让我们构建一个简单的解释器。第 11 部分。 (https://ruslanspivak.com/lsbasi-part11/)

日期 2016 年 9 月 20 日，星期二

前几天我坐在我的房间里，想着我们已经涵盖了多少，我想我会回顾一下我们到目前为止学到的东西以及我们面前的东西。

到目前为止，我们已经学会了：

- 如何将句子分解为标记。该过程称为**词法分析**，而解释器中执行此操作的部分称为**词法分析器**、**词法分析器**、**扫描器**或**标记器**。我们已经学会了如何在不使用正则表达式或任何其他工具（如Lex ）的 (https://en.wikipedia.org/wiki/Lex_(software))情况下从头开始编写我们自己的**词法分析器**。 (https://en.wikipedia.org/wiki/Lex_(software))
- 如何识别令牌流中的短语。在标记流中识别短语的过程，或者换句话说，在标记流中寻找结构的过程称为**解析**或**语法分析**。执行该工作的解释器或编译器的部分称为**解析器**或**语法分析器**。
- 如何用**语法图**来表示编程语言的语法规则，**语法图**是一种编程语言的语法规则的图形表示。 **语法图**直观地向我们展示了哪些语句在我们的编程语言中是允许的，哪些是不允许的。
- 如何使用另一种广泛使用的符号来指定编程语言的语法。它被称为**上下文无关文法**（简称为**文法**）或**BNF**（巴科斯-诺尔形式）。

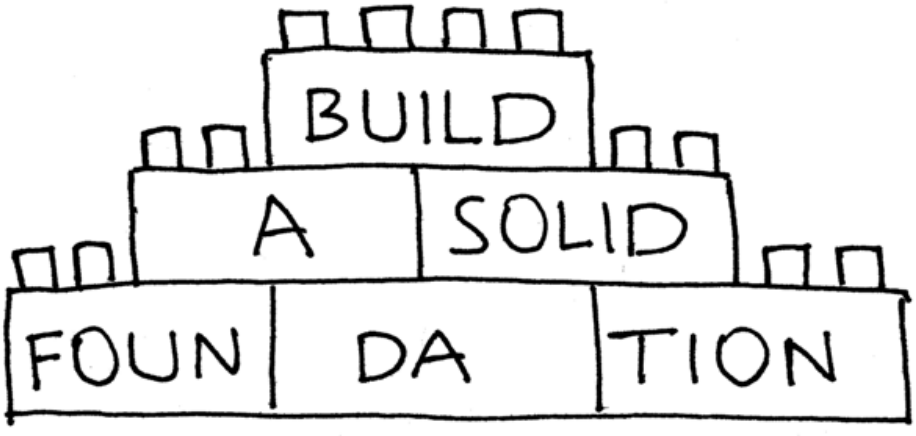
- 如何将**语法**映射到代码以及如何编写**递归下降解析器**。
- 如何编写一个非常基本的**解释器**。
- 如何**结合性**和**优先级**运营商的工作，以及如何构建使用优先顺序表的语法。
- 如何构建已解析句子的**抽象语法树**(AST)，以及如何在 Pascal 中将整个源程序表示为一个大**AST**。
- 如何走AST以及如何将我们的解释器实现为AST节点访问者。

凭借我们掌握的所有知识和经验，我们构建了一个可以扫描、解析和构建AST 的解释器，并通过执行AST 来解释我们第一个完整的 Pascal 程序。女士们，先生们，老实说，如果你们已经走到了这一步，那么你们应该得到一个鼓励。但不要让它冲昏头脑。继续。尽管我们已经涵盖了很多领域，但还有更多令人兴奋的部分即将到来。

到目前为止，我们已经涵盖了所有内容，我们几乎已准备好处理以下主题：

- 嵌套过程和函数
- 过程和函数调用
- 语义分析（类型检查，确保在使用变量之前声明变量，并基本上检查程序是否有意义）
- 控制流元素（如IF 语句）
- 聚合数据类型（记录）
- 更多内置类型
- 源码级调试器
- 杂项（上面没有提到的所有其他优点:)

