让我们构建一个简单的解释器。第 13 部分：语义分析。 (https://ruslanspivak.com/lsbasi-part13/)

日期 2017 年 4 月 27 日，星期四

> 任何值得做的事情都值得过度做。

在深入探讨作用域主题之前，我想“快速”绕道而行，更详细地讨论符号、符号表和语义分析。本着“任何值得做的事都值得过度做”的精神，我希望您会发现这些材料有助于在处理嵌套作用域之前建立更坚实的基础。今天，我们将继续增加关于如何编写解释器和编译器的知识。您将看到本文中介绍的一些材料包含的部分是您在第 11 部分中 (/lsbasi-part11/)看到的内容的扩展版本，我们在第 11 部分 (/lsbasi-part11/)中讨论了符号和符号表。

好的，让我们开始吧！

语义分析简介

虽然我们的 Pascal 程序可以在语法上正确并且解析器可以成功构建抽象语法树，但该程序仍然可能包含一些非常严重的错误。为了捕捉这些错误，我们需要使用抽象语法树和符号表中的信息。

为什么我们不能在解析过程中检查这些错误，即在语法分析过程中？为什么我们必须构建一个AST和一个叫做符号表的东西来做到这一点？

简而言之，为了方便和关注点的分离。通过将这些额外的检查转移到一个单独的阶段，我们可以一次专注于一项任务，而不会让我们的解析器和解释器做比他们应该做的更多的工作。

当解析器完成构建AST 时，我们知道程序在语法上是正确的；也就是说，它的语法根据我们的语法规则是正确的，现在我们可以分别专注于检查需要额外上下文和信息的错误，而这些错误是解析器在构建AST 时没有的。为了更具体，让我们看一下下面的 Pascal 赋值语句：

```
x := x + y ;
```

解析器会正确处理它，因为在语法上，该语句是正确的（根据我们之前定义的赋值语句和表达式的语法规则）。但这还不是故事的结尾，因为 Pascal 有一个要求，即变量必须在使用之前用其对应的类型声明。解析器如何知道**x**和**y**是否已经声明？

嗯，它没有，这就是为什么我们需要一个单独的语义分析阶段来回答变量是否在使用之前已经声明的问题（以及许多其他问题）。

什么是**语义分析**？基本上，它只是帮助我们根据语言定义确定程序是否有意义以及它是否有意义的过程。

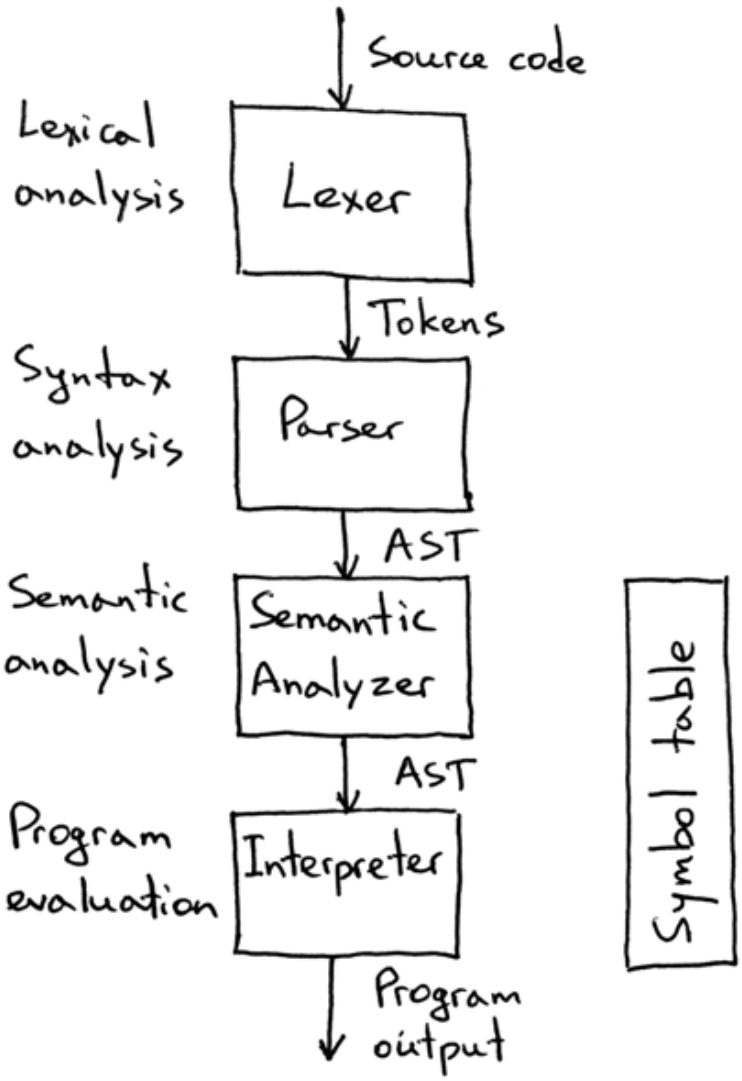
程序有意义意味着什么？这在很大程度上取决于语言定义和语言要求。

Pascal 语言，特别是 Free Pascal 的编译器，有某些要求，如果在程序中不遵循这些要求，将导致fpc编译器出现错误，表明该程序没有“意义”，它是不正确的，即使语法可能看起来不错。以下是其中一些要求：

- 变量必须在使用前声明
- 变量在算术表达式中使用时必须具有匹配类型（这是语义分析的重要部分，称为类型检查，我们将单独介绍）
- 不应有重复的声明（例如，Pascal 禁止在过程中使用与过程形式参数之一同名的局部变量）
- 调用过程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引用原始类型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匹配

当我们有足够的程序上下文时，执行上述要求会容易得多，即我们可以遍历的AST形式的中间表示和包含有关不同程序实体（如变量、过程和函数）的信息的符号表.

