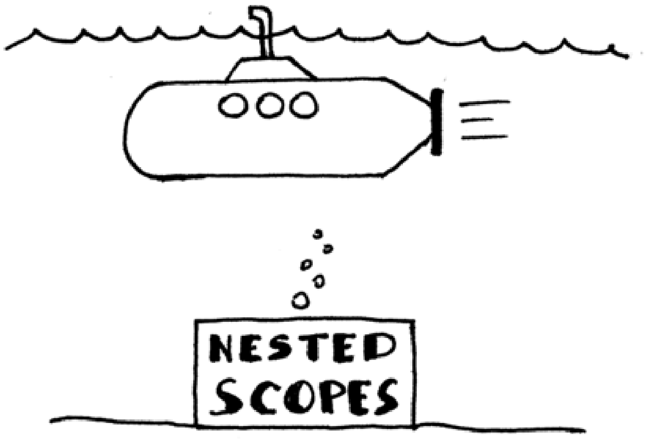
让我们构建一个简单的解释器。第 14 部分：嵌套作用域和源到源编译器。(https://ruslanspivak.com/lsbasi-part14/)

日期 2017 年 5 月 8 日，星期一

只有死鱼随波逐流。

正如我在上一篇文章中 (/lsbasi-part13)所承诺的 (/lsbasi-part13)，今天我们终于要深入探讨范围的主题。



这就是我们今天要学习的内容：

- 我们将学习作用域，它们为何有用，以及如何在带有符号表的代码中实现它们。
- 我们将学习嵌套作用域以及如何使用链式作用域符号表来实现嵌套作用域。
- 我们将学习如何解析带有形式参数的过程声明以及如何在代码中表示过程符号。
- 我们将学习如何扩展我们的语义分析器以在存在嵌套作用域的情况下进行语义检查。
- 我们将更多地了解名称解析以及当程序具有嵌套作用域时语义分析器如何将名称解析为它们的声明。
- 我们将学习如何构建范围树。
- 今天我们还将学习如何编写我们自己的**源到源编译器**！我们将在文章后面看到它与我们讨论范围的相关性。

让我们开始吧！或者我应该说，让我们潜入吧！

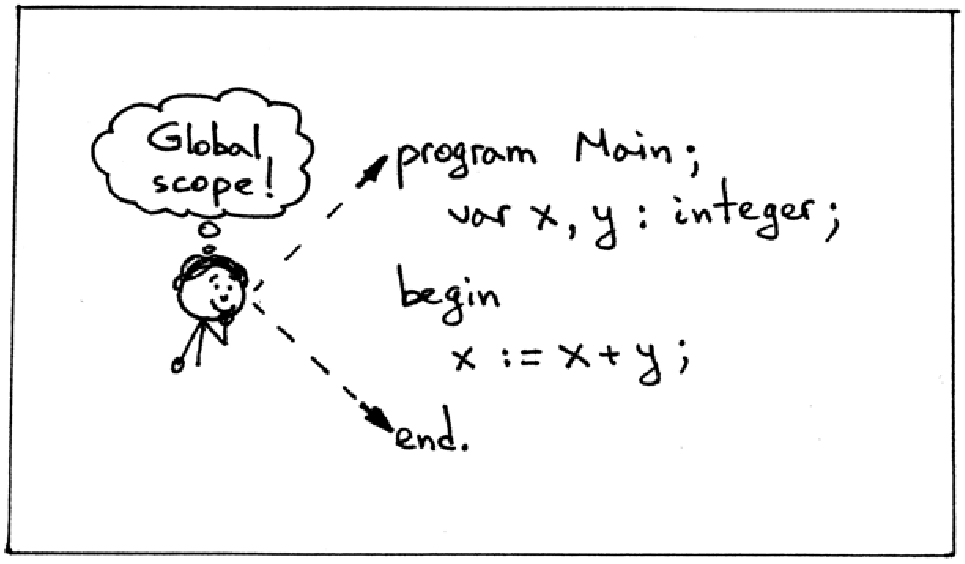
目录

范围和范围符号表

什么是范围？甲**范围**是其中可以使用的名称的程序的文本区域。我们来看下面的示例程序，例如：

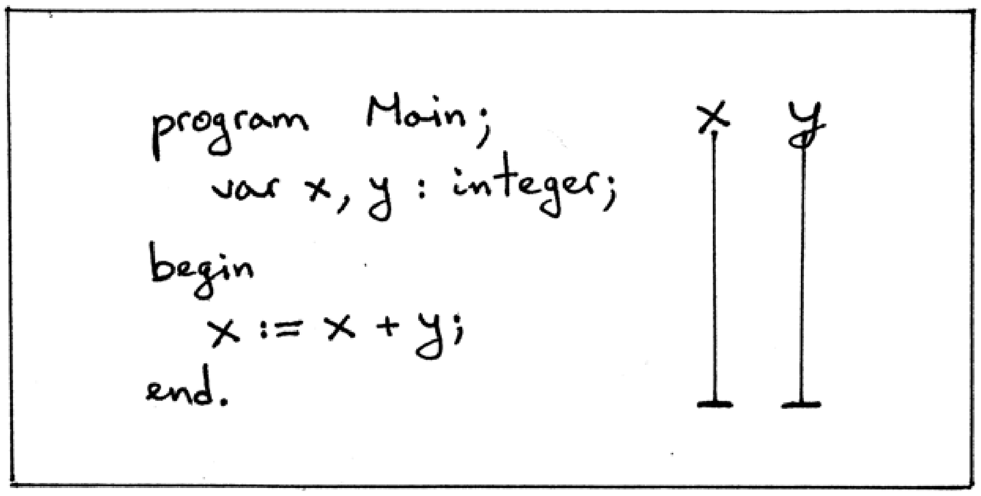
```
程序 主程序;
   var x , y : 整数;
开始
   x := x + y ;
结束。
```

在 Pascal 中，PROGRAM关键字（顺便说一下不区分大小写）引入了一个新的作用域，通常称为全局作用域，因此上面的程序具有一个全局作用域，并且声明的变量**x**和**y**在整个程序中都是可见和可访问的。在上面的例子中，文本区域以关键字program开始，以关键字end和一个点结束。在该文本区域中，名称**x**和**y**都可以使用，因此这些变量（变量声明）的范围是整个程序：



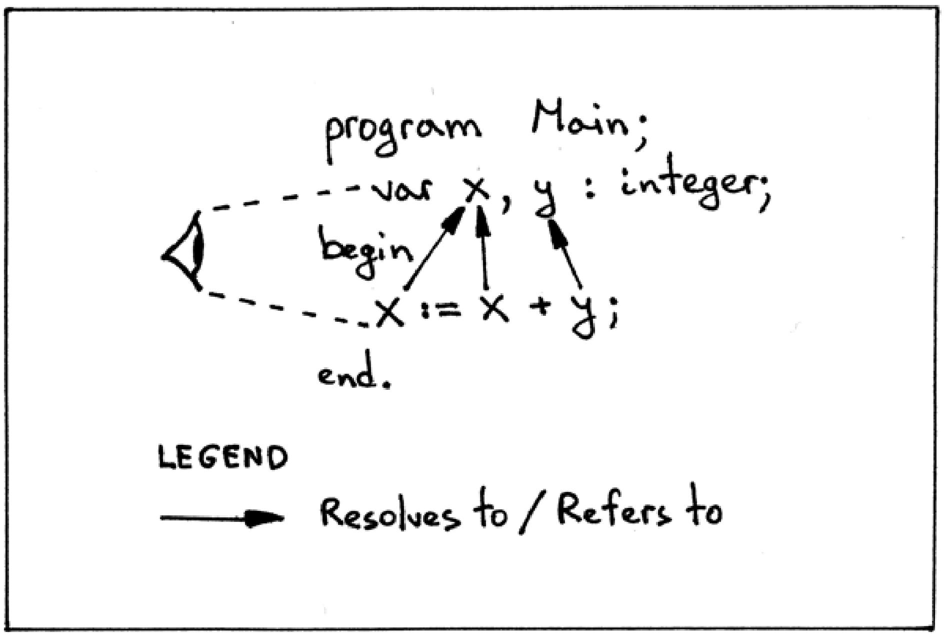
当您查看上面的源代码，特别是表达式**x := x + y 时**，您直观地知道它应该可以毫无问题地编译（或被解释），因为表达式中变量**x**和**y**的范围是全局范围和可变的引用**X**和**ÿ**在表达式**X：= X + Y**决心声明整数变量**X**和**ÿ**。如果您以前使用任何主流编程语言进行过编程，那么这里应该不会有任何意外。

当我们谈论一个变量的范围时，我们实际上是在谈论它的声明范围：



上图中，竖线表示声明变量的范围，可以使用声明名称**x**和**y**的文本区域，即它们可见的文本区域。如您所见，**x**和**y**的范围是整个程序，如垂直线所示。

Pascal 程序被称为**词法作用域**（或**静态作用域**），因为您可以查看源代码，甚至无需执行程序，纯粹根据文本规则确定哪些名称（引用）解析或引用哪些声明。例如，在 Pascal 中，诸如program和end 之类的词法关键字划定了范围的文本边界：



Why are scopes useful?

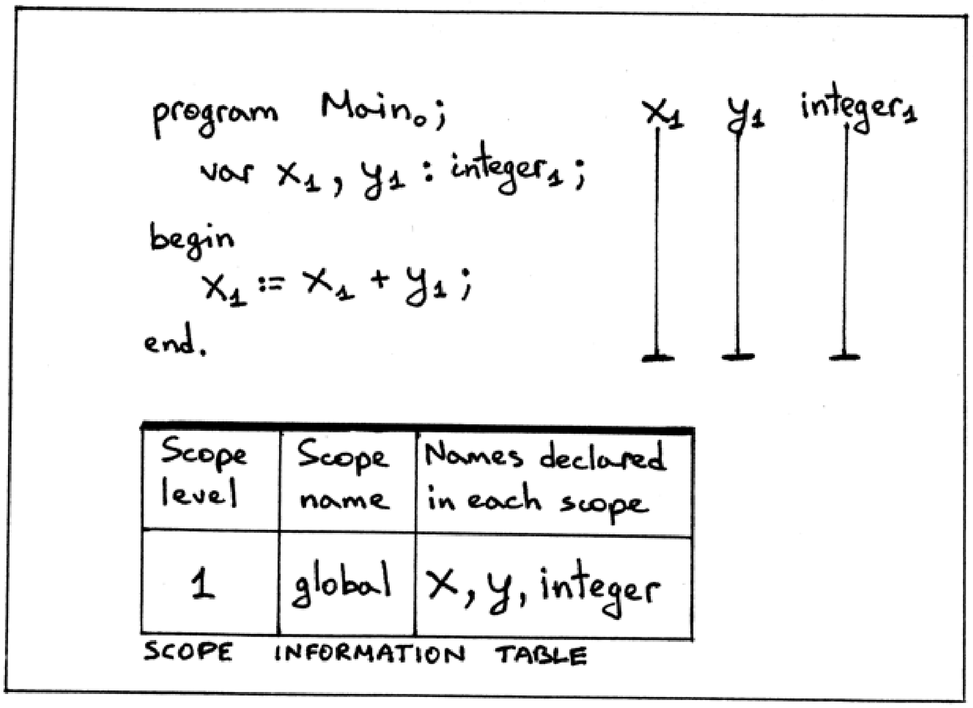
- Every scope creates an isolated name space, which means that variables declared in a scope cannot be accessed from outside of it.
- You can re-use the same name in different scopes and know exactly, just by looking at the program source code, what declaration the name refers to at every point in the program.
- In a nested scope you can re-declare a variable with the same name as in the outer scope, thus effectively hiding the outer declaration, which gives you control over access to different variables from the outer scope.

In addition to the global scope, Pascal supports nested procedures, and every procedure declaration introduces a new scope, which means that Pascal supports nested scopes.

When we talk about nested scopes, it’ s convenient to talk about scope levels to show their nesting relationships. It’ s also convenient to refer to scopes by name. We’ ll use both scope levels and scope names when we start our discussion of nested scopes.

Let’ s take a look at the following sample program and subscript every name in the program to make it clear:

1. At what level each variable (symbol) is declared
2. To which declaration and at what level a variable name refers to:



program $Main_0$;
   var x_1, y_1 : $integer_1$;
begin
   x_1 := x_1 + y_1;
end.

Scope level	Scope name	Names declared in each scope
1	global	x, y, integer

SCOPE INFORMATION TABLE

From the picture above we can see several things:

- We have a single scope, the global scope, introduced by the PROGRAM keyword
- Global scope is at level 1
- Variables (symbols) **x** and **y** are declared at level 1 (the global scope).
- integer built-in type is also declared at level 1
- The program name **Main** has a subscript 0. Why is the program’ s name at level zero, you might wonder? This is to make it clear that the program’ s name is not in the global scope and it’ s in some other outer scope, that has level zero.
- The scope of the variables **x** and **y** is the whole program, as shown by the vertical lines
- The scope information table shows for every level in the program the corresponding scope level, scope name, and names declared in the scope. The purpose of the table is to summarize and visually show different information about scopes in a program.

How do we implement the concept of a scope in code? To represent a scope in code, we’ ll need a scoped symbol table. We already know about symbol tables, but what is a scoped symbol table? A **scoped symbol table** is basically a symbol table with a few modifications, as you’ ll see shortly.

From now on, we’ ll use the word scope both to mean the concept of a scope as well as to refer to the scoped symbol table, which is an implementation of the scope in code.

Even though in our code a scope is represented by an instance of the ScopedSymbolTable class, we’ ll use the variable named scope throughout the code for convenience. So when you see a variable scope in the code of our interpreter, you should know that it actually refers to a scoped symbol table.

Okay, let’ s enhance our SymbolTable class by renaming it to ScopedSymbolTable class, adding two new fields scope_level and scope_name, and updating the scoped symbol table’ s constructor. And at the same time, let’ s update the __str__ method to print additional information, namely the scope_level and scope_name. Here is a new version of the symbol table, the ScopedSymbolTable:

```
class ScopedSymbolTable(object):
    def __init__(self, scope_name, scope_level):
        self._symbols = OrderedDict()
        self.scope_name = scope_name
        self.scope_level = scope_level
        self._init_builtins()

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

    def __str__(self):
        h1 = 'SCOPE (SCOPED SYMBOL TABLE)'
        lines = ['\n', h1, '=' * len(h1)]
        for header_name, header_value in (
            ('Scope name', self.scope_name),
            ('Scope level', self.scope_level),
        ):
            lines.append('%-15s: %s' % (header_name, header_value))
        h2 = 'Scope (Scoped symbol table) contents'
        lines.extend([h2, '-' * len(h2)])
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        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
```

Let’s also update the semantic analyzer’s code to use the variable scope instead of symtab, and remove the semantic check that was checking source programs for duplicate identifiers from the visit_VarDecl method to reduce the noise in the program output.

Here is a piece of code that shows how our semantic analyzer instantiates the ScopedSymbolTable class:

```
class SemanticAnalyzer(NodeVisitor):
    def __init__(self):
        self.scope = ScopedSymbolTable(scope_name='global', scope_level=1)

    ...
```

You can find all the changes in the file scope01.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope01.py). Download the file, run it on the command line, and inspect the output. Here is what I got:

```
$ python scope01.py
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: x
Lookup: INTEGER
Insert: y
Lookup: x
Lookup: y
Lookup: x


SCOPE (SCOPED SYMBOL TABLE)
==========================
Scope name     : global
Scope level    : 1
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>
```

Most of the output should look very familiar to you.

Now that you know about the concept of scope and how to implement the scope in code by using a scoped symbol table, it’ s time we talked about nested scopes and more dramatic modifications to the scoped symbol table than just adding two simple fields.

Procedure declarations with formal parameters

Let’ s take a look at a sample program in the file nestedscopes02.pas (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes02.pas) that contains a procedure declaration:

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

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

The first thing that we notice here is that we have a procedure with a parameter, and we haven’ t learned how to handle that yet. Let’ s fill that gap by making a quick detour and learning how to handle formal procedure parameters before continuing with scopes.*

> *ASIDE: Formal parameters are parameters that show up in the declaration of a procedure. Arguments (also called actual parameters) are different variables and expressions passed to a procedure in a particular procedure call.

Here is a list of changes we need to make to support procedure declarations with parameters:

1. Add the Param AST node

```
class Param(AST):
    def __init__(self, var_node, type_node):
        self.var_node = var_node
        self.type_node = type_node
```

2. Update the ProcedureDecl node’s constructor to take an additional argument: params

```
class ProcedureDecl(AST):
    def __init__(self, proc_name, params, block_node):
        self.proc_name = proc_name
        self.params = params  # a list of Param nodes
        self.block_node = block_node
```

3. Update the declarations rule to reflect changes in the procedure declaration sub-rule

```
def declarations(self):
    """declarations : (VAR (variable_declaration SEMI)+)*
                    | (PROCEDURE ID (LPAREN formal_parameter_list RPAREN)? SEMI block SEMI)*
                    | empty
    """
```

4. Add the formal_parameter_list rule and method

```
def formal_parameter_list(self):
    """ formal_parameter_list : formal_parameters
                              | formal_parameters SEMI formal_parameter_list
    """
```

5. Add the formal_parameters rule and method

```
def formal_parameters(self):
    """ formal_parameters : ID (COMMA ID)* COLON type_spec """
    param_nodes = []
```

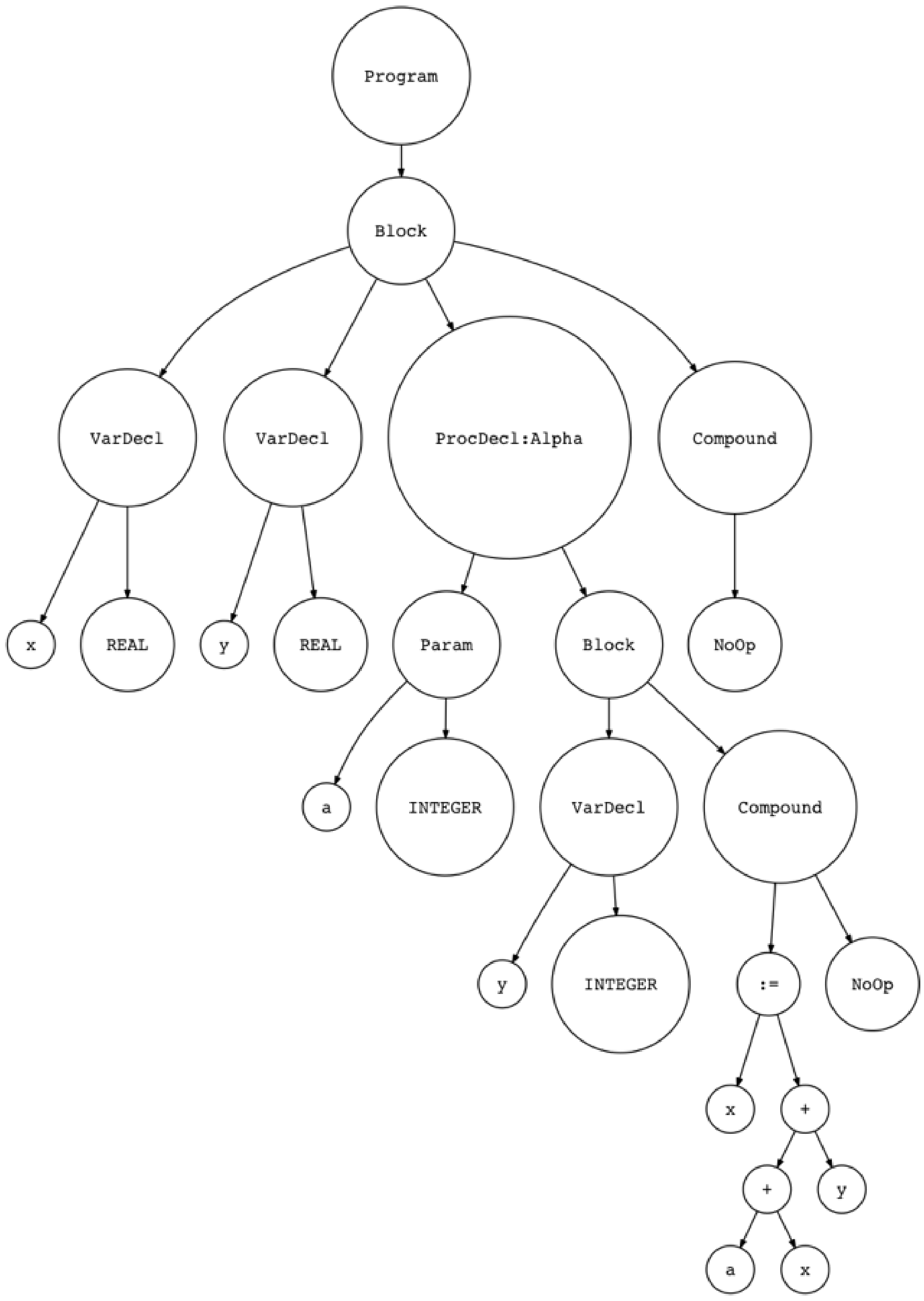
With the addition of the above methods and rules our parser will be able to parse procedure declarations like these (I’m not showing the body of declared procedures for brevity):

```
procedure Foo;

procedure Foo(a : INTEGER);

procedure Foo(a, b : INTEGER);

procedure Foo(a, b : INTEGER; c : REAL);
```

Let’s generate an AST for our sample program. Download genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part14/genastdot.py) and run the following command on the command line:

```
$ python genastdot.py nestedscopes02.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```

Here is a picture of the generated AST:



You can see now that the ProcedureDecl node in the picture has the Param node as its child.

You can find the complete changes in the spi.py (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py) file. Spend some time and study the changes. You’ ve done similar changes before; they should be pretty easy to understand and you should be able to implement them by yourself.

Procedure symbols

While we’ re on the topic of procedure declarations, let’ s also talk about procedure symbols.

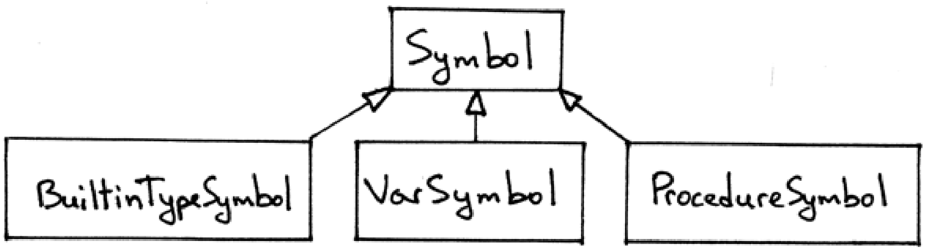
As with variable declarations, and built-in type declarations, there is a separate category of symbols for procedures. Let’ s create a separate symbol class for procedure symbols:

```
class ProcedureSymbol(Symbol):
    def __init__(self, name, params=None):
        super(ProcedureSymbol, self).__init__(name)
        # a list of formal parameters
        self.params = params if params is not None else []

    def __str__(self):
        return '<{class_name}(name={name}, parameters={params})>'.format(
            class_name=self.__class__.__name__,
            name=self.name,
            params=self.params,
        )

    __repr__ = __str__
```

Procedure symbols have a name (it’s a procedure’s name), their category is procedure (it’s encoded in the class name), and the type is None because in Pascal procedures don’t return anything.

Procedure symbols also carry additional information about procedure declarations, namely they contain information about the procedure’s formal parameters as you can see in the code above.

With the addition of procedure symbols, our new symbol hierarchy looks like this:



Nested scopes

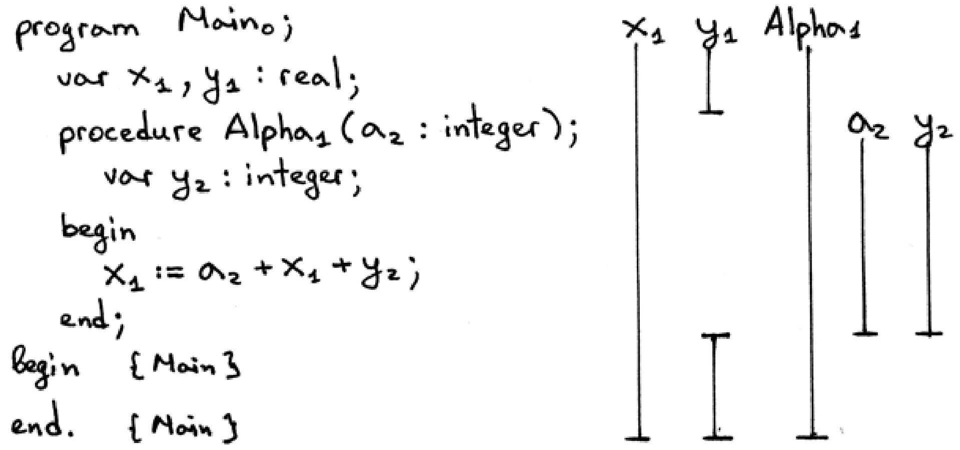
After that quick detour let’s get back to our program and the discussion of nested scopes:

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

   procedure Alpha(a : integer);
      var y : integer;
   begin
      x := a + x + y;
   end;

begin { Main }

end.  { Main }
```

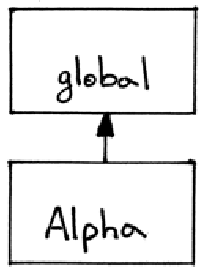
Things are actually getting more interesting here. By declaring a new procedure, we introduce a new scope, and this scope is nested within the global scope introduced by the PROGRAM statement, so this is a case where we have nested scopes in a Pascal program.

The scope of a procedure is the whole body of the procedure. The beginning of the procedure scope is marked by the PROCEDURE keyword and the end is marked by the END keyword and a semicolon.

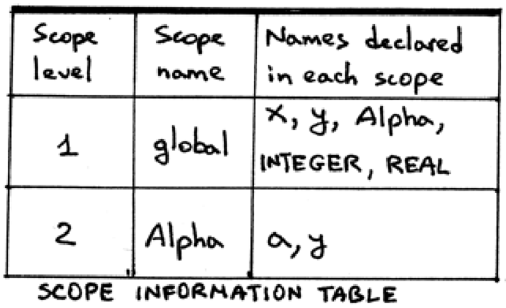
Let’s subscript names in the program and show some additional information:



```
program Main0;
   var x1, y1 : real;
   procedure Alpha1(a2 : integer);
      var y2 : integer;
   begin
      x1 := a2 + x1 + y2;
   end;
begin  {Main}
end.  {Main}
```



NESTING RELATIONSHIPS



Scope level	Scope name	Names declared in each scope
1	global	x, y, Alpha, INTEGER, REAL
2	Alpha	a, y

SCOPE INFORMATION TABLE

Some observations from the picture above:

- This Pascal program has two scope levels: level 1 and level 2
- The nesting relationships diagram visually shows that the scope Alpha is nested within the global scope, hence there are two levels: the global scope at level 1, and the Alpha scope at level 2.
- The scope level of the procedure declaration Alpha is one less than the level of the variables declared inside the procedure Alpha. You can see that the scope level of the procedure declaration Alpha is 1 and the scope level of the variables **a** and **y** inside the procedure is 2.
- The variable declaration of **y** inside Alpha hides the declaration of **y** in the global scope. You can see the hole in the vertical bar for **y1** (by the way, 1 is a subscript, it’ s not part of the variable name, the variable name is just **y**) and you can see that the scope of the **y2** variable declaration is the Alpha procedure’ s whole body.
- The scope information table, as you are already aware, shows scope levels, scope names for those levels, and respective names declared in those scopes (at those levels).
- In the picture, you can also see that I omitted showing the scope of the integer and real types (except in the scope information table) because they are always declared at scope level 1, the global scope, so I won’ t be subscripting the integer and real types anymore to save visual space, but you will see the types again and again in the contents of the scoped symbol table representing the global scope.

The next step is to discuss implementation details.

First, let’ s focus on variable and procedure declarations. Then, we’ ll discuss variable references and how name resolution works in the presence of nested scopes.

For our discussion, we’ ll use a stripped down version of the program. The following version does not have variable references: it only has variable and procedure declarations:

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

   procedure Alpha(a : integer);
      var y : integer;
   begin

   end;

begin { Main }

end.  { Main }
```

You already know how to represent a scope in code with a scoped symbol table. Now we have two scopes: the global scope and the scope introduced by the procedure Alpha. Following our approach we should now have two scoped symbol tables: one for the global scope and one for the Alpha scope. How do we implement that in code? We’ ll extend the semantic analyzer to create a separate scoped symbol table for every scope instead of just for the global scope. The scope construction will happen, as usual, when walking the AST.

First, we need to decide where in the semantic analyzer we’ re going to create our scoped symbol tables. Recall that PROGRAM and PROCEDURE keywords introduce new scope. In AST, the corresponding nodes are Program and ProcedureDecl. So we’ re going to update our visit_Program method and add the visit_ProcedureDecl method to create scoped symbol tables. Let’ s start with the visit_Program method:

```
def visit_Program(self, node):
    print('ENTER scope: global')
    global_scope = ScopedSymbolTable(
        scope_name='global',
        scope_level=1,
    )
    self.current_scope = global_scope

    # visit subtree
    self.visit(node.block)

    print(global_scope)
    print('LEAVE scope: global')
```

The method has quite a few changes:

1. When visiting the node in AST, we first print what scope we’ re entering, in this case global.
2. We create a separate scoped symbol table to represent the global scope. When we construct an instance of ScopedSymbolTable, we explicitly pass the scope name and scope level arguments to the class constructor.
3. We assign the newly created scope to the instance variable current_scope. Other visitor methods that insert and look up symbols in scoped symbol tables will use the current_scope.
4. We visit a subtree (block). This is the old part.
5. Before leaving the global scope we print the contents of the global scope (scoped symbol table)
6. We also print the message that we’ re leaving the global scope

Now let’ s add the visit_ProcedureDecl method. Here is the complete source code for it:

```
def visit_ProcedureDecl(self, node):
    proc_name = node.proc_name
    proc_symbol = ProcedureSymbol(proc_name)
    self.current_scope.insert(proc_symbol)

    print('ENTER scope: %s' %  proc_name)
    # Scope for parameters and local variables
    procedure_scope = ScopedSymbolTable(
        scope_name=proc_name,
        scope_level=2,
    )
    self.current_scope = procedure_scope

    # Insert parameters into the procedure scope
    for param in node.params:
        param_type = self.current_scope.lookup(param.type_node.value)
        param_name = param.var_node.value
        var_symbol = VarSymbol(param_name, param_type)
        self.current_scope.insert(var_symbol)
        proc_symbol.params.append(var_symbol)

    self.visit(node.block_node)

    print(procedure_scope)
    print('LEAVE scope: %s' %  proc_name)
```

Let’s go over the contents of the method:

1. The first thing that the method does is create a procedure symbol and insert it into the current scope, which is the global scope for our sample program.
2. Then the method prints the message about entering the procedure scope.
3. Then we create a new scope for the procedure’s parameters and variable declarations.
4. We assign the procedure scope to the self.current_scope variable indicating that this is our current scope and all symbol operations (insert and lookup) will use the current scope.
5. Then we handle procedure formal parameters by inserting them into the current scope and adding them to the procedure symbol.
6. Then we visit the rest of the AST subtree - the body of the procedure.
7. And, finally, we print the message about leaving the scope before leaving the node and moving to another AST node, if any.

Now, what we need to do is update other semantic analyzer visitor methods to use self.current_scope when inserting and looking up symbols. Let’s do that:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.current_scope.lookup(type_name)

    # We have all the information we need to create a variable symbol.
    # Create the symbol and insert it into the symbol table.
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)

    self.current_scope.insert(var_symbol)

def visit_Var(self, node):
    var_name = node.value
    var_symbol = self.current_scope.lookup(var_name)
    if var_symbol is None:
        raise Exception(
            "Error: Symbol(identifier) not found '%s'" % var_name
        )
```

Both the visit_VarDecl and visit_Var will now use the current_scope to insert and/or look up symbols. Specifically, for our sample program, the current_scope can point either to the global scope or the Alpha scope.

We also need to update the semantic analyzer and set the current_scope to None in the constructor:

```
class SemanticAnalyzer(NodeVisitor):
    def __init__(self):
        self.current_scope = None
```

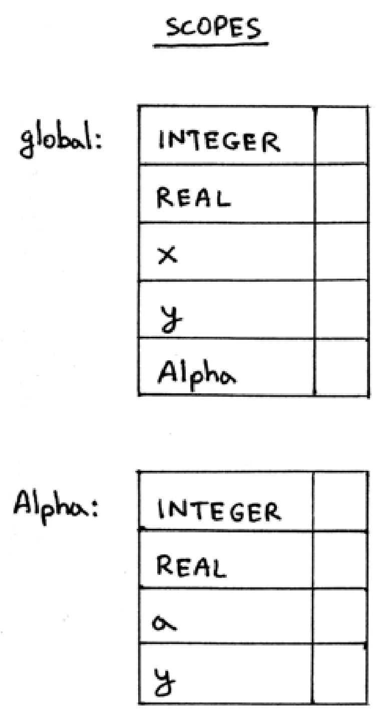
Clone the GitHub repository for the article (https://github.com/rspivak/lsbasi), run scope02.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope02.py) (it has all the changes we just discussed), inspect the output, and make sure you understand why every line is generated:

```
$ python scope02.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: Alpha
ENTER scope: Alpha
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

Some things about the output above that I think are worth mentioning:

1. You can see that the two lines Insert: INTEGER and Insert: REAL are repeated twice in the output and the keys INTEGER and REAL are present in both scopes (scoped symbol tables): global and Alpha. The reason is that we create a separate scoped symbol table for every scope and the table initializes the built-in type symbols every time we create its instance. We’ll change it later when we discuss nesting relationships and how they are expressed in code.
2. See how the line Insert: Alpha is printed before the line ENTER scope: Alpha. This is just a reminder that a name of a procedure is declared at a level that is one less than the level of the variables declared in the procedure itself.
3. You can see by inspecting the printed contents of the scoped symbol tables above what declarations they contain. See, for example, that global scope has the Alpha symbol in it.
4. From the contents of the global scope you can also see that the procedure symbol for the Alpha procedure also contains the procedure’s formal parameters.

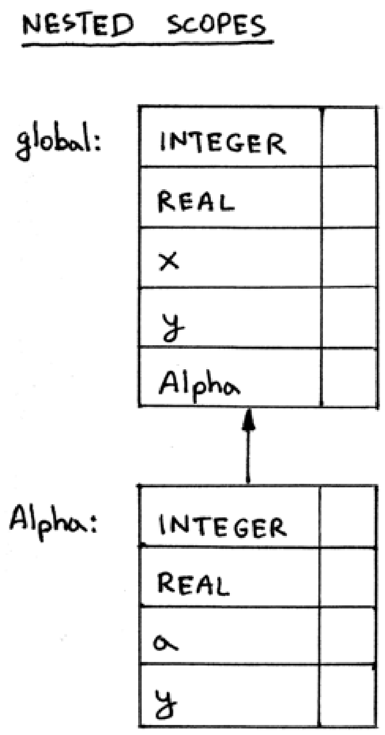
After we run the program, our scopes in memory would look something like this, just two separate scoped symbol tables:



Scope tree: Chaining scoped symbol tables

Okay, now every scope is represented by a separate scoped symbol table, but how do we represent the nesting relationship between the global scope and the scope Alpha as we showed in the nesting relationship diagram before? In other words, how do we express in code that the scope Alpha is nested within the global scope? The answer is chaining the tables together.

We’ll chain the scoped symbol tables together by creating a link between them. In a way it’ll be like a tree (we’ll call it a scope tree), just an unusual one, because in this tree a child will be pointing to a parent, and not the other way around. Let’s take a look the following scope tree:



In the scope tree above you can see that the scope Alpha is linked to the global scope by pointing to it. To put it differently, the scope Alpha is pointing to its enclosing scope, which is the global scope. It all means that the scope Alpha is nested within the global scope.

How do we implement scope chaining/linking? There are two steps:

1. We need to update the ScopedSymbolTable class and add a variable enclosing_scope that will hold a pointer to the scope’ s enclosing scope. This will be the link between scopes in the picture above.
2. We need to update the visit_Program and visit_ProcedureDecl methods to create an actual link to the scope’ s enclosing scope using the updated version of the ScopedSymbolTable class.

Let’ s start with updating the ScopedSymbolTable class and adding the enclosing_scope field. Let’ s also update the __init__ and __str__ methods. The __init__ method will be modified to accept a new parameter, enclosing_scope, with the default value set to None. The __str__ method will be updated to output the name of the enclosing scope. Here is the complete source code of the updated ScopedSymbolTable class:

```
class ScopedSymbolTable(object):
    def __init__(self, scope_name, scope_level, enclosing_scope=None):
        self._symbols = OrderedDict()
        self.scope_name = scope_name
        self.scope_level = scope_level
        self.enclosing_scope = enclosing_scope
        self._init_builtins()

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

    def __str__(self):
        h1 = 'SCOPE (SCOPED SYMBOL TABLE)'
        lines = ['\n', h1, '=' * len(h1)]
        for header_name, header_value in (
            ('Scope name', self.scope_name),
            ('Scope level', self.scope_level),
            ('Enclosing scope',
             self.enclosing_scope.scope_name if self.enclosing_scope else None
            )
        ):
            lines.append('%-15s: %s' % (header_name, header_value))
        h2 = 'Scope (Scoped symbol table) contents'
        lines.extend([h2, '-' * len(h2)])
        lines.extend(
            ('%7s: %r' % (key, value))
            for key, value in self._symbols.items()
        )
        lines.append('\n')
        s = '\n'.join(lines)
        return s

    __repr__ = __str__

    def insert(self, symbol):
        print('Insert: %s' % symbol.name)
        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
```

Now let’ s switch our attention to the visit_Program method:

```
def visit_Program(self, node):
    print('ENTER scope: global')
    global_scope = ScopedSymbolTable(
        scope_name='global',
        scope_level=1,
        enclosing_scope=self.current_scope, # None
    )
    self.current_scope = global_scope

    # visit subtree
    self.visit(node.block)

    print(global_scope)

    self.current_scope = self.current_scope.enclosing_scope
    print('LEAVE scope: global')
```

There are a couple of things here worth mentioning and repeating:

1. We explicitly pass the self.current_scope as the enclosing_scope argument when creating a scope
2. We assign the newly created global scope to the variable self.current_scope
3. We restore the variable self.current_scope to its previous value right before leaving the Program node. It’s important to restore the value of the current_scope after we’ve finished processing the node, otherwise the scope tree construction will be broken when we have more than two scopes in our program. We’ll see why shortly.

And, finally, let’s update the visit_ProcedureDecl method:

```
def visit_ProcedureDecl(self, node):
    proc_name = node.proc_name
    proc_symbol = ProcedureSymbol(proc_name)
    self.current_scope.insert(proc_symbol)

    print('ENTER scope: %s' %  proc_name)
    # Scope for parameters and local variables
    procedure_scope = ScopedSymbolTable(
        scope_name=proc_name,
        scope_level=self.current_scope.scope_level + 1,
        enclosing_scope=self.current_scope
    )
    self.current_scope = procedure_scope

    # Insert parameters into the procedure scope
    for param in node.params:
        param_type = self.current_scope.lookup(param.type_node.value)
        param_name = param.var_node.value
        var_symbol = VarSymbol(param_name, param_type)
        self.current_scope.insert(var_symbol)
        proc_symbol.params.append(var_symbol)

    self.visit(node.block_node)

    print(procedure_scope)

    self.current_scope = self.current_scope.enclosing_scope
    print('LEAVE scope: %s' %  proc_name)
```

Again, the main changes compared to the version in scope02.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope02.py) are:

1. We explicitly pass the self.current_scope as an enclosing_scope argument when creating a scope.
2. We no longer hard code the scope level of a procedure declaration because we can calculate the level automatically based on the scope level of the procedure’s enclosing scope: it’s the enclosing scope’s level plus one.
3. We restore the value of the self.current_scope to its previous value (for our sample program the previous value would be the global scope) right before leaving the ProcedureDecl node.

Okay, let’s see what the contents of the scoped symbol tables look like with the above changes. You can find all the changes in scope03a.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03a.py). Our sample program is:

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

   过程 阿尔法（a  ：  整数）;
      var  y  ：  整数;
   开始

   结束;

开始 {主要}

结束。  { 主要的 }
```

在命令行上运行 scope03a.py 并检查输出：

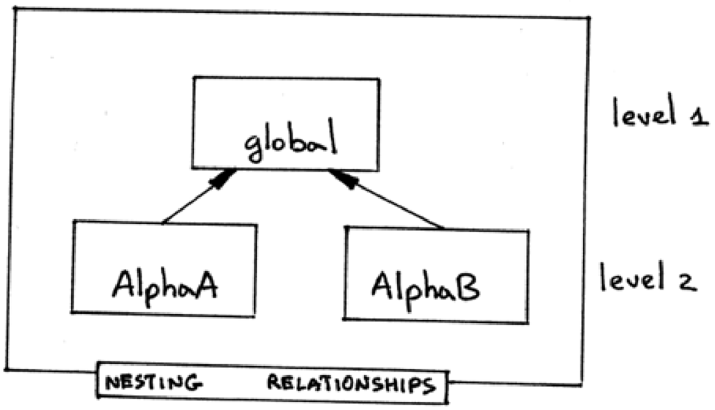
```
$ python scope03a.py
输入范围：全局
插入：整数
插入：真实
查找：真实
插入：x
查找：真实
插入：y
插入：阿尔法
输入范围：Alpha
插入：整数
插入：真实
查找：整数
插入：一个
查找：整数
插入：y


范围（范围符号表）
============================
范围名称：阿尔法
范围级别：2
封闭范围：全球
范围（ Scoped Symbol table ）内容
---------------------
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      a: <VarSymbol ( name = 'a' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >


离开范围：阿尔法


范围（范围符号表）
============================
范围名称：全球
范围级别：1
封闭范围：无
范围（ Scoped Symbol table ）内容
---------------------
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'REAL' ) >
      y: <VarSymbol ( name = 'y' , type = 'REAL' ) >
  Alpha: <ProcedureSymbol ( name = Alpha, parameters =[ <VarSymbol ( name = 'a' , type = 'INTEGER' ) > ]) >


LEAVE 范围：全球
```

您可以在上面的输出中看到全局作用域没有封闭作用域，并且Alpha的封闭作用域是全局作用域，这正是我们所期望的，因为Alpha作用域嵌套在全局作用域内。

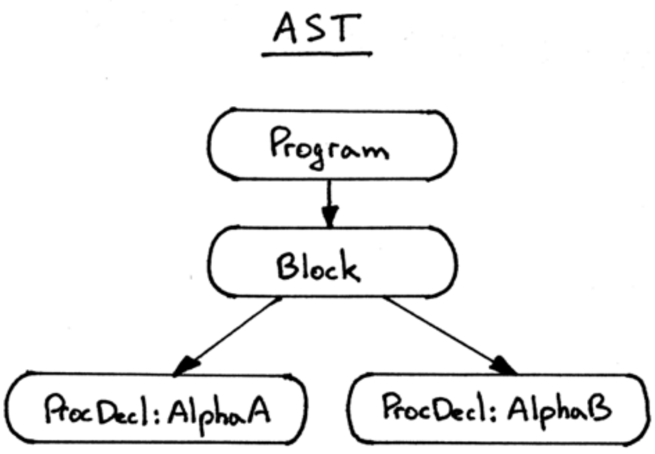
现在，正如所承诺的，让我们考虑为什么设置和恢复self.current_scope变量的值很重要。让我们看看下面的程序，我们在全局范围内有两个过程声明：

```
程序 主程序;
   var  x ,  y  : 实数;

   过程 AlphaA ( a  : 整数) ;
      var  y  :  整数;
   开始 { AlphaA }

   结束;   {阿尔法A}

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

   end;  { AlphaB }

begin { Main }

end.  { Main }
```

The nesting relationship diagram for the sample program looks like this:



An AST for the program (I left only the nodes that are relevant to this example) is something like this:



If we don’ t restore the current scope when we leave the Program and ProcedureDecl nodes what is going to happen? Let’ s see.

The way our semantic analyzer walks the tree is depth first, left-to-right, so it will traverse the ProcedureDecl node for AlphaA first and then it will visit the ProcedureDecl node for AlphaB. The problem here is that if we don’t restore the self.current_scope before leaving AlphaA the self.current_scope will be left pointing to AlphaA instead of the global scope and, as a result, the semantic analyzer will create the scope AlphaB at level 3, as if it was nested within the scope AlphaA, which is, of course, incorrect.

To see the broken behavior when the current scope is not being restored before leaving Program and/or ProcedureDecl nodes, download and run the scope03b.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03b.py) on the command line:

```
$ python scope03b.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL
Insert: x
Lookup: REAL
Insert: y
Insert: AlphaA
ENTER scope: AlphaA
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: y


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaA
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: AlphaA
Insert: AlphaB
ENTER scope: AlphaB
Insert: INTEGER
Insert: REAL
Lookup: INTEGER
Insert: a
Lookup: INTEGER
Insert: b


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : AlphaB
Scope level    : 3
Enclosing scope: AlphaA
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      a: <VarSymbol(name='a', type='INTEGER')>
      b: <VarSymbol(name='b', type='INTEGER')>


LEAVE scope: AlphaB


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
------------------------------------
```

```
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
 AlphaA: <ProcedureSymbol(name=AlphaA, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

如您所见，我们的语义分析器中的作用域树构建在存在两个以上作用域时完全被破坏：

1. 我们有三个级别，而不是嵌套关系图中所示的两个范围级别
2. 在全球范围内的内容不具有AlphaB在里面，只有AlphaA。

要正确构建作用域树，我们需要遵循一个非常简单的过程：

1. 当我们ENTER一个程序或ProcedureDecl节点，我们创建了一个新的范围，并将其分配给self.current_scope。
2. 当我们即将LEAVE的计划或ProcedureDecl节点，我们恢复的价值self.current_scope。

您可以将self.current_scope视为堆栈指针，将范围树视为堆栈的集合：

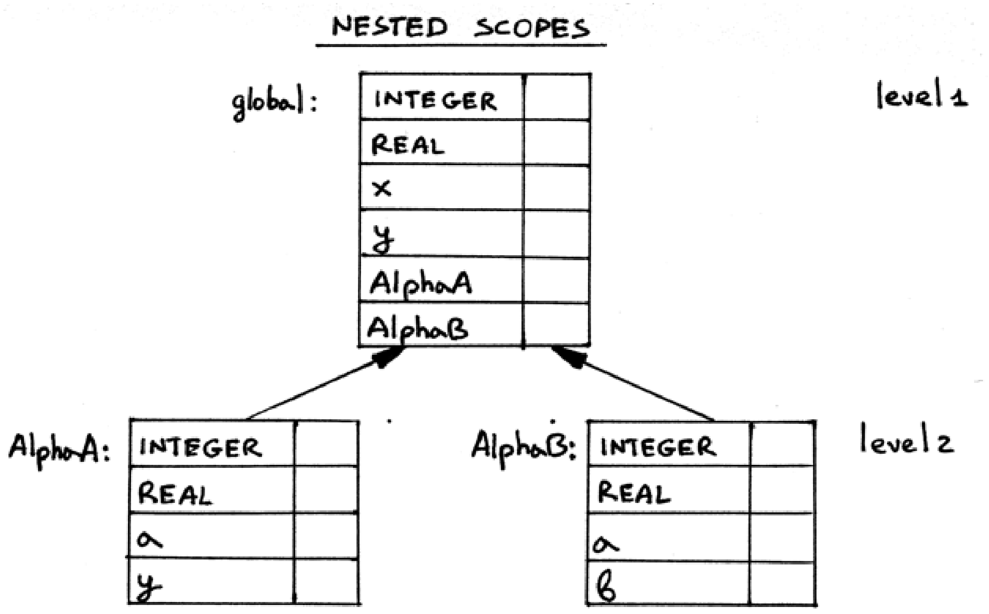
1. 当您访问Program或ProcedureDecl节点时，您将一个新的作用域推入堆栈，并将堆栈指针self.current_scope调整为指向堆栈的顶部，现在是最近推入的作用域。
2. 当您将要离开节点时，您将作用域从堆栈中弹出，并且您还将堆栈指针调整为指向堆栈上的前一个作用域，它现在是新的栈顶。

要查看存在多个作用域时的正确行为，请在命令行上下载并运行scope03c.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03c.py)。研究输出。确保您了解正在发生的事情：

```
$ python scope03c.py
输入范围：全局
插入：整数
插入：真实
查找：真实
插入：x
查找：真实
插入：y
插入：AlphaA
输入范围：AlphaA
插入：整数
插入：真实
查找：整数
插入：一个
查找：整数
插入：y


范围（范围符号表）
============================
范围名称：AlphaA
范围级别：2
封闭范围：全球
范围（ Scoped Symbol table ）内容
---------------------
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      a: <VarSymbol ( name = 'a' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >


离开范围：AlphaA
插入：AlphaB
输入范围：AlphaB
插入：整数
插入：真实
查找：整数
插入：一个
查找：整数
插入：b


范围（范围符号表）
============================
范围名称：AlphaB
范围级别：2
封闭范围：全球
范围（ Scoped Symbol table ）内容
---------------------
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      a: <VarSymbol ( name = 'a' , type = 'INTEGER' ) >
      b: <VarSymbol ( name = 'b' , type = 'INTEGER' ) >


离开范围：AlphaB


范围（范围符号表）
============================
范围名称：全球
范围级别：1
封闭范围：无
范围（ Scoped Symbol table ）内容
---------------------
```

```
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'REAL' ) >
      y: <VarSymbol ( name = 'y' , type = 'REAL' ) >
 AlphaA: <ProcedureSymbol ( name = AlphaA, parameters =[ <VarSymbol ( name = 'a' , type = 'INTEGER' ) > ]) >
 AlphaB: <ProcedureSymbol ( name = AlphaB, parameters =[ <VarSymbol ( name = 'a' , type = 'INTEGER' ) > ]) >


LEAVE 范围：全球
```

这就是我们运行scope03c.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope03c.py)并正确构建作用域树后作用域符号表的样子：



同样，正如我上面提到的，您可以将上面的作用域树视为作用域堆栈的集合。

现在让我们继续讨论当我们有嵌套作用域时名称解析是如何工作的。

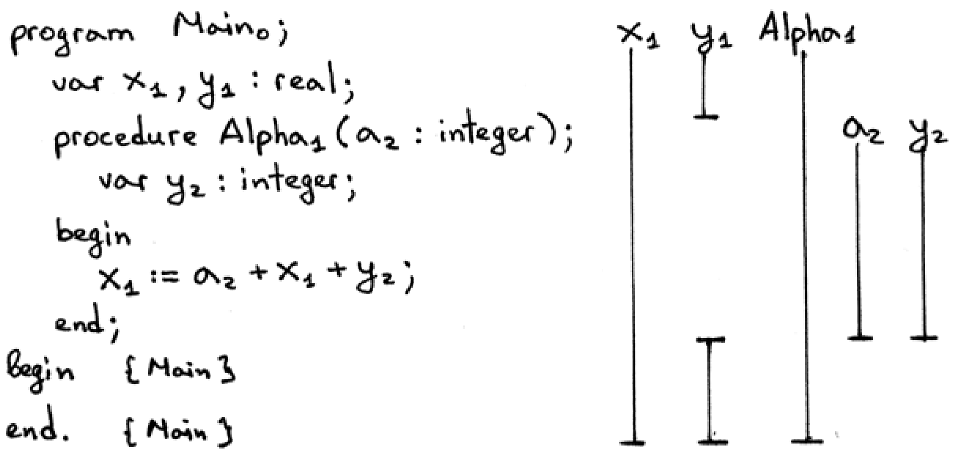
嵌套范围和名称解析

我们之前的重点是变量和过程声明。让我们在混合中添加变量引用。

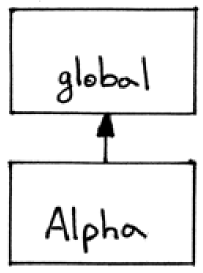
这是一个包含一些变量引用的示例程序：

```
程序 主程序;
   var  x ,  y : 实数;

   过程 阿尔法（a  ：  整数）;
      var  y  ：  整数;
   开始
      x  :=  a  +  x  +  y ;
   结束;

开始 {主要}

结束。  { 主要的 }
```

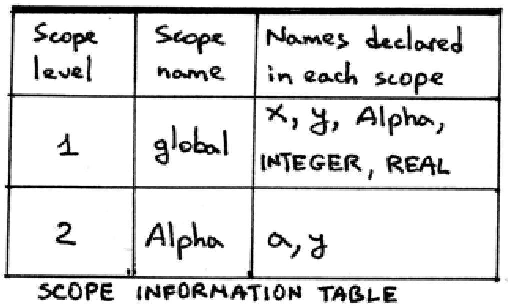
或者在视觉上添加一些附加信息：



```
program Main0;
   var x1, y1 : real;
   procedure Alpha1(a2 : integer);
      var y2 : integer;
   begin
      x1 := a2 + x1 + y2;
   end;
begin  {Main}
end.  {Main}
```



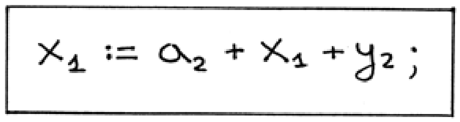
NESTING RELATIONSHIPS



Scope level	Scope name	Names declared in each scope
1	global	x, y, Alpha, INTEGER, REAL
2	Alpha	a, y

SCOPE INFORMATION TABLE

让我们把注意力转向赋值语句**x := a + x + y;** 这是带下标的：

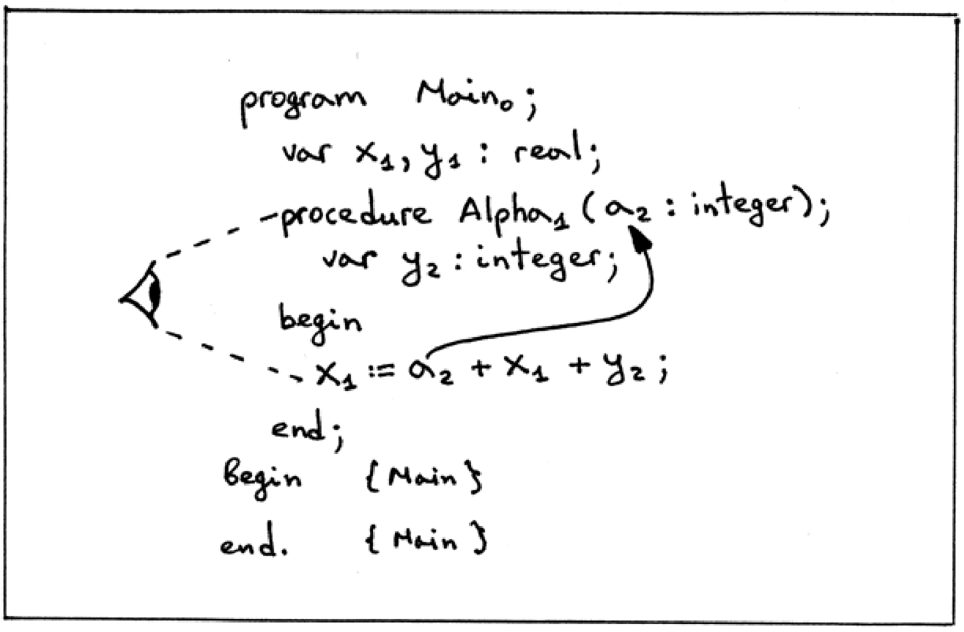


$$x_1 := a_2 + x_1 + y_2;$$

我们看到**x**解析为级别 1 的声明，**a**解析为级别 2 的声明，而**y**也解析为级别 2的声明。该解析如何工作？让我们看看如何。

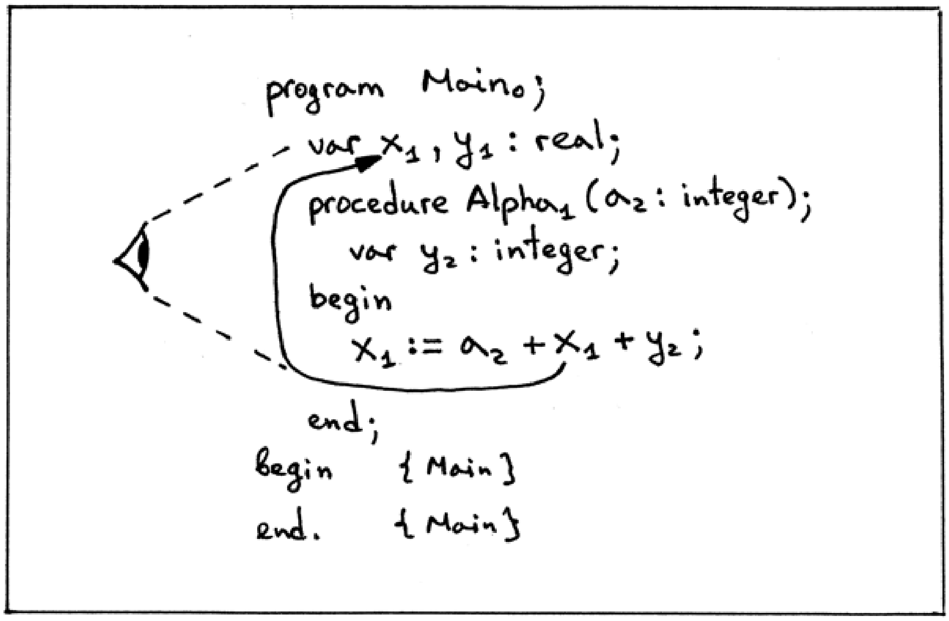
在名称解析方面，像 Pascal 这样的词法（静态）范围语言遵循**最紧密嵌套的范围**规则。这意味着，在每个作用域中，名称都指向其词法上最接近的声明。对于我们的赋值语句，让我们回顾一下每个变量引用，看看规则在实践中是如何工作的：

1. 因为我们的语义分析器首先访问赋值的右侧，所以我们将从算术表达式**a + x + y 中**的变量引用**a**开始。我们开始在词法上最接近的范围（即Alpha范围）中搜索**a**的声明。本阿尔法范围包含在变量声明阿尔法程序，包括程序的正式参数。我们找到的声明**—**在阿尔法范围：它的形参**—**对的阿尔法程序-有类型的变量符号**整数**. 我们在解析名字的时候一般都是用眼睛扫描源码来进行搜索的（记住，**a2**不是变量名，这里2是下标，变量名是**a**）：



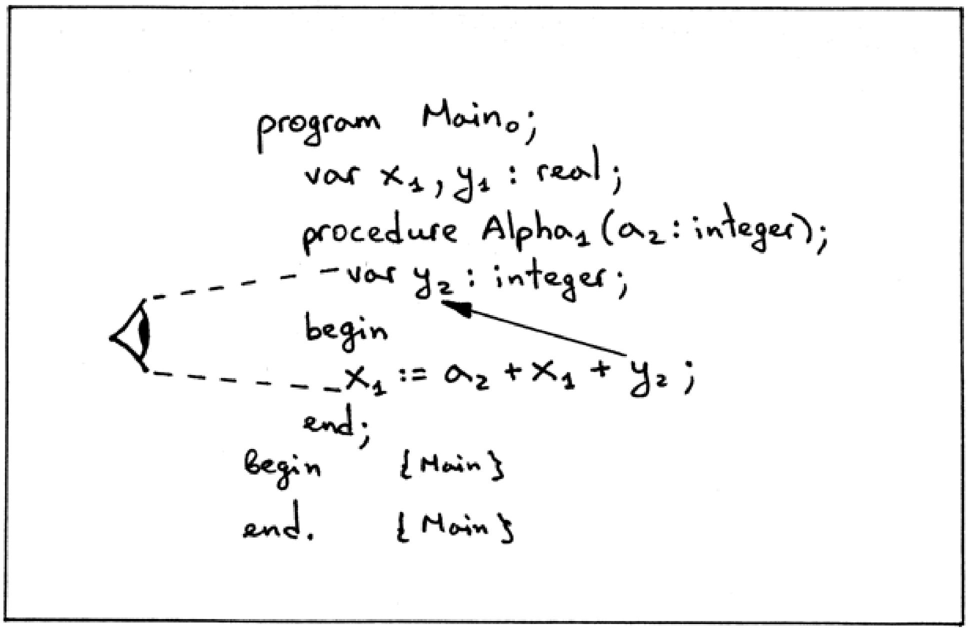
```
program Main0;
  var x1, y1 : real;
  procedure Alpha1(a2 : integer);
    var y2 : integer;
  begin
    x1 := a2 + x1 + y2;
  end;
begin {Main}
end. {Main}
```

2. 现在从算术表达式**a + x + y**到变量引用**x**。同样，首先我们在词法最接近的范围内搜索**x**的声明。词法上最接近的范围是级别 2的Alpha范围。该范围包含Alpha过程中的声明，包括过程的形式参数。我们没有在这个范围级别（在Alpha范围内）找到**x**，所以我们沿着链向上到全局范围并在那里继续我们的搜索。我们的搜索成功了，因为全局作用域中有一个名称为**x**的变量符号：

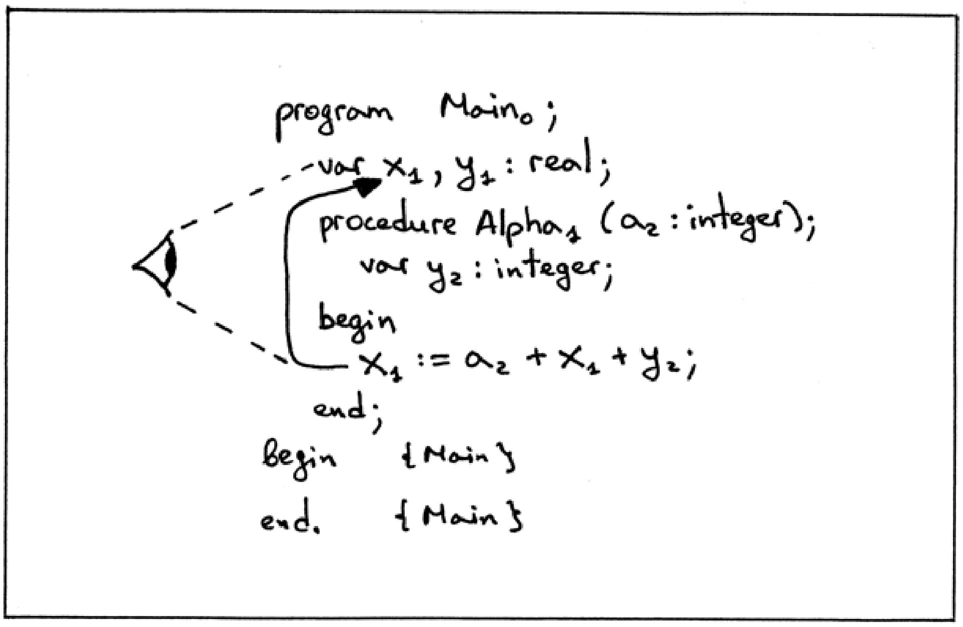


```
program Main0;
  var x1, y1 : real;
  procedure Alpha1(a2 : integer);
    var y2 : integer;
  begin
    x1 := a2 + x1 + y2;
  end;
begin {Main}
end. {Main}
```

3. 现在，让我们来看看变量引用**Ÿ**从算术表达式 **A + X + Y**。我们在词法上最接近的范围内找到它的声明，即Alpha范围。在Alpha作用域中，变量**y**具有**整数**类型（如果在Alpha作用域中没有**y**的声明，我们将扫描文本并在外部/全局作用域中找到**y**，在这种情况下它将具有**真实**类型）：



4. 最后，赋值语句左边的变量**x := a + x + y;** 它解析为与右侧算术表达式中的变量引用**x**相同的声明：



我们如何实现在当前作用域中查找，然后在封闭作用域中查找的行为，依此类推，直到我们找到我们正在寻找的符号，或者我们已经到达了作用域树的顶部并且没有还剩下更多的范围？我们只需要扩展ScopedSymbolTable类中的lookup方法以继续在作用域树中向上搜索：

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

    if symbol is not None:
        return symbol

    # recursively go up the chain and lookup the name
    if self.enclosing_scope is not None:
        return self.enclosing_scope.lookup(name)
```

The way the updated lookup method works:

1. Search for a symbol by name in the current scope. If the symbol is found, then return it.

2. If the symbol is not found, recursively traverse the tree and search for the symbol in the scopes up the chain. You don’t have to do the lookup recursively, you can rewrite it into an iterative form; the important part is to follow the link from a nested scope to its enclosing scope and search for the symbol there and up the tree until either the symbol is found or there are no more scopes left because you’ve reached the top of the scope tree.
3. The lookup method also prints the scope name, in parenthesis, where the lookup happens to make it clearer that lookup goes up the chain to search for a symbol, if it can’t find it in the current scope.

Let’s see what our semantic analyzer outputs for our sample program now that we’ve modified the way the lookup searches the scope tree for a symbol. Download scope04a.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope04a.py) and run it on the command line:

```
$ python scope04a.py
ENTER scope: global
Insert: INTEGER
Insert: REAL
Lookup: REAL. (Scope name: global)
Insert: x
Lookup: REAL. (Scope name: global)
Insert: y
Insert: Alpha
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: y
Lookup: a. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)
Lookup: y. (Scope name: Alpha)
Lookup: x. (Scope name: Alpha)
Lookup: x. (Scope name: global)


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : Alpha
Scope level    : 2
Enclosing scope: global
Scope (Scoped symbol table) contents
------------------------------------
      a: <VarSymbol(name='a', type='INTEGER')>
      y: <VarSymbol(name='y', type='INTEGER')>


LEAVE scope: Alpha


SCOPE (SCOPED SYMBOL TABLE)
===========================
Scope name     : global
Scope level    : 1
Enclosing scope: None
Scope (Scoped symbol table) contents
------------------------------------
INTEGER: <BuiltinTypeSymbol(name='INTEGER')>
   REAL: <BuiltinTypeSymbol(name='REAL')>
      x: <VarSymbol(name='x', type='REAL')>
      y: <VarSymbol(name='y', type='REAL')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>])>


LEAVE scope: global
```

Inspect the output above and pay attention to the ENTER and Lookup messages. A couple of things worth mentioning here:

1. Notice how the semantic analyzer looks up the INTEGER built-in type symbol before inserting the variable symbol **a**. It searches INTEGER first in the current scope, Alpha, doesn’ t find it, then goes up the tree all the way to the global scope, and finds the symbol there:

```
ENTER scope: Alpha
Lookup: INTEGER. (Scope name: Alpha)
Lookup: INTEGER. (Scope name: global)
Insert: a
```

2. Notice also how the analyzer resolves variable references from the assignment statement **x := a + x + y**:

   ```
   Lookup: a. (Scope name: Alpha)
   Lookup: x. (Scope name: Alpha)
   Lookup: x. (Scope name: global)
   Lookup: y. (Scope name: Alpha)
   Lookup: x. (Scope name: Alpha)
   Lookup: x. (Scope name: global)
   ```

   The analyzer starts its search in the current scope and then goes up the tree all the way to the global scope.

Let’s also see what happens when a Pascal program has a variable reference that doesn’t resolve to a variable declaration as in the sample program below:

```
程序 主程序;
   var  x , y : 实数;

   过程 阿尔法(a  :  整数);
      var  y  :  整数;
   开始
      x  :=  b  +  x  +  y ;  {这里有错误! }
   结束;

开始 {主要}

结束。  { 主要的 }
```

下载scope04b.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope04b.py)并在命令行上运行它:

```
$ python scope04b.py
输入范围:全局
插入:整数
插入:真实
查找:真实。(范围名称:全局)
插入:x
查找:真实。(范围名称:全局)
插入:y
插入:阿尔法
输入范围:Alpha
查找:整数。(范围名称:Alpha )
查找:INTEGER。(范围名称:全局)
插入:一个
查找:整数。(范围名称:Alpha )
查找:INTEGER。(范围名称:全局)
插入:y
查找: B. (范围名称:Alpha )
查找:b。(范围名称:全局)
错误:找不到符号(标识符)'b'
```

如您所见,分析器尝试解析变量引用**b**并首先在Alpha范围内搜索它,然后在全局范围内搜索它,但由于找不到名称为**b**的符号,它抛出了语义错误。

好的,现在我们知道如何编写语义分析器,当程序具有嵌套作用域时,它可以分析程序的语义错误。

源到源编译器

现在，进行完全不同的事情。让我们编写一个源代码到源代码的编译器！我们为什么要这样做？我们不是在谈论解释器和嵌套作用域吗？是的，我们是，但让我解释一下为什么我认为现在学习如何编写源到源编译器可能是一个好主意。

首先，让我们谈谈定义。什么是源到源编译器？出于本文的目的，让我们将**源到源编译器**定义为将某种源语言的程序翻译成相同（或几乎相同）源语言的程序的编译器。

因此，如果您编写一个将 Pascal 程序作为输入并输出可能已修改或增强的 Pascal 程序的翻译器，则这种情况下的翻译器称为源到源编译器。

供我们研究的源到源编译器的一个很好的例子是这样一种编译器，它将 Pascal 程序作为输入并输出一个类似 Pascal 的程序，其中每个名称都带有相应的范围级别的下标，并且，除了也就是说，每个变量引用也有一个类型指示符。因此，我们需要一个采用以下 Pascal 程序的源到源编译器：

```
程序 主程序;
   var  x ,  y : 实数;

   过程 阿尔法（a  :  整数）;
      var  y  :  整数;
   开始
      x  :=  a  +  x  +  y ;
   结束;

开始 {主要}

结束。  { 主要的 }
```

并将其转换为以下类似 Pascal 的程序：

```
程序 Main0 ;
   无功 x1  :  真实;
   var  y1  :  真实;
   程序 Alpha1 ( a2  :  INTEGER ) ;
      var  y2  :  整数;

   开始
      < X1 : REAL >  : =  < A2 : INTEGER >  +  < X1 : REAL >  +  < Y2 : INTEGER >;
   结束;  {阿尔法结束}

开始

结束。 {主要内容结束}
```

以下是我们的源到源编译器应对输入 Pascal 程序进行的修改列表：

1. 每个声明都应该打印在单独的行上，因此如果我们在输入 Pascal 程序中有多个声明，则编译后的输出应该在单独的行上显示每个声明。例如，我们可以在上面的文本中看到var x, y : real; 被转换成多行。
2. 每个名称都应该带有一个与相应声明的范围级别相对应的数字。
3. 每个变量引用，除了下标之外，还应该以下列形式打印：<var_name_with_subscript:type>
4. 编译器还应该在每个块的末尾以{ END OF ... }的形式添加注释，其中省略号将被替换为程序名称或过程名称。这将帮助我们更快地识别程序的文本边界。

从上面生成的输出中可以看出，这个源到源编译器可能是理解名称解析如何工作的有用工具，尤其是当程序具有嵌套作用域时，因为编译器生成的输出将允许我们快速查看某个变量引用解析到什么声明和在什么范围内。这在学习符号、嵌套作用域和名称解析时很有帮助。

我们如何实现这样的源到源编译器？我们实际上已经涵盖了所有必要的部分来做到这一点。我们现在需要做的就是稍微扩展我们的语义分析器以生成增强的输出。您可以在此处查看 (https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py)编译器的完整源代码。它基本上是一个关于药物的语义分析器，经过修改以生成和返回某些AST 节点的字符串。

下载src2srccompiler.py (https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py)，研究它，并通过将不同的 Pascal 程序作为输入传递给它来试验它。

对于以下程序，例如：

```
程序 主程序;
   var  x , y  : 实数;
   var  z  :  整数;

   过程 AlphaA ( a  : 整数) ;
      var  y  :  整数;
   开始 { AlphaA }
      x  :=  a  +  x  +  y ;
   结束;   {阿尔法A}

   过程 AlphaB ( a  :  integer ) ;
      var  b  :  整数;
   开始 { AlphaB}
   结束;   {字母B}

开始 {主要}
结束。  { 主要的 }
```

编译器生成以下输出：

```
$ python  src2src编译器。py 嵌套范围03 。PAS
程序 Main0 ;
   无功 x1  :  真实;
   var  y1  :  真实;
   var  z1  :  整数;
   程序 AlphaA1 ( a2  :  INTEGER ) ;
      var  y2  :  整数;

   开始
      < X1 : REAL >  : =  < A2 : INTEGER >  +  < X1 : REAL >  +  < Y2 : INTEGER >;
   结束; {END OF AlphaA}
   过程 AlphaB1 ( a2  :  INTEGER ) ;
      var  b2  :  整数;

   开始

   结束; {字母B结束}

开始

结束。 {主要内容结束}
```

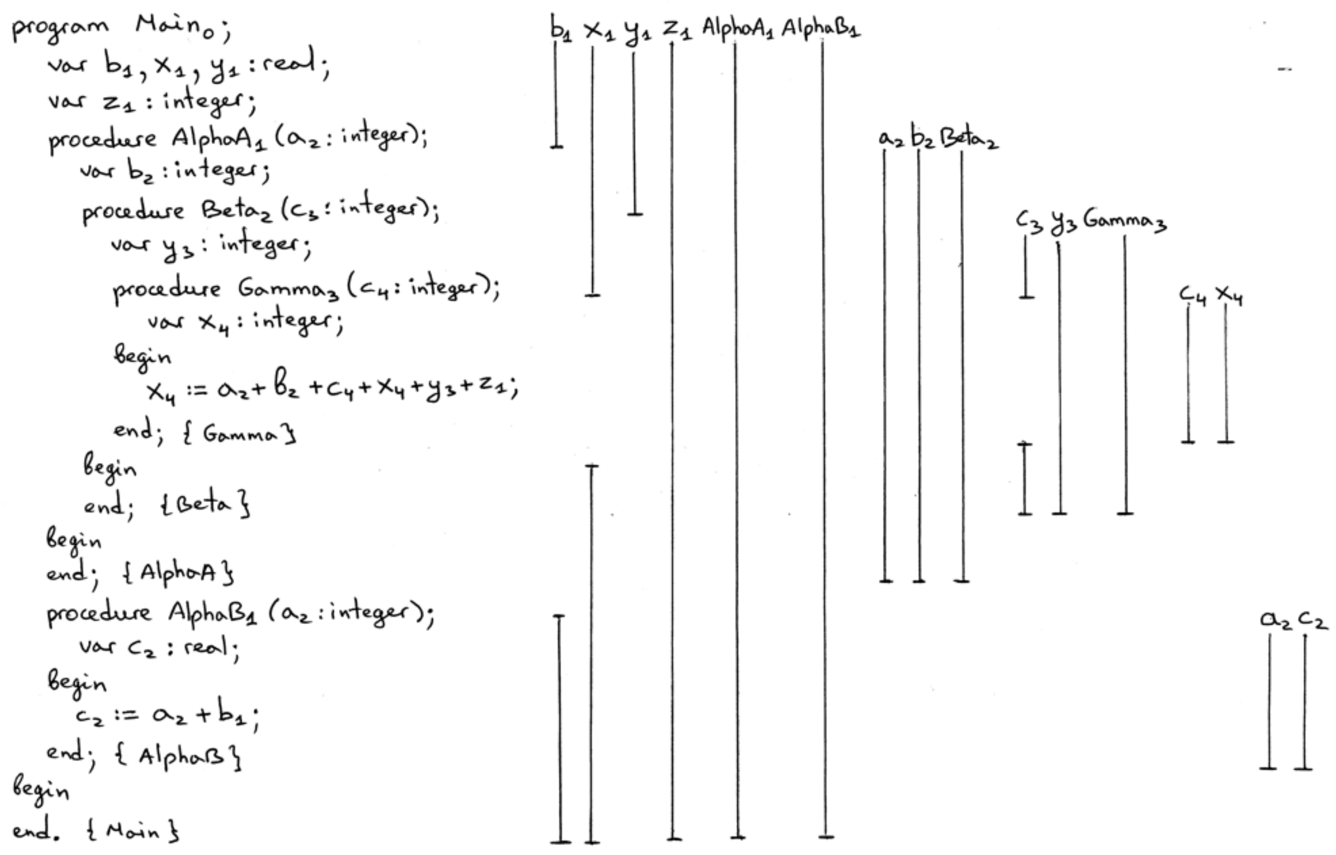
酷豆，恭喜你，现在你知道如何编写一个基本的源到源编译器了！

使用它可以进一步了解嵌套作用域、名称解析以及当您拥有AST和一些以符号表形式表示的程序的额外信息时可以做什么。

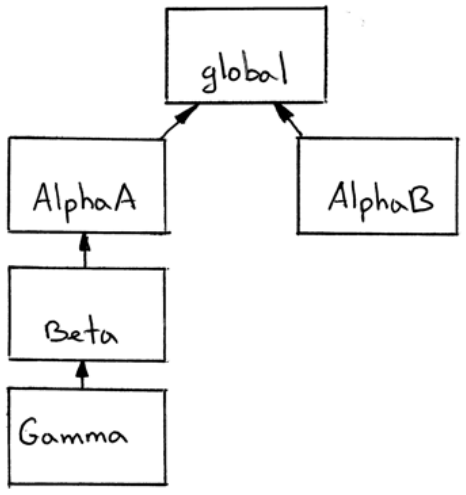
现在我们有了一个有用的工具来为我们的程序添加下标，让我们看一个更大的嵌套作用域示例，您可以在nestedscopes04.pas 中 (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas)找到它 (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas)：

```
程序 主程序;
   var  b ,  x ,  y  : 实数;
   var  z  :  整数;

   过程 AlphaA ( a  : 整数) ;
      var  b  :  整数;

      程序 Beta （c  :  整数）;
         var  y  :  整数;

         程序 伽玛（c  :  整数）;
            var  x  :  整数;
         begin  { Gamma }
            x  :=  a  +  b  +  c  +  x  +  y  +  z ;
         结束;   {伽玛}

      开始 {测试}

      结束;   { 测试版 }

   开始 { AlphaA }

   结束;   {阿尔法A}

   过程 AlphaB ( a  :  integer ) ;
      var  c  : 真实的;
   开始 { AlphaB }
      c  :=  a  +  b ;
   结束;   {字母B}

开始 {主要}
结束。  { 主要的 }
```

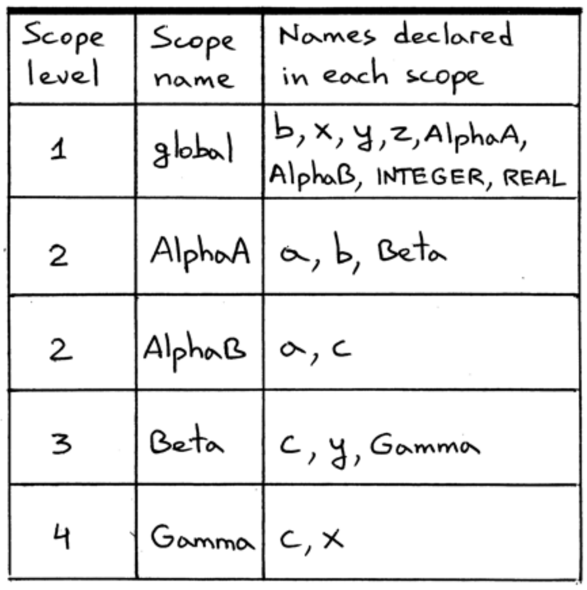
您可以在下面看到声明的范围、嵌套关系图和范围信息表:



```
program Main0;
   var b1, x1, y1 : real;
   var z1 : integer;
   procedure AlphaA1(a2 : integer);
      var b2 : integer;
      procedure Beta2(c3 : integer);
         var y3 : integer;
         procedure Gamma3(c4 : integer);
            var x4 : integer;
         begin
            x4 := a2 + b2 + c4 + x4 + y3 + z1;
         end; {Gamma}
      begin
      end; {Beta}
   begin
   end; {AlphaA}
   procedure AlphaB1(a2 : integer);
      var c2 : real;
   begin
      c2 := a2 + b1;
   end; {AlphaB}
begin
end. {Main}
```



NESTING RELATIONSHIPS



Scope level	Scope name	Names declared in each scope
1	global	b, x, y, z, AlphaA, AlphaB, INTEGER, REAL
2	AlphaA	a, b, Beta
2	AlphaB	a, c
3	Beta	c, y, Gamma
4	Gamma	c, x

SCOPE INFORMATION TABLE

让我们运行我们的源到源编译器并检查输出。下标应与上图中范围信息表中的下标匹配：

```
$ python  src2src编译器。py 嵌套范围04 。PAS
程序 Main0 ;
   var  b1  :  真实;
   无功 x1  :  真实;
   var  y1  :  真实;
   var  z1  :  整数;
   程序 AlphaA1 ( a2  :  INTEGER ) ;
      var  b2  :  整数;
      程序 Beta2 （c3  :  整数）；
         var  y3  :  整数;
         程序 Gamma3 ( c4  :  INTEGER ) ;
            var  x4  :  整数;

         begin
            < x4 : INTEGER >  :=  < a2 : INTEGER >  +  < b2 : INTEGER >  +  < c4 : INTEGER >  +  < x4 : INTEGE
         结束;  {伽马结束}

      开始

      结束;  {测试结束}

   开始

   结束;  {END OF AlphaA}
   过程 AlphaB1 ( a2  :  INTEGER ) ;
      var  c2  :  真实;

   begin
      < c2 : REAL >  :=  < a2 : INTEGER >  +  < b1 : REAL >;
   结束;  {字母B结束}

开始

结束。 {主要内容结束}
```

花一些时间研究源到源编译器的图片和输出。确保您了解以下要点：

- 绘制垂直线以显示声明范围的方式。
- 作用域中的空洞表明在嵌套作用域中重新声明了一个变量。
- 这AlphaA和AlphaB在全球范围内声明。
- 这AlphaA和AlphaB声明引入新的领域。
- 范围如何相互嵌套，以及它们的嵌套关系。
- 为什么不同的名称，包括赋值语句中的变量引用，以它们的方式下标。换句话说，名称解析，特别是链式作用域符号表的查找方法是如何工作的。

还运行以下程序 (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py)：

```
$ python scope05.py nestedscopes04.pas
```

并检查链接的作用域符号表的内容，并将其与您在上图中的作用域信息表中看到的内容进行比较。并且不要忘记genastdot.py (https://github.com/rspivak/lsbasi/blob/master/part14/genastdot.py)，您可以使用它来生成AST的可视化图表，以查看过程如何在树中相互嵌套。

在我们结束今天对嵌套作用域的讨论之前，回想一下之前我们删除了检查源程序是否有重复标识符的语义检查。让我们把它放回去。但是，为了在存在嵌套作用域和查找方法的新行为的情况下进行检查，我们需要进行一些更改。首先，我们需要更新查找方法并添加一个额外的参数，以允许我们将搜索限制在当前范围内：

```
def  lookup ( self ,  name ,  current_scope_only = False ):
    print ( 'Lookup: %s . (Scope name: %s )'  %  ( name ,  self . scope_name ))
    # 'symbol' 要么是 Symbol 类的实例，要么无
    符号 =  self 。_符号。得到（名字）

    如果 符号 是 不 无:
        回归 符号

    如果 current_scope_only ：
        返回 None

    # 递归地向上链并查找名称
    if  self 。enclosing_scope 是 不 无:
        回归 自我。封闭范围。查找（名称）
```

其次，我们需要修改visit_VarDecl方法并在lookup 方法中使用新的current_scope_only参数添加检查：

```
def  visit_VarDecl ( self ,  node ):
    type_name  =  node 。类型节点。值
    type_symbol  =  self 。current_scope 。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。current_scope 。查找（var_name ，  current_scope_only = True ）：
        引发 异常（
            “错误：发现重复标识符' %s '”  %  var_name
        ）

    自我。current_scope 。插入（var_symbol ）
```

如果我们不将搜索重复标识符限制在当前作用域内，查找可能会在外部作用域中找到同名的变量符号，结果会抛出错误，而实际上没有语义错误开始。

这是使用没有重复标识符错误的程序运行scope05.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py)的输出。您可以在下面注意到输出中有更多行，这是由于我们的重复标识符检查在插入新符号之前查找重复名称：

```
$ python scope05.py nestedscopes02.pas
输入范围：全局
插入：整数
插入：真实
查找：真实。（范围名称：全局）
查找：x。（范围名称：全局）
插入：x
查找：真实。（范围名称：全局）
查找：y。（范围名称：全局）
插入：y
插入：阿尔法
输入范围：Alpha
查找：整数。（范围名称：Alpha ）
查找：INTEGER。（范围名称：全局）
插入：一个
查找：整数。（范围名称：Alpha ）
查找：INTEGER。（范围名称：全局）
查找：y。（范围名称：阿尔法）
插入：y
查找：（范围名称：Alpha ）
查找：x。（范围名称：Alpha ）
查找：x。（范围名称：全局）
查找：y。（范围名称：Alpha ）
查找：x。（范围名称：Alpha ）
查找：x。（范围名称：全局）


范围（范围符号表）
============================
范围名称：阿尔法
范围级别：2
封闭范围：全球
范围（ Scoped Symbol table ）内容
---------------------
      a: <VarSymbol ( name = 'a' , type = 'INTEGER' ) >
      y: <VarSymbol ( name = 'y' , type = 'INTEGER' ) >


离开范围：阿尔法


范围（范围符号表）
============================
范围名称：全球
范围级别：1
封闭范围：无
范围（ Scoped Symbol table ）内容
---------------------
整数：<BuiltinTypeSymbol ( name = 'INTEGER' ) >
   REAL: <BuiltinTypeSymbol ( name = 'REAL' ) >
      x: <VarSymbol ( name = 'x' , type = 'REAL' ) >
      y: <VarSymbol ( name = 'y' , type = 'REAL' ) >
  Alpha: <ProcedureSymbol ( name = Alpha, parameters =[ <VarSymbol ( name = 'a' , type = 'INTEGER' ) > ]) >


LEAVE 范围：全球
```

现在，让我们将scope05.py (https://github.com/rspivak/lsbasi/blob/master/part14/scope05.py)用于另一个测试驱动器，看看它如何捕获重复标识符语义错误。

例如，对于以下 (https://github.com/rspivak/lsbasi/blob/master/part14/dupiderror.pas)在Alpha 范围内重复声明**a 的**错误程序 (https://github.com/rspivak/lsbasi/blob/master/part14/dupiderror.pas)：

```
程序 主程序;
   var  x ,  y : 实数;

   过程 阿尔法（a  ：  整数）;
      var  y  ：  整数;
      var  a  : 真实的;   {这里有错误! }
   开始
      x  :=  a  +  x  +  y ;
   结束;

开始 {主要}

结束。  { 主要的 }
```

该程序生成以下输出：

```
$ python scope05.py dupiderror.pas
输入范围：全局
插入：整数
插入：真实
查找：真实。（范围名称：全局）
查找：x。（范围名称：全局）
插入：x
查找：真实。（范围名称：全局）
查找：y。（范围名称：全局）
插入：y
插入：阿尔法
输入范围：Alpha
查找：整数。（范围名称：Alpha ）
查找：INTEGER。（范围名称：全局）
插入：一个
查找：整数。（范围名称：Alpha ）
查找：INTEGER。（范围名称：全局）
查找：y。（范围名称：阿尔法）
插入：y
查找：真实。（范围名称：Alpha ）
查找：REAL。（范围名称：全局）
查找：a。（范围名称：Alpha ）
错误：发现
重复的标识符“a”
```

它按预期捕获了错误。

在这个积极的方面，让我们结束今天对作用域、作用域符号表和嵌套作用域的讨论。

概括

我们已经涵盖了很多领域。让我们快速回顾一下我们在本文中学到的东西：

- 我们了解了scopes，它们为什么有用，以及如何在代码中实现它们。
- 我们了解了嵌套作用域以及如何使用链式作用域符号表来实现嵌套作用域。
- 我们学习了如何编写一个语义分析器来遍历AST，构建作用域符号表，将它们链接在一起，并进行各种语义检查。
- 我们了解了名称解析以及语义分析器如何使用链式作用域符号表（范围）将名称解析为它们的声明，以及查找方法如何在作用域树中递归地向上查找与某个名称对应的声明。
- 我们了解到，在语义分析器中构建作用域树涉及遍历AST，在进入某个AST节点时将新作用域“推”到作用域符号表堆栈的顶部，并在离开节点时将作用域从堆栈中“弹出”，使作用域树看起来像作用域符号表堆栈的集合。
- 我们学习了如何编写源到源编译器，这在学习嵌套作用域、作用域级别和名称解析时可能是一个有用的工具。

练习

是时候练习了，哦耶！

1. 您已经在整篇文章的图片中看到，程序语句中的Main名称的下标为零。我还提到该程序的名称不在全局范围内，而是在其他级别为零的外部范围内。扩展spi.py (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py)并创建一个内置作用域，一个级别为 0 的新作用域，并将内置类型INTEGER和REAL移到该作用域中。为了好玩和练习，您还可以更新代码以将程序名称也放入该范围内。
2. 对于nestedscopes04.pas 中 (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas)的源程序，请 (https://github.com/rspivak/lsbasi/blob/master/part14/nestedscopes04.pas)执行以下操作：
    1. 在一张纸上写下 Pascal 源程序
    2. 为程序中的每个名称添加下标，指示名称解析到的声明的范围级别。
    3. 为每个名称声明（变量和过程）绘制垂直线以直观地显示其范围。绘制时不要忘记范围孔及其含义。
    4. 为程序编写一个源到源编译器，无需查看本文中的示例源到源编译器。
    5. 使用原始的src2srccompiler.py (https://github.com/rspivak/lsbasi/blob/master/part14/src2srccompiler.py)程序来验证编译器的输出以及您是否在练习 (2.2) 中正确下标了名称。
3. 修改源码编译器为内置类型INTEGER和REAL添加下标
4. 取消注释spi.py 中 (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py)的以下块 (https://github.com/rspivak/lsbasi/blob/master/part14/spi.py)

    ```
    # interpreter = Interpreter(tree)
    # result = interpreter.interpret()
    # print('')
    # print('Run-time GLOBAL_MEMORY contents:')
    # for k, v in sorted(interpreter.GLOBAL_MEMORY.items()):
    # 打印('%s = %s' % (k, v))
    ```

    使用part10.pas (https://github.com/rspivak/lsbasi/blob/master/part10/python/part10.pas)文件作为输入运行解释器：

    ```
    $ python spi.py part10.pas
    ```

    发现问题并将缺失的方法添加到语义分析器中。

这就是今天的内容。在下一篇文章中，我们将了解运行时、调用堆栈、实现过程调用，并编写我们的递归阶乘函数的第一个版本。请继续关注，我们很快就会见到你！

如果您有兴趣，这里是我在准备文章时最常参考的书籍列表（附属链接）：

1. 语言实现模式：创建您自己的特定领域和通用编程语言（实用程序员）(https://www.amazon.com/gp/product/193435645X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=193435645X&linkCode=as2&tag=russblo0b-20&linkId=5d5ca8c07bff5452ea443d8319e7703d)
2. 编译器工程，第二版 (https://www.amazon.com/gp/product/012088478X/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=012088478X&linkCode=as2&tag=russblo0b-20&linkId=74578959d7d04bee4050c7bff1b7d02e)
3. 编程语言语用学，第四版 (https://www.amazon.com/gp/product/0124104096/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0124104096&linkCode=as2&tag=russblo0b-20&linkId=8db1da254b12fe6da1379957dda717fc)
4. 编译器：原理、技术和工具（第 2 版）(https://www.amazon.com/gp/product/0321486811/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0321486811&linkCode=as2&tag=russblo0b-20&linkId=31743d76157ef1377153dba78c54e177)
5. 编写编译器和解释器：一种软件工程方法 (https://www.amazon.com/gp/product/0470177071/ref=as_li_tl?ie=UTF8&camp=1789&creative=9325&creativeASIN=0470177071&linkCode=as2&tag=russblo0b-20&linkId=542d1267e34a529e0f69027af20e27f3)

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 Web Server. Part 3.
6 years ago • 45 comments
“We learn most when we have to invent” —Piaget In Part 2 you created a …

Let’s Build A Simple Interpreter. Part 5.
6 years ago • 19 comments
How do you tackle something as complex as understanding how to …

Let’s Build A Simple Interpreter. Part 17: …
2 years ago • 8 comments
You may have to fight a battle more than once to win it. - Margaret Thatcher

Let’s Interp
5 years
“Be no slowly standi

52 Comments **Ruslan's Blog** **Disqus' Privacy Policy** 1 **Login**

Recommend 5 Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

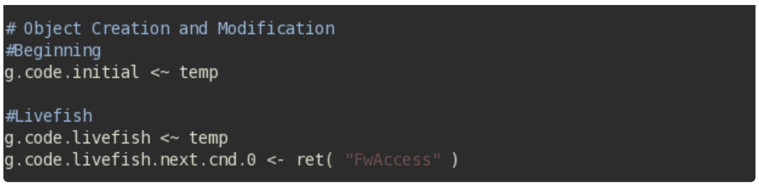
Name

pmst • 4 years ago
Thank you so much for this awesome series of tutorials. By the way, the series ends here? I notice you haven't been updated for eight months
5 | 1 • Reply • Share ›

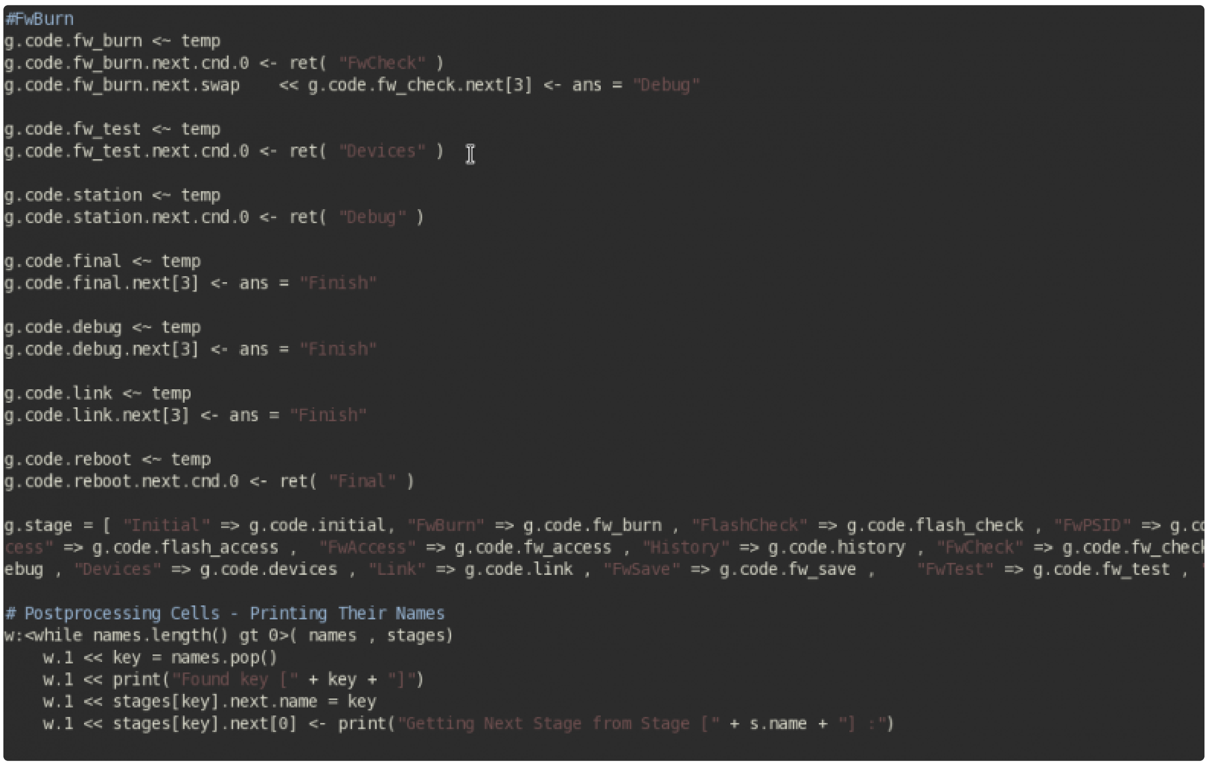
Ilya Nonename • 4 years ago • edited
Thank you, Made a new language with code that allows a code post modification. Actually all token saved as tokens and i added command to modify them from code. Gives me a lot of flexibility Added this to my project :)

Some code Example. It actually runs as any other script from command line :)
I feel like i made a real beast))))

```
print("Running Cell Script\n")

g.code <~ cell
temp <~ cell
temp.next = t:(data)
    t.show_data = j:<while data.length() gt 0>(data)
        j.1 << keys = data.keys()
        j.1 << key  = keys[0]
        j.1 << res  = data.delete(key)
        j.1 << print("Key [" + key + "]            \tRes is [" + res + "]")

    t.swap = k:()
        k << print( "Making Swap\n" )

    t.cnd = i:<if var gt 0 >(var)
        i.1 <- ret( "Debug" )
        i.0 <- ret( "Livefish" )

    # Next Function Content
    t << print("Running Next Stage:")
    t << copy <~ data
    t << s.show_data( copy )
    t << ans = s.cnd( data["NEXT_DEBUG"] )
    t << s.swap()
    t << print( "Next Stage Is [" +  ans + "]\n\n" )
    t << ret( ans )
```



```
# Object Creation and Modification
#Beginning
g.code.initial <~ temp

#Livefish
g.code.livefish <~ temp
g.code.livefish.next.cnd.0 <- ret( "FwAccess" )
```



```
#FwBurn
g.code.fw_burn <~ temp
g.code.fw_burn.next.cnd.0 <- ret( "FwCheck" )
g.code.fw_burn.next.swap    << g.code.fw_check.next[3] <- ans = "Debug"

g.code.fw_test <~ temp
g.code.fw_test.next.cnd.0 <- ret( "Devices" )

g.code.station <~ temp
g.code.station.next.cnd.0 <- ret( "Debug" )

g.code.final <~ temp
g.code.final.next[3] <- ans = "Finish"

g.code.debug <~ temp
g.code.debug.next[3] <- ans = "Finish"

g.code.link <~ temp
g.code.link.next[3] <- ans = "Finish"

g.code.reboot <~ temp
g.code.reboot.next.cnd.0 <- ret( "Final" )

g.stage = [ "Initial" => g.code.initial, "FwBurn" => g.code.fw_burn , "FlashCheck" => g.code.flash_check , "FwPSID" => g.co
cess" => g.code.flash_access ,  "FwAccess" => g.code.fw_access , "History" => g.code.history , "FwCheck" => g.code.fw_check
ebug , "Devices" => g.code.devices , "Link" => g.code.link , "FwSave" => g.code.fw_save ,    "FwTest" => g.code.fw_test , "

# Postprocessing Cells - Printing Their Names
w:<while names.length() gt 0>( names , stages)
    w.1 << key = names.pop()
    w.1 << print("Found key [" + key + "]")
    w.1 << stages[key].next.name = key
    w.1 << stages[key].next[0] <- print("Getting Next Stage from Stage [" + s.name + "] :")
```

2 ^ | ∨ • Reply • Share ›

Irwin Rodriguez • 8 months ago

Holy mother of god! I finished this chapter already! it took me the whole week but I did it! I think this is the longest chapter of the series. Thanks for sharing this tremendous content.

1 ^ | ∨ • Reply • Share ›

yaanhyy • a year ago

Thanks a lot for your work.

1 ^ | ∨ • Reply • Share ›

⚜Philippe†Aucaz¿uЯ • 4 years ago

As many others before me, I landed here looking for a tutorial about parsing. I've found a goldmine here : your articles are really awesome, since they are well constructed, well explained, and with a slight touch of humour too. I am a complete beginner in this field and I can now build something acceptable. So, thank you very much, and I hope next part will come soon (maybe you're a little bit too busy with your new job ?)
Thanks again.

P.S. Please excuse my mistakes in english: Frenchmen are not very good at other languages... ;-)

1 ^ | ∨ • Reply • Share ›

JCFF • 4 years ago

I’m developer since many years. Nevertheless, I’ve never had to create an interpreter, but 4 weeks ago this changed: I have to create a little language with some Excel functions like IF, ROUND, and so on (besides adding, subtracting, multiplying, …).
I found a book on this subject ("Parsing Techniques: A Practical Guide", by David Gries), but it was very hard to read (theoretical, especially).
With time I had it was impossible for me (someone with no experience in compilers or interpreters), reading it, understand it, create the language, test it and integrate it into my application.
So I kept looking for one less hard. Then I found "Language Implementation Patterns", by Terence Parr. The latter was a more affordable book but missing examples and my problem - the eternal problem - was the time. I needed to read less and to advance (program) more.
Then God Himself came down, he embraced me and whispered in my ear: “Don’t be silly. Search Ruslan Spivak’s blog and don’t waste any more time”... x-DDDDD

Although it is a joke, discovery of your blog it has been something like that...
With first book would be by chapter 10, and instead I have already in VB.NET an interpreter that adds, subtracts, multiply and divide (with priority rules), supports module and exponentiation operators, parenthesis and variables.
I still have work to do (some Excel functions like IF, ROUND, ...), but I’m in the right direction and with an acceptable pace.
And that thanks to your blog!
You're one of those people who have a special ability to explain. Each part is easy to understand, with examples...
For me and anyone who needs to do something similar, it's a treasure. Priceless. Really.
Thanks very much.
My little language will take your name! x-DDDD

PS.
I highly recommend using TDD to program the interpreter/compiler.
I have been continually adding operators, changing their priority, refactoring, ... and always quiet thanks to my safety net: my tests.

1 ^ | ˅ • Reply • Share ›

rspivak Mod ➦ JCFF • 4 years ago

Thanks a lot for sharing your experience and good luck with your project!

1 ^ | ˅ • Reply • Share ›

JCFF ➦ rspivak • 4 years ago

I hope so, thanks!
With only the first 9 parts, today I finished an interpreter that supports the grammar shown in the picture. It's amazing!
And Ruslan language it's very powerful and fun ;D

^ | ˅ 1 • Reply • Share ›

I_love_han_hye_jin • a year ago

Thank you for this awsome series!

^ | ˅ • Reply • Share ›

Igor Pissolati • 2 years ago

Thanks a lot for your amazing tutorials! it's a shame you did not continue it... But, anyway, I continued by myself with the C# language, looking for a point where I could create an interpreter in the interpreter =)
After some time planning how to do callstacks and classes I got there and now my interpreter is able to interpret the calculator from tutorial 6 (that I rewrote in pascal)! =D

Now I'm planning to do a refactoring on the whole project to then make a repository in github, and then implement the things I think matter and I have not implemented yet and maybe even turn the project into an (very simple) embedded language.

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Igor Pissolati • 2 years ago

That looks great. Good luck with your project!

1 ^ | ˅ • Reply • Share ›

Russell Coleman • 3 years ago

One of the best tutorials I've ever read. Lots of fun and learned a lot. This has helped me so much, even in mundane tasks like parsing user input. I always build a grammar and/or an AST now because it just works so smooth. Doing excellent work Ruslan!

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Russell Coleman • 3 years ago

Thank you

^ | ˅ • Reply • Share ›

Hermes Passer • 3 years ago

So, it's dead?

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Hermes Passer • 3 years ago

More like on extended sabbatical. :)

1 ^ | ˅ • Reply • Share ›

Dan Boxall ➦ rspivak • 3 years ago

Oh good, so you are going to continue? This is brilliant by the way, thanks! I'm implementing your python code in java, great fun

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Dan Boxall • 3 years ago

Yes, it's in the works. You're welcome and good luck with your interpreter!

^ | ˅ • Reply • Share ›

Yuipas Unknown ➦ rspivak • 3 years ago

Wow, It's great to see you back here! Thanks for all you've done for us.

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Yuipas Unknown • 3 years ago

You're welcome :)

^ | ˅ • Reply • Share ›

roachsinai ➦ rspivak • 2 years ago

Thank you so much, can't wait for the next article!

^ | ˅ • Reply • Share ›

rspivak Mod ➦ roachsinai • 2 years ago

You're welcome!

^ | ˅ • Reply • Share ›

N. Chauhan • 3 years ago

N. Chauhan • 3 years ago

Thanks so much for these tutorials! I think after reading up to around part 13 I decided I would actually start writing an interpreter. What I've come out with is a half-decent programming language with custom syntax and now its 2nd release. See my work here: https://github.com/nchauhan... (There's some demos too!)

^ | ˅ • Reply • Share ›

rspivak Mod ➦ N. Chauhan • 3 years ago

You're welcome. That's really great you started working on your own interpreter, way to go!

^ | ˅ • Reply • Share ›

N. Chauhan ➦ rspivak • 3 years ago

Thanks. It was great fun (and a challenge!) to implement things like user-defined functions, but I've managed to get it running smoothly and I'm proud of myself :D

^ | ˅ • Reply • Share ›

Grayson Dubois • 3 years ago • edited

Excellent tutorials, Ruslan! I have to say, I've always loved the idea of designing my own programming language, but after my Compiler Design course in college, I had about decided that programming language implementation was far more complex than anything I wanted to deal with. Your tutorials have proven that does not have to be the case. What you've managed to do better than anyone else I've seen so far is explain the concepts and implementations in a down-to-earth, easily understood manner that makes the prospect of building an interpreter far less intimidating and even - dare I say it? - fun!

Following your tutorials, I have implemented a Pascal (subset) interpreter in JavaScript https://github.com/MellowCo..., which has proven to be an extremely rewarding experience.
I look forward to more tutorials as you post them, and plan on one day using what I've learned from you to design and implement my own interpreter in Rust.

Keep up the great work, and thank you so much for the time you've put into these articles!

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Grayson Dubois • 3 years ago

You're welcome!

^ | ˅ • Reply • Share ›

Generic Commenter • 4 years ago

You're a legend, dude!

^ | ˅ • Reply • Share ›

Moon • 4 years ago

Thank you so much for this awesome series of tutorials.

Before I landed on your website, I tried working alone with the Dragon Book, but I wasn't able to write the grammar rules' method on my own, they were messy and full of useless tests. I think I can use the book on my own now, thanks to you.

Good luck writing the next parts, I can wait to study them!

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Moon • 4 years ago

Thanks!

^ | ˅ • Reply • Share ›

Igor Kulman • 4 years ago • edited

Thanks for the series, I just finished it, writing a Pascal compiler in Swift as I go https://github.com/igorkulm.... Cannot wait for the next part. Which of the 4 books would you recommend if I want to implement procedure calls while I wait for the part 15?

^ | ˅ • Reply • Share ›

rspivak Mod ➦ Igor Kulman • 4 years ago • edited

You're welcome. Take a look at "Writing Compilers and Interpreters: A Software Engineering Approach" and "Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages." Hope that helps. And thanks a lot for showing your project!

^ | ˅ • Reply • Share ›

Igor Kulman ➦ rspivak • 4 years ago
Thanks, got to a point where I can call procedures and evaluate conditions so I can execute a basic factorial procedure. I am really curious to see how your implementation of call stack will look like, if it will be completely different from mine (other than python vs swift).

1 ^ | ˅ • Reply • Share ›

alexsander • 4 years ago
The series ends here? thank's for make this available.
^ | ˅ • Reply • Share ›

Luca Cassarino • 4 years ago
Where is part 15 ?
^ | ˅ • Reply • Share ›

LucasAraujo • 4 years ago
Thank you so much for this tutorial series. I made a little spreadsheet language interpreter based on your posts: https://github.com/LucasNev...
^ | ˅ • Reply • Share ›

rspivak Mod ➦ LucasAraujo • 4 years ago
You're very welcome. Thanks for showing your project!
^ | ˅ • Reply • Share ›

Gilles Page • 4 years ago
Are you going to add loops and conditionals and procedure calling, I just skimmed the last 5 blogs, sorry if it's in there, I did not notice
^ | ˅ • Reply • Share ›

DescriptiveName • 4 years ago
I wish there was more to this tutorial series! Thanks for what you managed to get out, it's been very interesting and fun writing little languages.
^ | ˅ • Reply • Share ›

CSY • 4 years ago
I really like your work! I recently want to build an interpreter, but I can't find such great articles like yours, they helped me a lot to face so many concepts. You use simple but complete way to explain them, I like that style! Hope the next article will come soon!
^ | ˅ • Reply • Share ›

evanxg852000 • 4 years ago • edited
As a system internal lover, I really enjoyed this series. I have implemented debuggers in a CHIP8 emulator. but I am eager to see your technique of implementing a debugger for this interpreter.
I also wrote an ultra tiny compiler for simpler approach https://gist.github.com/eva...

Thank you for this awesome work

Reply • Share ›

cpsthrume • 4 years ago
Hi, Ruslan, thanks a lot for your effort to put this guide online! It's one of the most interesting tutorials I've come across, so I hope you are still planning to release to last parts, I am super eager to dive into them :)

Reply • Share ›

rspivak Mod → cpsthrume • 4 years ago
You're welcome! Don't worry, I will finish the series. It just takes a bit longer than I thought it would. :)

1 • Reply • Share ›

cpsthrume → rspivak • 4 years ago
Great, glad to know you're fine and still working on this course. Good luck!

Reply • Share ›

rspivak Mod → cpsthrume • 4 years ago
Thanks!

Reply • Share ›

Ilya Nonename → rspivak • 4 years ago • edited
Actually i made already in my interpreter support for Objects,Lists and Function calls. I am making it without AST. And i had to rewrite half of the code (I even had to rewrite if conditions and loops) to make proper object support with function calls and list indexing [That was a mess] . But i think i did it the best way i could and it's finally done.

Reply • Share ›

Peter Mahler • 4 years ago • edited
Hi Ruslan,
thx a lot for this great job.

I had to develop a parser for a properity markup language and I failed with lex/yacc, the syntax was somehow strange and I had to switch the syntax inbetween the lexer :-(. With Lex/yacc i got horrable regexp or i had to do a multi-pass parsing. Your lexer did the job!

Independet from the syntax I had some problems (more python related, i usually develop in C/C++). Maybe someone else find this information helpful:
- I have to deal with qoted strings in the style of "blabla....", in the first step it was no problem to add a _quoted method But when i figured out that a string can contain escape sequences ("blabla\r\nblabla\"QuotedBlaBla\"TabulatorBlaBla\t") python faild - I found no way to evaluate these sequences text mode. Switching to binary mode in file open did also this job - but the predefined text block was not usable any more
- I have also to deal with files that are up to 200MB. Reading the files at once into a global variable is not a good idea. I modified the lexer.advance() and the lexer.peek() method to operate with a BufferedReader object, a five minute job. Together with the binary mode now it works perfect, It maybe also a solution when you want to deal with network streams.

I usally do prototype development in python and switch later on to C/C++ to increase the performance so the next step will be a swtich from python to C/C++

BR, Peter

Reply • Share ›

pejman hkh • 4 years ago
My simple php interpreter :
https://github.com/pejman-h...

Reply • Share ›

Ilya Nonename • 4 years ago • edited
Thanks man. I am working on Interpreter for new language that will have better capabilities than any OO. I am trying to represent a code as a data, so it can be modifyed during program execution

represent a code as a data, so it can be modifyed during program execution.
Thanks for tutorial. It is Very helpful.

This is a first program written in this language:

see more

^ | v • Reply • Share ›

Dhaval Shownkani • 4 years ago
cannot wait for the next part !!!! When will it be out ?
Thanks :D
^ | v • Reply • Share ›

António Poças • 4 years ago
Great tutorials, thanks a lot! I've wanted to learn more about interpreters and compilers for a long time and I think your approach of starting with just a simple calculator made things a lot easier to understand. I've implemented the whole thing in C#. It was too interesting to stop here though so I've also added types, functions, function and procedure calls, a call stack and if branches, although I have no idea if I did any of these things right. (I haven't tested recursion yet, but I'm fairly certain it will break something)

The repo is available here if anybody wants to take a look https://github.com/antonio-... it has no unit tests :(
^ | v • Reply • Share ›

Load more comments

社会的

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

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

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

Popular posts

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

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

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

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

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

Disclaimer

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

Back to top