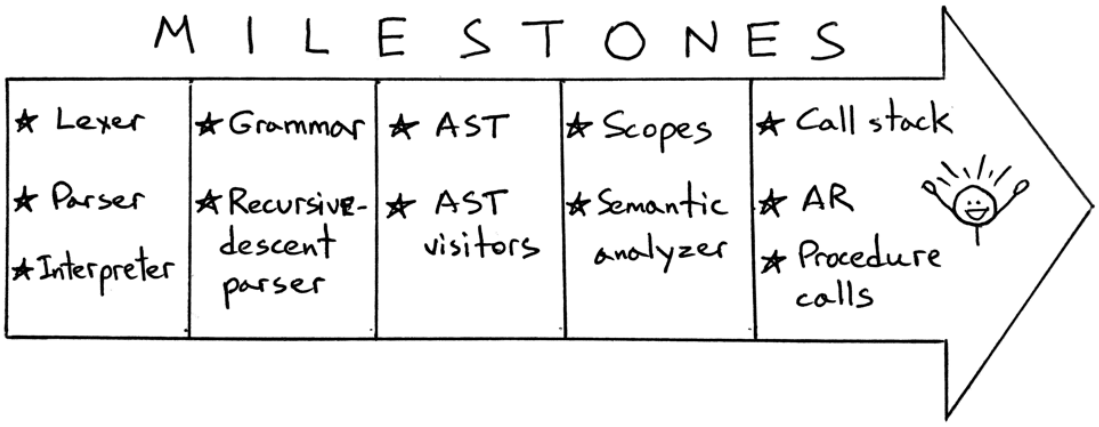
Let’s Build A Simple Interpreter. Part 18: Executing Procedure Calls (https://ruslanspivak.com/lsbasi-part18/)

Date Thu, February 20, 2020

> *“Do the best you can until you know better. Then when you know better, do better.” — Maya Angelou*

It’s a huge milestone for us today! Because today we will extend our interpreter to execute procedure calls. If that’s not exciting, I don’t know what is. :)



Are you ready? Let’s get to it!

Here is the sample program we’ll focus on in this article:

```
program Main;

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

begin { Main }

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

end.  { Main }
```

It has one procedure declaration and one procedure call. We will limit our focus today to procedures that can access their parameters and local variables only. We will cover nested procedure calls and accessing non-local variables in the next two articles.

Let's describe an algorithm that our interpreter needs to implement to be able to execute the *Alpha(3 + 5, 7)* procedure call in the program above.

Here is the algorithm for executing a procedure call, step by step:

1. Create an activation record
2. Save procedure arguments (actual parameters) in the activation record
3. Push the activation record onto the call stack
4. Execute the body of the procedure
5. Pop the activation record off the stack

Procedure calls in our interpreter are handled by the *visit_ProcedureCall* method. The method is currently empty:

```
class Interpreter(NodeVisitor):
    ...

    def visit_ProcedureCall(self, node):
        pass
```

Let's go over each step in the algorithm and write code for the *visit_ProcedureCall* method to execute procedure calls.

Let's get started!

Step 1. Create an activation record

If you remember from the previous article (https://ruslanspivak.com/lsbasi-part17), an *activation record (AR)* is a dictionary-like object for maintaining information about the currently executing invocation of a procedure or function, and also the program itself. The activation record for a procedure, for example, contains the current values of its formal parameters and the current values of its local variables. So, to store the procedure's arguments and local variables, we need to create an AR first. Recall that the *ActivationRecord* constructor takes 3 parameters: *name*, *type*, and *nesting_level*. And here's what we need to pass to the constructor when creating an AR for a procedure call:

- We need to pass the procedure's name as the *name* parameter to the constructor
- We also need to specify PROCEDURE as the *type* of the AR
- And we need to pass 2 as the *nesting_level* for the procedure call because the program's nesting level is set to 1 (You can see that in the *visit_Program* method of the interpreter)

Before we extend the *visit_ProcedureCall* method to create an activation record for a procedure call, we need to add the PROCEDURE type to the *ARType* enumeration. Let's do this first:

```
class ARType(Enum):
    PROGRAM   = 'PROGRAM'
    PROCEDURE = 'PROCEDURE'
```

Now, let's update the *visit_ProcedureCall* method to create an activation record with the appropriate arguments that we described earlier in the text:

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

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

Writing code to create an activation record was easy once we figured out what to pass to the *ActivationRecord* constructor as arguments.

