Let's Build A Simple Interpreter. Part 19: Nested Procedure Calls (https://ruslanspivak.com/lsbasi-part19/)

Date Thu, March 19, 2020

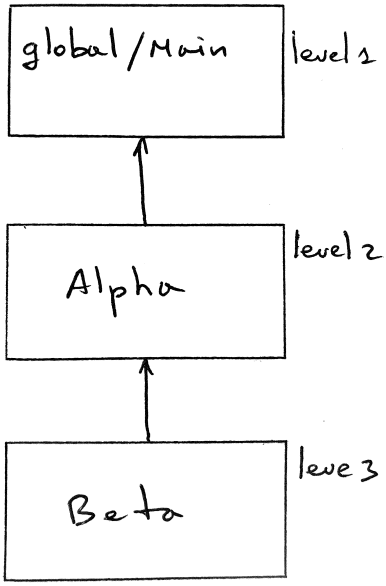
> *"What I cannot create, I do not understand." — Richard Feynman*

As I promised you last time, today we're going to expand on the material covered in the previous article and talk about executing nested procedure calls. Just like last time, we will limit our focus today to procedures that can access their parameters and local variables only. We will cover accessing non-local variables in the next article.

Here is the sample program for today:

```
program Main;

procedure Alpha(a : integer; b : integer);
var x : integer;

   procedure Beta(a : integer; b : integer);
   var x : integer;
   begin
      x := a * 10 + b * 2;
   end;

begin
   x := (a + b ) * 2;

   Beta(5, 10);      { procedure call }
end;

begin { Main }

   Alpha(3 + 5, 7);  { procedure call }

end.  { Main }
```

The nesting relationships diagram for the program looks like this:



Some things to note about the above program:

- it has two procedure declarations, *Alpha* and *Beta*
- *Alpha* is declared inside the main program (the *global* scope)
- the *Beta* procedure is declared inside the *Alpha* procedure
- both *Alpha* and *Beta* have the same names for their formal parameters: integers *a* and *b*
- and both *Alpha* and *Beta* have the same local variable *x*
- the program has nested calls: the *Beta* procedure is called from the *Alpha* procedure, which, in turn, is called from the main program

Now, let's do an experiment. Download the part19.pas (https://github.com/rspivak/lsbasi/blob/master/part19/part19.pas) file from GitHub (https://github.com/rspivak/lsbasi/blob/master/part19) and run the interpreter from the previous article (https://github.com/rspivak/lsbasi/blob/master/part18/spi.py) with the part19.pas (https://github.com/rspivak/lsbasi/blob/master/part19/part19.pas) file as its input to see what happens when the interpreter executes nested procedure calls (the main program calling *Alpha* calling *Beta*):

```
$ python spi.py part19.pas --stack

ENTER: PROGRAM Main
CALL STACK
1: PROGRAM Main


...


ENTER: PROCEDURE Beta
CALL STACK
2: PROCEDURE Beta
   a                   : 5
   b                   : 10
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
   x                   : 30
1: PROGRAM Main


LEAVE: PROCEDURE Beta
CALL STACK
2: PROCEDURE Beta
   a                   : 5
   b                   : 10
   x                   : 70
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
   x                   : 30
1: PROGRAM Main


...


LEAVE: PROGRAM Main
CALL STACK
1: PROGRAM Main
```

It just works! There are no errors. And if you study the contents of the ARs(activation records), you can see that the values stored in the activation records for the *Alpha* and *Beta* procedure calls are correct. So, what's the catch then? There is one small issue. If you take a look at the output where it says 'ENTER: PROCEDURE Beta', you can see that the nesting level for the *Beta* and *Alpha* procedure call is the same, it's 2 (two). The nesting level for *Alpha* should be 2 and the nesting level for *Beta* should be 3. That's the issue that we need to fix. Right now the *nesting_level* value in the *visit_ProcedureCall* method is hardcoded to be 2 (two):

```
def visit_ProcedureCall(self, node):
    proc_name = node.proc_name

    ar = ActivationRecord(
        name=proc_name,
        type=ARType.PROCEDURE,
        nesting_level=2,
    )

    proc_symbol = node.proc_symbol
    ...
```

Let's get rid of the hardcoded value. How do we determine a nesting level for a procedure call? In the method above we have a procedure symbol and it is stored in a scoped symbol table that has the right scope level that we can use as the value of the *nesting_level* parameter (see Part 14 (https://ruslanspivak.com/lsbasi-part14/) for more details about scopes and scope levels).

How do we get to the scoped symbol table's scope level through the procedure symbol?

