Let's Build A Simple Interpreter. Part 16: Recognizing Procedure Calls (https://ruslanspivak.com/lsbasi-part16/)

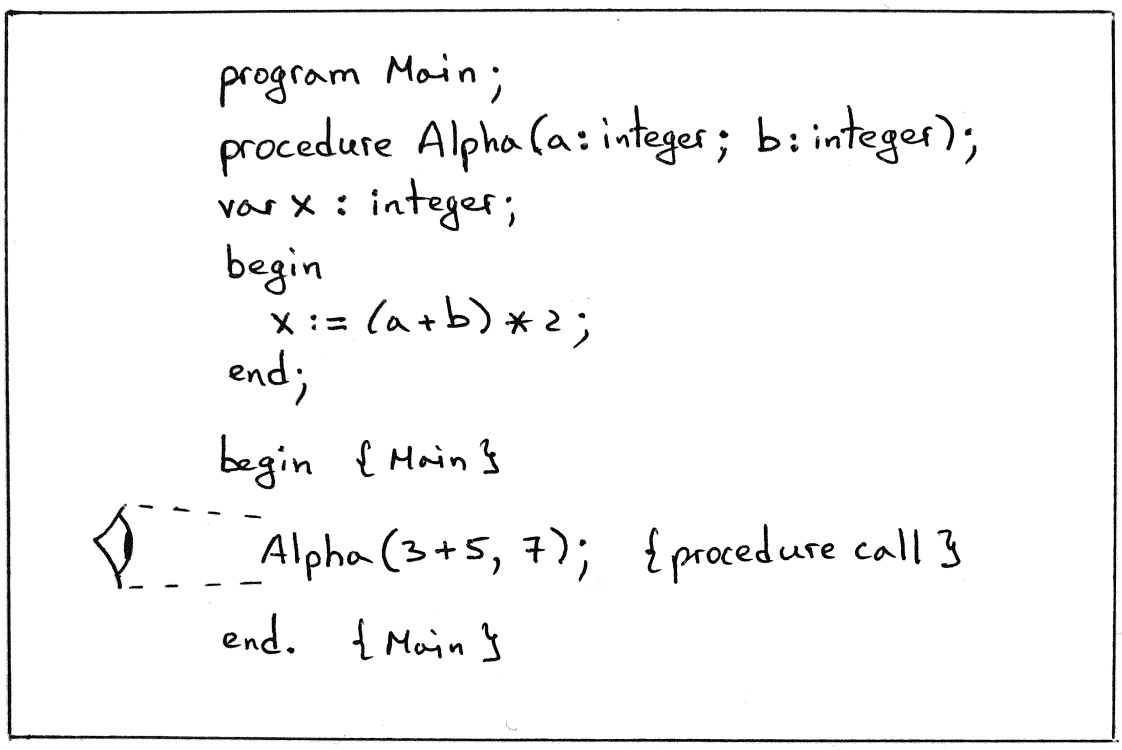
Date Tue, July 23, 2019

> *"Learning is like rowing upstream: not to advance is to drop back." — Chinese proverb*

Today we're going to extend our interpreter to recognize procedure calls. I hope by now you've flexed your coding muscles and are ready to tackle this step. This is a necessary step for us before we can learn how to execute procedure calls, which will be a topic that we will cover in great detail in future articles.

The goal for today is to make sure that when our interpreter reads a program with a procedure call, the parser constructs an Abstract Syntax Tree (AST) with a new tree node for the procedure call, and the semantic analyzer and the interpreter don't throw any errors when walking the AST.

Let's take a look at a sample program that contains a procedure call *Alpha(3 + 5, 7)*:



Making our interpreter recognize programs like the one above will be our focus for today.

As with any new feature, we need to update various components of the interpreter to support this feature. Let's dive into each of those components one by one.

First, we need to update the parser. Here is a list of all the parser changes that we need to make to be able to parse procedure calls and build the right AST:

1. We need to add a new AST node to represent a procedure call
2. We need to add a new grammar rule for procedure call statements; then we need to implement the rule in code
3. We need to extend the *statement* grammar rule to include the rule for procedure call statements and update the *statement* method to reflect the changes in the grammar

1. Let's start by creating a separate class to represent a procedure call AST node. Let's call the class *ProcedureCall*:

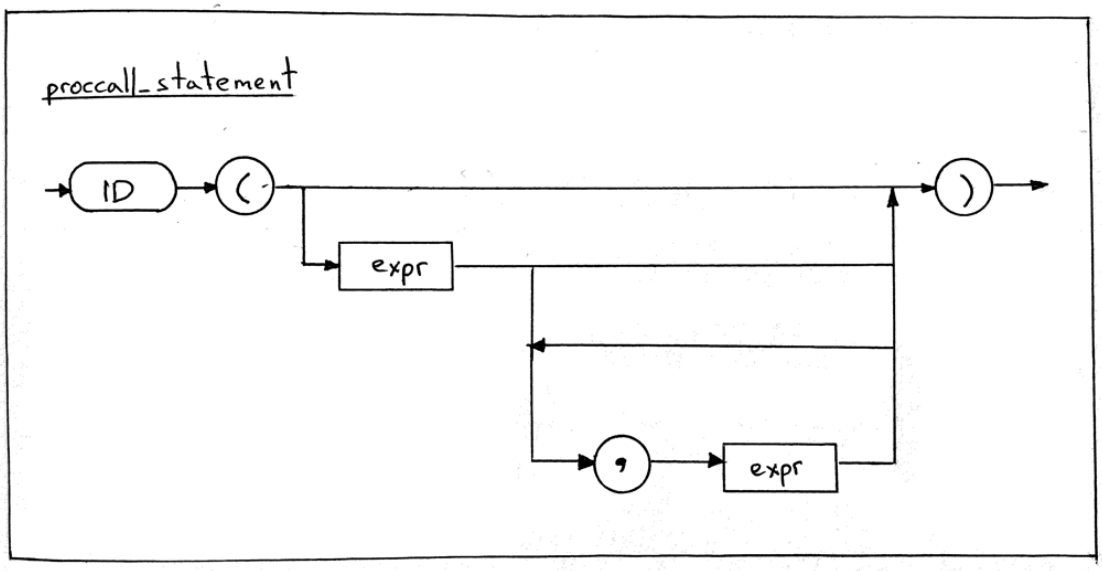
```
class ProcedureCall(AST):
    def __init__(self, proc_name, actual_params, token):
        self.proc_name = proc_name
        self.actual_params = actual_params  # a list of AST nodes
        self.token = token
```

The *ProcedureCall* class constructor takes three parameters: a procedure name, a list of actual parameters (a.k.a arguments), and a token. Nothing really special here, just enough information for us to capture a particular procedure call.

2. The next step that we need to take is to extend our grammar and add a grammar rule for procedure calls. Let's call the rule *proccall_statement*:

```
proccall_statement : ID LPAREN (expr (COMMA expr)*)? RPAREN
```

Here is a corresponding syntax diagram for the rule:



From the diagram above you can see that a procedure call is an ID token followed by a left parenthesis, followed by zero or more expressions separated by commas, followed by a right parenthesis. Here are some of the procedure call examples that fit the rule:

```
Alpha();
Alpha(1);
Alpha(3 + 5, 7);
```

Next, let's implement the rule in our parser by adding a *proccall_statement* method

```
def proccall_statement(self):
    """proccall_statement : ID LPAREN (expr (COMMA expr)*)? RPAREN"""
    token = self.current_token

    proc_name = self.current_token.value
    self.eat(TokenType.ID)
    self.eat(TokenType.LPAREN)
    actual_params = []
    if self.current_token.type != TokenType.RPAREN:
        node = self.expr()
        actual_params.append(node)

    while self.current_token.type == TokenType.COMMA:
        self.eat(TokenType.COMMA)
        node = self.expr()
        actual_params.append(node)

    self.eat(TokenType.RPAREN)

    node = ProcedureCall(
        proc_name=proc_name,
        actual_params=actual_params,
        token=token,
    )
    return node
```

The implementation is pretty straightforward and follows the grammar rule: the method parses a procedure call and returns a new *ProcedureCall* AST node.

3. And the last changes to the parser that we need to make are: extend the *statement* grammar rule by adding the *proccall_statement* rule and update the *statement* method to call the *proccall_statement* method.

Here is the updated *statement* grammar rule, which includes the *proccall_statement* rule:

```
statement : compound_statement
          | proccall_statement
          | assignment_statement
          | empty
```

Now, we have a tricky situation on hand where we have two grammar rules - *proccall_statement* and *assignment_statement* - that start with the same token, the ID token. Here are their complete grammar rules put together for comparison:

```
proccall_statement : ID LPAREN (expr (COMMA expr)*)? RPAREN

assignment_statement : variable ASSIGN expr
variable: ID
```

How do you distinguish between a procedure call and an assignment in a case like that? They are both statements and they both start with an ID token. In the fragment of code below, the ID token's value(lexeme) for both statements is *foo*:

```
foo();     { procedure call }
foo := 5;  { assignment }
```