在我们实现语义分析阶段后，我们的 Pascal 解释器的结构将如下所示：



从上图可以看出，我们的词法分析器将获取源代码作为输入，将其转换为解析器将使用的标记，并用于验证程序语法正确，然后它将生成一个抽象语法树，我们新的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，并产生程序输出。

让我们进入语义分析阶段的细节。

符号和符号表

在下一节中，我们将讨论如何实现一些语义检查以及如何构建符号表：换句话说，我们将讨论如何对我们的 Pascal 程序进行语义分析。请记住，尽管语义分析听起来很花哨和深入，但这只是在解析我们的程序并创建AST以检查源程序是否存在一些由于缺少附加信息（上下文）。

今天我们将重点关注以下两个静态语义检查*：

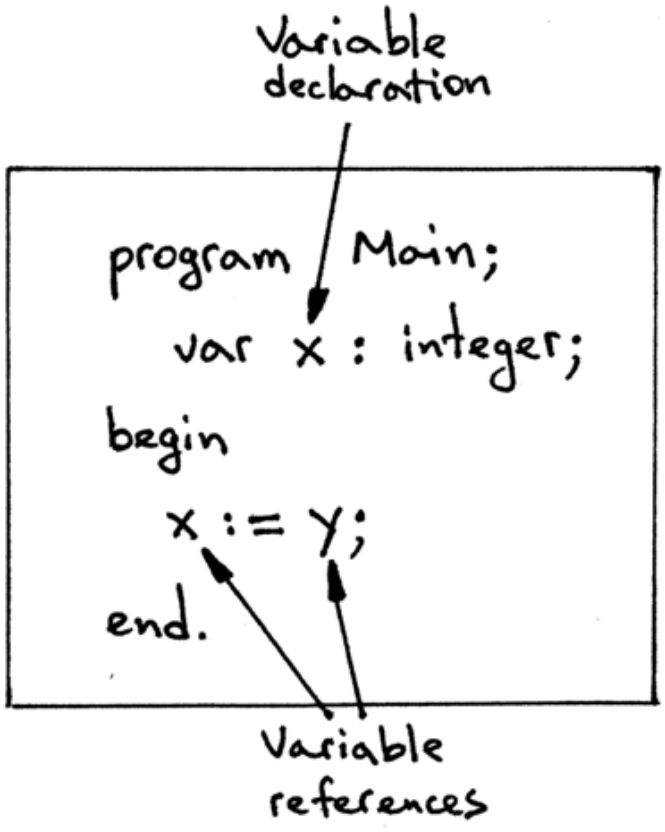
1. 变量在使用之前被声明
2. 没有重复的变量声明

> * ASIDE：静态语义检查是我们可以在解释（评估）程序之前进行的检查，即在对 Interpreter 类的实例调用解释方法之前。前面提到的所有 Pascal 要求都可以通过遍历AST并使用符号表中的信息，通过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，检查没有被零除以及数组索引没有越界就是动态语义检查。我们今天的重点是静态语义检查。

让我们从第一次检查开始，并确保在我们的 Pascal 程序中变量在使用之前已声明。看看以下语法正确但语义不正确的程序（呃……一个句子中有太多难以发音的单词。:)

```
程序 主程序;
   var x : 整数;

开始
    x := y ;
结束。
```

上面的程序有一个变量声明和两个变量引用。您可以在下图中看到：



让我们实际验证我们的程序在语法上是正确的，并且我们的解析器在解析它时没有抛出错误。正如他们所说，信任但要验证。:) 下载spi.py (https://github.com/rspivak/lsbasi/blob/master/part13/spi.py)，启动一个

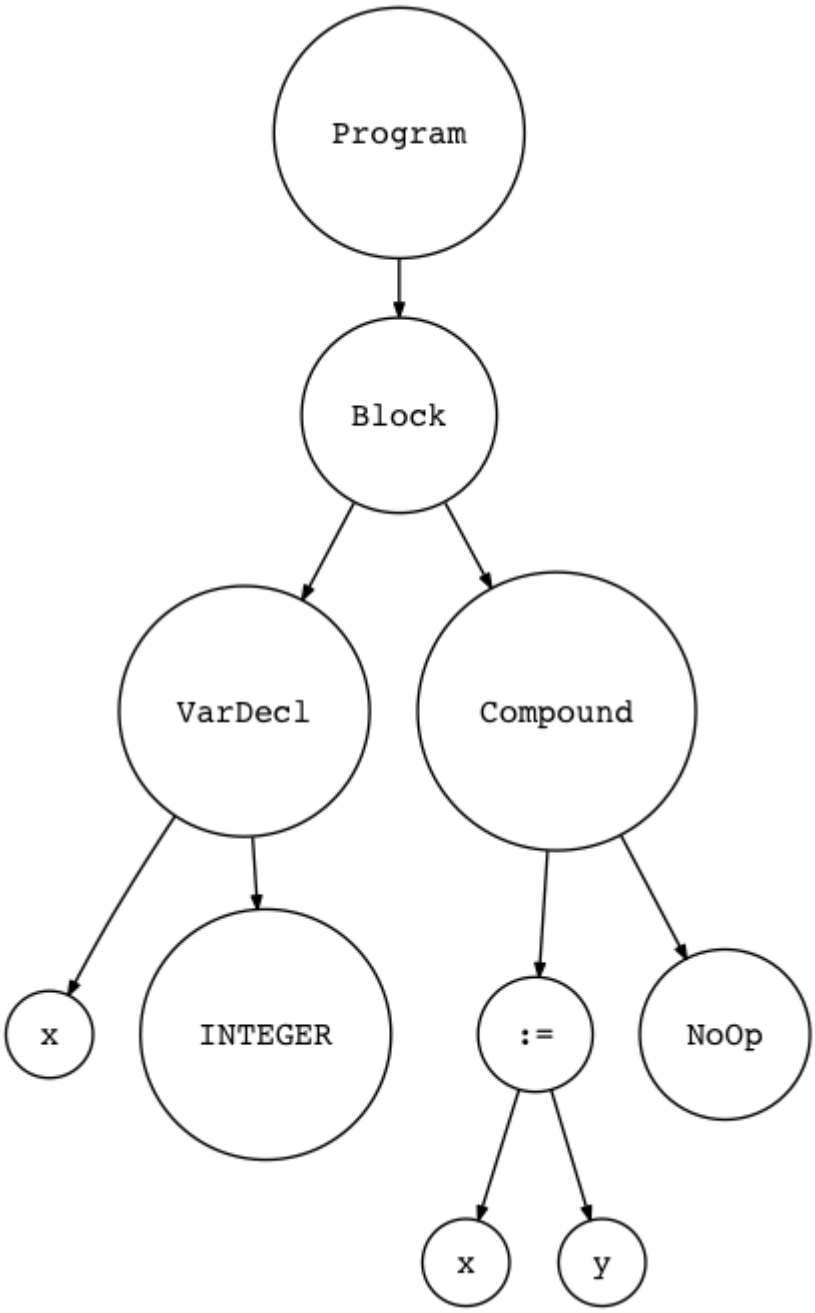
Python shell，然后自己看看：

```
>>> from spi 导入词法分析器、解析器
>>> text  =  """
program Main;
   var x : integer;

开始
    x := y;
结尾。
"""
>>>
>>>词法 =词法分析器（文本）
>>>解析器 =分析器（词法分析器）
>>>树 = parser.parse （）
>>>
```