Let's look at the following parts of the *ScopedSymbolTable* class:

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

    def insert(self, symbol):
        self.log(f'Insert: {symbol.name}')
        self._symbols[symbol.name] = symbol
```

Looking at the code above, we can see that we could assign the scope level of a scoped symbol table to a symbol when we store the symbol in the scoped symbol table (scope) inside the *insert* method. This way we will have access to the procedure symbol's scope level in the *visit_Procedure* method during the interpretation phase. And that's exactly what we need.

Let's make the necessary changes:

- First, let's add a *scope_level* member to the *Symbol* class and give it a default value of zero:

  ```
  class Symbol:
      def __init__(self, name, type=None):
          ...
          self.scope_level = 0
  ```

- Next, let's assign the corresponding scope level to a symbol when storing the symbol in a scoped symbol table:

```
class ScopedSymbolTable:
    ...
    def insert(self, symbol):
        self.log(f'Insert: {symbol.name}')

        symbol.scope_level = self.scope_level

        self._symbols[symbol.name] = symbol
```

Now, when creating an AR for a procedure call in the *visit_ProcedureCall* method, we have access to the scope level of the procedure symbol. All that's left to do is use the scope level of the procedure symbol as the value of the *nesting_level* parameter:

```
class Interpreter(NodeVisitor):
    ...
    def visit_ProcedureCall(self, node):
        proc_name = node.proc_name
        proc_symbol = node.proc_symbol

        ar = ActivationRecord(
            name=proc_name,
            type=ARType.PROCEDURE,
            nesting_level=proc_symbol.scope_level + 1,
        )
```

That's great, no more hardcoded nesting levels. One thing worth mentioning is why we put *proc_symbol.scope_level + 1* as the value of the *nesting_level* parameter and not just *proc_symbol.scope_level.* In Part 17 (https://ruslanspivak.com/lsbasi-part17/), I mentioned that the nesting level of an AR corresponds to the scope level of the respective procedure or function declaration plus one. Let's see why.

In our sample program for today, the *Alpha* procedure symbol - the symbol that contains information about the *Alpha* procedure declaration - is stored in the *global* scope at level 1 (one). So 1 is the value of the *Alpha* procedure symbol's *scope_level*. But as we know from Part14 (https://ruslanspivak.com/lsbasi-part14/), the scope level of the procedure declaration *Alpha* is one less than the level of the variables declared inside the procedure *Alpha*. So, to get the scope level of the scope where the *Alpha* procedure's parameters and local variables are stored, we need to increment the procedure symbol's scope level by 1. That's the reason we use *proc_symbol.scope_level + 1* as the value of the *nesting_level* parameter when creating an AR for a procedure call and not simply *proc_symbol.scope_level*.

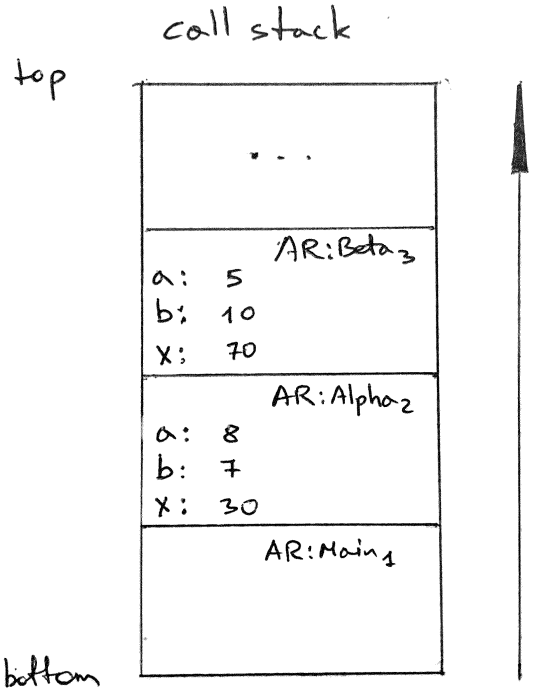
Let's see the changes we've made so far in action. Download the updated interpreter (https://github.com/rspivak/lsbasi/blob/master/part19) and test it again with the part19.pas (https://github.com/rspivak/lsbasi/blob/master/part19) file as its input:

```
$ python spi.py part19.pas --stack

ENTER: PROGRAM Main
CALL STACK
1: PROGRAM Main


ENTER: PROCEDURE Alpha
CALL STACK
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
1: PROGRAM Main


ENTER: PROCEDURE Beta
CALL STACK
3: PROCEDURE Beta
   a                   : 5
   b                   : 10
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
   x                   : 30
1: PROGRAM Main


LEAVE: PROCEDURE Beta
CALL STACK
3: PROCEDURE Beta
   a                   : 5
   b                   : 10
   x                   : 70
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
   x                   : 30
1: PROGRAM Main


LEAVE: PROCEDURE Alpha
CALL STACK
2: PROCEDURE Alpha
   a                   : 8
   b                   : 7
   x                   : 30
1: PROGRAM Main