The parser should recognize *foo();* above as a procedure call and *foo := 5;* as an assignment. But what can we do to help the parser to distinguish between procedure calls and assignments? According to our new *proccall_statement* grammar rule, procedure calls start with an ID token followed by a left parenthesis. And that's what we are going to rely on in the parser to distinguish between procedure calls and assignments to variables - the presence of a left parenthesis after the ID token:

```
if (self.current_token.type == TokenType.ID and
    self.lexer.current_char == '('
):
    node = self.proccall_statement()
elif self.current_token.type == TokenType.ID:
    node = self.assignment_statement()
```

As you can see in the code above, first we check if the current token is an ID token and then we check if it's followed by a left parenthesis. If it is, we parse a procedure call, otherwise we parse an assignment statement.

Here is the full updated version of the *statement* method:

```
def statement(self):
    """
    statement : compound_statement
              | proccall_statement
              | assignment_statement
              | empty
    """
    if self.current_token.type == TokenType.BEGIN:
        node = self.compound_statement()
    elif (self.current_token.type == TokenType.ID and
          self.lexer.current_char == '('
    ):
        node = self.proccall_statement()
    elif self.current_token.type == TokenType.ID:
        node = self.assignment_statement()
    else:
        node = self.empty()
    return node
```

So far so good. The parser can now parse procedure calls. One thing to keep in mind though is that Pascal procedures don't have return statements, so we can't use procedure calls in expressions. For example, the following example will not work if *Alpha* is a procedure:

```
x := 10 * Alpha(3 + 5, 7);
```

That's why we added *proccall_statement* to the *statements* method only and nowhere else. Not to worry, later in the series we'll learn about Pascal functions that can return values and also can be used in expressions and assignments.

These are all the changes for our parser. Next up is the semantic analyzer changes.

The only change we need to make in our semantic analyzer to support procedure calls is to add a *visit_ProcedureCall* method:

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

All the method does is iterate over a list of actual parameters passed to a procedure call and visit each parameter node in turn. It's important not to forget to visit each parameter node because each parameter node is an AST sub-tree in itself.

That was easy, wasn't it? Okay, now moving on to interpreter changes.

The interpreter changes, compared to the changes to the semantic analyzer, are even simpler - we only need to add an empty *visit_ProcedureCall* method to the *Interpreter* class:

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

With all the above changes in place, we now have an interpreter that can recognize procedure calls. And by that I mean the interpreter can parse procedure calls and create an AST with *ProcedureCall* nodes corresponding to those procedure calls. Here is the sample Pascal program we saw at the beginning of the article that we want our interpreter to be tested on:

```
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 }
```

