FINAL PROJECT – PROGRAMMING LANGUAGES

LANGUAGE USER GUIDE

Submitted by:

Daniel Cohen

Shani Haker

Michal Ben Haim

TABLE OF CONTENTS

1. Interpreter

1.1 Interactive Mode

Interactive Mode allows the interpreter to execute code line by line. It's immediately invoked when running 'shell.py' file.

```
basic > command
```

Possible commands and syntax are further explained in later paragraphs.

1.2 Script Mode

Script mode is designated for reading multiple lines of code through a file instead of writing them one by one. Nonetheless, the interpreter will print the result of each command in that file separately.

To invoke script mode, run 'shell.py' file and enter the following command:

```
basic > -F filename.lambda
```

- Replace 'filename' with any file of your liking, note that only files with '.lambda' suffix are legible to read and execute.

Upon completion, the interpreter will once more return to interactive mode allowing to either insert a one-line command or execute another file.

1.3 Termination

To exit the interpreter, insert the following command:

```
basic > QUIT
```

2. Defining Variables

Variables are memory stored information. In this language, variables can hold two forms of data – numerical or binary (True/False) information [Later on, we will introduce variables holding anonymous functions]. The following examples demonstrate simple assignation of those two variable types.

```
basic > VAR a = 5

basic > VAR b = True
```

The example above shows the basic syntax for declaring and initializing a variable.

```
VAR variable_name = value
```

A successful assignment will invoke the interpreter into returning the assigned value:

```
basic > VAR a = 5

5

basic > VAR b = True

True
```

3. Basic Functions on Numbers

The language allows performing all simple calculator functions; add, subtract, multiply, divide, modulo and power. Those functions are invoked through using their appropriate symbols. Moreover, the Interpreter allows performing them either directly through entering the numbers or through stored variables information.

Symbol	Action
+	Addition
-	Subtraction
*	Multiply
/	dividing two full numbers and without fractional part
%	Modulo; the remainder of a division
^	Power

➢ Direct Performance on Inserted Numbers

In this case, the numbers used aren't accessed through variables but rather inserted straightforwardly to the command line. Their values and eventual result will not be stored. The expected output will the result of the specified action.

```
basic > 5+5

10

basic > 8^2

64
```

- **Performance on Stored Values**

  In this case, the numerical values are accessed through program memory. Unless the result is assigned to a new variable – it won't be stored or possible to access in the future. The expected output will the result of the specified action.

```
basic > VAR a = 5

5

basic > a + 4

9

basic > VAR b = a + 4  # this result will be stored

9
```

3.1 Comparison Expressions

After introducing the rudimentary numerical functions, the language can assist in evaluating basic mathematical expressions:

Symbol	Action
==	Equal to
!=	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Similarly to the basic mathematical functions – the operations can be placed on numerical values either direct or stored; A combination of a basic numerical function inside an expression is also allowed (ref. example below). The interpreter will be invoked to return either 'True' or 'False' for each expression accordingly. Results can be stored for future access.

```
basic > 5 < 3 + 1

False

basic > VAR b = 1 == 1

True
```

4. Logical Expressions

The language allows performing logical operations on appropriate values. Only variables with assigned True or False values are legible for performing the following functions: '&&' (And), '||' (Or) and '!' (Not). Those operators will be higher in hierarchy, allowing a combination of numerous comparison expressions (introduced above) into a large logical expression. The interpreter will be invoked to return either 'True' or 'False' for each expression accordingly. Results can be stored for future access.

```
basic > VAR a = (5 > 3) && (0 != 2)
True
basic > 5 > 3 && 0 != 2  # 4.1
True
basic > VAR b = !((a) || 1 != 1) # 4.2
False
basic > !a  # 4.3
Invalid Syntax Error
```

As shown in the example above, for simple binary operation (as shown in 4.1) – there's no need for consecutive parentheses. However, when dealing with more complex logical expressions, or when with an integrated hierarchy, parentheses are required (as shown in 4.2).

Note: Not (!) is regarded as a unary operator and always requires consecutive parentheses (shown in 4.3).

5. Control Flow Tools

Control flow refers to the order in which a program's instructions are executed. It determines how the program decides which path to follow based on conditions, loops, and function calls. The language currently supports IF statements and Functions (ref. to paragraph 5.2) and implements *while* loops through recursive calls (ref to paragraph 5.2.1).

5.1 IF-statements

IF-statements are condition driven statements. To implement those statements, three keywords are used – 'IF', 'THEN', 'ELSE'.

The example below demonstrates the use of an IF command:

```
basic > IF 3 > 0 THEN VAR a = True ELSE 0

True
```

Thus, an IF keyword is followed by an expression that can evaluated to either be 'True' or 'False'. If the condition holds 'True', the code block reaches 'THEN' and executes the operation within that block, otherwise, it jumps to 'ELSE' and executes that segment. [NOTE: 'ELSE' is optional].

```
IF {comp_exp} THEN {operation} ELSE {operation}
```

The interpreter will reply with result of the 'IF' statement which won't be stored unless assigned to a variable first.

5.2 Defining Functions

Functions allow us reuse blocks of code upon calling them. In this segment, we will cover basic functions and definitions and actions.

For example, the following code block defines a simple function of adding two numerical arguments:

```
basic > FUN sum(a,b) -> a + b

<function sum>
```

To define a function, the following syntax needs to be implemented:

```
FUN func_name(args) -> {operation}
```

Operation can be mathematical expressions or function or logical expressions. Upon a successful definition, the interpreter will reply with a '<function func_name>'. When the function is called, the interpreter will then access the previously defined functions and operate with the inserted arguments [NOTE: multiple arguments must be divided with commas]. Similarly to other expression results, function results won't be stored unless stored to a variable.

```
basic > FUN sum(a,b) -> a + b

<function sum>

basic > sum(1,3)

4
```

5.2.1 Anonymous Functions

The language allows *anonymous functions,* or more broadly, functions without a name. However, to properly use those functions, they must be stored in a variable. The following example demonstrates the definition of an anonymous function:

```
basic > VAR mystery = FUN (a,b) -> a * b

<function <anonymous>>
```

To define an anonymous function, the following syntax needs to be implemented:

```
VAR func_name = FUN (args) -> {operation}
```

Upon successful assignment of an anonymous function, the interpreter will reply with '<function <anonymous>>'. Any future calls to the anonymous function must be through the assigned variable while passing the correct type and number of arguments.

Thus, calling the argument mentioned above will have the following code:

```
basic > mystery(3,5)

15
```

5.2.2 Recursive Functions

Recursive functions are functions that call themselves. To prevent an infinite loop of calls, a base condition is required. The syntax for defining recursive functions is similar to that of any regular function (see paragraph 5). The example below demonstrates a recursive function counting to a certain number.

```
basic > FUN increment_counter(counter) -> IF counter < 10 THEN
increment_counter(counter + 1) ELSE counter

<function increment_counter>
```

6. Errors

Error Type	Cause	Example
Invalid Syntax Error	Inappropriate use of keywords and/or command formatting	• a = 5 [missing 'VAR'] • FUN add(a,b) a+b [missing '->']
Expected Char Error	Missing symbols of defined operations	a & b [expected '&&']
Illegal Char Error	Using undefined symbols	'$' [undefined language symbol]
Runtime Error	• Attempting to divide by zero or modulo by zero • calling an illegal function	10 / 0 [divide by zero]

7. Reserved Keywords

'VAR', 'IF', 'THEN', 'ELSE', 'FUN', 'True', 'TRUE', 'FALSE', 'False', 'QUIT', '-F'