节点：Program , Block , VarDecl , Type
- 与新语法规则相对应的新方法：block , declaration , variable\_declaration和type\_spec。
- 对现有解析器方法的更新：program, term, and factor

更新解释器：

有四种新方法来访问我们的新节点：

- visit\_Program
- visit\_Block
- visit\_VarDecl
- visit\_Type

实例展示：

```

PROGRAM Part10;
VAR
    number      : INTEGER;
    a, b, c, x  : INTEGER;
    y           : REAL;

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

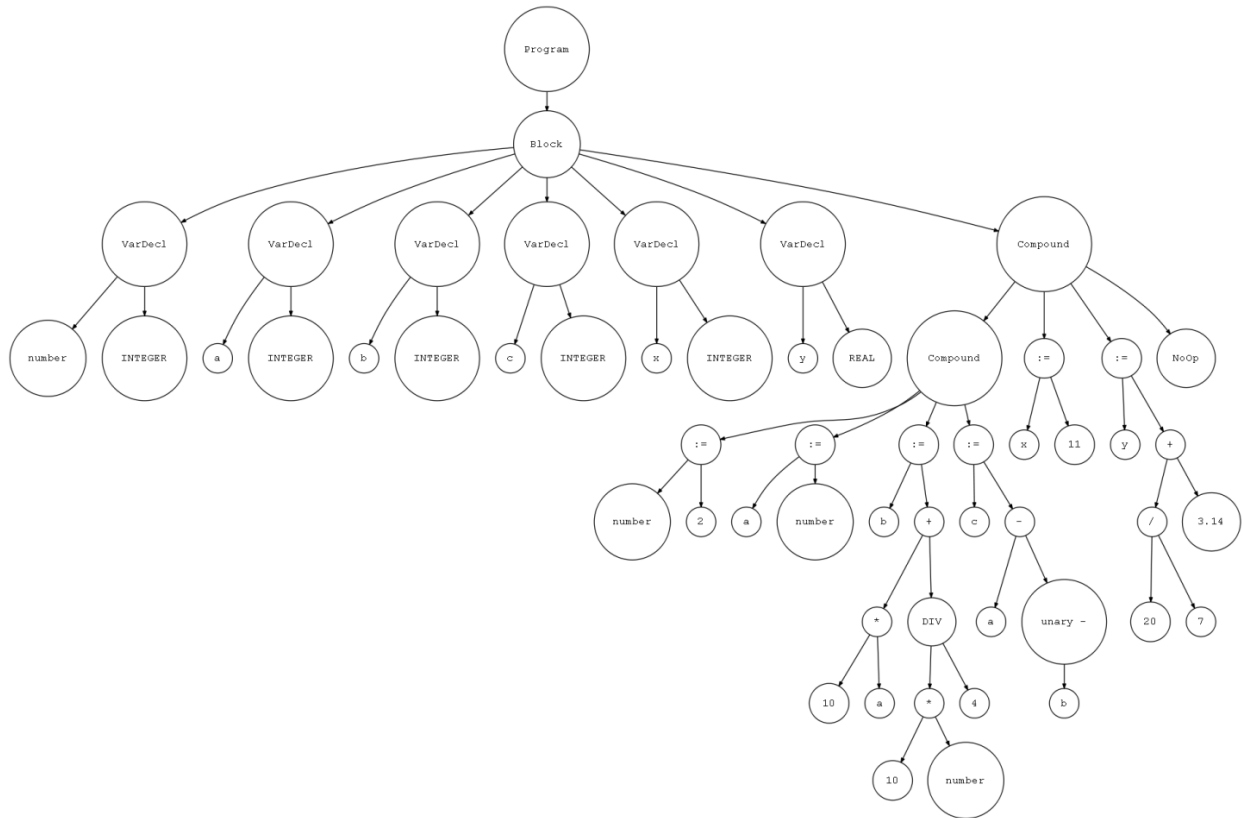
```

```

{ writeln('b = ', b); }
{ writeln('c = ', c); }
{ writeln('number = ', number); }
{ writeln('x = ', x); }
{ writeln('y = ', y); }
END. {Part10}