但在我们讨论这些主题之前，我们需要建立一个坚实的基础和基础设施。



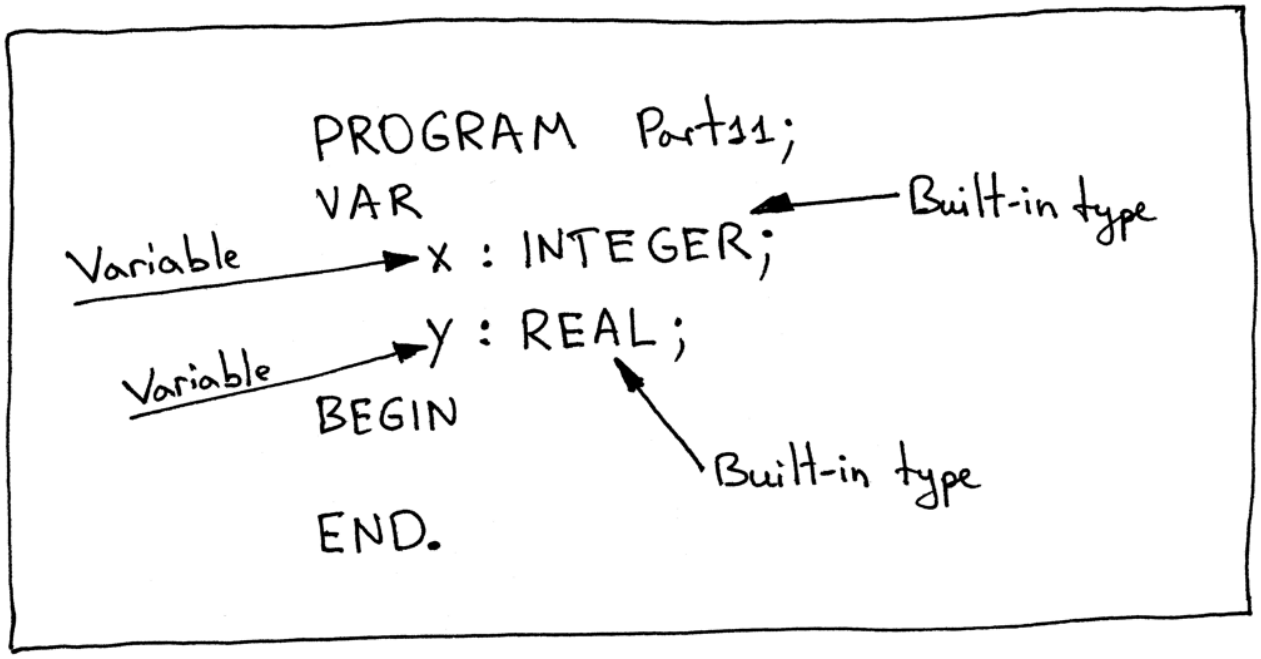
这是我们开始深入研究符号、符号表和范围的超级重要主题的地方。该主题本身将跨越多篇文章。这很重要，你会明白为什么。好的，让我们开始建立基础和基础设施，好吗？

首先，让我们谈谈符号以及我们为什么需要跟踪它们。什么是**符号**？出于我们的目的，我们将非正式地将**符号**定义为某些程序实体的标识符，例如变量、子例程或内置类型。要使符号有用，它们至少需要具有关于它们识别的程序实体的以下信息：

- 名称（例如，“x”、“y”、“数字”）
- 类别（它是变量、子程序还是内置类型？）
- 类型（INTEGER，REAL）

今天我们将处理变量符号和内置类型符号，因为我们之前已经使用过变量和类型。顺便说一下，“内置”类型只是指一种尚未由您定义并且开箱即用的类型，例如您以前见过和使用过的INTEGER和REAL类型。

下面我们来看看下面的Pascal程序，特别是变量声明部分。您可以在下图中看到该部分有四个符号：两个变量符号（x和y）和两个内置类型符号（INTEGER和REAL）。



我们如何在代码中表示符号？让我们在 Python 中创建一个基本的Symbol类：

```
class  Symbol ( object ):
    def  __init__ ( self ,  name ,  type = None ):
        self 。姓名 = 姓名
        自我。类型 = 类型
```

如您所见，该类采用name参数和一个可选的type参数（并非所有符号都可能具有与其关联的类型）。符号的类别呢？我们将在类名本身中对符号的类别进行编码，这意味着我们将创建单独的类来表示不同的符号类别。

让我们从基本的内置类型开始。到目前为止，当我们声明变量时，我们已经看到了两种内置类型：INTEGER和REAL。我们如何在代码中表示内置类型符号？这是一种选择：

```
class  BuiltinTypeSymbol ( Symbol ):
    def  __init__ ( self ,  name ):
        super () 。__init__ (名称)

    def  __str__ ( self ):
        返回 self 。姓名

    __repr__  =  __str__
```

该类继承自Symbol类，构造函数只需要该类型的名称。类别在类名中编码，内置类型符号的基类的类型参数是None。双下划线或dunder（如“Double UNDERscore”）方法__str__和__repr__是特殊的 Python 方法，我们将它们定义为在打印符号对象时具有良好的格式化消息。

下载解释器文件(https://github.com/rspivak/lsbasi/blob/master/part11/python/spi.py)并保存为spi.py；从保存 spi.py 文件的同一目录启动一个 python shell，并以交互方式使用我们刚刚定义的类：

```
$蟒蛇
>>> from spi import BuiltinTypeSymbol
>>> int_type  = BuiltinTypeSymbol ( 'INTEGER' )
>>> int_type
整数
>>> real_type  = BuiltinTypeSymbol ( 'REAL' )
>>> 真实类型
真实的
```

我们如何表示变量符号？让我们创建一个VarSymbol 类：

```
class  VarSymbol ( Symbol ):
    def  __init__ ( self ,  name ,  type ):
        super () 。__init__ （名称， 类型）

    def  __str__ ( self ):
        返回 '<{name}:{type}>' 。格式（名称=自我。名称， 类型=自我。类型）

    __repr__  =  __str__
```

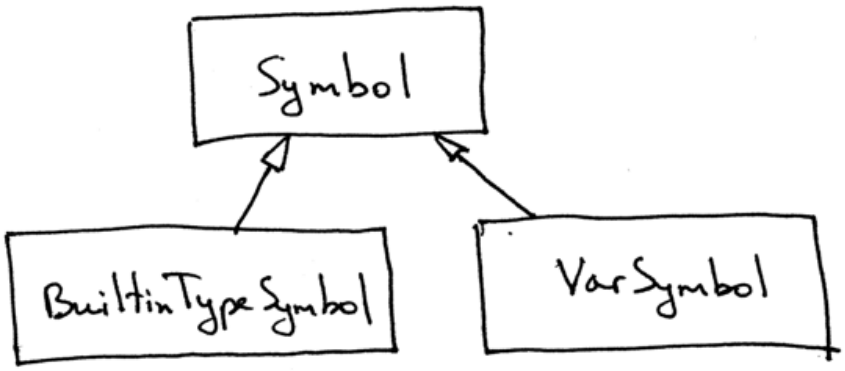
在类中我们将名称和类型参数都设为必需参数，类名VarSymbol清楚地表明类的实例将标识一个变量符号（类别为variable。）

回到交互式 python shell，看看我们如何为我们的变量符号手动构造实例，现在我们知道如何构造BuiltinTypeSymbol类实例：

```
$蟒蛇
>>> from spi import BuiltinTypeSymbol, VarSymbol
>>> int_type  = BuiltinTypeSymbol ( 'INTEGER' )
>>> real_type  = BuiltinTypeSymbol ( 'REAL' )
>>>
>>> var_x_symbol  = VarSymbol ( 'x' , int_type )
>>> var_x_symbol
<x:整数>
>>> var_y_symbol  = VarSymbol ( 'y' , real_type )
>>> var_y_symbol
<y:真实>
```

如您所见，我们首先创建一个内置类型符号的实例，然后将其作为参数传递给VarSymbol的构造函数。

这是我们以视觉形式定义的符号层次结构：

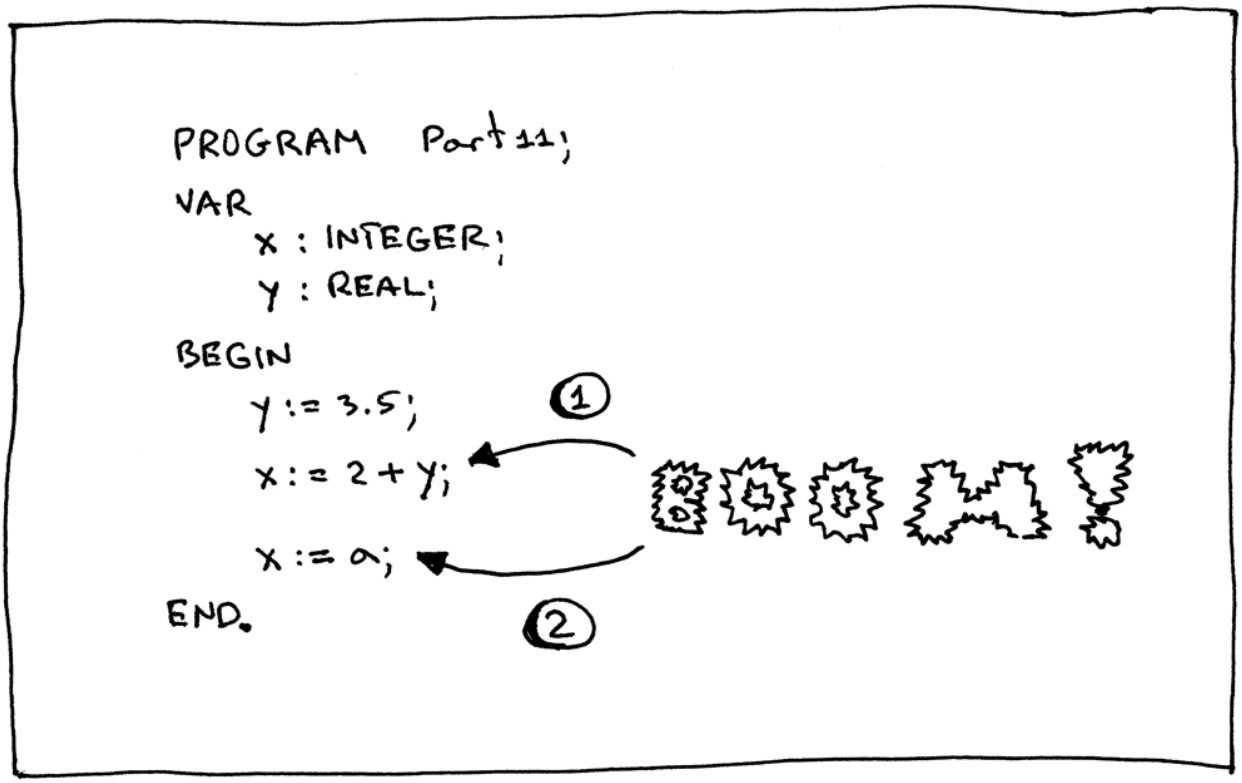


到目前为止一切顺利，但我们还没有回答为什么我们甚至需要首先跟踪这些符号的问题。

以下是一些原因：

- 确保当我们为变量赋值时，类型是正确的（类型检查）
- 确保在使用之前声明变量

看看下面不正确的Pascal程序，例如：



上面的程序有两个问题（可以用fpc (http://www.freepascal.org/)编译自己看看）：

1. 在表达式“x := 2 + y;”中 我们为声明为整数的变量“x”分配了一个十进制值。这不会编译，因为类型不兼容。
2. 在赋值语句“x := a;”中 我们引用了未声明的变量“a”——错了！

为了能够在运行时解释/评估程序的源代码之前识别出类似的情况，我们需要跟踪程序符号。我们在哪里存储我们跟踪的符号？我想你猜对了 - 在符号表中！

什么是**符号表**？甲**符号表**是一个抽象数据类型（**ADT**用于在源代码中追踪各种符号）。今天，我们将使用一些辅助方法将符号表实现为一个单独的类：

```
class  SymbolTable ( object ):
    def  __init__ ( self ):
        self 。_symbols  =  {}

    def  __str__ ( self ):
        s  =  'Symbols: {symbols}' 。格式（
            码元= [值 对于 值 在 自我。_symbols 。值（）]
        ）
        返回 小号

    __repr__  =  __str__

    DEF 定义（自， 符号）：
        打印（'定义：%S '  % 符号）
        自我。_symbols [符号. 名称]  = 符号

    def  lookup ( self ,  name ):
        print ( 'Lookup: %s '  %  name )
        symbol  =  self . _符号。get ( name )
        # 'symbol' 要么是 Symbol 类的实例，要么是 'None'
        返回 符号
```

我们将对符号表执行两个主要操作：存储符号并按名称查找它们：因此，我们需要两个辅助方法 -定义和查找。

方法定义将符号作为参数并使用符号的名称作为键和符号实例作为值将其内部存储在其_symbols有序字典中。方法查找将交易品种名称作为参数，如果找到则返回一个交易品种，如果没有则返回“无”。

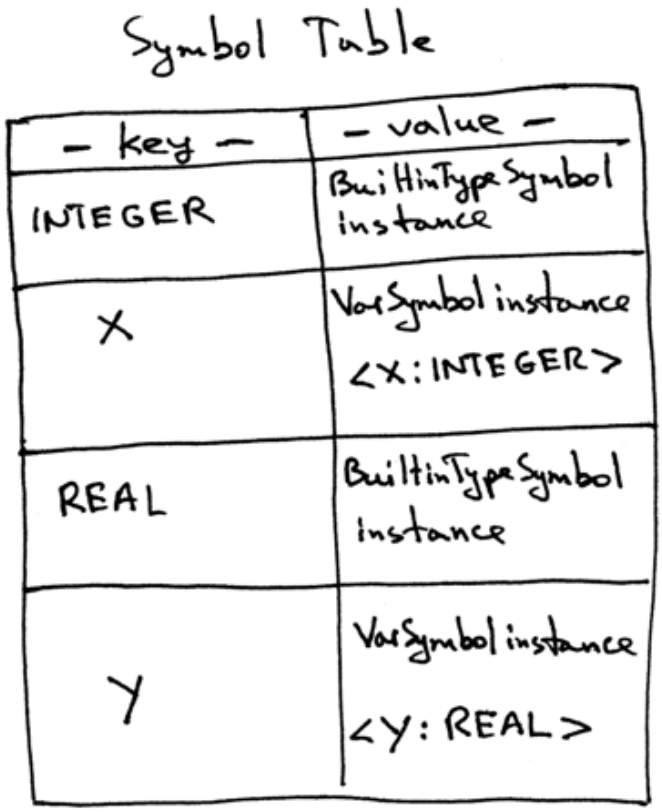
让我们手动填充我们最近使用的相同 Pascal 程序的符号表，我们手动创建变量和内置类型符号：

```
程序 第 11 部分；
VAR
   x  ：  整数；
   y  ：  真实；

开始

结束。
```

