Let’s Build A Simple Interpreter. Part 11. (https://ruslanspivak.com/lsbasi-part11/)

Date Tue, September 20, 2016

I was sitting in my room the other day and thinking about how much we had covered, and I thought I would recap what we’ve learned so far and what lies ahead of us.

Up until now we’ve learned:

- How to break sentences into tokens. The process is called ***lexical analysis*** and the part of the interpreter that does it is called a ***lexical analyzer***, ***lexer***, ***scanner***, or ***tokenizer***. We’ve learned how to write our own ***lexer*** from the ground up without using regular expressions or any other tools like Lex (https://en.wikipedia.org/wiki/Lex_(software)).
- How to recognize a phrase in the stream of tokens. The process of recognizing a phrase in the stream of tokens or, to put it differently, the process of finding structure in the stream of tokens is called ***parsing*** or ***syntax analysis***. The part of an interpreter or compiler that performs that job is called a ***parser*** or ***syntax analyzer***.
- How to represent a programming language’s syntax rules with ***syntax diagrams***, which are a graphical representation of a programming language’s syntax rules.

Syntax diagrams visually show us which statements are allowed in our programming language and which are not.

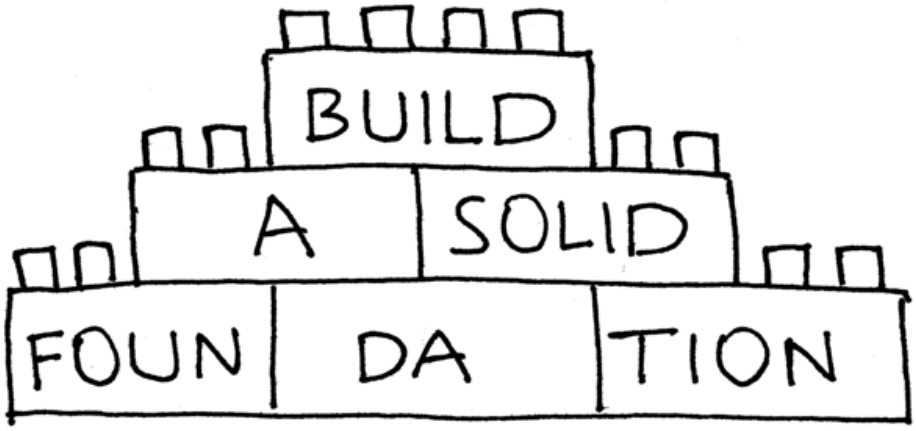
- How to use another widely used notation for specifying the syntax of a programming language. It's called ***context-free grammars*** (***grammars***, for short) or ***BNF*** (Backus-Naur Form).
- How to map a ***grammar*** to code and how to write a ***recursive-descent parser***.
- How to write a really basic ***interpreter***.
- How ***associativity*** and ***precedence*** of operators work and how to construct a grammar using a precedence table.
- How to build an ***Abstract Syntax Tree*** (AST) of a parsed sentence and how to represent the whole source program in Pascal as one big ***AST***.
- How to walk an AST and how to implement our interpreter as an AST node visitor.

With all that knowledge and experience under our belt, we've built an interpreter that can scan, parse, and build an AST and interpret, by walking the AST, our very first complete Pascal program. Ladies and gentlemen, I honestly think if you've reached this far, you deserve a pat on the back. But don't let it go to your head. Keep going. Even though we've covered a lot of ground, there are even more exciting parts coming our way.

With everything we've covered so far, we are almost ready to tackle topics like:

- Nested procedures and functions
- Procedure and function calls
- Semantic analysis (type checking, making sure variables are declared before they are used, and basically checking if a program makes sense)
- Control flow elements (like IF statements)
- Aggregate data types (Records)
- More built-in types
- Source-level debugger
- Miscellanea (All the other goodness not mentioned above :)

But before we cover those topics, we need to build a solid foundation and infrastructure.



