

Let's Build A Simple Interpreter. Part 17: Call Stack and Activation Records

(<https://ruslanspivak.com/lsbasi-part17/>)

Date 📅 Wed, August 28, 2019

"You may have to fight a battle more than once to win it." — Margaret Thatcher

In 1968 during the Mexico City Summer Olympics, a marathon runner named John Stephen Akhwari found himself thousands miles away from his home country of Tanzania, in East Africa. While running the marathon at the high altitude of Mexico City he got hit by other athletes jockeying for position and fell to the ground, badly wounding his knee and causing a dislocation. After receiving medical attention, instead of pulling out of the competition after such a bad injury, he stood up and continued the race.

Mamo Wolde of Ethiopia, at 2:20:26 into the race, crossed the finish line in first place. More than an hour later at 3:25:27, after the sun had set, Akhwari, hobbling, with a bloody leg and his bandages dangling and flapping in the wind, crossed the finish line, in last place.

When a small crowd saw Akhwari crossing the line, they cheered him in disbelief, and the few remaining reporters rushed onto the track to ask him why he continued to run the race with his injuries. His response went down in history: "My country did not send me 5,000 miles to start the race. They sent me 5,000 miles to finish the race."

This story has since inspired many athletes and non-athletes alike. You might be thinking at this point, "That's great, it's an inspiring story, but what does it have to do with me?" The main message for you and me is this: "Keep going!" This has been a long series spun over a long period of time and at times it may feel daunting to go along with it, but we're approaching an important milestone in the series, so we need to keep going.



Okay, let's get to it!

We have a couple of goals for today:

1. Implement a new memory system that can support programs, procedure calls, and function calls.
2. Replace the interpreter's current memory system, represented by the *GLOBAL_MEMORY* dictionary, with the new memory system.

Let's start by answering the following questions:

1. What is a memory system?
2. Why do we need a new memory system?
3. What does the new memory system look like?
4. Why would we want to replace the *GLOBAL_MEMORY* dictionary?

1. *What is a memory system?*

To put it simply, it is a system for storing and accessing data in memory. At the hardware level, it is the physical memory (RAM) where values are stored at particular physical addresses. At the interpreter level, because our interpreter stores values according to their variable names and not physical addresses, we represent memory with a dictionary that maps names to values. Here is a simple demonstration where we store the value of 7 by the variable name *y*, and then immediately access the value associated with the name *y*:

```
>>> GLOBAL_MEMORY = {}
>>>
>>> GLOBAL_MEMORY['y'] = 7    # store value by name
>>>
>>> GLOBAL_MEMORY['y']        # access value by name
7
>>>
```

We've been using this dictionary approach to represent global memory for a while now. We've been storing and accessing variables at the PROGRAM level (the global level) using the *GLOBAL_MEMORY* dictionary. Here are the parts of the interpreter concerned with the "memory" creation, handling assignments of values to variables in memory and accessing values by their names:

```
class Interpreter(NodeVisitor):
    def __init__(self, tree):
        self.tree = tree
        self.GLOBAL_MEMORY = {}

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

    def visit_Var(self, node):
        var_name = node.value
        var_value = self.GLOBAL_MEMORY.get(var_name)
        return var_value
```

Now that we've described how we currently represent memory in our interpreter, let's find out an answer to the next question.

2. *Why do we need a new memory system for our interpreter?*

It turns out that having just one dictionary to represent global memory is not enough to support procedure and function calls, including recursive calls.

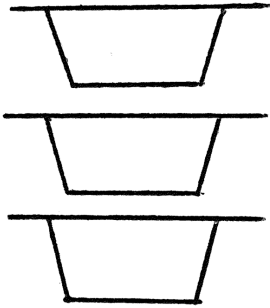
To support nested calls, and a special case of nested calls, recursive calls, we need multiple dictionaries to store information about each procedure and function invocation. And we need those dictionaries organized in a particular way. That's the reason we need a new memory system. Having this memory system in place is a stepping-stone for executing procedure calls, which we will implement in future articles.

3. *What does the new memory system look like?*

At its core, the new memory system is a stack data structure that holds dictionary-like objects as its elements. This stack is called the "**call stack**" because it's used to track what procedure/function call is being currently executed. The *call stack* is also known as the run-time stack, execution stack, program stack, or just "the stack". The dictionary-like objects that the *call stack* holds are called **activation records**. You may know them by another name: "stack frames", or just "frames".

Let's go into more detail about the *call stack* and *activation records*.