```

如图：



## part11

本part主要是为编译器添加符号表，以用来检查所有的符号是否已经声明，在part10中若使用未声明变量不会报错，而本部分主要通过符号表来解决这个问题。

symbols and Symbol Table

```

#####
#####
#
#
# SYMBOLS and SYMBOL TABLE
#
#
#
#####
#####

class Symbol(object):
    def __init__(self, name, type=None):

```



```

        self.name = name
        self.type = type

class VarSymbol(Symbol):
    def __init__(self, name, type):
        super().__init__(name, type)

    def __str__(self):
        return '<{name}:{type}>'.format(name=self.name, type=self.type)

    __repr__ = __str__

class BuiltinTypeSymbol(Symbol):
    def __init__(self, name):
        super().__init__(name)

    def __str__(self):
        return self.name

    __repr__ = __str__

class SymbolTable(object):
    def __init__(self):
        self._symbols = {}
        self._init_builtins()

    def _init_builtins(self):
        self.define(BuiltinTypeSymbol('INTEGER'))
        self.define(BuiltinTypeSymbol('REAL'))

    def __str__(self):
        s = 'Symbols: {symbols}'.format(
            symbols=[value for value in self._symbols.values()]
        )
        return s

    __repr__ = __str__

    def define(self, symbol):
        print('Define: %s' % symbol)
        self._symbols[symbol.name] = symbol

    def lookup(self, name):
        print('Lookup: %s' % name)
        symbol = self._symbols.get(name)
        # 'symbol' is either an instance of the Symbol class or 'None'
        return symbol

```

```

class SymbolTableBuilder(NodeVisitor):
    def __init__(self):
        self.symtab = SymbolTable()

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

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

    def visit_BinOp(self, node):
        self.visit(node.left)
        self.visit(node.right)

    def visit_Num(self, node):
        pass

    def visit_UnaryOp(self, node):
        self.visit(node.expr)

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

    def visit_NoOp(self, node):
        pass

    def visit_VarDecl(self, node):
        type_name = node.type_node.value
        type_symbol = self.symtab.lookup(type_name)
        var_name = node.var_node.value
        var_symbol = VarSymbol(var_name, type_symbol)
        self.symtab.define(var_symbol)

    def visit_Assign(self, node):
        var_name = node.left.value
        var_symbol = self.symtab.lookup(var_name)
        if var_symbol is None:
            raise NameError(repr(var_name))

        self.visit(node.right)

    def visit_Var(self, node):
        var_name = node.value
        var_symbol = self.symtab.lookup(var_name)

        if var_symbol is None:
            raise NameError(repr(var_name))

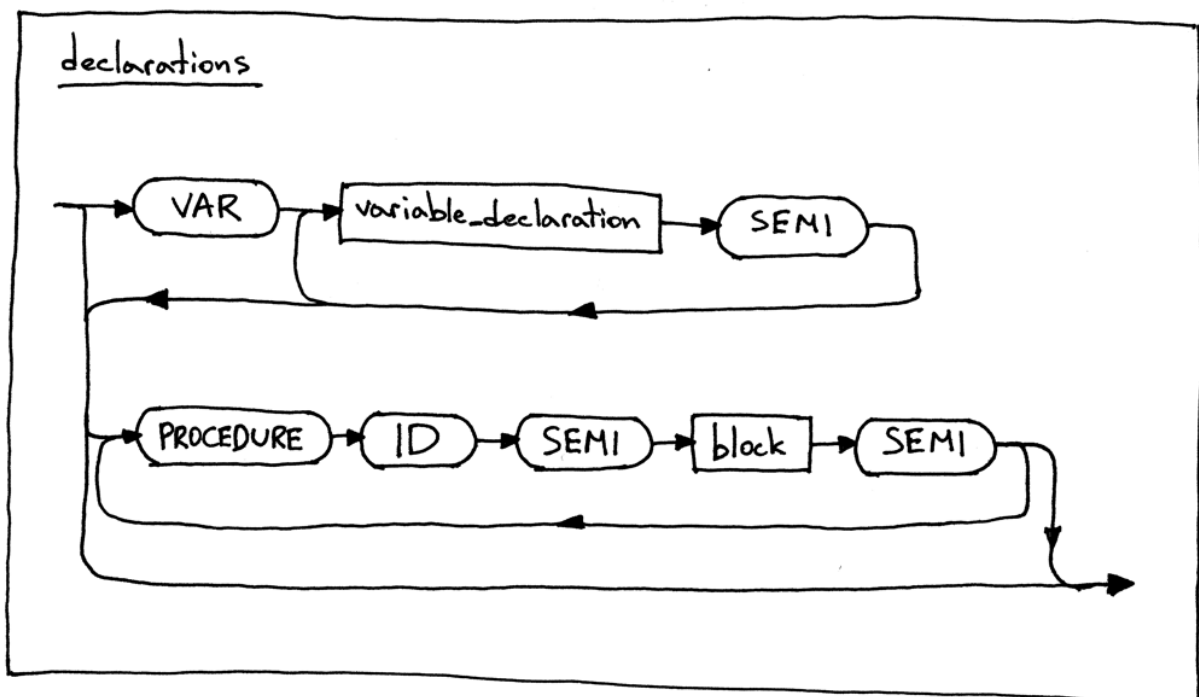
```

## part12

这部分主要是如何解析嵌套程序。

首先需要更新一下声明部分的语法，以便声明程序：

declarations : VAR (variable\_declaration SEMI)+  
              | (PROCEDURE ID SEMI block SEMI)\*  
              | empty



先是procedure再声明对应procedure上的变量等等，有点像program的子程序的感觉，类似如下形式：

```
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}

```

主要操作是对声明部分添加解析和添加procedure的节点，以下是一个例子：

```

PROGRAM Part12;
VAR
    a : INTEGER;

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

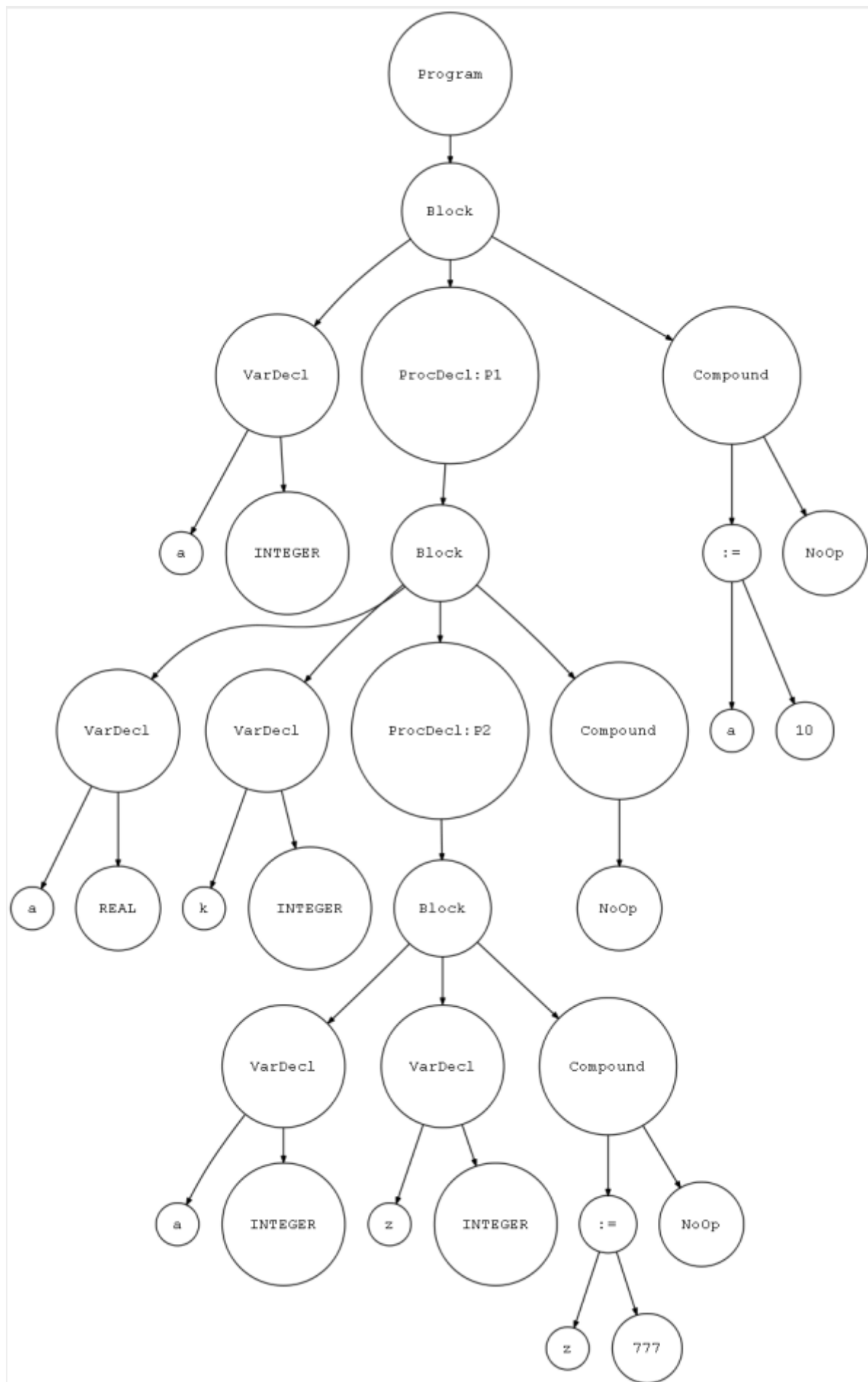
PROCEDURE P2;
VAR
    a, z : INTEGER;
BEGIN {P2}
    z := 777;
END; {P2}

BEGIN {P1}

END; {P1}

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

```



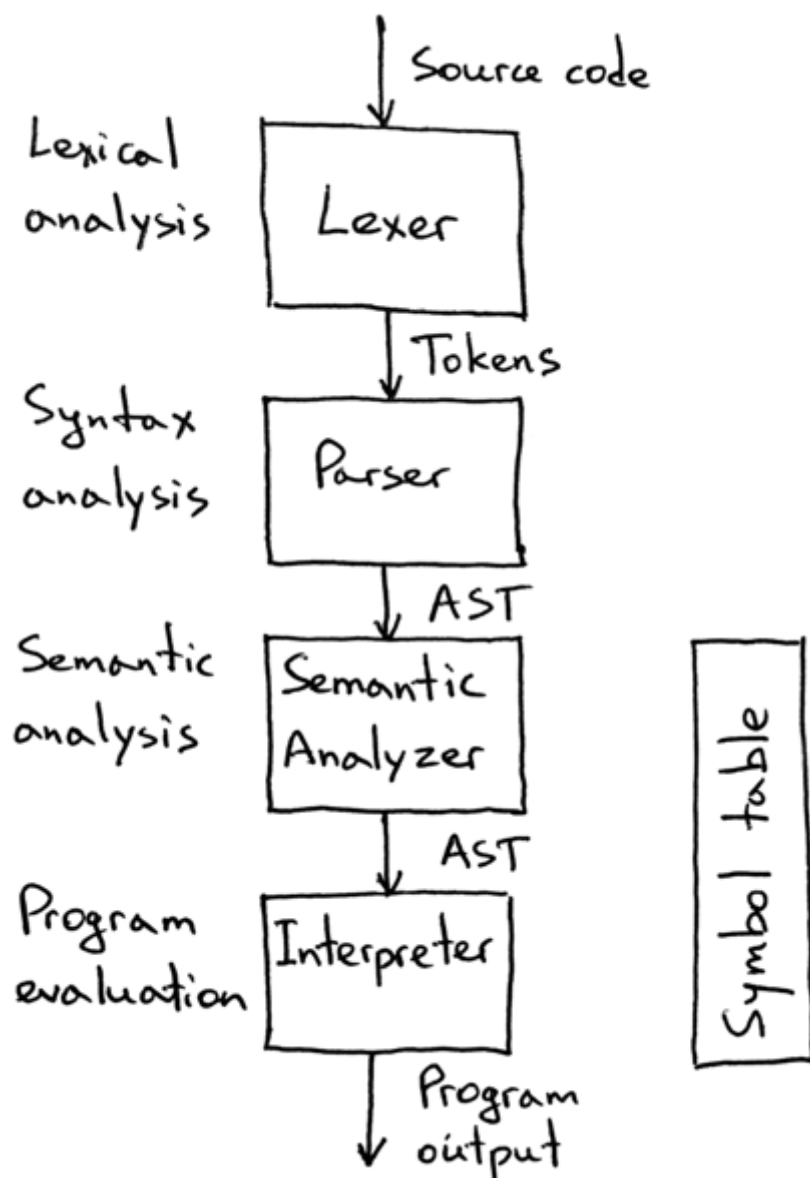
这一部分主要介绍语义分析部分

关于语义分析我们主要有一些特别的要求：

- 必须在使用变量之前声明变量
- 当在算术表达式中使用时，变量必须具有匹配类型（这是语义分析的一个重要部分，称为类型检查，我们将单独介绍）
- 应该没有重复的声明（Pascal禁止，例如，在一个过程中有一个局部变量，其名称与过程的形式参数之一相同）
- 在对过程的调用中的名称引用必须引用实际声明的过程（如果在过程调用foo（）中，名称foo引用基本类型INTEGER的变量foo，则在Pascal中没有意义）
- 过程调用必须具有正确数量的参数，并且参数的类型必须与过程声明中的形式参数的类型匹配

当我们有足够的关于程序的上下文时，执行上述要求要容易得多，即我们可以走的AST形式的中间表示和带有关于变量，procedure和函数等不同程序实体的信息的符号表。

Pascal解释器的结构将如下所示



从上图中你可以看到我们的词法分析器会将源代码作为输入，将其转换为解析器将使用的标记，并用于验证程序在语法上是否正确，然后它将生成一个新的抽象语法树语义分析阶段将用于强制执行

不同的Pascal语言要求。在语义分析阶段，语义分析器还将构建和使用符号表。在语义分析之后，我们的解释器将采用AST，通过遍历AST来评估程序，并生成程序输出。

语义分析检查可以分为静态语义检查和动态语义检查，这里先介绍静态语义检查。

静态语义检查是我们在解释（评估）程序之前可以进行的检查，也就是在对Interpreter类的实例调用解释方法之前。之前提到的所有Pascal要求都可以通过使用AST并使用符号表中的信息进行静态语义检查来强制执行。

另一方面，动态语义检查需要在程序的解释（评估）期间执行检查。例如，检查没有除零，并且数组索引不超出范围将是动态语义检查。

我们今天主要解决的部分是静态语义检查。

- 变量在使用之前声明
- 没有重复的变量声明
- 添加大小写关键字都满足的情况
- 修改declarations()函数，让其满足多个var定义，类似于：  
`var x, y : integer; var y : real;`

## part14

这部分也是一个进阶，主要有以下内容：

- 如何解析定义一个scopes(范围)。
- 范围和范围符号表
- 带有形式参数的过程声明
- 程序符号
- 嵌套范围
- 范围树：链接范围符号表
- 嵌套范围和名称解析
- 源到源编译器

关于范围

PROGRAM关键字表示该作用域为一个全局作用域；当我们谈论变量的范围时，我们实际上谈论了它的声明范围。除了全局作用域之外，Pascal还支持嵌套过程，并且每个过程声明都引入了一个新作用域，这意味着Pascal支持嵌套作用域。

例子：

```
program Main;
  var x, y : real;
  var z : integer;

  procedure AlphaA(a : integer);
    var y : integer;
  begin { AlphaA }
    x := a + x + y;
```

```

end; { AlphaA }

procedure AlphaB(a : integer);
  var b : integer;
begin { AlphaB }
end; { AlphaB }

begin { Main }
end. { Main }

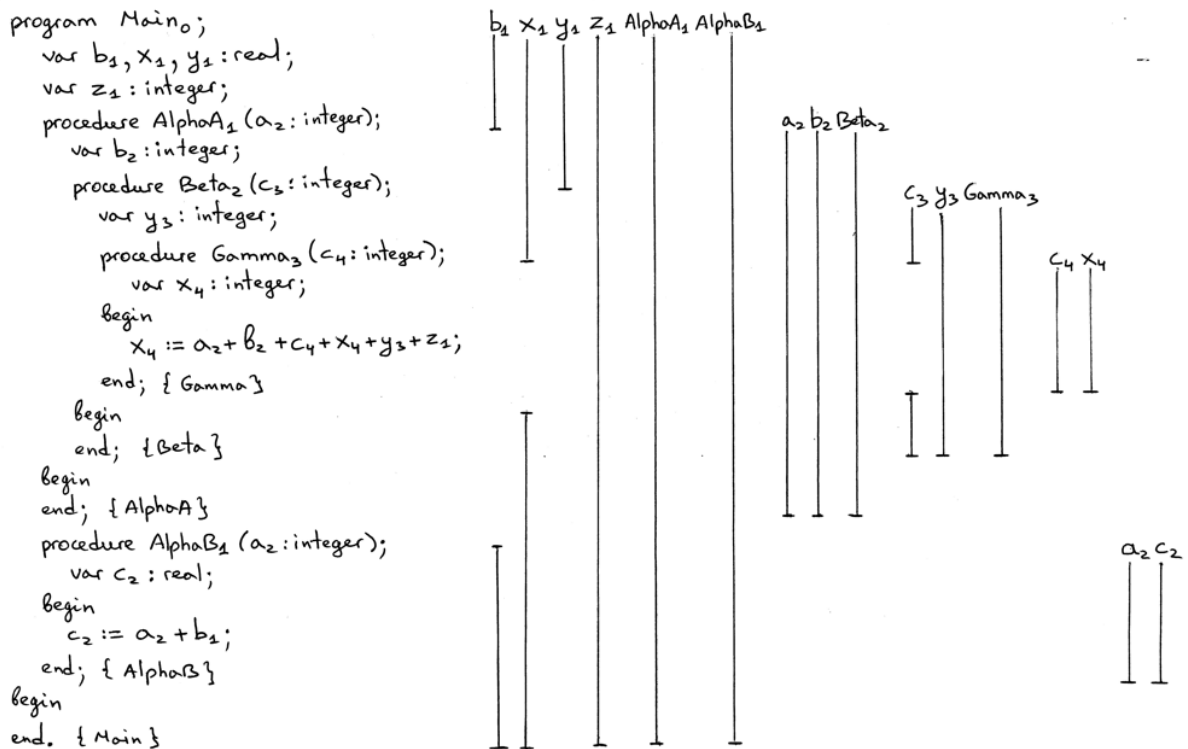
```



从上面这个例子我们可以看到在AST上主要是添加了param节点，除此之外，我们还设置了优先级和通过设定封闭节点来确定某变量的范围。

下图是一个程序中变量的范围：





## part15

代码升级大改造~

词法分析器改造：

- 我们将添加错误代码和自定义异常：LexerError，ParserError和SemanticError
- 我们将向Lexer类添加新成员以帮助跟踪令牌的位置：lineno和column
- 我们将修改advance方法来更新词法分析器的lineno和column 变量
- 我们将更新错误方法以使用有关当前行和列的信息引发LexerError异常
- 我们将在TokenType枚举类中定义令牌类型（在Python 3.4中添加了对枚举的支持）
- 我们将添加代码以自动从TokenType枚举成员创建保留关键字
- 我们将向Token类中添加新成员：lineno和column，以便相应地在文本中跟踪令牌的行号和列号
- 我们将重构get\_next\_token方法代码以使其更短并具有处理单字符令牌的通用代码

解析器代码改造：

- 我们将更新解析器的错误方法，以抛出带有错误代码和当前令牌的ParserError异常
- 我们将更新next方法来调用修改后的错误 方法
- 我们将重构声明方法并将解析过程声明的代码移动到单独的方法中。

语义分析器代码改造：

- 我们将向SemanticAnalyzer类添加一个新的错误方法，以抛出带有一些附加信息的SemanticError异常
- 我们将通过调用带有相关错误代码和令牌的错误方法来更新visit\_VarDecl以发出错误信号
- 我们还将更新visit\_Var以通过使用相关错误代码和令牌调用error方法来发出错误信号
- 我们将为ScopedSymbolTable和SemanticAnalyzer添加一个log方法，并在相应的类中调用self.log替换所有print语句

- 我们将添加一个命令行选项“-scope”来打开和关闭范围记录（默认情况下它将关闭）来控制我们希望我们的解释器如何“嘈杂”
- 我们将添加空的visit\_Num和visit\_UnaryOp 方法

## part16

更新解析器，，以便能够解析过程调用并构建正确的AST：

- 我们需要添加一个新的AST节点来表示一个过程调用
- 我们需要为过程调用语句添加一个新的语法规则; 那么我们需要在代码中实现规则
- 我们需要扩展语句语法规则以包含过程调用语句的规则并更新语句方法以反映语法中的更改