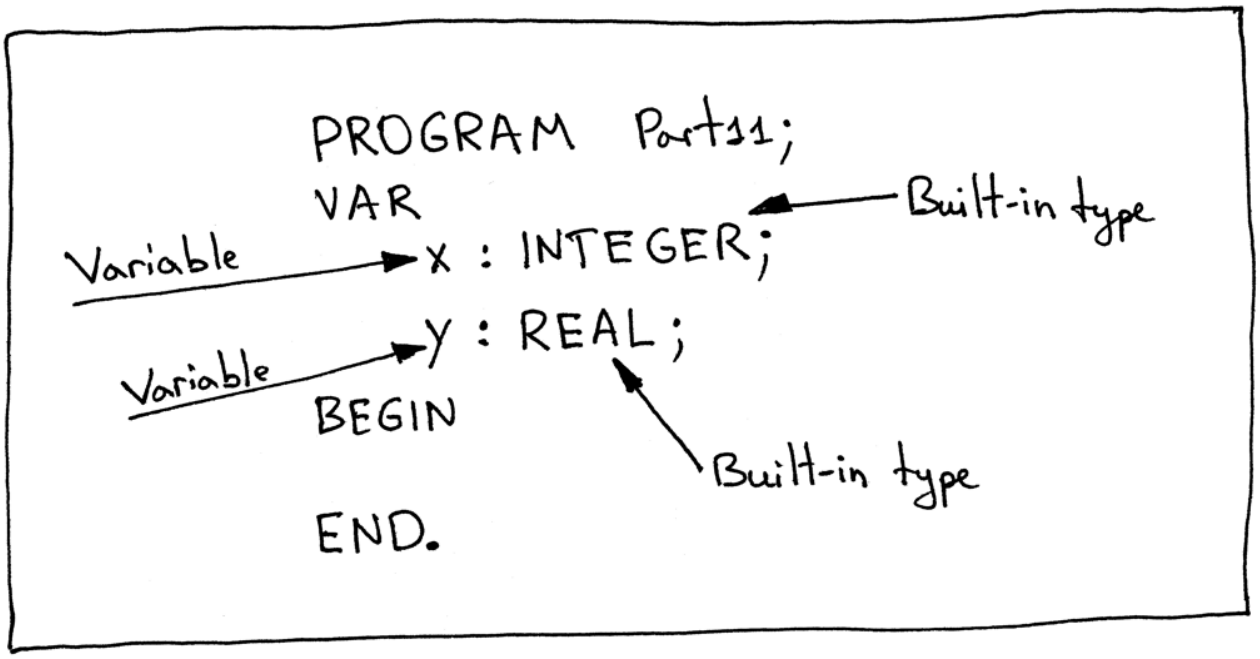
This is where we start diving deeper into the super important topic of symbols, symbol tables, and scopes. The topic itself will span several articles. It's that important and you'll see why. Okay, let's start building that foundation and infrastructure, then, shall we?

First, let's talk about symbols and why we need to track them. What is a ***symbol***? For our purposes, we'll informally define ***symbol*** as an identifier of some program entity like a variable, subroutine, or built-in type. For symbols to be useful they need to have at least the following information about the program entities they identify:

- Name (for example, 'x', 'y', 'number')
- Category (Is it a variable, subroutine, or built-in type?)
- Type (INTEGER, REAL)

Today we'll tackle variable symbols and built-in type symbols because we've already used variables and types before. By the way, the "built-in" type just means a type that hasn't been defined by you and is available for you right out of the box, like INTEGER and REAL types that you've seen and used before.

Let's take a look at the following Pascal program, specifically at the variable declaration part. You can see in the picture below that there are four symbols in that section: two variable symbols (*x* and *y*) and two built-in type symbols (*INTEGER* and *REAL*).



How can we represent symbols in code? Let's create a base *Symbol* class in Python:

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

As you can see, the class takes the *name* parameter and an optional *type* parameter (not all symbols may have a type associated with them). What about the category of a symbol? We'll encode the category of a symbol in the class name itself, which means we'll create separate classes to represent different symbol categories.