再次启动 Python shell 并按照以下步骤操作：

```
$蟒蛇
>>> 从 spi 导入 SymbolTable、BuiltinTypeSymbol、VarSymbol
>>> symtab  = SymbolTable ()
>>> int_type  = BuiltinTypeSymbol ( 'INTEGER' )
>>> symtab.define ( int_type )
定义：整数
>>> 符号表
符号：[整数]
>>>
>>> var_x_symbol  = VarSymbol ( 'x' , int_type )
>>> symtab.define ( var_x_symbol )
定义：<x:INTEGER>
>>> 符号表
符号：[ INTEGER, <x:INTEGER> ]
>>>
>>> real_type  = BuiltinTypeSymbol ( 'REAL' )
>>> symtab.define ( real_type )
定义：真实
>>> 符号表
符号：[ INTEGER, <x:INTEGER>, REAL ]
>>>
>>> var_y_symbol  = VarSymbol ( 'y' , real_type )
>>> symtab.define ( var_y_symbol )
定义：<y:REAL>
>>> 符号表
符号：[ INTEGER, <x:INTEGER>, REAL, <y:REAL> ]
```

如果你查看_symbols字典的内容，它看起来像这样：



我们如何自动化构建符号表的过程？我们将编写另一个节点访问者来遍历我们的解析器构建的AST！这是另一个例子，说明拥有像AST这样的中间形式是多么有用。我们没有扩展我们的解析器来处理符号表，而是分离关注点并编写一个新的节点访问者类。漂亮干净。:)

不过，在此之前，让我们扩展SymbolTable类以在创建符号表实例时初始化内置类型。这是今天的SymbolTable 类的完整源代码：

```
class  SymbolTable ( object ):
    def  __init__ ( self ):
        self 。_symbols  =  OrderedDict ()
        self 。_init_builtins ()

    def  _init_builtins ( self ):
        self 。定义( BuiltinTypeSymbol ( 'INTEGER' ))
        self 。定义( BuiltinTypeSymbol ( 'REAL' ))

    def  __str__ ( self ):
        s  =  'Symbols: {symbols}' 。格式（
            码元= [值 对于 值 在 自我。_symbols 。值（）]
        ）
        返回 小号

    __repr__  =  __str__

    DEF 定义（自， 符号）：
        打印（'定义：%S '  % 符号）
        自我。_symbols [符号. 名称]  = 符号

    def  lookup ( self ,  name ):
        print ( 'Lookup: %s '  %  name )
        symbol  =  self . _符号。get ( name )
        # 'symbol' 要么是 Symbol 类的实例，要么是 'None'
        返回 符号
```

现在到SymbolTableBuilder AST节点访问者：

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

    高清 visit_Block （自我， 节点）：
        用于 报关 的 节点。声明：
            自我。访问（声明）
        自我。访问（节点。复合语句）

    def  visit_Program ( self ,  node ):
        self 。访问（节点。块）

    def  visit_BinOp ( self ,  node ):
        self 。访问（节点。左）
        自我。访问（节点。右）

    def  visit_Num ( self ,  node ):
        pass

    def  visit_UnaryOp ( self ,  node ):
        self 。访问（节点。EXPR ）

    高清 visit_Compound （自我， 节点）：
        为 孩子 的 节点。孩子：
            自我。拜访（孩子）

    def  visit_NoOp ( self ,  node ):
        通过

    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)
```

You’ve seen most of those methods before in the Interpreter class, but the visit_VarDecl method deserves some special attention. Here it is again:

```
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)
```

This method is responsible for visiting (walking) a VarDecl AST node and storing the corresponding symbol in the symbol table. First, the method looks up the built-in type symbol by name in the symbol table, then it creates an instance of

the VarSymbol class and stores (defines) it in the symbol table.

Let’s take our SymbolTableBuilder AST walker for a test drive and see it in action:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM Part11;
... VAR
...    x : INTEGER;
...    y : REAL;
...
... BEGIN
...
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <x:INTEGER>
Lookup: REAL
Define: <y:REAL>
>>> # Let’s examine the contents of our symbol table
…
>>> symtab_builder.symtab
Symbols: [INTEGER, REAL, <x:INTEGER>, <y:REAL>]
```

In the interactive session above, you can see the sequence of “Define: …” and “Lookup: …” messages that indicate the order in which symbols are defined and looked up in the symbol table. The last command in the session prints the contents of the symbol table and you can see that it’s exactly the same as the contents of the symbol table that we’ve built manually before. The magic of AST node visitors is that they pretty much do all the work for you. :)

We can already put our symbol table and symbol table builder to good use: we can use them to verify that variables are declared before they are used in assignments and expressions. All we need to do is just extend the visitor with two more methods: visit_Assign and visit_Var:

```
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))
```

These methods will raise a NameError exception if they cannot find the symbol in the symbol table.

Take a look at the following program, where we reference the variable “b” that hasn’t been declared yet:

```
PROGRAM NameError1;
VAR
   a : INTEGER;

BEGIN
   a := 2 + b;
END.
```

Let’s see what happens if we construct an AST for the program and pass it to our symbol table builder to visit:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError1;
... VAR
...    a : INTEGER;
...
... BEGIN
...    a := 2 + b;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: a
Lookup: b
Traceback (most recent call last):
  ...
  File "spi.py", line 674, in visit_Var
    raise NameError(repr(var_name))
NameError: 'b'
```

Exactly what we were expecting!

Here is another error case where we try to assign a value to a variable that hasn’t been defined yet, in this case the variable ‘a’:

```
PROGRAM NameError2;
VAR
   b : INTEGER;

BEGIN
   b := 1;
   a := b + 2;
END.
```

Meanwhile, in the Python shell:

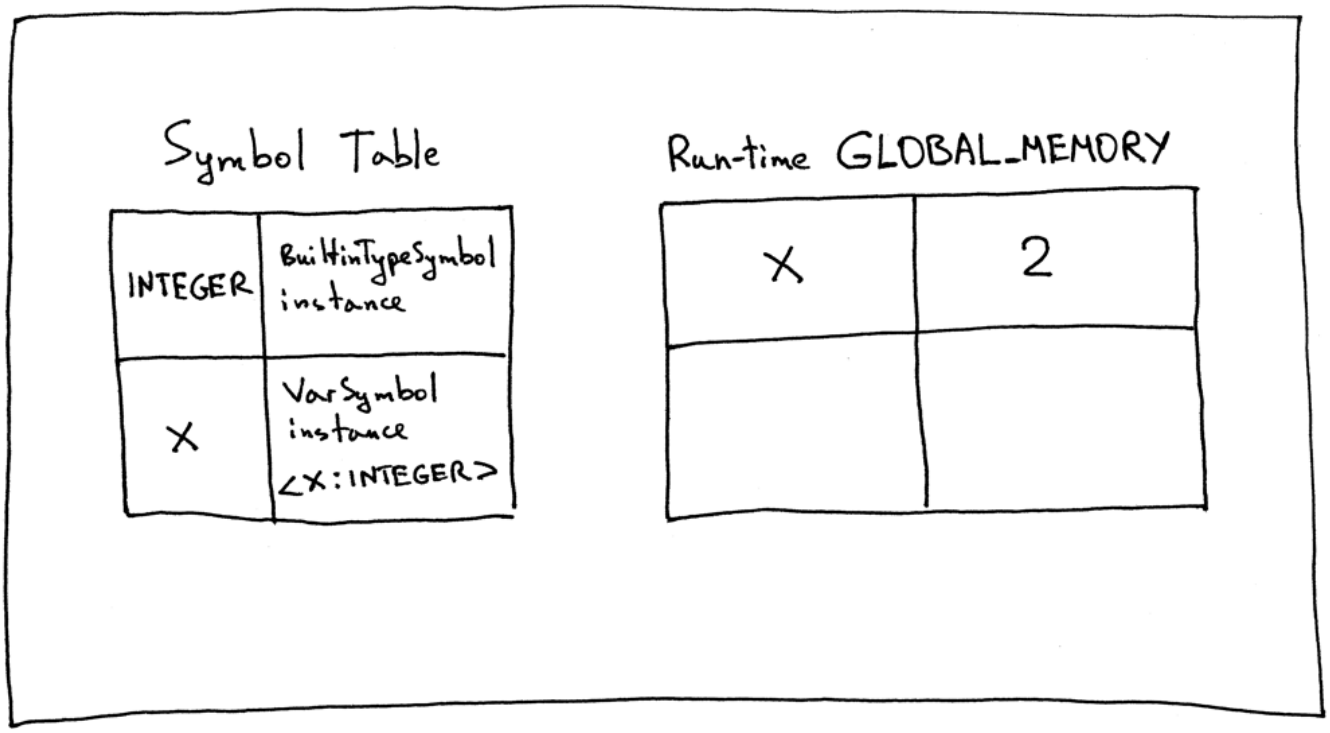
```
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError2;
... VAR
...    b : INTEGER;
...
... BEGIN
...    b := 1;
...    a := b + 2;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <b:INTEGER>
Lookup: b
Lookup: a
Traceback (most recent call last):
  ...
  File "spi.py", line 665, in visit_Assign
    raise NameError(repr(var_name))
NameError: 'a'
```

Great, our new visitor caught this problem too!

我想强调的一点是，SymbolTableBuilder AST访问者所做的所有检查都是在运行时之前进行的，因此在我们的解释器实际评估源程序之前进行。如果我们要解释以下程序，请说明这一点：

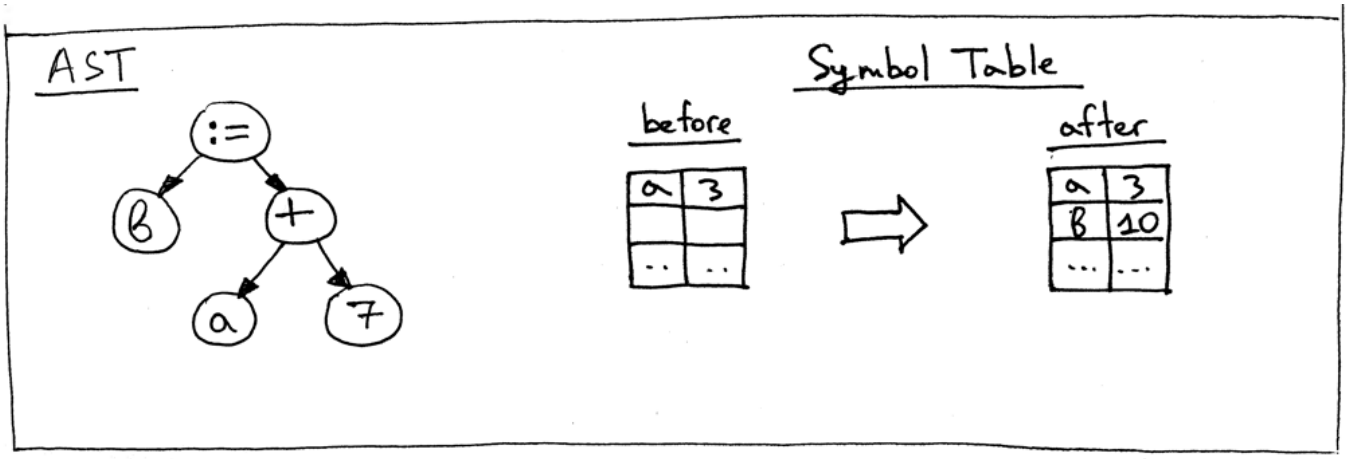
```
程序 第 11 部分；
VAR
   x  :  整数；
开始
   x  :=  2 ;
结束。
```

程序退出前的符号表和运行时 GLOBAL_MEMORY 的内容如下所示：



你看得到差别吗？你能看到符号表没有保存变量“x”的值 2 吗？现在这只是口译员的工作。

还记得第 9 部分 (/lsbasi-part9/)中符号表用作全局存储器的图片吗？



不再！我们有效地摆脱了符号表作为全局内存的双重作用的黑客攻击。

让我们把它们放在一起，并使用以下程序测试我们的新解释器：

```
程序 第 11 部分;
VAR
   数 : 整数;
   a 、 b   : 整数;
   y       : 真实;

BEGIN  {Part11}
   number  :=  2 ;
   一个 : = 数字 ;
   b  :=  10  *  a  +  10  * 数字 DIV  4 ;
   y  :=  20  /  7  +  3.14
结束。  {Part11}
```

将程序保存为 part11.pas 并启动解释器：

```
$ python spi.py part11.pas
定义：整数
定义：真实
查找：整数
定义：<number:INTEGER>
查找：整数
定义：<a:INTEGER>
查找：整数
定义：<b:INTEGER>
查找：真实
定义：<y:REAL>
查找：数字
查找：一个
查找：数字
查找：b
查找：一个
查找：数字
查找：y

符号表内容：
符号：[ INTEGER, REAL, <number:INTEGER>, <a:INTEGER>, <b:INTEGER>, <y:REAL> ]

运行时 GLOBAL_MEMORY 内容：
a  =  2
b  =  25
数字 =  2
y  =  5 .99714285714
```

我想再次提请您注意Interpreter类与构建符号表无关，它依赖于SymbolTableBuilder来确保源代码中的变量在被使用之前正确声明口译员。

检查你的理解

- 什么是符号?
- 为什么我们需要跟踪符号?
- 什么是符号表?
- 定义符号和解析/查找符号有什么区别?
- 鉴于以下的小帕斯卡尔程序，这将是符号表，全局内存（GLOBAL_MEMORY字典，是一部分的内容翻译）？

```
程序 第 11 部分;
VAR
   x , y  : 整数;
开始
   x  :=  2 ;
   y  :=  3  +  x ;
结束。
```

这就是今天的全部内容。在下一篇文章中，我将讨论作用域，我们将深入分析嵌套过程。请继续关注，我们很快就会见到你！并记住无论如何，“继续前进！”

PS我对符号和符号表管理主题的解释深受 Terence Parr所著《语言实现模式 (http://amzn.to/2cHsHT1)》一书的影响。这是一本了不起的书。我认为它对我所见过的主题有最清晰的解释，并且还涵盖了类作用域，我不会在本系列中讨论这个主题，因为我们不会讨论面向对象的 Pascal。

PP S.：如果您迫不及待想要开始深入研究编译器，我强烈推荐 Jack Crenshaw 免费提供的经典著作“Let's Build a Compiler”。 (http://compilers.iecc.com/crenshaw/)

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”！

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 Web Server. Part 2.

6 years ago • 61 comments

Remember, in Part 1 I asked you a question: "How do you run a Django application, …

EOF is not a character

2 years ago • 16 comments

I was reading Computer Systems: A Programmer's Perspective the other day …

Let's Build A Simple Interpreter. Part 12.

5 years ago • 29 comments

"Be not afraid of going slowly; be afraid only of standing still." - Chinese …

Let's B Interpr

2 years a

What I c not unde Richard

Sung Won Cho • 5 years ago

Is there a reason we are using OrderedDict rather than a regular python dictionary?

60 • Reply • Share ›

Sebastian Sung Won Cho • 3 years ago

I think he is using OrderedDict so at the end we can see at what order the symbols were added to the SymbolTable. If you would use the default Python dictionary, there is a chance that when you print the content of the dictionary, the order would be different than what you see in the tutorial, so that might confuse some people.
If you are following this tutorial using Python 3.6 or above, you can safely use the regular Python dictionary, as the insertion order is preserved:
https://stackoverflow.com/a...

• Reply • Share ›

pmst • 3 years ago

This is the awesome tutorial !! Thank you so much. BTW, I am a iOS developer from China(member of swift.gg), I hope to translate your tutrioal into Chinese, This tutorial is so helpful to newbies like me who want to learn interpreter. And I will reserved original artical url in the top of translation artical. Would you authorize me to do it? my email address colourful98765@gmail.com

• Reply • Share ›

Quetzalcoalt • 4 years ago

Uh i'm having a problem, i'm writing a compiler for a grammar given from my university as a project, and this is one of the best tutorials i've found. I'm writing it in java and the grammar is

similar to C, so far everything is working (i think, during lots of testing and debugging, writing on paper because i don't have AST visualizer). My problem is that i don't know where is the method that checks for the variable type. If i have " int a = 3; int b = a/2; " b will be 1.5 even though its an int. The interpreter should not check the types right ? that's the symbolTables job. I just can't figure where the testing is made. Struggling from the past 4 days on this =/. Also is the symbolTable like a hashmap<string, string=""> or it's a hashmap<string, symbol=""> because symbol has two strings in it for var name and var type. so it prints a but i made it as a <string, string=""> and have it as int <int>, double <double> , a <int> ect ect. I also don't understand what visitAsign and visitVar does in the symbolBuilder.

Reply • Share ›

Atit Shetty • 4 years ago

Thank you so much for this series.
Loved the content and superb explanation.

Reply • Share ›

rspivak Mod ➚ Atit Shetty • 4 years ago

You're welcome!

Reply • Share ›

exFil ➚ rspivak • 4 years ago

i mean self.eat(PROGRAM)...

Reply • Share ›

exFil ➚ rspivak • 4 years ago

Hi rspivak. I got an error in this line "p.eat(PROGRAM)" when i inputted this command tree=parser.parse() in the shell.

Reply • Share ›

Richard Rast • 5 years ago

I'm loving this series but this post seemed a bit unfinished. For example you talked about these different types (INTEGER vs REAL) but didn't actually use them anywhere. It seems like semantic analysis would be really great here! I suppose it's a bigger topic and deserves its own blog post, but still ...

Reply • Share ›

Diego Híjar Ruiz • 5 years ago

So.. im having a problem with the code, when i run the spi.py it says:

"Traceback (most recent call last):
File "C:\Users\Diego\Documents\proyecto corona\Calculadoras\python\spi.py", line 773, in <module>
main()
File "C:\Users\Diego\Documents\proyecto corona\Calculadoras\python\spi.py", line 752, in main
text = open(sys.argv[1], 'r').read()
IndexError: list index out of range"

Im new with python so i dont know if im missing something, someone help? :S

Reply • Share ›

rspivak Mod ➚ Diego Híjar Ruiz • 5 years ago

Hi Diego,

You need to pass a filename to the interpreter as an argument:

$ python spi.py part11.pas

Hope that helps.

Reply • Share ›

Mark • 5 years ago

This is the best tutorial I have ever read. I stayed up till 1AM just to get to the end. The sad part is I see it's not the end and I have wait for rest of the knowledge. Maybe Netflix it and release the whole season at once next time? lol Thanks Again! Nothing like binging on Interpreters. I wish you would discuss some basics on OO things. I am writing my own high level language and it is OO. I think I have the concepts now but it would be nice to see an example of how to handle objects and their defined methods that you do not know about, like the code referencing a created class. The second OO topic is code referencing a class that is available to your interpreter. How these two cases are handled would be very interesting to me.

Reply • Share ›

Diego Marcia ➚ Mark • 5 years ago • edited

Being the reference language Pascal, I don't think OOP will be discussed... How have you thought to implement Objects? Off the top of my head, I'd use some special Structures which carry (obfuscated to the user) a set of function signatures. It's a bit naive solution, and not very "strict" wrt Languages Theory (a user-defined Class should be a new type in your type system).
What's your idea?

Reply • Share ›

Russell Coleman • 5 years ago

Yo thank you so much dude this tutorial is incredible!! I've been translating this into C to work on an embedded system because I needed a way to write simple easily modifiable scripts on the device. Can't wait for functions! Thank you so much for taking time out of your day to write all this, I'm learning a lot!

Reply • Share ›

rspivak Mod ➚ Russell Coleman • 5 years ago

You're welcome! I'm glad you liked it. :)

Reply • Share ›

Hugo Dufour • 5 years ago • edited

I'm working on a C compiler project, but it's written in C, how do you suggest I manage data structures? :/
(if you want to see it's on my GitHub history "NuclearCoder")

Reply • Share ›

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