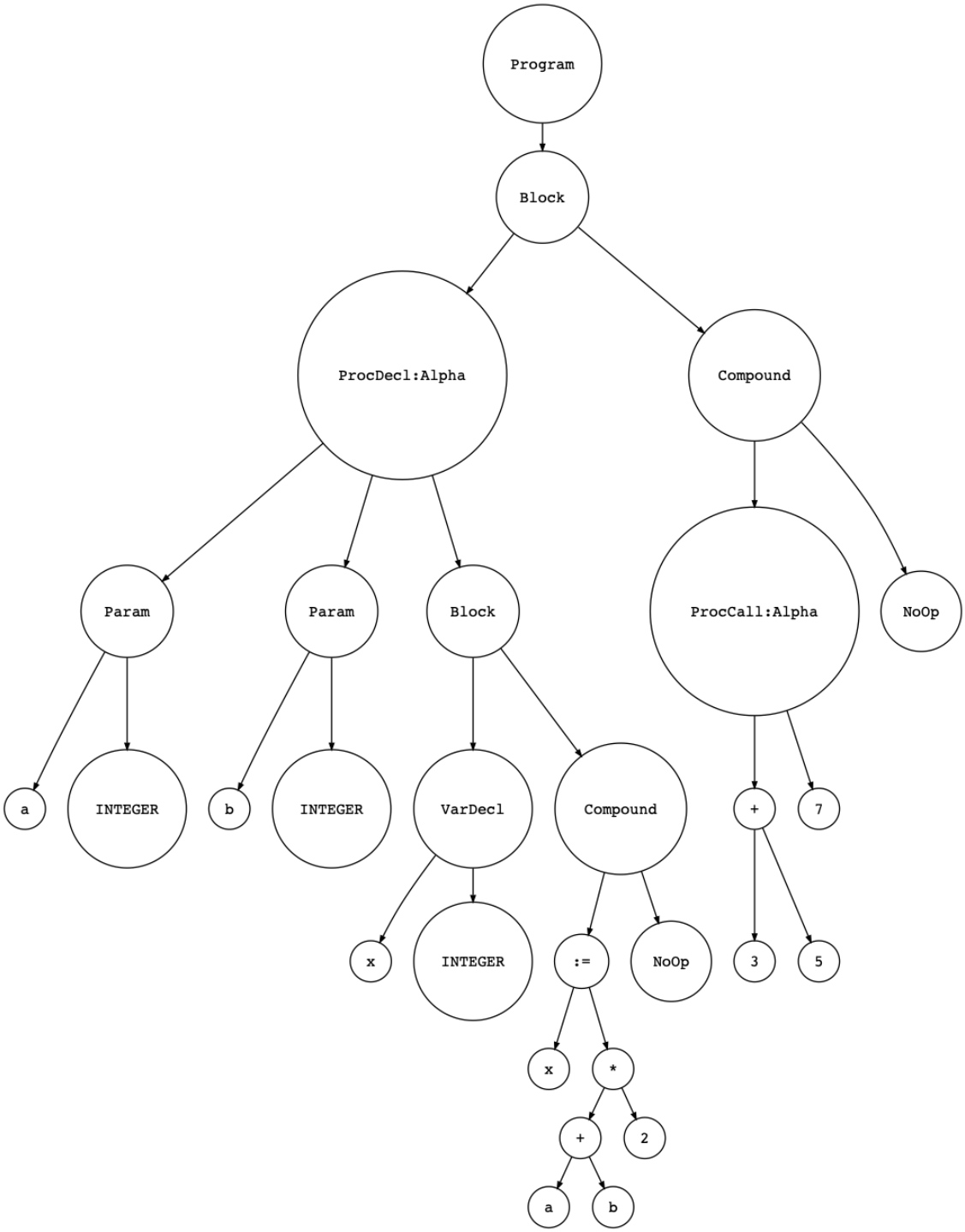
Download the above program from GitHub (https://github.com/rspivak/lsbasi/tree/master/part16) or save the code to the file part16.pas

See for yourself that running our updated interpreter (https://github.com/rspivak/lsbasi/tree/master/part16) with the part16.pas as its input file does not generate any errors:

```
$ python spi.py part16.pas
$
```

So far so good, but no output is not that exciting. :) Let's get a bit visual and generate an AST for the above program and then visualize the AST using an updated version of the genastdot.py (https://github.com/rspivak/lsbasi/tree/master/part16/genastdot.py) utility:

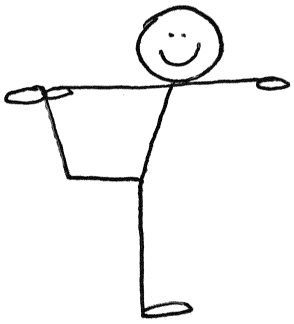
```
$ python genastdot.py part16.pas > ast.dot && dot -Tpng -o ast.png ast.dot
```



That's better. In the picture above you can see our new *ProcCall* AST node labeled *ProcCall:Alpha* for the *Alpha(3 + 5, 7)* procedure call. The two children of the *ProcCall:Alpha* node are the subtrees for the arguments *3 + 5* and *7* passed to the *Alpha(3 + 5, 7)* procedure call.

Okay, we have accomplished our goal for today: when encountering a procedure call, the parser constructs an AST with a *ProcCall* node for the procedure call, and the semantic analyzer and the interpreter don't throw any errors when walking the AST.

Now, it's time for an exercise.



Exercise: Add a check to the semantic analyzer that verifies that the number of arguments (actual parameters) passed to a procedure call equals the number of formal parameters defined in the corresponding procedure declaration. Let's take the *Alpha* procedure declaration we used earlier in the article as an example:

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

The number of formal parameters in the procedure declaration above is two (integers *a* and *b*). Your check should throw an error if you try to call the procedure with a number of arguments other than two:

```
Alpha();          { 0 arguments –> ERROR }
Alpha(1);         { 1 argument  –> ERROR }
Alpha(1, 2, 3);   { 3 arguments –> ERROR }
```

You can find a solution to the exercise in the file *solutions.txt* on GitHub (https://github.com/rspivak/lsbasi/tree/master/part16), but try to work out your own solution first before peeking into the file.

That's all for today. In the next article we'll begin to learn how to interpret procedure calls. We will cover topics like call stack and activation records. It is going to be a wild ride :) So stay tuned and see you next time!

Resources used in preparation for this article (some 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?