你看？没有错误。我们甚至可以使用genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part13/genastdot.py)为该程序生成AST图。首先，将源代码保存到一个文件中，假设semanticerror01.pas，然后运行以下命令：
(https://github.com/rspivak/lsbasi/blob/master/part13/genastdot.py)

```
$ python genastdot.py语义错误01.pas>语义错误01.dot
$ dot -Tpng -o ast.png semanticerror01.dot
```

这是AST 图：



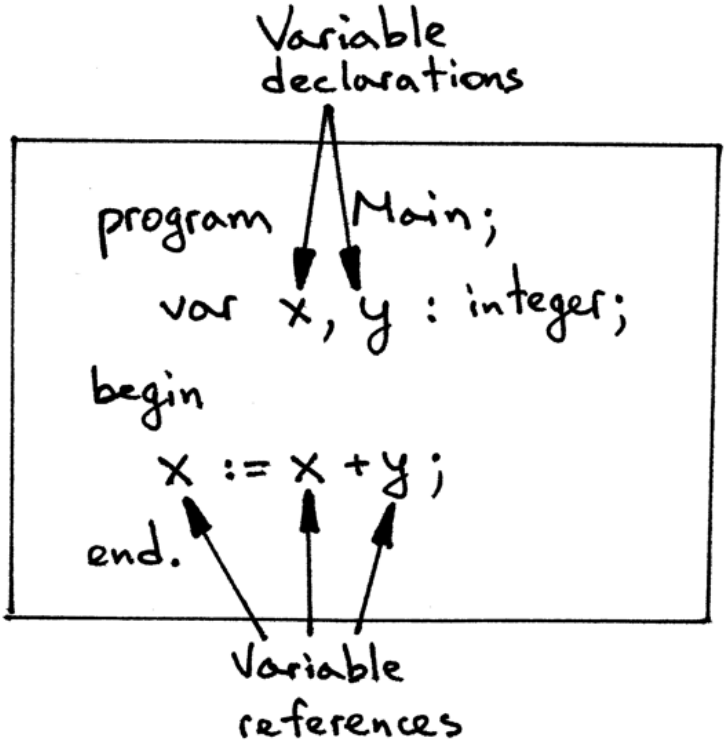
因此，它是一个语法（语法）正确的程序，但是程序没有任何意义，因为我们甚至不知道什么类型的变量**Ÿ**有（这就是为什么我们需要声明），如果它会是有意义的分配**ÿ**到**×**。如果**y**是字符串，那么将字符串分配给整数是否有意义？它没有，至少在 Pascal 中没有。

所以上面的程序有一个语义错误，因为变量**y**没有声明，我们不知道它的类型。为了让我们能够捕捉到这样的错误，我们需要学习如何在使用变量之前检查它们是否已声明。所以让我们学习如何做到这一点。

让我们仔细看看以下语法和语义正确的示例程序：

```
程序 主程序;
   var  x ,  y  : 整数;

开始
    x  :=  x  +  y ;
结束。
```

- 它有两个变量声明：**x**和**y**
- 它还在赋值语句**x := x + y 中**具有三个变量引用（**x**、另一个**x**和**y**）；

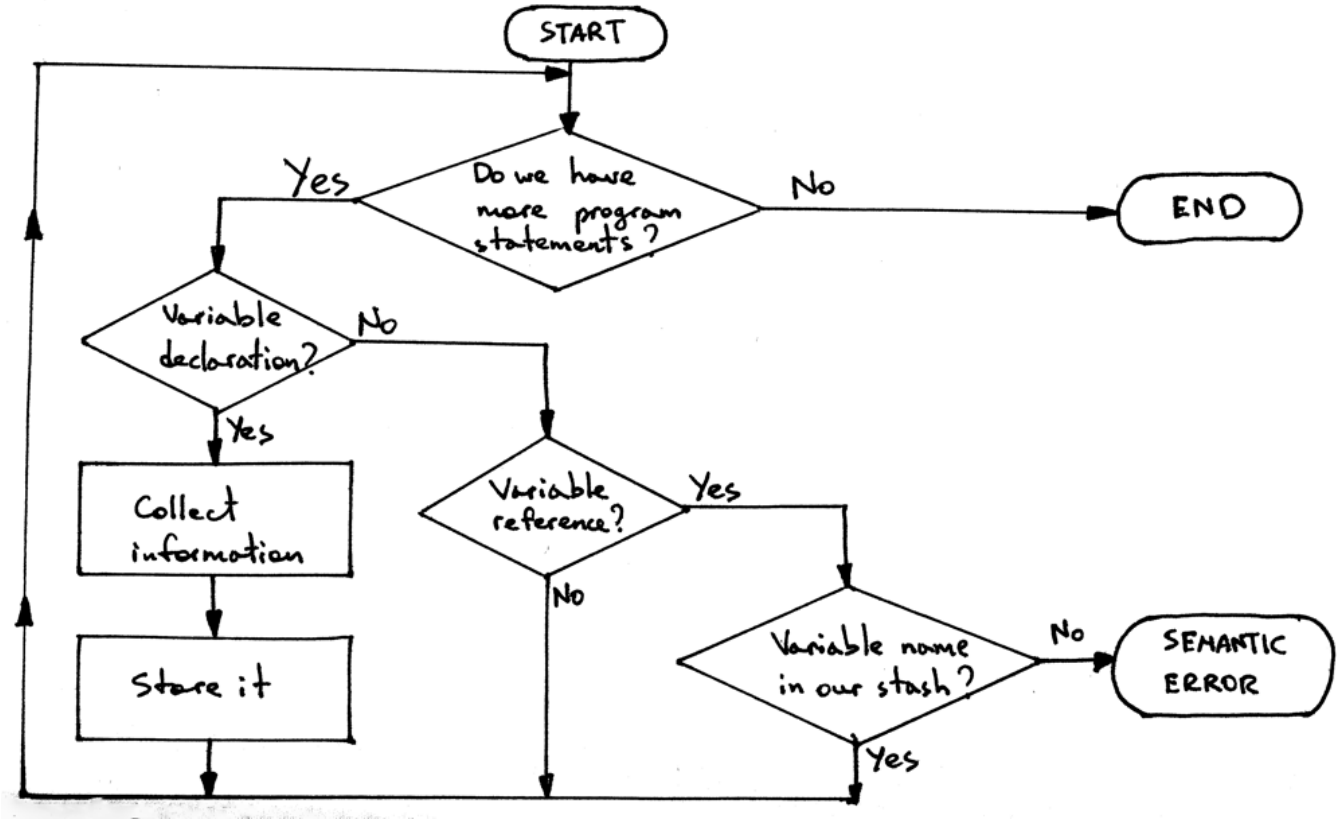


程序语法正确，声明了所有变量，我们可以看到将两个整数相加并将结果赋给一个整数是非常有意义的。这是伟大的，但我们如何编程检查变量（变量引用）**X**和**Ÿ**在赋值语句**X: = X + Y;** 被宣布了吗？

我们可以通过实现以下算法分几个步骤来做到这一点：

1. 遍历所有变量声明
2. 对于您遇到的每个变量声明，收集有关声明变量的所有必要信息
3. 通过使用变量的名称作为键，将收集到的信息存储在一些存储中以备将来参考
4. 当您看到变量引用时，例如在赋值语句**x := x + y 中**，请按变量名称搜索存储以查看存储是否包含有关该变量的任何信息。如果是，则该变量已被声明。如果没有，则变量尚未声明，这是语义错误。

这是我们算法的流程图：



在实现算法之前，我们需要回答几个问题：

- A. 我们需要收集哪些关于变量的信息？
- B. 我们应该在哪里以及如何存储收集到的信息？
- C. 我们如何实现“检查所有变量声明”步骤？

我们的攻击计划如下：

1. 找出以上问题 A、B 和 C 的答案。
2. 使用 A、B 和 C 的答案来实现我们第一次静态语义检查的算法步骤：检查变量在使用之前是否已声明。

好的，让我们开始吧。

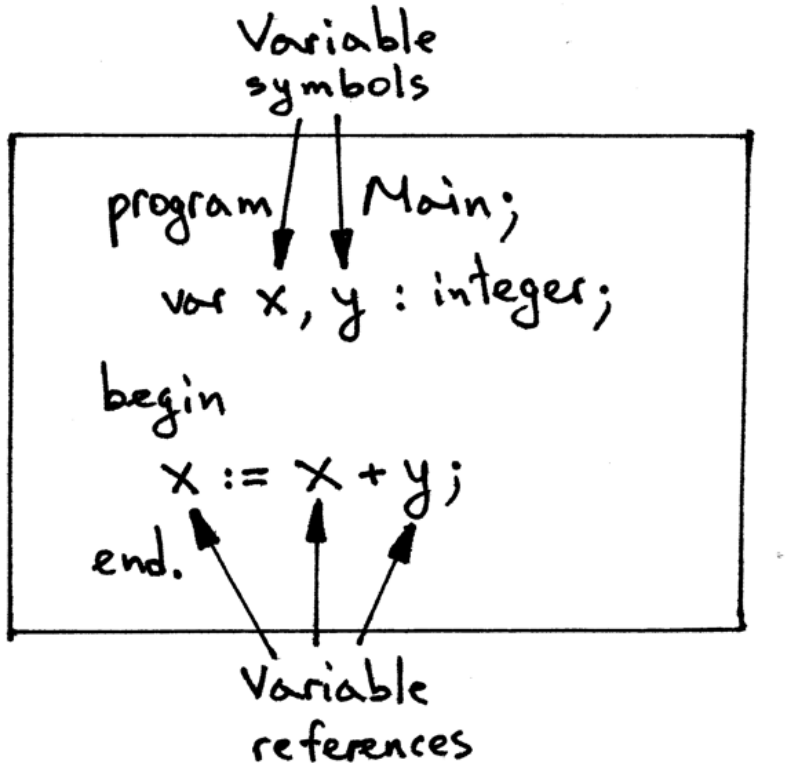
让我们找到“我们需要收集关于变量的哪些信息？”这个问题的答案。

那么，我们需要收集关于变量的哪些必要信息？以下是重要部分：

- 名称（我们需要知道声明变量的名称，因为稍后我们将通过名称查找变量）
- 类别（我们需要知道它是什么类型的标识符：变量、类型、过程等）
- 类型（我们需要此信息进行类型检查）

符号将保存有关我们变量的信息（名称、类别和类型）。什么是符号？甲**符号**是一些节目实体的象的可变，子例程的标识符，或内置型。

在下面的示例程序中，我们有两个变量声明，我们将使用它们来创建两个变量符号：**x** 和**y**。



在代码中，我们将使用一个名为Symbol的类来表示符号，该类具有字段name和type：

```
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参数和一个可选的类型参数（并非所有符号都具有与其关联的类型信息，我们很快就会看到）。

怎么样的类别？我们将类别编码为类名。或者，我们可以将符号的类别存储在Symbol类的专用类别字段中，如下所示：

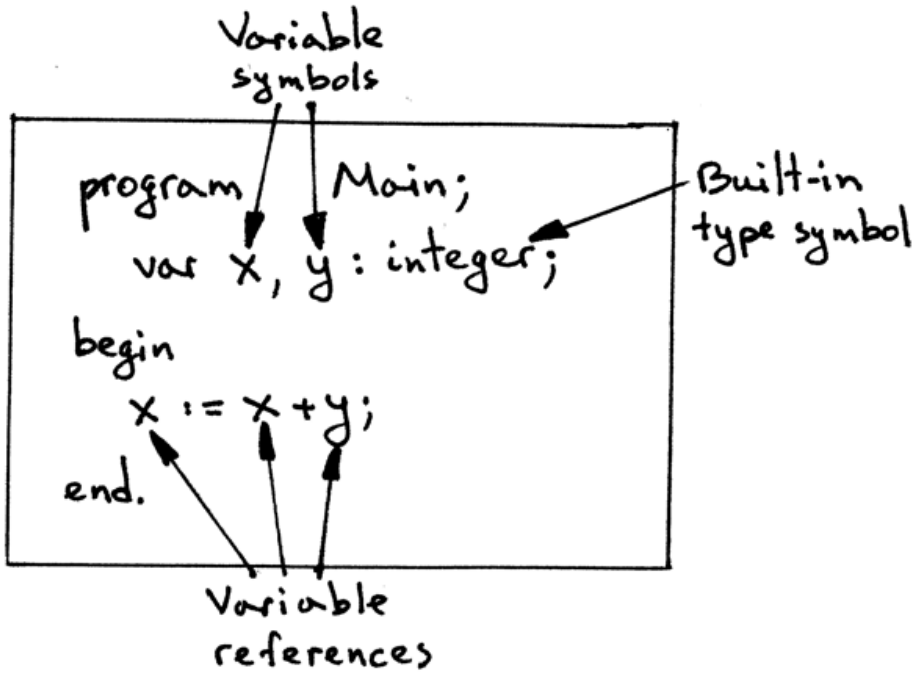
```
class  Symbol ( object ):
    def  __init__ ( self ,  name ,  type = None ):
        self 。姓名 = 姓名
        自我。类型 = 类型
        self 。类别 = 类别
```

但是，创建类的层次结构更明确，其中类的名称指示其类别。

到目前为止，我一直围绕着一个主题，即内置类型。如果您再次查看我们的示例程序：