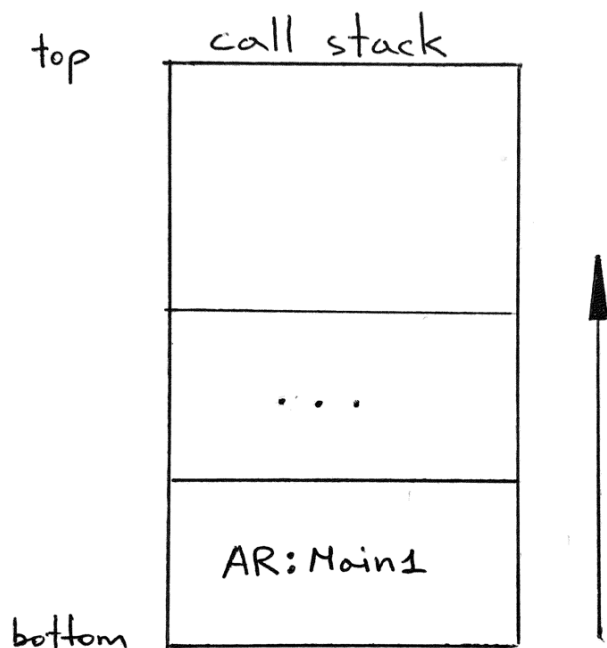
What is a **stack**? A **stack** is a data structure that is based on a “*last-in-first-out*” policy (*LIFO*), which means that the most recent item added to the stack is the first one that comes out. It’s like a collection of plates where you put (“push”) a plate on the top of the plate stack and, if you need to take a plate, you take one off the top of the plate stack (you “pop” the plate):



Our stack implementation will have the following methods:

- *push* (to push an item onto the stack)
- *pop* (to pop an item off the stack)
- *peek* (to return an item at the top of the stack without removing it)

And by our convention our stack will be growing upwards:



How would we implement a stack in code? A very basic implementation could look like this:

```

class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

```

That's pretty much how our call stack implementation will look as well. We'll change some variable names to reflect the fact that the *call stack* will store *activation records* and add a `__str__()` method to print the contents of the stack:

```

class CallStack:
    def __init__(self):
        self._records = []

    def push(self, ar):
        self._records.append(ar)

    def pop(self):
        return self._records.pop()

    def peek(self):
        return self._records[-1]

    def __str__(self):
        s = '\n'.join(repr(ar) for ar in reversed(self._records))
        s = f'CALL STACK\n{s}\n'
        return s

    def __repr__(self):
        return self.__str__()

```

The `__str__()` method generates a string representation of the contents of the *call stack* by iterating over *activation records* in reverse order and concatenating a string representation of each record to produce the final result. The `__str__()` method prints the contents in the reverse order so that the standard output shows our stack growing up.

Now, what is an **activation record**? For our purposes, an *activation record* 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 invocation, for example, will contain the current values of its formal parameters and its local variables.

Let's take a look at how we will represent *activation records* in code:

```

class ARType(Enum):
    PROGRAM = 'PROGRAM'

class ActivationRecord:
    def __init__(self, name, type, nesting_level):
        self.name = name
        self.type = type
        self.nesting_level = nesting_level
        self.members = {}

    def __setitem__(self, key, value):
        self.members[key] = value

    def __getitem__(self, key):
        return self.members[key]

    def get(self, key):
        return self.members.get(key)

    def __str__(self):
        lines = [
            '{level}: {type} {name}'.format(
                level=self.nesting_level,
                type=self.type.value,
                name=self.name,
            )
        ]
        for name, val in self.members.items():
            lines.append(f'    {name:<20}: {val}')

        s = '\n'.join(lines)
        return s

    def __repr__(self):
        return self.__str__()

```

There are a few things worth mentioning:

a. The *ActivationRecord* class constructor takes three parameters:

- the *name* of the activation record (AR for short); we'll use a program name as well as a procedure/function name as the name for the corresponding AR
- the *type* of the activation record (for example, PROGRAM); these are defined in a separate enumeration class called *ARType* (*activation record type*)
- the *nesting_level* of the activation record; the nesting level of an AR corresponds to the scope level of the respective procedure or function declaration plus one; the nesting level will always be set to 1 for programs, which you'll see shortly

b. The *members* dictionary represents memory that will be used for keeping information about a particular invocation of a routine. We'll cover this in more detail in the next article

- c. The *ActivationRecord* class implements special `__setitem__()` and `__getitem__()` methods to give activation record objects a dictionary-like interface for storing key-value pairs and for accessing values by keys: `ar['x'] = 7` and `ar['x']`
- d. The `get()` method is another way to get a value by key, but instead of raising an exception, the method will return *None* if the key doesn't exist in the *members* dictionary yet.
- e. The `__str__()` method returns a string representation of the contents of an activation record

Let's see the call stack and activation records in action using a Python shell:

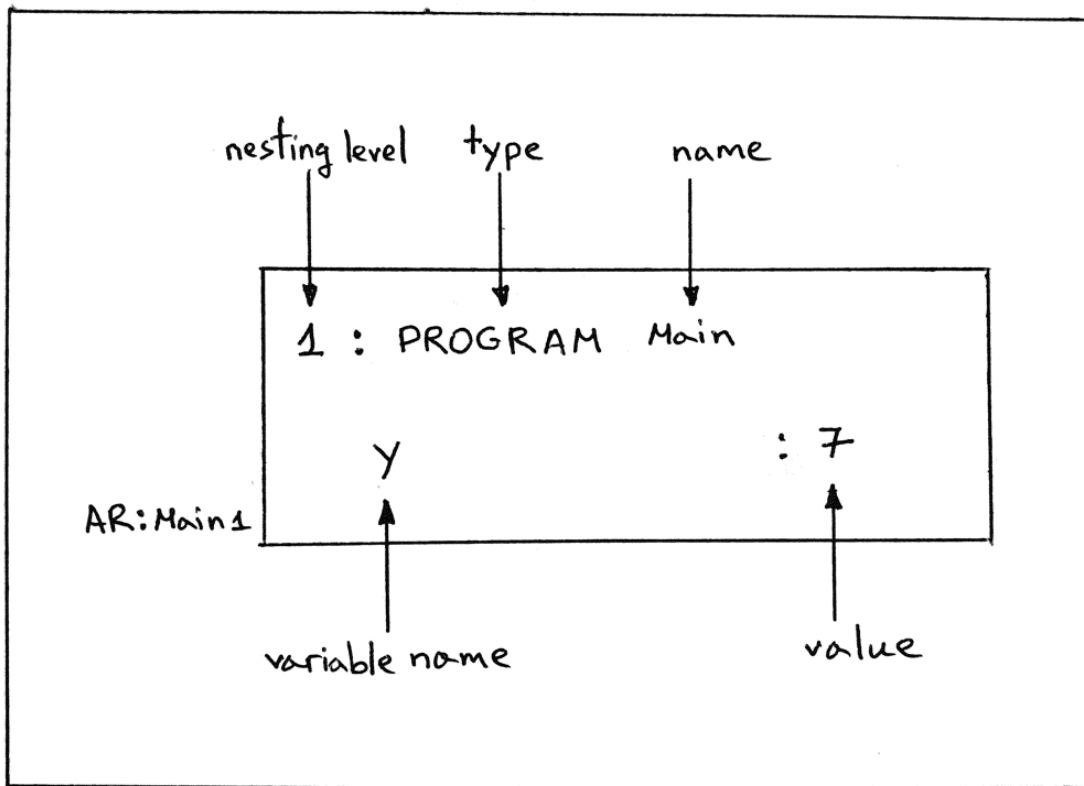
```
>>> from spi import CallStack, ActivationRecord, ARTYPE
>>> stack = CallStack()
>>> stack
CALL STACK

>>> ar = ActivationRecord(name='Main', type=ARTYPE.PROGRAM, nesting_level=1)
>>>
>>> ar
1: PROGRAM Main
>>>
>>> ar['y'] = 7
>>>
>>> ar
1: PROGRAM Main
   y                : 7
>>>
>>> stack
CALL STACK

>>> stack.push(ar)
>>>
>>> stack
CALL STACK
1: PROGRAM Main
   y                : 7

>>>
```

In the picture below, you can see the description of the contents of the activation record from the interactive session above:



AR:Main1 denotes an activation record for the program named *Main* at nesting level 1.

Now that we've covered the new memory system, let's answer the following question.

4. Why would we want to replace the *GLOBAL_MEMORY* dictionary with the *call stack*?

The reason is to simplify our implementation and to have unified access to global variables defined at the *PROGRAM* level as well as to procedure and function parameters and their local variables.

In the next article we'll see how it all fits together, but for now let's get to the *Interpreter* class changes where we put the *call stack* and *activation records* described earlier to good use.

Here are all the interpreter changes we're going to make today:

1. Replace the *GLOBAL_MEMORY* dictionary with the *call stack*
2. Update the *visit_Program* method to use the *call stack* to push and pop an *activation record* that will hold the values of global variables
3. Update the *visit_Assign* method to store a key-value pair in the activation record at the top of the *call stack*
4. Update the *visit_Var* method to access a value by its name from the activation record at the top of the *call stack*
5. Add a *log* method and update the *visit_Program* method to use it to print the contents of the *call stack* when interpreting a program

Let's get started, shall we?