Step 2. Save procedure arguments in the activation record

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

Here is a list of steps that describes the high-level actions the interpreter needs to take to save procedure arguments in the activation record:

a. Get a list of the procedure's formal parameters

b. Get a list of the procedure's actual parameters (arguments)
c. For each formal parameter, get the corresponding actual parameter and save the pair in the procedure's activation record by using the formal parameter's name as a key and the actual parameter (argument), after having evaluated it, as the value

If we have the following procedure declaration and procedure call:

```
procedure Alpha(a : integer; b : integer);

Alpha(3 + 5, 7);
```

Then after the above three steps have been executed, the procedure's AR contents should look like this:

```
2: PROCEDURE Alpha
   a                  : 8
   b                  : 7
```

Here is the code that implements the steps above:

```
proc_symbol = node.proc_symbol

formal_params = proc_symbol.formal_params
actual_params = node.actual_params

for param_symbol, argument_node in zip(formal_params, actual_params):
    ar[param_symbol.name] = self.visit(argument_node)
```

Let's take a closer look at the steps and the code.

a) First, we need to get a list of the procedure's formal parameters. Where can we get them from? They are available in the respective procedure symbol created during the semantic analysis phase. To jog your memory, here is the definition of the *ProcedureSymbol* class:

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


class ProcedureSymbol(Symbol):
    def __init__(self, name, formal_params=None):
        super().__init__(name)
        # a list of VarSymbol objects
        self.formal_params = [] if formal_params is None else formal_params
```

And here's the contents of the *global* scope (program level), which shows a string representation of the *Alpha* procedure symbol with its formal parameters:

```
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')>
  Alpha: <ProcedureSymbol(name=Alpha, parameters=[<VarSymbol(name='a', type='INTEGER')>, <VarSymbol(n
```

Okay, we now know where to get the formal parameters from. How do we get to the procedure symbol from the *ProcedureCall* AST *node* variable? Let's take a look at the *visit_ProcedureCall* method code that we've written so far:

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

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

We can get access to the procedure symbol by adding the following statement to the code above:

```
proc_symbol = node.proc_symbol
```

But if you look at the definition of the *ProcedureCall* class from the previous article (https://ruslanspivak.com/lsbasi-part17), you can see that the class doesn't have *proc_symbol* as a member:

```
class ProcedureCall(AST):
    def __init__(self, proc_name, actual_params, token):
        self.proc_name = proc_name
        self.actual_params = actual_params  # a list of AST nodes
        self.token = token
```

Let's fix that and extend the *ProcedureCall* class to have the *proc_symbol* field:

```
class ProcedureCall(AST):
    def __init__(self, proc_name, actual_params, token):
        self.proc_name = proc_name
        self.actual_params = actual_params  # a list of AST nodes
        self.token = token
        # a reference to procedure declaration symbol
        self.proc_symbol = None
```

That was easy. Now, where should we set the *proc_symbol* so that it has the right value (a reference to the respective procedure symbol) for the interpretation phase? As I've mentioned earlier, the procedure symbol gets created during the semantic analysis phase. We can store it in the *ProcedureCall* AST node during the node traversal done by the semantic analyzer's *visit_ProcedureCall* method.

Here is the original method:

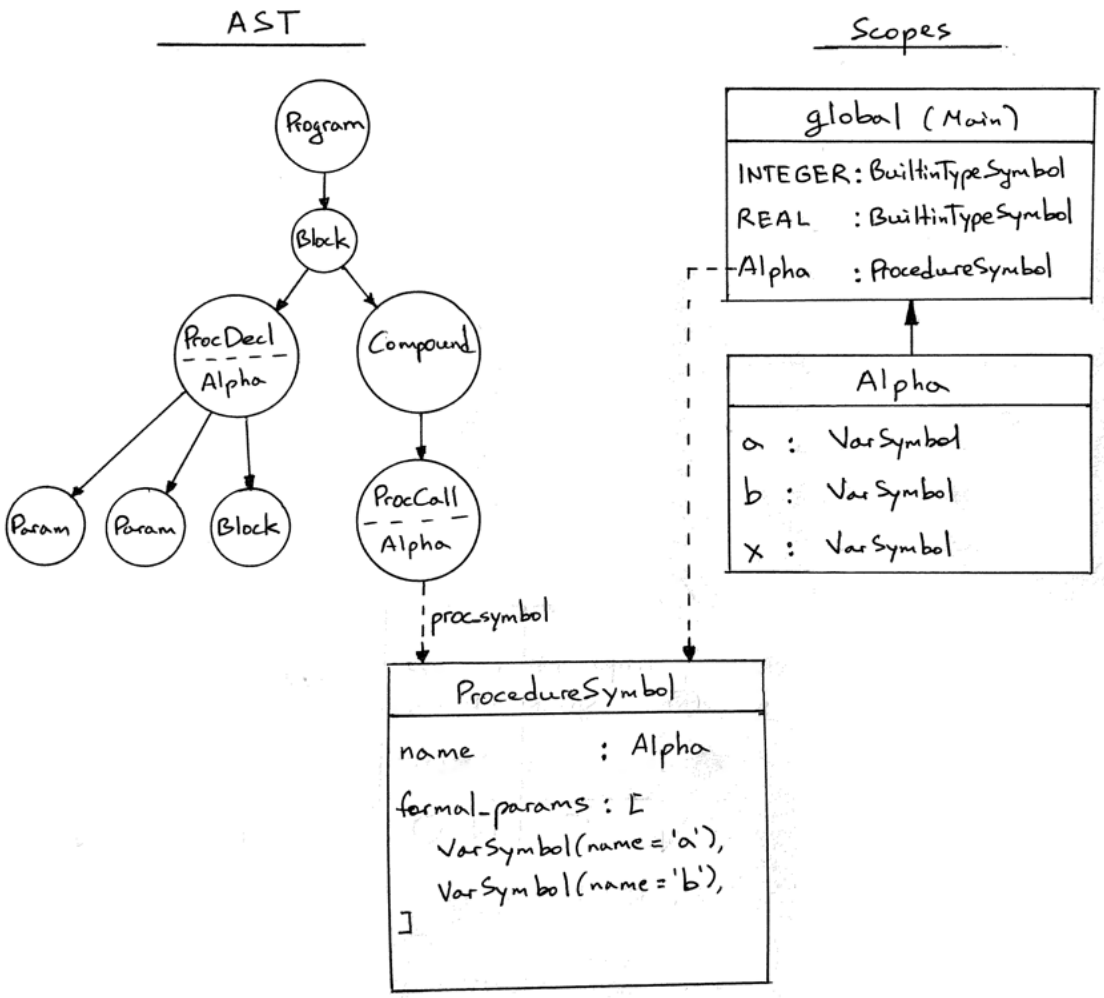
```
class SemanticAnalyzer(NodeVisitor):
    ...

    def visit_ProcedureCall(self, node):
        for param_node in node.actual_params:
            self.visit(param_node)
```

Because we have access to the current scope when traversing the AST tree in the semantic analyzer, we can look up the procedure symbol by a procedure name and then store the procedure symbol in the *proc_symbol* variable of the *ProcedureCall* AST node. Let's do this:

```
class SemanticAnalyzer(NodeVisitor):
    ...

    def visit_ProcedureCall(self, node):
        for param_node in node.actual_params:
            self.visit(param_node)

        proc_symbol = self.current_scope.lookup(node.proc_name)
        # accessed by the interpreter when executing procedure call
        node.proc_symbol = proc_symbol
```

In the code above, we simply resolve a procedure name to its procedure symbol, which is stored in one of the scoped symbol tables (in our case in the *global* scope, to be exact), and then assign the procedure symbol to the *proc_symbol* field of the *ProcedureCall* AST node.

For our sample program, after the semantic analysis phase and the actions described above, the AST tree will have a link to the *Alpha* procedure symbol in the global scope:



As you can see in the picture above, this setup allows us to get the procedure's formal parameters from the interpreter's *visit_ProcedureCall* method - when evaluating a *ProcedureCall* node - by simply accessing the *formal_params* field of the *proc_symbol* variable stored in the *ProcedureCall* AST node:

```
proc_symbol = node.proc_symbol

proc_symbol.formal_params  # aka parameters
```

b) After we get the list of formal parameters, we need to get a list of the procedure's actual parameters (arguments). Getting the list of arguments is easy because they are readily available from the *ProcedureCall* AST *node* itself:

```
node.actual_params  # aka arguments
```

c) And the last step. For each formal parameter, we need to get the corresponding actual parameter and save the pair in the procedure's activation record by using the formal parameter's name as the key and the actual parameter (argument), after having evaluated it, as the value

Let's take a look at the code that does building of the key-value pairs using the Python zip() (https://docs.python.org/3/library/functions.html#zip) function:

```
proc_symbol = node.proc_symbol

formal_params = proc_symbol.formal_params
actual_params = node.actual_params

for param_symbol, argument_node in zip(formal_params, actual_params):
    ar[param_symbol.name] = self.visit(argument_node)
```

Once you know how the Python zip() (https://docs.python.org/3/library/functions.html#zip) function works, the *for* loop above should be easy to understand. Here's a Python shell demonstration of the zip() (https://docs.python.org/3/library/functions.html#zip) function in action:

```
>>> formal_params = ['a', 'b', 'c']
>>> actual_params = [1, 2, 3]
>>>
>>> zipped = zip(formal_params, actual_params)
>>>
>>> list(zipped)
[('a', 1), ('b', 2), ('c', 3)]
```

The statement to store the key-value pairs in the activation record is very straightforward:

```
ar[param_symbol.name] = self.visit(argument_node)
```

The key is the name of a formal parameter, and the value is the evaluated value of the argument passed to the procedure call.

Here is the interpreter's *visit_ProcedureCall* method with all the modifications we've done so far:

```
class Interpreter(NodeVisitor):
    ...

    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

        formal_params = proc_symbol.formal_params
        actual_params = node.actual_params

        for param_symbol, argument_node in zip(formal_params, actual_params):
            ar[param_symbol.name] = self.visit(argument_node)
```

Step 3. Push the activation record onto the call stack

After we've created the AR and put all the procedure's parameters into the AR, we need to push the AR onto the stack. It's super easy to do. We need to add just one line of code:

```
self.call_stack.push(ar)
```

Remember: an AR of a currently executing procedure is always at the top of the stack. This way the currently executing procedure has easy access to its parameters and local variables. Here is the updated *visit_ProcedureCall* method:

```
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

    formal_params = proc_symbol.formal_params
    actual_params = node.actual_params

    for param_symbol, argument_node in zip(formal_params, actual_params):
        ar[param_symbol.name] = self.visit(argument_node)

    self.call_stack.push(ar)
```

Step 4. Execute the body of the procedure

Now that everything has been set up, let's execute the body of the procedure. The only problem is that neither the *ProcedureCall* AST *node* nor the procedure symbol *proc_symbol* knows anything about the body of the respective procedure declaration.

How do we get access to the body of the procedure declaration during execution of a procedure call? In other words, when traversing the AST tree and visiting the *ProcedureCall* AST node during the interpretation phase, we need to get access to the *block_node* variable of the corresponding *ProcedureDecl* node. The *block_node* variable holds a reference to an AST sub-tree that represents the body of the procedure. How can we access that variable from the *visit_ProcedureCall* method of the *Interpreter* class? Let's think about it.

We already have access to the procedure symbol that contains information about the procedure declaration, like the procedure's formal parameters, so let's find a way to store a reference to the *block_node* in the procedure symbol itself. The right spot to do that is the semantic analyzer's *visit_ProcedureDecl* method. In this method we have access to both the procedure symbol and the procedure's body, the *block_node* field of the *ProcedureDecl* AST node that points to the procedure body's AST sub-tree.

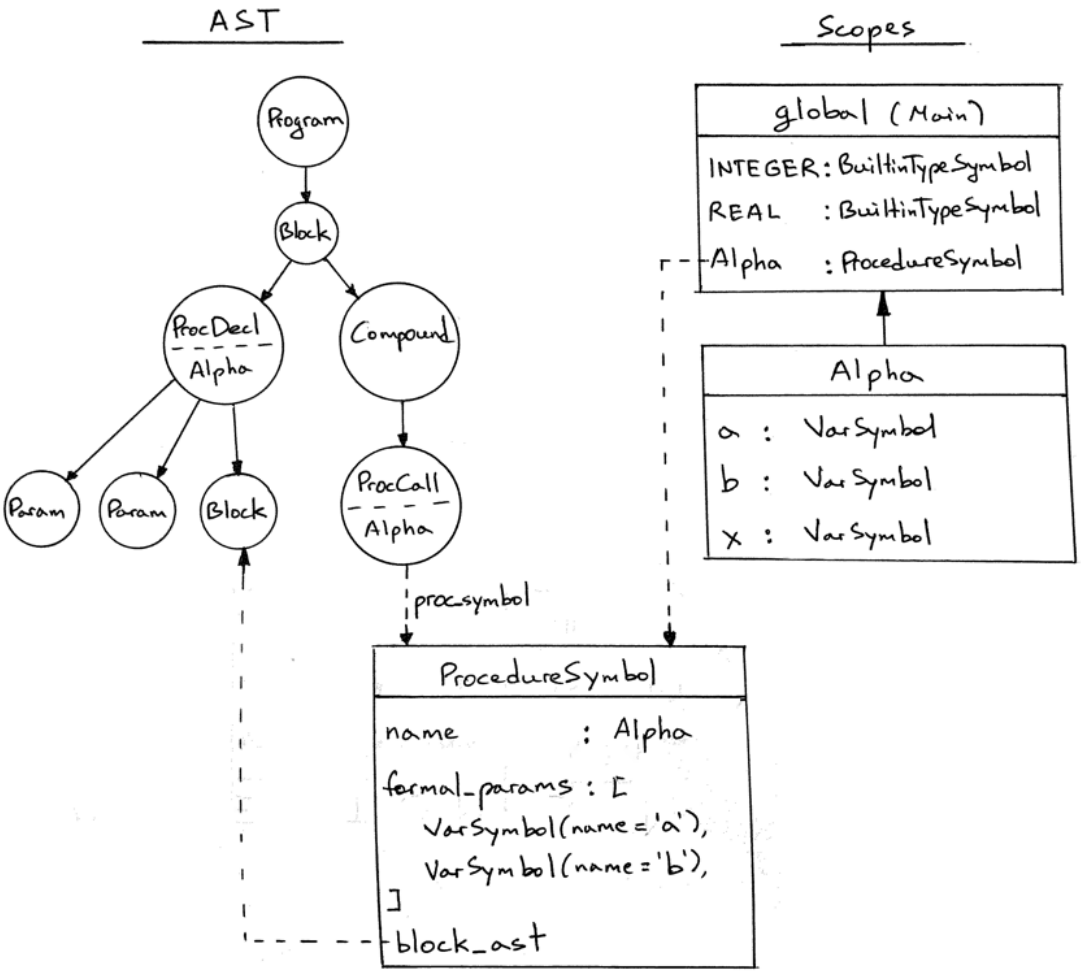
We have a procedure symbol, and we have a *block_node*. Let's store a pointer to the *block_node* in the *block_ast* field of the *proc_symbol*:

```
class SemanticAnalyzer(NodeVisitor):

    def visit_ProcedureDecl(self, node):
        proc_name = node.proc_name
        proc_symbol = ProcedureSymbol(proc_name)
        ...
        self.log(f'LEAVE scope: {proc_name}')

        # accessed by the interpreter when executing procedure call
        proc_symbol.block_ast = node.block_node
```

And to make it explicit, let's also extend the *ProcedureSymbol* class and add the *block_ast* field to it:

```
class ProcedureSymbol(Symbol):
    def __init__(self, name, formal_params=None):
        ...
        # a reference to procedure's body (AST sub-tree)
        self.block_ast = None
```

In the picture below you can see the extended *ProcedureSymbol* instance that stores a reference to the corresponding procedure's body (a *Block* node in the AST):



With all the above, executing the body of the procedure in the procedure call becomes as simple as visiting the procedure declaration's *Block* AST node accessible through the *block_ast* field of the procedure's *proc_symbol*:

```
self.visit(proc_symbol.block_ast)
```

Here is the fully updated *visit_ProcedureCall* method of the *Interpreter* class:

```
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

    formal_params = proc_symbol.formal_params
    actual_params = node.actual_params

    for param_symbol, argument_node in zip(formal_params, actual_params):
        ar[param_symbol.name] = self.visit(argument_node)

    self.call_stack.push(ar)

    # evaluate procedure body
    self.visit(proc_symbol.block_ast)
```

If you remember from the previous article (https://ruslanspivak.com/lsbasi-part17), the *visit_Assignment* and *visit_Var* methods use an AR at the top of the call stack to access and store variables:

```
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
```

These methods stay unchanged. When interpreting the body of a procedure, these methods will store and access values from the AR of the currently executing procedure, which will be at the top of the stack. We'll see shortly how it all fits and works together.

Step 5. Pop the activation record off the stack

After we're done evaluating the body of the procedure, we no longer need the procedure's AR, so we pop it off the call stack right before leaving the *visit_ProcedureCall* method. Remember, the top of the call stack contains an AR for a currently executing procedure, function, or program, so once we're done evaluating one of those routines, we need to pop their respective AR off the call stack using the call stack's *pop()* method:

```
self.call_stack.pop()
```

Let's put it all together and also add some logging to the *visit_ProcedureCall* method to log the contents of the *call stack* right after pushing the procedure's AR onto the *call stack* and right before popping it off the stack:

```
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

    formal_params = proc_symbol.formal_params
    actual_params = node.actual_params

    for param_symbol, argument_node in zip(formal_params, actual_params):
        ar[param_symbol.name] = self.visit(argument_node)

    self.call_stack.push(ar)

    self.log(f'ENTER: PROCEDURE {proc_name}')
    self.log(str(self.call_stack))

    # evaluate procedure body
    self.visit(proc_symbol.block_ast)

    self.log(f'LEAVE: PROCEDURE {proc_name}')
    self.log(str(self.call_stack))

    self.call_stack.pop()
```

Let's take our modified interpreter for a ride and see how it executes procedure calls. Download the following sample program from GitHub (https://github.com/rspivak/lsbasi/tree/master/part18) or save it as part18.pas (https://github.com/rspivak/lsbasi/blob/master/part18/part18.pas):

```
program Main;

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

begin { Main }

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

end.  { Main }
```

Download the interpreter file spi.py (https://github.com/rspivak/lsbasi/blob/master/part18/spi.py) from GitHub (https://github.com/rspivak/lsbasi/tree/master/part18/) and run it on the command line with the following arguments:

```
$ python spi.py part18.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


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
```

So far, so good. Let's take a closer look at the output and inspect the contents of the call stack during program and procedure execution.

1. The interpreter first prints

```
ENTER: PROGRAM Main
CALL STACK
1: PROGRAM Main
```

when visiting the *Program* AST node before executing the body of the program. At this point the *call stack* has one *activation record.* This activation record is at the top of the call stack and it's used for storing global variables. Because we don't have any global variables in our sample program, there is nothing in the activation record.

2. Next, the interpreter prints

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

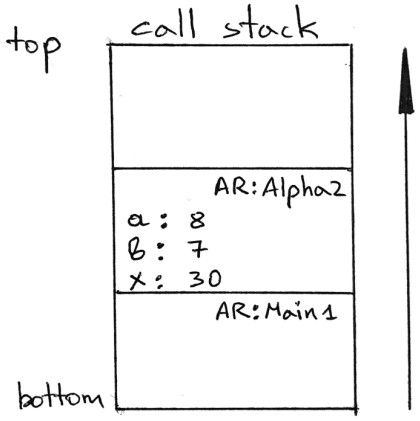
when it visits the *ProcedureCall* AST node for the *Alpha(3 + 5, 7)* procedure call. At this point the body of the *Alpha* procedure hasn't been evaluated yet and the *call stack* has two activation records: one for the *Main* program at the bottom of the stack (nesting level 1) and one for the *Alpha* procedure call, at the top of the stack (nesting level 2). The AR at the top of the stack holds the values of the procedure arguments *a* and *b* only; there is no value for the local variable *x* in the AR because the body of the procedure hasn't been evaluated yet.

3. Up next, the interpreter prints

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

when it's about to leave the *ProcedureCall* AST node for the *Alpha(3 + 5, 7)* procedure call but before popping off the AR for the *Alpha* procedure.

From the output above, you can see that in addition to the procedure arguments, the AR for the currently executing procedure *Alpha* now also contains the result of the assignment to the local variable *x*, the result of executing the *x := (a + b) * 2;* statement in the body of the procedure. At this point the call stack visually looks like this:

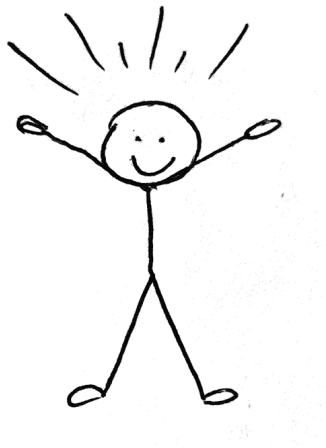


4. And finally the interpreter prints

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

when it leaves the *Program* AST node but before it pops off the AR for the main program. As you can see, the activation record for the main program is the only AR left in the stack because the AR for the *Alpha* procedure call got popped off the stack earlier, right before finishing executing the *Alpha* procedure call.

That's it. Our interpreter successfully executed a procedure call. If you've reached this far, congratulations!



It is a huge milestone for us. Now you know how to execute procedure calls. And if you've been waiting for this article for a long time, thank you for your patience.

That's all for today. In the next article, we'll expand on the current material and talk about executing nested procedure calls. So stay tuned and see you next time!

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