```
程序 主程序;
   var  x ,  y  : 整数;

开始
    x  :=  x  +  y ;
结束。
```

您可以看到变量**x**和**y**被声明为整数。什么是整数类型？整数类型是另一种符号，内置类型符号。它被称为内置的，因为它不必在 Pascal 程序中显式声明。声明该类型符号并使其可供程序员使用是我们的解释器的责任：



我们将为内置类型创建一个单独的类，称为BuiltinTypeSymbol。这是我们内置类型的类定义：

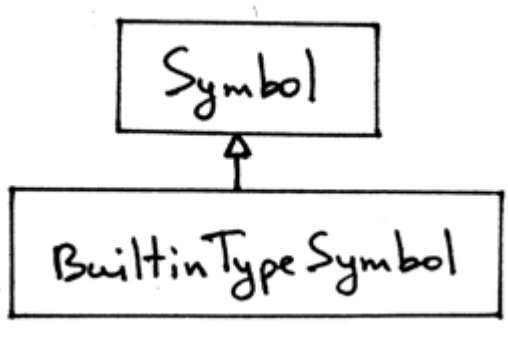
```
class  BuiltinTypeSymbol ( Symbol ):
    def  __init__ ( self ,  name ):
        super () 。__init__ (名称)

    def  __str__ ( self ):
        返回 self 。姓名

    def  __repr__ ( self ):
        返回 "<{class_name}(name='{name}')>" 。格式(
            class_name = self . __class__ . __name__ ,
            name = self . name ,
        )
```

类BuiltinTypeSymbol继承自Symbol类，其构造函数只需要类型的名称，如integer或real。'builtin type' 类别编码在类名中，正如我们之前所讨论的，当我们创建 BuiltinTypeSymbol 类的新实例时，来自基类的类型参数会自动设置为None。

> 在旁边
>
> 双下划线或dunder（如在“ **d** ouble **UNDER**分数”）方法__str__和__repr__是特殊的Python方法。当我们将符号对象打印到标准输出时，我们已经将它们定义为具有良好格式化的消息。

顺便说一句，内置类型是 Symbol 类构造函数中的类型参数是可选参数的原因。

到目前为止，这是我们的符号类层次结构：



让我们在 Python shell 中使用内置类型。下载解释器文件 (https://github.com/rspivak/lsbasi/blob/master/part13/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 ( '整数' )
>>> int_type
<BuiltinTypeSymbol ( name = 'integer' ) >
>>>
>>> real_type  = BuiltinTypeSymbol ( 'real' )
>>> 真实类型
<BuiltinTypeSymbol ( name = 'real' ) >
```

这就是目前内置类型符号的全部内容。现在回到我们的变量符号。

我们如何在代码中表示它们？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 ):
        return  "<{class_name}(name='{name}', type='{type}')>" 。格式（
            class_name = self . __class__ . __name__ ，
            name = self . name ，
            type = self . type ，
        ）

    __repr__  =  __str__
```

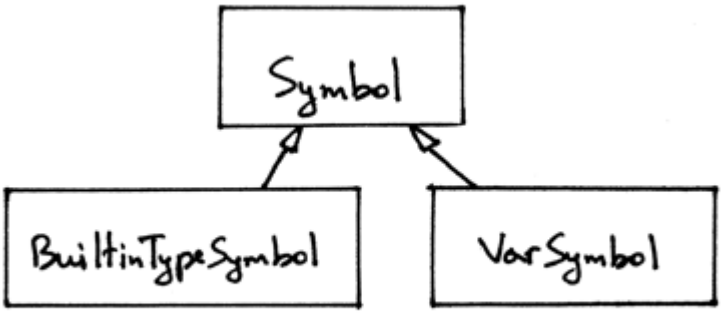
在这个类中，我们将名称和类型参数都做了需要，类名VarSymbol清楚地表明类的一个实例将标识一个变量符号（类别是variable）。所述类型参数是的一个实例 BuiltinTypeSymbol 类。

既然我们知道如何构造BuiltinTypeSymbol类实例，让我们回到交互式 Python shell，看看如何手动构造变量符号的实例：

```
$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
<VarSymbol ( name = 'x' , type = 'integer' ) >
>>>
>>> var_y_symbol  = VarSymbol ( 'y' , real_type )
>>> var_y_symbol
<VarSymbol ( name = 'y' , type = 'real' ) >
>>>
```

如您所见，我们首先创建内置类型符号的实例，然后将其作为第二个参数传递给VarSymbol的构造函数：变量符号必须具有与其关联的名称和类型，正如您在各种变量声明，如**var x : integer;**

这是我们迄今为止以视觉形式定义的完整符号层次结构：



好的，现在开始回答“我们应该在哪里以及如何存储收集到的信息？”的问题。

既然我们已经拥有代表所有变量声明的所有符号，那么我们应该将这些符号存储在何处，以便以后在遇到变量引用（名称）时可以搜索它们？

正如您可能已经知道的那样，答案在符号表中。

什么是符号表？甲**符号表**是用于在源代码中追踪各种符号的抽象数据类型。把它想象成一个字典，其中键是符号的名称，值是符号类（或其子类之一）的实例。为了在代码中表示符号表，我们将使用一个专门的类来恰当地命名为SymbolTable。:) 为了在符号表中存储符号，我们将向符号表类添加插入方法。方法insert将接受一个符号作为参数并将其内部存储在_symbols有序字典中，使用符号的名称作为键和符号实例作为值：

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

    def  __str__ ( self ):
        symtab_header  =  '符号表内容'
        lines  =  [ ' \n ' ,  symtab_header ,  '_'  *  len ( symtab_header )]
        lines 。延伸（
             （' %7S : %R '  %  （键， 值））
            为 键， 值 在 自我。_symbols 。项目（）
        ）
        线. 追加( ' \n ' )
        s  =  ' \n ' 。加入（行）
        返回 s

    __repr__  =  __str__

    高清 插入（自我， 符号）：
        打印（'插入：%s的'  %  符号。名称）
        自我。_symbols [符号. 名称]  = 符号
```

让我们为以下示例程序手动填充符号表。因为我们还不知道如何搜索我们的符号表，我们的程序不会包含任何变量引用，只有变量声明：

```
程序 SymTab1 ;
   var  x ,  y  : 整数;

开始

结束。
```

下载symtab01.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab01.py)，它包含我们新的SymbolTable类并在命令行上运行它。这是我们上面程序的输出：

```
$ python symtab01.py
插入：整数
插入：x
插入：y


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >
```

现在让我们在 Python shell 中手动构建和填充符号表：

```
$蟒蛇
>>> 从 symtab01 导入 SymbolTable、BuiltinTypeSymbol、VarSymbol
>>> symtab  = SymbolTable ()
>>> int_type  = BuiltinTypeSymbol ( 'INTEGER' )
>>> # 现在让我们将内置类型符号存储在符号表中
...
>>> symtab.insert ( int_type )
插入：整数
>>>
>>> 符号表


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >



>>> var_x_symbol  = VarSymbol ( 'x' , int_type )
>>> symtab.insert ( var_x_symbol )
插入：x
>>> 符号表


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >


>>> var_y_symbol  = VarSymbol ( 'y' , int_type )
>>> symtab.insert ( var_y_symbol )
插入：y
>>> 符号表


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >



>>>
```

至此，我们对之前提出的两个问题有了答案：

- A. 我们需要收集哪些关于变量的信息？

  名称、类别和类型。我们使用符号来保存这些信息。

- B. 我们应该在哪里以及如何存储收集到的信息？

我们使用其插入方法将收集到的符号存储在符号表中。

现在让我们找到第三个问题的答案：**“我们如何实现“检查所有变量声明”步骤?**

这是一个非常容易的。因为我们已经有一个由我们的解析器构建的AST，我们只需要创建一个新的AST访问者类，该类将负责遍历树并在访问VarDecl AST 节点时执行不同的操作!

现在我们对所有三个问题都有了答案：

- A. 我们需要收集哪些关于变量的信息?

  名称、类别和类型。我们使用符号来保存这些信息。

- B. 我们应该在哪里以及如何存储收集到的信息?

  我们使用其插入 方法将收集到的符号存储在符号表中。

- C. 我们如何实现“检查所有变量声明”步骤?

  我们将创建一个新的AST访问者，该访问者将对访问VarDecl AST 节点执行一些操作。

让我们创建一个新的树访问者类并将其命名为SemanticAnalyzer。看看下面的示例程序，例如：

```
程序 SymTab2 ;
   var  x ,  y  : 整数;

开始

结束。
```

为了能够分析上面的程序，我们不需要实现所有的visit_xxx方法，只需要实现它们的一个子集。下面是SemanticAnalyzer类的骨架，它有足够的visit_xxx方法来成功地遍历上面示例程序的AST：

```
class  SemanticAnalyzer ( 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）

    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 ):
        # Actions go here
        pass
```

现在，我们已经完成了第一次静态语义检查算法的前三个步骤的所有部分，该检查验证变量在使用之前是否已声明。

这里再次介绍算法的步骤：

1. 遍历所有变量声明
2. 对于您遇到的每个变量声明，收集有关声明变量的所有必要信息
3. 通过使用变量的名称作为键，将收集到的信息存储在一些存储中以备将来参考
4. 当您在赋值语句**x := x + y 中**看到变量引用时，请按变量名称搜索存储以查看存储是否包含有关该变量的任何信息。如果是，则该变量已被声明。如果没有，则变量尚未声明，这是语义错误。

让我们实施这些步骤。实际上，我们唯一需要做的就是填写SemanticAnalyzer类的visit_VarDecl方法。在这里，填写：

```
def  visit_VarDecl ( self ,  node ):
    # 目前，手动为 INTEGER 内置类型创建一个符号
    # 并将类型符号插入符号表中。
    type_symbol  =  BuiltinTypeSymbol ( 'INTEGER' )
    self 。符号表。插入（type_symbol ）

    # 我们拥有创建变量符号所需的所有信息。
    # 创建符号并将其插入到符号表中。
    var_name  = 节点。变量节点。value
    var_symbol  =  VarSymbol ( var_name ,  type_symbol )
    self 。符号表。插入（var_symbol ）
```

如果您查看方法的内容，您会发现它实际上包含了所有三个步骤：

1. 一旦我们调用了SemanticAnalyzer实例的访问方法，就会为每个变量声明调用该方法。这涵盖了算法的第 1 步： “检查所有变量声明”
2. 对于每个变量声明，方法visit_VarDecl将收集必要的信息并创建一个变量符号实例。这涵盖了算法的第 2 步： “对于您遇到的每个变量声明，收集有关声明变量的所有必要信息”
3. 方法visit_VarDecl将使用符号表的插入方法将收集到的关于变量声明的信息存储在符号表中。这涵盖了算法的第 3 步： “将收集到的信息存储在某个存储库中，以使用变量名作为关键字，以备将来参考”

要查看所有这些步骤的实际效果，请先(https://github.com/rspivak/lsbasi/blob/master/part13/symtab02.py)下载文件symtab02.py(https://github.com/rspivak/lsbasi/blob/master/part13/symtab02.py)并研究其源代码。然后在命令行上运行它并检查输出：

```
$ python symtab02.py
插入：整数
插入：x
插入：整数
插入：y


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >
```

您可能已经注意到，有两行表示Insert: INTEGER。我们将在下一节中解决这种情况，我们将讨论语义检查算法的最后一步（第 4 步）的实现。

好的，让我们实现算法的第 4 步。这是第 4 步的更新版本，以反映符号和符号表的引入：当您在赋值语句**x := x + y 中**看到变量引用（名称）时，按变量名称搜索符号表以

查看该表是否具有与名称关联的变量符号。如果是，则该变量已被声明。如果没有，则变量尚未声明，这是语义错误。

要实现第 4 步，我们需要对符号表和语义分析器进行一些更改：

1. 我们需要向符号表添加一个方法，该方法将能够按名称查找符号。
2. 我们需要更新我们的语义分析器以在每次遇到变量引用时在符号表中查找名称。

首先，让我们通过添加负责按名称搜索符号的查找方法来更新SymbolTable类。换句话说，查找方法将负责将变量名（变量引用）解析为它的声明。将变量引用映射到其声明的过程称为**名称解析**。这是我们的查找方法，名称解析：

```
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
```

该方法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None。就如此容易。

在此过程中，让我们也更新SymbolTable类以初始化内置类型。我们将通过添加一个方法_init_builtins并在SymbolTable的构造函数中调用它来做到这一点。该_init_builtins方法将插入一类符号整数和一类符号真实到符号表中。

这是我们更新的SymbolTable 类的完整代码：

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

    def  _init_builtins ( self ):
        self 。插入( BuiltinTypeSymbol ( 'INTEGER' ))
        自我。插入( BuiltinTypeSymbol ( 'REAL' ))

    def  __str__ ( self ):
        symtab_header  =  '符号表内容'
        lines  =  [ ' \n ' ,  symtab_header ,  '_'  *  len ( symtab_header )]
        lines 。延伸（
            （' %7S : %R '  %  （键， 值））
            为 键， 值 在 自我。_symbols 。项目（）
        ）
        线. 追加( ' \n ' )
        s  =  ' \n ' 。加入（行）
        返回 s

    __repr__  =  __str__

    高清 插入（自我， 符号）：
        打印（'插入：%s的'  % 符号。名称）
        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
```

现在我们有内置的类型符号和查找方法来在遇到变量名（以及其他名称，如类型名）时搜索我们的符号表，让我们更新SemanticAnalyzer的visit_VarDecl方法并替换我们手动创建的两行对INTEGER内置型符号，并将其手动插入到符号表中的代码来查找INTEGER类型的符号。

此更改还将解决我们之前看到的Insert: INTEGER行的双重输出问题。

这是更改前的visit_VarDecl方法：

```
def visit_VarDecl(self, node):
    # 目前，手动为 INTEGER 内置类型创建一个符号
    # 并将类型符号插入符号表中。
    type_symbol = BuiltinTypeSymbol('INTEGER')
    self.符号表。插入（type_symbol）

    # 我们拥有创建变量符号所需的所有信息。
    # 创建符号并将其插入到符号表中。
    var_name = 节点。变量节点。value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.符号表。插入（var_symbol）
```

更改后：

```
def visit_VarDecl(self, node):
    type_name = node.类型节点。值
    type_symbol = self.符号表。查找（type_name）

    # 我们拥有创建变量符号所需的所有信息。
    # 创建符号并将其插入到符号表中。
    var_name = 节点。变量节点。value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.符号表。插入（var_symbol）
```

让我们将更改应用于只有变量声明的熟悉的 Pascal 程序：

```
程序 SymTab3;
   var x, y : 整数;

开始

结束。
```

下载包含 (https://github.com/rspivak/lsbasi/blob/master/part13/symtab03.py) 我们刚刚讨论的所有更改的symtab03.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab03.py)文件，在命令行上运行它，然后看到程序输出中不再有重复的Insert: INTEGER行：

```
$ python symtab03.py
插入：整数
插入：真实
查找：整数
插入：x
查找：整数
插入：y


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >
```

您还可以在上面的输出中看到，我们的语义分析器两次查找INTEGER内置类型：第一次查找变量**x**的声明，第二次查找变量**y**的声明。

现在让我们将注意力转移到变量引用（名称）以及如何将变量名称解析为它的变量声明（变量符号），例如在算术表达式中。让我们看看下面的示例程序，例如，它有一个赋值语句**x := x + y;** 具有三个变量引用：**x**、另一个**x**和**y**：

```
程序 SymTab4 ;
    var  x ,  y  : 整数;

开始
    x  :=  x  +  y ;
结束。
```

我们已经在符号表实现中使用了查找方法。我们现在需要做的是扩展我们的语义分析器，以便每次遇到变量引用时，它都会使用符号表的查找名称通过变量引用名称搜索符号表。当分析器遍历AST时，每次遇到变量引用时，会调用SemanticAnalyzer 的什么方法？这是方法visit_Var。让我们将它添加到我们的类中。这很简单：它所做的就是按名称查找变量符号：

```
def  visit_Var ( self ,  node ):
    var_name  =  node 。值
    var_symbol  =  self 。符号表。查找（var_name ）
```

因为我们的示例程序SymTab4与算术加在其右边的赋值语句，我们需要两个方法添加到我们的SemanticAnalyzer，使其能够真正走AST的的SymTab4程序并调用visit_Var方法都无功节点。我们需要添加的方法是visit_Assign和visit_BinOp。它们并不新鲜：您以前见过这些方法。他们来了：

```
def  visit_Assign ( self ,  node ):
    # 右侧
    self . 访问( node . right )
    # 左侧
    self . 访问（节点。左）

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

您可以在文件symtab04.py 中(https://github.com/rspivak/lsbasi/blob/master/part13/symtab04.py)找到包含我们刚刚讨论的更改的完整源代码。下载该文件，在命令行上运行它，并使用赋值语句检查为我们的示例程序SymTab4生成的输出。

这是我的笔记本电脑上的输出：

```
$ python symtab04.py
插入：整数
插入：真实
查找：整数
插入：x
查找：整数
插入：y
查找：x
查找：y
查找：x


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >
```

花一些时间分析输出并确保您了解如何以及为什么按该顺序生成输出。

在这一点上，我们已经实现了我们算法的所有步骤，用于静态语义检查，验证程序中的所有变量在使用之前都已声明！

语义错误

到目前为止，我们已经查看了声明了变量的程序，但是如果我们的程序有一个不能解析为任何声明的变量引用呢？也就是说，它没有声明？这是一个语义错误，我们需要扩展我们的语义分析器以发出该错误信号。

看看以下语义不正确的程序，其中变量**y**未声明但在赋值语句中使用：

```
程序 SymTab5 ;
    var  x  :  整数；

开始
    x  :=  y ;
结束。
```

要发出错误信号，我们需要修改SemanticAnalyzer的visit_Var方法，以便在查找方法无法将名称解析为符号并返回None 时抛出异常。这是visit_Var的更新代码：

```
def  visit_Var ( self ,  node ):
    var_name  =  node 。值
    var_symbol  =  self 。符号表。查找（var_name ）
    如果 var_symbol 是 None ：
        引发 异常（
            “错误：找不到符号（标识符）' %s '”  %  var_name
        ）
```

下载symtab05.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab05.py)，在命令行上运行它，看看会发生什么：

```
$ python symtab05.py
插入：整数
插入：真实
查找：整数
插入：x
查找：y
错误：找不到符号（标识符）“y”


符号表内容
_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
```

您可以看到错误消息**Error: Symbol(identifier) not found 'y'**和符号表的内容。

恭喜您完成了我们语义分析器的当前版本，它可以静态检查程序中的变量是否在使用前声明，如果没有，则抛出一个异常指示语义错误！

让我们暂停一下，庆祝这个重要的里程碑。好的，第二个结束了，我们需要继续进行另一个静态语义检查。为了好玩和盈利，让我们扩展我们的语义分析器来检查声明中的重复标识符。

我们来看看下面的程序，SymTab6：

```
程序 SymTab6 ;
   var  x ,  y  : 整数;
   变量 y  : 真实的；
开始
   x  :=  x  +  y ;
结束。
```

变量**y**被声明了两次：第一次是整数，第二次是实数。

为了捕捉语义错误，我们需要修改我们的visit_VarDecl方法，以在插入新符号之前检查符号表是否已经有一个同名的符号。这是我们新版本的方法：

```
def  visit_VarDecl ( self ,  node ):
    type_name  =  node 。类型节点。值
    type_symbol  =  self 。符号表。查找（type_name ）

    # 我们拥有创建变量符号所需的所有信息。
    # 创建符号并将其插入到符号表中。
    var_name  = 节点。变量节点。值
    var_symbol  =  VarSymbol ( var_name ,  type_symbol )

    # 如果表已经有一个
    同名
    的符号# if  self ，则发出错误信号。符号表。查找（VAR_NAME ） 是 不 无：
        加注 异常（
            “错误：重复识别符' %S '发现” % VAR_NAME
        ）

    自我。符号表。插入（var_symbol ）
```

文件symtab06.py (https://github.com/rspivak/lsbasi/blob/master/part13/symtab06.py)包含所有更改。下载它并在命令行上运行它：

```
$ python symtab06.py
插入：整数
插入：真实
查找：整数
查找：x
插入：x
查找：整数
查找：y
插入：y
查找：真实
查找：y
错误：发现重复的标识符“y”


符号表内容

_____________________
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >
```

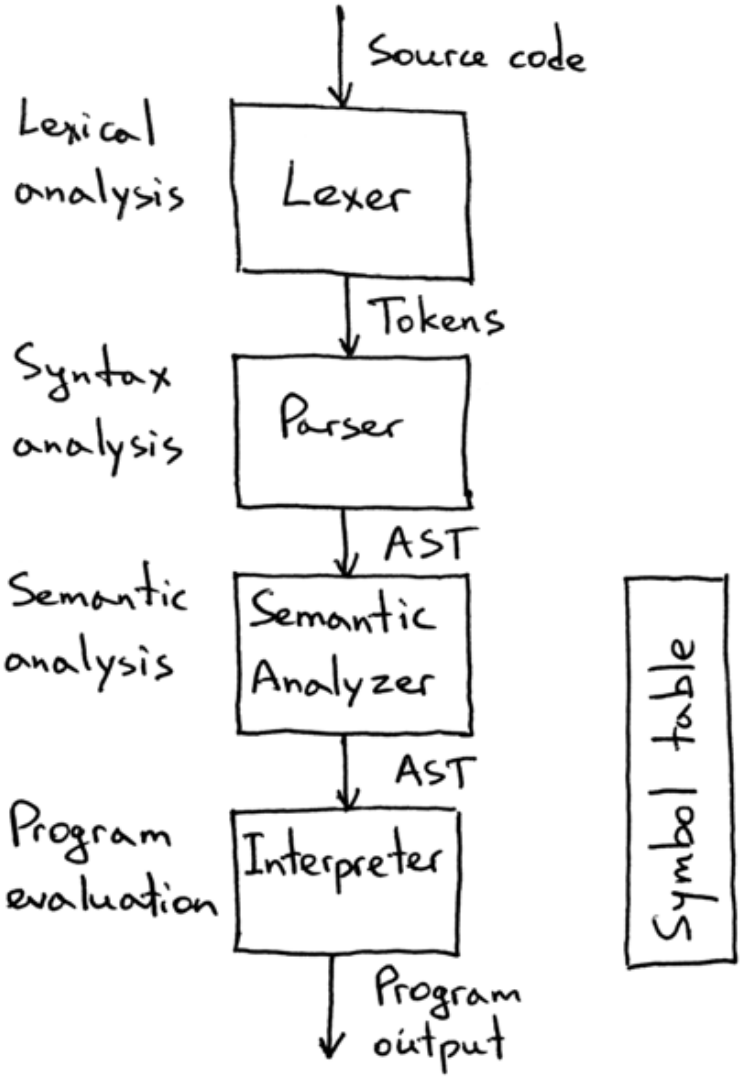
研究符号表的输出和内容。确保您了解正在发生的事情。

概括

让我们快速回顾一下我们今天学到的东西：

- 我们学习了更多关于符号、符号表和一般语义分析的知识
- 我们了解了名称解析以及语义分析器如何将名称解析为其声明
- 我们学习了如何编写一个语义分析器来遍历AST，构建符号表，并进行基本的语义检查

而且，提醒一下，我们的解释器的结构现在看起来像这样：



我们已经完成了今天的语义检查，我们终于准备好处理作用域的主题，它们如何与符号表相关，以及存在嵌套作用域时的语义检查主题。这些将是下一篇文章的中心主题。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

Zenon • 4 years ago

Ruslan, thank you for your hard work. I have learned so much from your tutorials.

Last grammar rule of 'declarations' from Part 12 doesn't allow for multiple VAR reserved keywords so a test program:

```
program SymTab6;
var x, y : integer;
var y : real;
begin
x := x + y;
end.
```

won't parse.

Also this part of the tutorial, number 13, is almost exaclty the same as part 11. Was this an oversight? :)

3 • Reply • Share ›

Howard ➚ Zenon • 3 years ago

In parser.py in declarations method:

```
if self.current_token.type == VAR:
while self.current_token.type == .VAR: <== ADD THIS LINE and INDENT NEXT 5 LINES
self.match(VAR)
while self.current_token.type == ID:
var_decl = self.variable_declaration()
declarations.extend(var_decl)
self.match(Token.SEMI)

while self.current_token.type == PROCEDURE:
..... rest of code
```

1 • Reply • Share ›

solstice333 ➚ Zenon • a year ago

On top of @Howard's comment, the updated grammar for the declarations rule would be

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

I believe.

Reply • Share ›

Chengyi Wu • 4 years ago

This looks exactly the same as SymbolTableBuilder class in part11. I guess the SymbolTableBuilder calss should be the semantic analysis.

1 Reply • Share ›

Anon • 4 years ago

Would you please update your mini pascal grammar in EBNF? It's hard to keep track the differences between implements.

1 Reply • Share ›

Mykola Zekter • 4 years ago

First to comment! I almost thought you forgot about this series, the last part was almost 5 months ago. This series has been really useful in my pursuit of understanding how compilers and interpreters work.

Your fellow Ukrainian reader.

1 Reply • Share ›

Trats • 3 months ago

Ruslan I really appreciate your effort. Thanks for many usefull hints.

Reply • Share ›

Irwin Rodriguez • 8 months ago

I did it! Finally after 8 days of constant study I finally understood all the way through chapter 9 to 13. When I saw the full grammar implementation in part 9 my mind get overwhelmed and I thought I needed a breath to tackle all that content step by step until I hear the "click" in my head. Now I finally understand all the grammar implementation and the basic semantics checks (including the symbol table implementation) so let's keep moving forward! thanks for this terrific series Ruslan!

Reply • Share ›

Steve Freed • 9 months ago

How would dynamic Semantic analysis be done? Specifically in the Interpreter class for BinOps you cannot get the node object from each visitor method if you return the value of the node from the visitor methods. Even if you return the node instead then you'll run into issues in other visit methods in the recursive decent.

Reply • Share ›

roachsinai • 2 years ago

So `self.category = category` is useless in definition of class Symbol?

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