1. First things first, let's replace the *GLOBAL_MEMORY* dictionary with our *call stack* implementation. All we need to do is change the *Interpreter* constructor from this:

```
class Interpreter(NodeVisitor):
    def __init__(self, tree):
        self.tree = tree
        self.GLOBAL_MEMORY = {}
```

to this:

```
class Interpreter(NodeVisitor):
    def __init__(self, tree):
        self.tree = tree
        self.call_stack = CallStack()
```

2. Now, let's update the *visit_Program* method:

Old code:

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

New code:

```
def visit_Program(self, node):
    program_name = node.name

    ar = ActivationRecord(
        name=program_name,
        type=ARType.PROGRAM,
        nesting_level=1,
    )
    self.call_stack.push(ar)

    self.visit(node.block)

    self.call_stack.pop()
```

Let's unpack what's going on in the updated method above:

- First, we create an activation record, giving it the name of the program, the PROGRAM type, and the nesting level 1
- Then we push the activation record onto the call stack; we do this before anything else so that the rest of the interpreter can use the call stack with the single activation record at the top of the stack to store and access global variables
- Then we evaluate the body of the program as usual. Again, as our interpreter evaluates the body of the program, it uses the activation record at the top of the call stack to store and access global variables

- Next, right before exiting the *visit_Program* method, we pop the activation record off the call stack; we don't need it anymore because at this point the execution of the program by the interpreter is over and we can safely discard the activation record that is no longer used

3. Up next, let's update the *visit_Assign* method to store a key-value pair in the activation record at the top of the call stack:

Old code:

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

New code:

```
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
```

In the code above we use the *peek()* method to get the activation record at the top of the stack (the one that was pushed onto the stack by the *visit_Program* method) and then use the record to store the value *var_value* using *var_name* as a key.

4. Next, let's update the *visit_Var* method to access a value by its name from the activation record at the top of the call stack:

Old code:

```
def visit_Var(self, node):
    var_name = node.value
    var_value = self.GLOBAL_MEMORY.get(var_name)
    return var_value
```

New code:

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

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

    return var_value
```

Again as you can see, we use the *peek()* method to get the top (and only) activation record - the one that was pushed onto the stack by the *visit_Program* method to hold all the global variables and their values - and then get a value associated with the *var_name* key.

5. And the last change in the *Interpreter* class that we're going to make is to add a *log* method and use the *log* method to print the contents of the call stack when the interpreter evaluates a program:

```
def log(self, msg):
    if _SHOULD_LOG_STACK:
        print(msg)

def visit_Program(self, node):
    program_name = node.name
    self.log(f'ENTER: PROGRAM {program_name}')

    ar = ActivationRecord(
        name=program_name,
        type=ARType.PROGRAM,
        nesting_level=1,
    )
    self.call_stack.push(ar)

    self.log(str(self.call_stack))

    self.visit(node.block)

    self.log(f'LEAVE: PROGRAM {program_name}')
    self.log(str(self.call_stack))

    self.call_stack.pop()
```

The messages will be logged only if the global variable `_SHOULD_LOG_STACK` is set to true. The variable's value will be controlled by the “—stack” command line option. First, let's update the main function and add the “—stack” command line option to turn the logging of the call stack contents on and off:

```
def main():
    parser = argparse.ArgumentParser(
        description='SPI - Simple Pascal Interpreter'
    )
    parser.add_argument('inputfile', help='Pascal source file')
    parser.add_argument(
        '--scope',
        help='Print scope information',
        action='store_true',
    )
    parser.add_argument(
        '--stack',
        help='Print call stack',
        action='store_true',
    )
    args = parser.parse_args()

    global _SHOULD_LOG_SCOPE, _SHOULD_LOG_STACK

    _SHOULD_LOG_SCOPE, _SHOULD_LOG_STACK = args.scope, args.stack
```

Now, let's take our updated interpreter for a test drive. Download the interpreter from [GitHub](https://github.com/rspivak/lbasi/tree/master/part17) (<https://github.com/rspivak/lbasi/tree/master/part17>) and run it with the `-h` command line option to see available command line options:

```
$ python spi.py -h
usage: spi.py [-h] [--scope] [--stack] inputfile

SPI - Simple Pascal Interpreter

positional arguments:
  inputfile    Pascal source file

optional arguments:
  -h, --help  show this help message and exit
  --scope     Print scope information
  --stack     Print call stack
```

Download the following sample program from [GitHub](https://github.com/rspivak/lbasi/tree/master/part17) (<https://github.com/rspivak/lbasi/tree/master/part17>) or save it to file `part17.pas`

```
program Main;
var x, y : integer;
begin { Main }
  y := 7;
  x := (y + 3) * 3;
end. { Main }
```

Run the interpreter with the `part17.pas` file as its input file and the “`--stack`” command line option to see the contents of the call stack as the interpreter executes the source program:

```
$ python spi.py part17.pas --stack
ENTER: PROGRAM Main
CALL STACK
1: PROGRAM Main

LEAVE: PROGRAM Main
CALL STACK
1: PROGRAM Main
  y          : 7
  x          : 30
```

Mission accomplished! We have implemented a new memory system that can support programs, procedure calls, and function calls. And we've replaced the interpreter's current memory system, represented by the `GLOBAL_MEMORY` dictionary, with the new system based on the call stack and activation records.