LEAVE: PROGRAM Main
CALL STACK
1: PROGRAM Main
```

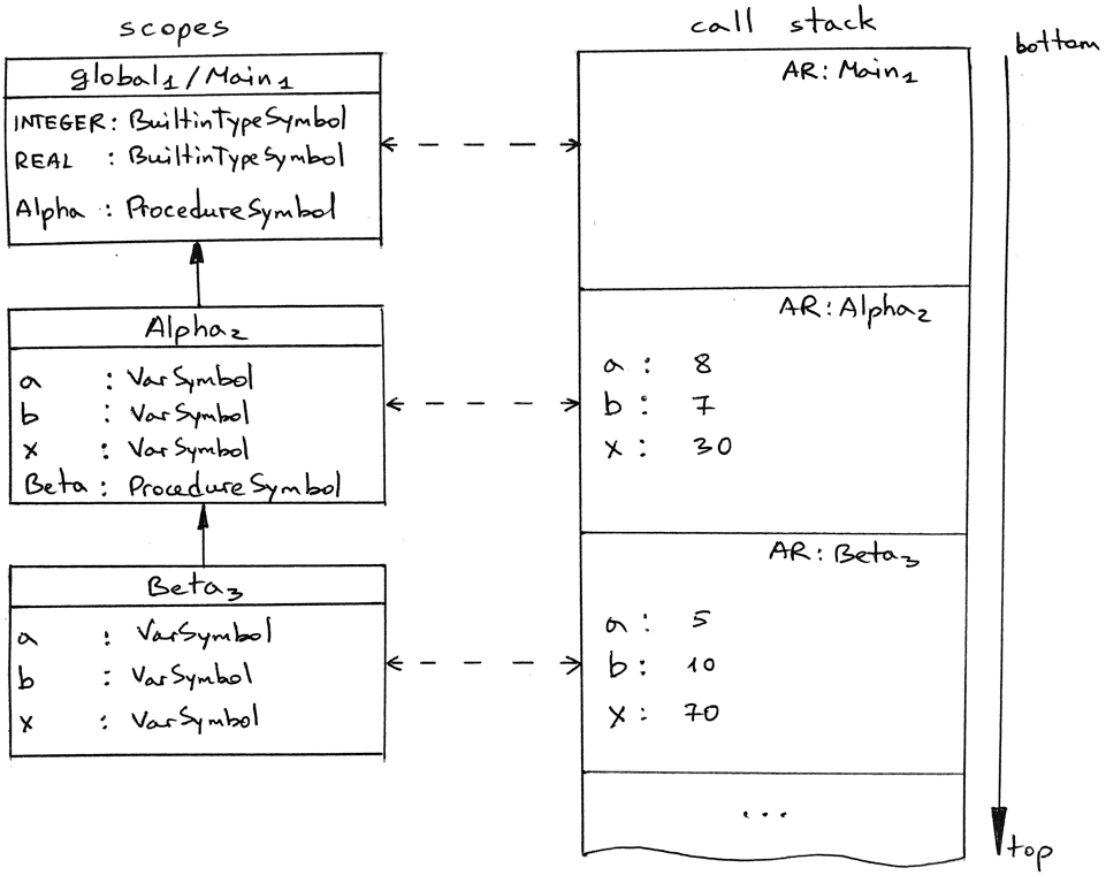
As you can see from the output above, the nesting levels of the activation records (AR) now have the correct values:

- The *Main* program AR: nesting level 1
- The *Alpha* procedure AR: nesting level 2
- The *Beta* procedure AR: nesting level 3

Let's take a look at how the scope tree (scoped symbol tables) and the call stack look visually during the execution of the program. Here is how the call stack looks right after the message "LEAVE: PROCEDURE Beta" and before the AR for the *Beta* procedure call is popped off the stack:



And if we flip the call stack (so that the top of the stack is at the "bottom"), you can see how the call stack with activation records relates to the scope tree with scopes (scoped symbol tables). In fact, we can say that activation records are run-time equivalents of scopes. Scopes are created during semantic analysis of a source program (the source program is read, parsed, and analyzed at this stage, but not executed) and the call stack with activation records is created at run-time when the interpreter executes the source program:



As you've seen in this article, we haven't made a lot of changes to support the execution of nested procedure calls. The only real change was to make sure the nesting level in ARs was correct. The rest of the codebase stayed the same. The main reason why our code continues to work pretty much unchanged with nested procedure calls is because the *Alpha* and *Beta* procedures in the sample program access the values of local variables only (including their own parameters). And because those values are stored in the AR at the top of the stack, this allows us to continue to use the methods *visit_Assignment* and *visit_Var* without any change, when executing the body of the procedures. Here is the source code of the methods again:

```
def visit_Assign(self, node):
    var_name = node.left.value
    var_value = self.visit(node.right)

    ar = self.call_stack.peek()
    ar[var_name] = var_value

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

    ar = self.call_stack.peek()
    var_value = ar.get(var_name)

    return var_value
```

Okay, today we've been able to successfully execute nested procedure calls with our interpreter with very few changes. And now we're one step closer to properly executing recursive procedure calls.

That's it for today. In the next article, we'll talk about how procedures can access non-local variables during run-time.

Stay safe, stay healthy, and take care of each other! See you next time.

Resources used in preparation for this article (links are affiliate links):

1. Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers) (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. Writing Compilers and Interpreters: A Software Engineering Approach (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)
3. Programming Language Pragmatics, Fourth Edition (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)

If you want to get my newest articles in your inbox, then enter your email address below and click "Get Updates!"

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series: