PyLit

Language Reference Manual

Ryan LoPrete (rjl2172)

October 15, 2018

I. Introduction

PyLit aims to provide a simple, lightweight implementation of the common scripting language, Python. The language has many of the basic data structures that are native to Python, such as dictionaries, strings, and lists – which allow the user to define functions and generate algorithms. The data types support common operations such as retrieving the first element in a list, or pushing a value to a dictionary. Indentation to signify code blocks is replaced by brackets, as used in most other languages. The compiler for PyLit is coded in Ocaml.

II. Types

Types are not explicitly declared. Rather, PyLit infers types at runtime similar to Python. There are four primitive types and two complex types, as specified below.

string: String literals contain text consisting of ASCII characters between double quotes.

```
let str = ([' '-'!' '#'-'[' ']'-'~']| '\\' ['\\' '"' 'n' 't'
'r'])*
```

int: Integer literals are any whole number without a decimal or fractional portion.

```
let digit = ['0'-'9']
```

float: Floats are used to represent decimal numbers or those containing exponents. For example, 0.5 and 1E7 are considered floats. The matching rule for floats is given below.

```
let exponent = ['e' 'E'] ['+' '-']? ['0'-'9'] (['0'-'9'] | '_')*
let float =
   ('-'? ['0'-'9'] (['0'-'9'] | '_')*)? (('.' (['0'-'9'] | '_')*
(exponent)?) | exponent)
```

bool: Booleans represent values which are either logical true / false. These are reserved keywords

list: List data structures can store multiple values of the same type. They are denoted by square brackets surrounding comma separated values. Lists are useful for performing iterations or keeping track of sequenced data.

dict: Dictionaries are useful for storing key / value pairs. The keys must be of the same type, and the values must be of the same type. The key and value types do not need to match.

Examples:

```
var = 1          # int
var = 1.5        # float
var = "PyLit"    # string
var = true       # bool
var = [1,2,3,4]  # list(int)
var = {"day1": "Monday", "day2: "Tuesday"}  # dict(string, string)
```

III. Lexical Conventions

Statement Termination

Statements should be terminated with the semicolon ‘;’. This character tells PyLit to evaluate the statement preceding that character.

Comments

Single line comments are allowed in PyLit and are denoted by the ‘#’ character.

```
# This sentence is a comment

x = 10; # Comments can also appear after statements
```

Whitespace

Spaces, horizontal tab, carriage return, and new line are ignored by PyLit. Tokens can be separated by any of the above characters without affecting code execution.

```
whitespace = [' ' '\t' '\r' '\n']
```

Identifiers

Identifiers are used to represent variables. An identifier can consist of the below characters, but cannot be one of PyLit’s reserved keywords.

```
id = ['a'-'z'] (['a'-'z' 'A'-'Z'] | ['0'-'9'] | '_')*
```

Valid Identifiers	**Invalid Identifiers**
`var`	`Var (no uppercase)`
`first_name`	`for (reserved keyword)`
`option_1`	`option-1 (invalid character)`

Keywords

PyLit consists of a number of keywords that hold special meaning. These are reserved and cannot be used as variable declarations.

```
if, elif, else, for, true, false, in, not, and, or, def, return,
print, none
```

Separators

The following list contains characters that separate tokens, other than whitespace. They are used in function declarations and list creation, for example.

```
'('          { LP }

')'          { RP }

'['          { LSB }

']'          { RSB }

'{'          { LBRACE }

'}'          { RBRACE }

','          { COMMA }

';'          { SEMICOLON }

':'          { COLON }
```

Operators

PyLit offers unary and binary operators. The two unary operators are negation via the ‘-‘ token, as well as ‘not’ for logical negation.

```
'+'          { PLUS }

'-'          { MINUS }

'*'          { MULTIPLY }

'/'          { DIVIDE }

'%'          { MOD }

'<'          { LT }

'>'          { GT }

'='          { ASSIGNMENT }

'!'          { NOT }

"<="         { LTE }

">="         { GTE }

"=="         { EQUALS }

"!="         { NEQUAL }
```

Arithmetic Operations

Basic arithmetic can be calculated on floats and integers. Two operands must have matching types or an error will be thrown. The five arithmetic operations as well as unary negation are shown below.

Operation	**Example**
Addition	`1 + 1;   # 2`
Subtraction	`5 - 4;   # 1`
Multiplication	`3 * 2;   # 6`
Division	`10 / 5;  # 2`
Modulus	`10 % 2;  # 0`
Unary Minus	`-5;      # -5`

Relational Operators

Relational operations can be performed on types float, int, or string. The operands must be of the same type. For comparison of string types, alphabetical ordering is used and returns a Boolean type (true or false). Variables can also be compared if the identifiers specify similar types. Examples of the relational operators are shown below.

Less than:

```
10 < 20;               # evaluates to true
```

Greater than:

```
20.0 > 10.0;           # evaluates to true
```

Less than or equal:

```
“PyLit” >= “PyLit”;    # evaluates to true
```

Greater than or equal:

```
“PyLit” > “PyLit”;     # evaluates to false
```

Variable comparison:

```
a = 5;
b = 3;
b > a;                 # evaluates to false
```

String Operators

PyLit offers one string operations for concatenation, given by the “+” token. Both operands must be string literals or variables that identify string literals.

```
“PyLit” + “ is great”; # evaluates to ”PyLit is great”
```

Comparison Operators

Comparison operators include equals “==” amd not equal “!=”. Any primitive operands can be compared but they must be of the same type. Strings are equal if they contain the same alphabetical characters in the same order, while lists and dictionaries are equal if they contain the same values or key/value pairs in the same order.

```
      “PyLit” == “PyLit”    # evaluates to true
      5.0 == 5.0            # evaluates to true
      5 == 5.0              # evaluates to false
      [1,2,3] == [1,2,3]    # evaluates to true
      {“day”: “Monday”} == {“day”: ”Tuesday”}     # evaluates to false
      {“day”: “Monday”} != {“day”: ”Monday”}      # evaluates to false
```

Logical Operators

Three logical operators are offered in PyLit: logical “and”, logical “or”, and logical “not”. Any expression that evaluates to a Boolean literal can use these operations. The returned type is also a boolean. Logical “and” / “or” take two operands, while “not” is unary.

```
      true and true    # evaluates to true
      true and false   # evaluates to false
      true or false    # evaluates to true
      false or false   # evaluates to false
      not true         # evaluates to false
```

List Operators

PyLit offers one list operation to add new values to a list. This is given by the cons symbol “::”. A value can be added to either the beginning or end of a list depending on where the operator is used.

```
      [1,2] :: [3]     # evaluates to [1,2,3]
      [1] :: [2,3]     # evaluates to [1,2,3]
```

Dictionaries

In PyLit, once dictionaries are created, they cannot be changed. Keys / values can be accessed with the “keys” / “values” operations.

```
      dict = {“day1”: “Monday”, “day2”:”Tuesday”}
      keys dict        # evaluates to [“day1”, “day2”]
      values dict      # evaluates to [“Monday”, “Tuesday”]
```

Operator Precedence

Precedence	Operator
Highest	`-  (unary)`
	`* , / , %`
	`+ , -`
	`<=, >=, <, >, ==, !=`
	`Not`
	`And`
Lowest	`Or`

All operations are left associative in PyLit except for assignment, which is right associative.

Functions

PyLit functions are similar to that of Python except for some structural changes to account for whitespace insensitivity. Instead of a colon, brackets are used to define where the function body begins, similar to the C programming language. Functions must be declared using the “def” keyword. Two example functions are given below.

```
def find_max (MyArray) {
max = 0;
for x in MyArray {
    if (x > max) {
        max = x;
        }
    }
return max
}

find_max([1,2,3,4])   # evaluates to 4
```

```
def gcd(x, y) {
    if (y>x) {
        return gcd(y,x)
    }
    else (x%y) == 0 {
        return y
    }
    return gcd (y, x%y)
}

gcd(10,2)  #evaluates to 2
```