Language Interpreter

Draft V0.1

**Change History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Revised By** | **Brief Outline of Changes** |
| 0.1 | 10/03/16 | Norwin T. Avila | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[1 Overview 3](#_Toc463298341)

[2 Top-down Design Concept 4](#_Toc463298342)

# Specification

**Variables**

Use single letter, case insensitive. Interpreter will only have up to 26 variables which is A-Z.

e.g.

A = 100;

**Functions**

Supports only 3 functions as follows.

* **print**

Syntax:

print “Hello, world”;

print <variable>;

print <variable>, <string>, ..., etc;

* **for**

Syntax:

for (i=<initial value> to <limit>) {

<expression>;

<statement>;

}

Example:

for (i=0 to 5) {

print “value of i: “, i;

}

* **if**

Syntax:

if (<expression>) {

<expression>;

<statement>;

}

If statement only supports only 3 conditional operator (=, <, >).

**Error Checking**

The language interpreter will not support robust error checking, in order to keep the code small, so if syntax error is encountered the behavior is undefined and may cause unresponsiveness.

# Example Script

print “This script prints hello world 10 times”;

for (i=1 to 10) {

print i, “Hello world!”;

}

# Flow

1. Tokenize the script in order to extract the following parts.
2. Function
3. Quoted string
4. Variable
5. Expression
6. If the script will encounter a function then execute the function. Only 3 functions are supported namely (if, for print).
7. If expression is encountered evaluate the expression (i.e. variable declaration, conditional, and mathematical expressions).
8. The interpreter will only contain 2 data structure (i.e. Stack and Table for variables)
9. The interpreter will only contain a single letter case insensitive variable (i.e. A-Z)

# Topdown Design

1. First iteration of top-down design.

Interpreter

Tokenizer

ExpressionParser

Functions

Stack

Variables

A **Tokenizer** module will parse the script and will return the necessary tokens that are required by the interpreter. Based on Section 2: Example Script, the tokens are as follows.

print

“This script prints hello world 10 times”

;

for

(

i

=

0

to

10

)

{

print

“Hello world!”

;

}

Tokenizer module will contain a **getToken()** function to break the script in to tokens as described above.

The moment we have extracted the token via **getToken()** function, that token will be removed in the source script but there are cases that we need to extract the token just for purpose of checking the next token for some logic and then put it back to the source script. In order to achieve this we need to create another function called **putbackToken().**

In order to properly manage the token we need to classify according to its type such as the following.

Delimeter 🡪 \*/+-=;,(){}

Number 🡪 0-9

PRINT 🡪 print function

FOR 🡪 for function

IF 🡪 if function

VARIABLE 🡪 for variables [A-Za-z] (case insensitive).

We can achieve this be creating another function called **getTokenType()**.

Tokenizer

putbackToken()

getToken()

getTokenType()

Next is to define the **ExpressionParser** module. Let’s look at some example expression so that we can define what are the necessary functions that composed our ExpressionParser module.

A = 10 + 2 \* 3 + 4 / 2

In Mathematics, the definition of an expression is as follows.

Expression 🡺 Term[+Term][-Term]

Term 🡺 Factor[\*Factor][/Factor]

Factor 🡺 variable, numer, Expression

Example:

A + 3\*C

In the above example, there are 2 terms: A and 3\*C. It has 3 factors: A, 3, C, consisting of two variables and 1 number.

Another example:

A\*(8-B)

In the above example, there is only one term. It has two factors A and (8-B), consisting of one variable and one expression.

In breaking down the expression parser using top-down design we again based on the information we have defined in our expression. So the following are being defined here.

processAssignment() -> evaluateExpression()

processTerm() 🡪 compute(left + or - right)

processFactor() 🡪 compute (left \* or / right)

ExpressionParser

processTerm()

processFactor()

processPrimitive()

evaluateExpression()

Tokenizer

processPrimitive() 🡪 translate numbers or variable.

Next is to define our **Functions** module. Our functions module will do nothing more than execute the 3 supported functions(print, if for) based on syntax defined in section 2.2. So it is clear that the top-down design for Functions module would be as follows.

Functions

processPrint()

processIf()

processFor() / processForNext()

Tokenizer

To summarize the overall top-down design of our Language Interpreter we will combine or conquer each functionality to form a single entity.

Interpreter

processTerm()

processFactor()

processPrimitive()

evaluateExpression()

Variables

ExpressionParser

processPrint()

processFor()

processIf()

processForNex()

Stack

Functions

popStack()

pushStack

Tokenizer

getToken()

putbackToken()

getTokenType()

Source Code:

