PyJ∀R

A language that was like Java, but wandered off toward Python

Group 32

John Abbott

Richa Srivastava

Vidhi Trivedi

Yousuf Hussain

Introduction

PyJar is a **dynamically typed language** with non local scopes for functions (but not loops): similar to Python.
It is modelled as a procedural language and supports scripting. It has static scoping and function arguments are passed by value. Functions are not first class objects.
Operations supported:

- Loops (while loop)
- Conditional expressions (if-else if-else)
- I/O on the console
- Functions (but not nested functions)
- Recursion
- Built in stack

Tools used

- Compiler:
  - Java - based.
  - Lexer and Parser - Built using Anltr v4.5.3
  - Grammar - Done through a .g4 file on Antlr.
  - Intermediate code generated through the ParseTreeWalker class, by creating and running a Listener file.
- Runtime / Interpreter:
  - Python - based
  - Runs the intermediate code file generated by the compiler part.
  - Prints output on the console

How to compile and run PyJar programs

- You need to have Java 1.7 and Python 2.
- Download and extract PyJar.zip from this github repo
- Create a new .txt file and write a program in PyJar (lets call it pgm.txt).
- Go to command line interface.
- Run: *java -jar PyJar.jar pgm.txt*
- You'll notice that there are 2 new files in the folder:
  - intermediate.pyj - The intermediate code generated
  - parseTree.pt - The parse tree for reference
- Run: python Interpreter.py
- You'll be prompted to enter a file name: Type *intermediate.pyj*
- The output of your program should be on the screen right about now.

PyJar Data types

PyJar has three data types:

- Boolean : BOOL - (True | False)
  - For assigning a true or false value to a variable
  - Syntax : x = True
- Integer : INT
  - For signed integer types
  - Syntax : x = 5
- Stacks are implemented as a built in data structure.
  - Syntax : stack stack_name

Declarations and Assignments

- Since PyJar is a dynamically typed language, we do not need to declare a variable before using it.
- The syntax uses block structure that is similar to Java. However, there is no need to write semicolon ‘ ; ’ at the end of each line.
- Also, there are no strict indentation rules that need to be followed in our language because whitespace is ignored.
- Programs can be written as a script.
- The language also supports complex assignments, evaluates arithmetic and boolean expressions.
  - Eg. Syntax : **x = y or (False and True or z)**

    : **x = (3 + 4) * 5 +3-2+(5/4)**

Some Intermediate Code Operations

- READ - gets input from user and pushes on to the stack
- STORE varname - pops a value from the stack and store it in variable varname
- PUSH varname - pushes variable varname on to the stack
- TESTFGOTO line_no - pops from stack, if popped value is False, sends execution to line_no
- TESTTGOTO line_no - pops from stack, if popped value is True, sends execution to line_no
- PRINT - pops from stack and prints on to the console
- DIVIDE, MULTIPLY, ADD, SUBTRACT - pop from the stack twice, perform the operation on the 2 popped values

More Intermediate Code Operations..

- GREATER, LESSER, EQUALS, GREATEREQUAL, LESSEREQUAL - Operations for the INT data type pop from the stack twice, perform the operation on the 2 popped values, and push the result (True or False) on to the stack.
- AND, OR, EQUALS - Operations for the BOOL data type.
- RET - returns the element at the top of the stack
- CALL funcname - Calls the function funcname; pops the number of parameters that funcname's signature has from the stack
- END - do nothing, designates the end of the program.

PyJar operators

PyJar includes integer operators and boolean operators:

`MULOP : ('*' | '/' | '%');` - Multpily, Divide, and Modular Division

`ADDOP : ('+' | '-') ;` - Addition and Subtraction

`INTCOMP : ('>' | '<' | '==' | '<=' | '>=');` - Greater than, Lesser than, Equals, Lesser or equal to, and Greater or equal to.