Let's start with basic built-in types. We've seen two built-in types so far, when we declared variables: INTEGER and REAL. How do we represent a built-in type symbol in code? Here is one option:

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

    def __str__(self):
        return self.name

    __repr__ = __str__
```

The class inherits from the *Symbol* class and the constructor requires only a name of the type. The category is encoded in the class name, and the *type* parameter from the base class for a built-in type symbol is *None*. The double underscore or *dunder* (as in "Double

UNDERscore") methods *__str__* and *__repr__* are special Python methods and we've defined them to have a nice formatted message when you print a symbol object.

Download the interpreter file (https://github.com/rspivak/lsbasi/blob/master/part11/python/spi.py) and save it as *spi.py*; launch a python shell from the same directory where you saved the spi.py file, and play with the class we've just defined interactively:

```
$ python
>>> from spi import BuiltinTypeSymbol
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> int_type
INTEGER
>>> real_type = BuiltinTypeSymbol('REAL')
>>> real_type
REAL
```

How can we represent a variable symbol? Let's create a *VarSymbol* class:

```
class VarSymbol(Symbol):
    def __init__(self, name, type):
        super().__init__(name, type)

    def __str__(self):
        return '<{name}:{type}>'.format(name=self.name, type=self.type)

    __repr__ = __str__
```

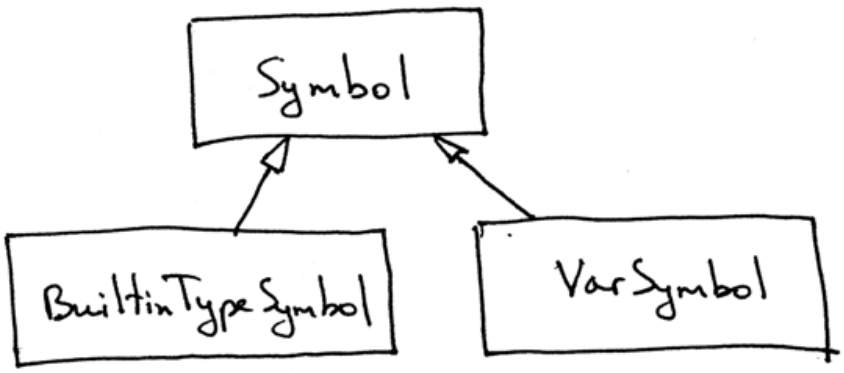
In the class we made both the *name* and the *type* parameters required parameters and the class name *VarSymbol* clearly indicates that an instance of the class will identify a variable symbol (the category is *variable*.)

Back to the interactive python shell to see how we can manually construct instances for our variable symbols now that we know how to construct *BuiltinTypeSymbol* class instances:

```
$ python
>>> from spi import BuiltinTypeSymbol, VarSymbol
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> real_type = BuiltinTypeSymbol('REAL')
>>>
>>> var_x_symbol = VarSymbol('x', int_type)
>>> var_x_symbol
<x:INTEGER>
>>> var_y_symbol = VarSymbol('y', real_type)
>>> var_y_symbol
<y:REAL>
```

As you can see, we first create an instance of a built-in type symbol and then pass it as a parameter to *VarSymbol*'s constructor.

Here is the hierarchy of symbols we've defined in visual form:

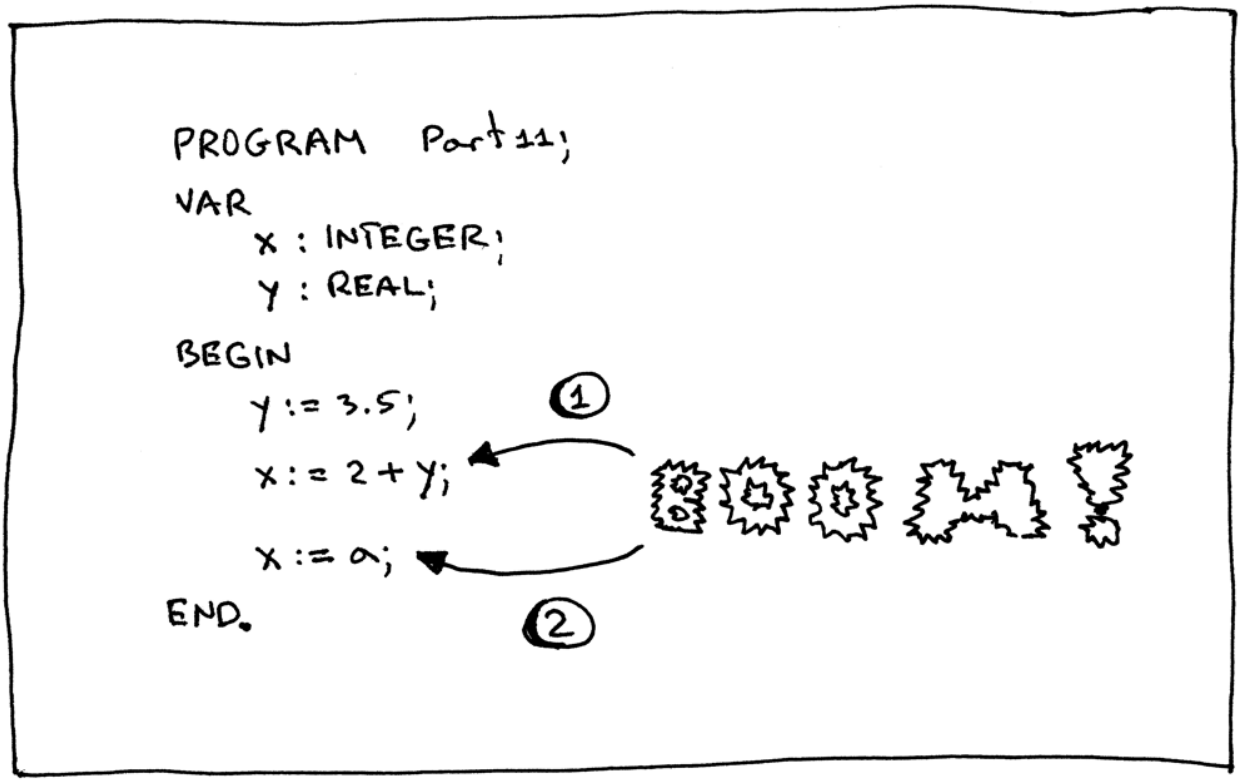


So far so good, but we haven't answered the question yet as to why we even need to track those symbols in the first place.

Here are some of the reasons:

- To make sure that when we assign a value to a variable the types are correct (type checking)
- To make sure that a variable is declared before it is used

Take a look at the following incorrect Pascal program, for example:



There are two problems with the program above (you can compile it with *fpc* (http://www.freepascal.org/) to see it for yourself):

1. In the expression *"x := 2 + y;"* we assigned a decimal value to the variable "x" that was declared as integer. That wouldn't compile because the types are incompatible.
2. In the assignment statement *"x := a;"* we referenced the variable "a" that wasn't declared - wrong!

To be able to identify cases like that even before interpreting/evaluating the source code of the program at run-time, we need to track program symbols. And where do we store the symbols that we track? I think you've guessed it right - in the symbol table!

What is a ***symbol table***? A ***symbol table*** is an abstract data type (***ADT***) for tracking various symbols in source code. Today we're going to implement our symbol table as a separate class with some helper methods:

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

    def __str__(self):
        s = 'Symbols: {symbols}'.format(
            symbols=[value for value in self._symbols.values()]
        )
        return s

    __repr__ = __str__

    def define(self, symbol):
        print('Define: %s' % symbol)
        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
```

There are two main operations that we will be performing with the symbol table: storing symbols and looking them up by name: hence, we need two helper methods - *define* and *lookup*.

The method *define* takes a symbol as a parameter and stores it internally in its *_symbols* ordered dictionary using the symbol's name as a key and the symbol instance as a value. The method *lookup* takes a symbol name as a parameter and returns a symbol if it finds it or "None" if it doesn't.

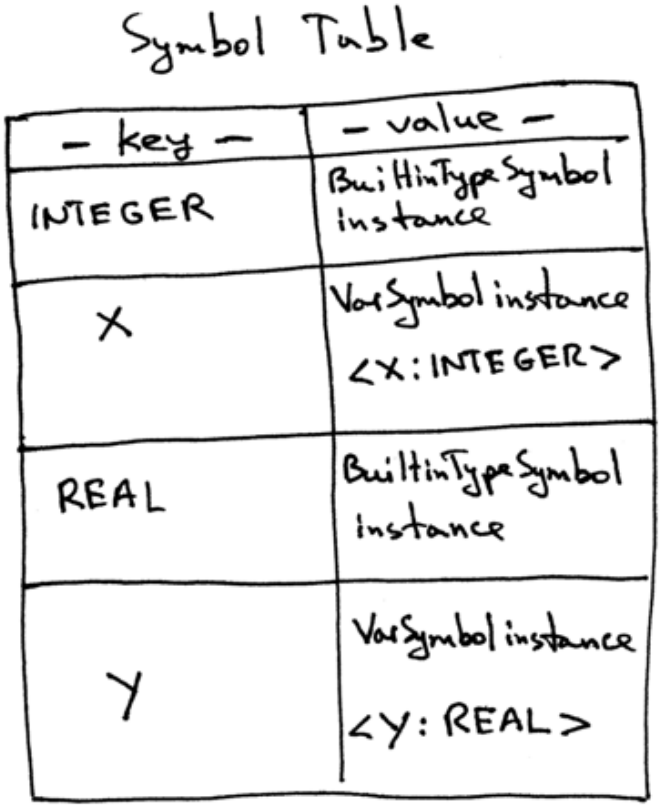
Let's manually populate our symbol table for the same Pascal program we've used just recently where we were manually creating variable and built-in type symbols:

```
PROGRAM Part11;
VAR
   x : INTEGER;
   y : REAL;

BEGIN

END.
```

Launch a Python shell again and follow along:

```
$ python
>>> from spi import SymbolTable, BuiltinTypeSymbol, VarSymbol
>>> symtab = SymbolTable()
>>> int_type = BuiltinTypeSymbol('INTEGER')
>>> symtab.define(int_type)
Define: INTEGER
>>> symtab
Symbols: [INTEGER]
>>>
>>> var_x_symbol = VarSymbol('x', int_type)
>>> symtab.define(var_x_symbol)
Define: <x:INTEGER>
>>> symtab
Symbols: [INTEGER, <x:INTEGER>]
>>>
>>> real_type = BuiltinTypeSymbol('REAL')
>>> symtab.define(real_type)
Define: REAL
>>> symtab
Symbols: [INTEGER, <x:INTEGER>, REAL]
>>>
>>> var_y_symbol = VarSymbol('y', real_type)
>>> symtab.define(var_y_symbol)
Define: <y:REAL>
>>> symtab
Symbols: [INTEGER, <x:INTEGER>, REAL, <y:REAL>]
```

If you looked at the contents of the *_symbols* dictionary it would look something like this:



How do we automate the process of building the symbol table? We'll just write another node visitor that walks the AST built by our parser! This is another example of how useful it is to have an intermediary form like AST. Instead of extending our parser to deal with the symbol table, we separate concerns and write a new node visitor class. Nice and clean. :)

Before doing that, though, let's extend our *SymbolTable* class to initialize the built-in types when the symbol table instance is created. Here is the full source code for today's *SymbolTable* class:

```
class SymbolTable(object):
    def __init__(self):
        self._symbols = OrderedDict()
        self._init_builtins()

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

    def __str__(self):
        s = 'Symbols: {symbols}'.format(
            symbols=[value for value in self._symbols.values()]
        )
        return s

    __repr__ = __str__

    def define(self, symbol):
        print('Define: %s' % symbol)
        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 onto the *SymbolTableBuilder* AST node visitor:

```
class SymbolTableBuilder(NodeVisitor):
    def __init__(self):
        self.symtab = SymbolTable()

    def visit_Block(self, node):
        for declaration in node.declarations:
            self.visit(declaration)
        self.visit(node.compound_statement)

    def visit_Program(self, node):
        self.visit(node.block)

    def visit_BinOp(self, node):
        self.visit(node.left)
        self.visit(node.right)

    def visit_Num(self, node):
        pass

    def visit_UnaryOp(self, node):
        self.visit(node.expr)

    def visit_Compound(self, node):
        for child in node.children:
            self.visit(child)

    def visit_NoOp(self, node):
        pass

    def visit_VarDecl(self, node):
        type_name = node.type_node.value
        type_symbol = self.symtab.lookup(type_name)
        var_name = node.var_node.value
        var_symbol = VarSymbol(var_name, type_symbol)
        self.symtab.define(var_symbol)
```

You've seen most of those methods before in the *Interpreter* class, but the *visit_VarDecl* method deserves some special attention. Here it is again:

```
def visit_VarDecl(self, node):
    type_name = node.type_node.value
    type_symbol = self.symtab.lookup(type_name)
    var_name = node.var_node.value
    var_symbol = VarSymbol(var_name, type_symbol)
    self.symtab.define(var_symbol)
```

This method is responsible for visiting (walking) a *VarDecl* AST node and storing the corresponding symbol in the symbol table. First, the method looks up the built-in type symbol by name in the symbol table, then it creates an instance of the *VarSymbol* class and stores (defines) it in the symbol table.

Let's take our *SymbolTableBuilder* AST walker for a test drive and see it in action:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM Part11;
... VAR
...    x : INTEGER;
...    y : REAL;
...
... BEGIN
...
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <x:INTEGER>
Lookup: REAL
Define: <y:REAL>
>>> # Let's examine the contents of our symbol table
…
>>> symtab_builder.symtab
Symbols: [INTEGER, REAL, <x:INTEGER>, <y:REAL>]
```

In the interactive session above, you can see the sequence of “Define: …” and “Lookup: …” messages that indicate the order in which symbols are defined and looked up in the symbol table. The last command in the session prints the contents of the symbol table and you can see that it’s exactly the same as the contents of the symbol table that we’ve built manually before. The magic of AST node visitors is that they pretty much do all the work for you. :)

We can already put our symbol table and symbol table builder to good use: we can use them to verify that variables are declared before they are used in assignments and expressions. All we need to do is just extend the visitor with two more methods: *visit_Assign* and *visit_Var*:

```
def visit_Assign(self, node):
    var_name = node.left.value
    var_symbol = self.symtab.lookup(var_name)
    if var_symbol is None:
        raise NameError(repr(var_name))

    self.visit(node.right)

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

    if var_symbol is None:
        raise NameError(repr(var_name))
```

These methods will raise a *NameError* exception if they cannot find the symbol in the symbol table.

Take a look at the following program, where we reference the variable “b” that hasn’t been declared yet:

```
PROGRAM NameError1;
VAR
   a : INTEGER;

BEGIN
   a := 2 + b;
END.
```

Let's see what happens if we construct an AST for the program and pass it to our symbol table builder to visit:

```
$ python
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError1;
... VAR
...    a : INTEGER;
...
... BEGIN
...    a := 2 + b;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: a
Lookup: b
Traceback (most recent call last):
  ...
  File "spi.py", line 674, in visit_Var
    raise NameError(repr(var_name))
NameError: 'b'
```

Exactly what we were expecting!

Here is another error case where we try to assign a value to a variable that hasn't been defined yet, in this case the variable 'a':

```
PROGRAM NameError2;
VAR
   b : INTEGER;

BEGIN
   b := 1;
   a := b + 2;
END.
```

Meanwhile, in the Python shell:

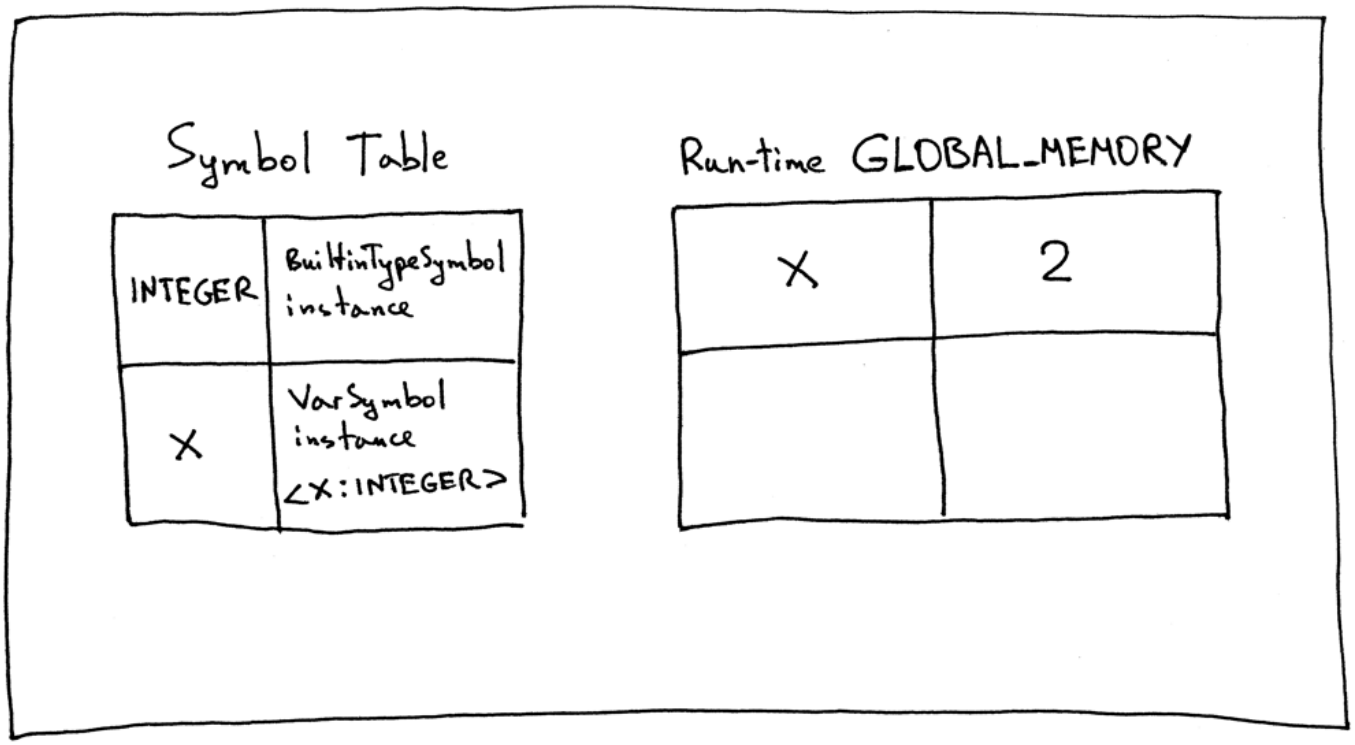
```
>>> from spi import Lexer, Parser, SymbolTableBuilder
>>> text = """
... PROGRAM NameError2;
... VAR
...    b : INTEGER;
...
... BEGIN
...    b := 1;
...    a := b + 2;
... END.
... """
>>> lexer = Lexer(text)
>>> parser = Parser(lexer)
>>> tree = parser.parse()
>>> symtab_builder = SymbolTableBuilder()
Define: INTEGER
Define: REAL
>>> symtab_builder.visit(tree)
Lookup: INTEGER
Define: <b:INTEGER>
Lookup: b
Lookup: a
Traceback (most recent call last):
  ...
  File "spi.py", line 665, in visit_Assign
    raise NameError(repr(var_name))
NameError: 'a'
```

Great, our new visitor caught this problem too!

I would like to emphasize the point that all those checks that our *SymbolTableBuilder* AST visitor makes are made before the run-time, so before our interpreter actually evaluates the source program. To drive the point home if we were to interpret the following program:

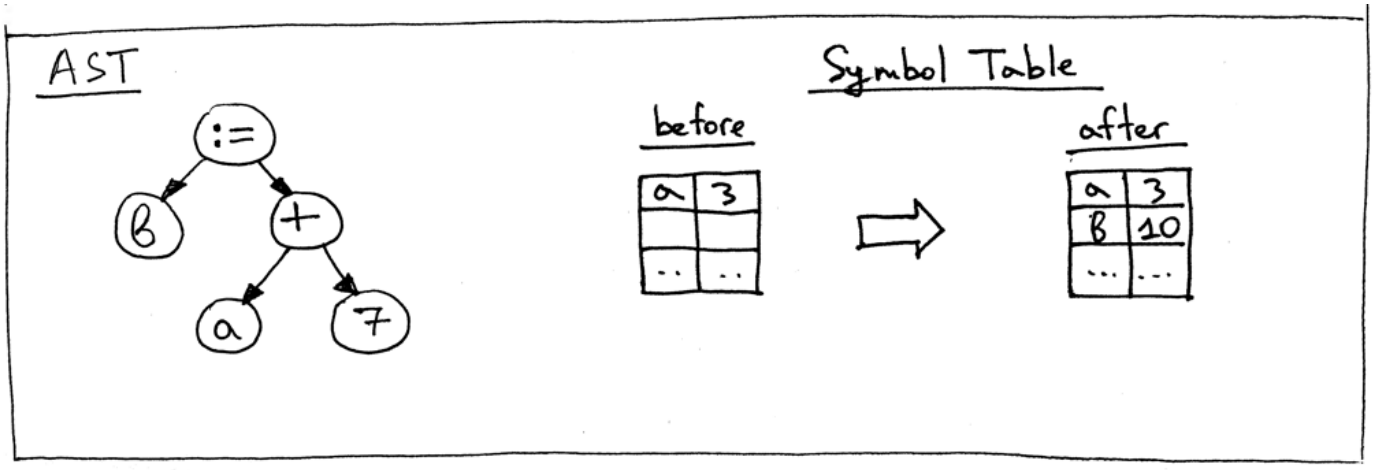
```
PROGRAM Part11;
VAR
   x : INTEGER;
BEGIN
   x := 2;
END.
```

The contents of the symbol table and the run-time GLOBAL_MEMORY right before the program exited would look something like this:



Do you see the difference? Can you see that the symbol table doesn't hold the value 2 for variable "x"? That's solely the interpreter's job now.

Remember the picture from Part 9 (/lsbasi-part9/) where the Symbol Table was used as global memory?



No more! We effectively got rid of the hack where symbol table did double duty as global memory.

Let's put it all together and test our new interpreter with the following program:

```
PROGRAM Part11;
VAR
   number : INTEGER;
   a, b   : INTEGER;
   y      : REAL;

BEGIN {Part11}
   number := 2;
   a := number ;
   b := 10 * a + 10 * number DIV 4;
   y := 20 / 7 + 3.14
END.  {Part11}
```

Save the program as part11.pas and fire up the interpreter:

```
$ python spi.py part11.pas
Define: INTEGER
Define: REAL
Lookup: INTEGER
Define: <number:INTEGER>
Lookup: INTEGER
Define: <a:INTEGER>
Lookup: INTEGER
Define: <b:INTEGER>
Lookup: REAL
Define: <y:REAL>
Lookup: number
Lookup: a
Lookup: number
Lookup: b
Lookup: a
Lookup: number
Lookup: y

Symbol Table contents:
Symbols: [INTEGER, REAL, <number:INTEGER>, <a:INTEGER>, <b:INTEGER>, <y:REAL>]

Run-time GLOBAL_MEMORY contents:
a = 2
b = 25
number = 2
y = 5.99714285714
```

I'd like to draw your attention again to the fact that the *Interpreter* class has nothing to do with building the symbol table and it relies on the *SymbolTableBuilder* to make sure that the variables in the source code are properly declared before they are used by the *Interpreter*.

Check your understanding

- What is a symbol?
- Why do we need to track symbols?
- What is a symbol table?
- What is the difference between defining a symbol and resolving/looking up the symbol?
- Given the following small Pascal program, what would be the contents of the symbol table, the global memory (the GLOBAL_MEMORY dictionary that is part of the *Interpreter*)?

```
PROGRAM Part11;
VAR
   x, y : INTEGER;
BEGIN
   x := 2;
   y := 3 + x;
END.
```

That's all for today. In the next article, I'll talk about scopes and we'll get our hands dirty with parsing nested procedures. Stay tuned and see you soon! And remember that no matter what, "Keep going!"

P.S. My explanation of the topic of symbols and symbol table management is heavily influenced by the book *Language Implementation Patterns (http://amzn.to/2cHsHT1)* by Terence Parr. It's a terrific book. I think it has the clearest explanation of the topic I've ever seen and it also covers class scopes, a subject that I'm not going to cover in the series because we will not be discussing object-oriented Pascal.

P.P.S.: If you can't wait and want to start digging into compilers, I highly recommend the freely available classic by Jack Crenshaw "Let's Build a Compiler." (http://compilers.iecc.com/crenshaw/)

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!