`BOOLAND : 'and' ;` - Boolean AND

`BOOLOR : 'or' ;` - Boolean OR

`BOOLCOMP : 'is' ;` - Boolean EQUALS

Branching

Branching is handled by an if/elseif/else statement:

```
ifelse : (ifStatement)(elseIfStatement)* (elseStatement)?;

ifStatement : prefixIf prefixContext;

prefixIf : 'if' '(' (boolCompare | integerCompare) ')'  ;

elseIfStatement : prefixElseIf prefixContext;

prefixElseIf : 'else if' '(' (boolCompare | integerCompare) ')' ;

elseStatement : prefixElse prefixContext;

prefixElse : 'else' ;

prefixContext : '{' context '}';
```

Looping

PyJar implements a while loop as its only built in looping mechanism

```
whileLoop : whilePrefix '{' context '}' ;
```

`whilePrefix : 'while' '(' (boolCompare | integerCompare) ')';` - As you might guess, the boolCompare and integerCompare are both comparison functions.

Functions

Some important features:

- Functions need to be defined before they are called.
- For functions, arguments and return statement are optional.
- Data types are not required in the argument list.
- Functions are preceded by the keyword ‘func’.

Function Syntax:

```
func function_name(argument){

function_body

}
```

```
call function_name(parameters)
```

Example 1 - nth Fibonacci no (iterative)

High level code

```
n = read
a = 0
b = 1
i = 0
if(n == 1){
        print 0
} else if(n == 2){
        print 1
} else{
        while(i < n - 2){
                c = a + b
                a = b
                b = c
                i = i + 1
        }
        print c
}
```

Intermediate code

```
READ
STORE n
PUSH 0
STORE a
PUSH 1
STORE b
PUSH 0
STORE i
PUSH n
PUSH 1
EQUALS
TESTFGOTO 17
PUSH 0
PRINT
PUSH True
TESTTGOTO 47
PUSH n
PUSH 2
EQUALS
TESTFGOTO 25
PUSH 1
PRINT
PUSH True
TESTTGOTO 47
PUSH i
PUSH n
PUSH 2
SUBTRACT
LESSER
TESTFGOTO 45
PUSH a
PUSH b
ADD
STORE c
PUSH b
STORE a
PUSH c
STORE b
PUSH i
PUSH 1
ADD
STORE i
PUSH True
TESTTGOTO 25
PUSH c
PRINT
END
```

Example 2: nth Fibonacci no. (recursive)

High level code

```
func fibo(n){
        if(n==1){
                return 0
        } else if(n==2){
                return 1
        }
        f1 = fibo(n-1)
        f2 = fibo(n-2)
        fsum = f1 + f2
        return fsum
}
x = read
print fibo(x)
```

Intermediate code

```
FUNC fibo
STORE n
PUSH n
PUSH 1
EQUALS
TESTFGOTO 10
RET 0
PUSH True
TESTTGOTO 17
PUSH n
PUSH 2
EQUALS
TESTFGOTO 17
RET 1
PUSH True
TESTTGOTO 17
PUSH n
PUSH 1
SUBTRACT
```

```
CALL fibo
STORE f1
PUSH n
PUSH 2
SUBTRACT
CALL fibo
STORE f2
PUSH f1
PUSH f2
ADD
STORE fsum
RET fsum
ENDFUNC
READ
STORE x
PUSH x
CALL fibo
PRINT
END
```

Example 3: Stack

High level code

```
stack s1
s1.push(3)
s1.push(4)
s1.push(5)
s1.push(True)
a=s1.pop()
print a
print s1.pop()
cond = s1.isEmpty()
while(cond is False){
        print s1.pop()
        cond = s1.
isEmpty()
}
```

Intermediate code

```
STACK s1
PUSH 3
STACKPUSH s1
PUSH 4
STACKPUSH s1
PUSH 5
STACKPUSH s1
PUSH z
STACKPUSH s1
PUSH True
STACKPUSH s1
STACKPOP s1
STORE a
PUSH a
PRINT
```

```
STACKPOP s1
PRINT
STACKISEMPTY s1
STORE cond
PUSH cond
PUSH False
EQUALS
TESTFGOTO 30
STACKPOP s1
PRINT
STACKISEMPTY s1
STORE cond
PUSH True
TESTTGOTO 20
END
```

Output

```
True
5
4
3
```