COMP251 - Assignment2
Deadline: Nov 6, 2018

Goal:

Well in this assignment you will design and implement your very first Compiler! It shows you the role of stacks in implementing compilers. I am sure you will enjoy doing this assignment.

Problem:

To make sure you enjoy this as much as possible, I have made it quite simple. A real compiler is more complicated than what we learn in this course. A compiler (like C++ or Java) takes a text file containing a number of statements in a particular language and produce an executable file (i.e. an equivalent program in machine language) as output.
Your compiler takes a file (program file) and executes it line by line, so it is more like an interpreter, not a real compiler. We will use a simple language in this assignment called ***Espresso language.***

Espresso Language description:

Statements: each line of a program in Espresso language contains a single statement. There are three types of statements in Espresso language:

1. **The assignment statement:** the syntax of an assignment statement is
   ```
   variable = infix-expr
   ```
   When an assignment statement is executed the `infix-expr` is evaluate and its value is assigned to the variable.
2. **The input statement:** the syntax of an input statement is
   ```
   read variable
   ```
   When an input statement is executed the program asks the user to enter an integer and assigns the entered value to the specified variable.
3. **The output statement:** the syntax of an output statement is
   ```
   print infix-expr
   ```
   When an output statement is executed the `infix-expr` is evaluate and its value is printed on the screen.

Variables: the variables in Espresso language are one letter characters: "a" to "z" and "A" to "Z". The language is case sensitive so "a" and "A" refer to two different variables. Also `Read` or `PRINT` are not valid keywords.

Infix-expr: the infix-expr's are used in assignment or print statements. An infix-expr is an algebraic expression in infix format. The operators in an infix-expr include: "+", "-", "*", "/" and "%". The operators are integer binary operators (there is no unary operator such as –x). The operands are either variables or positive integer constants. **You can assume that there is exactly one space between operands and operators (including parenthesis) in an infix expression**. **This means that your program will be tested for the programs that include statements with exactly one space between their operands and operators.** Below are some examples of valid infix-expr:

```
Y
X + 2 * y
( 0 – 45 )
( x – y ) * ( x + y )
G * ( ( m * n ) / ( r * r ) )
```

and here some invalid infix-expr's:

```
2y            //2y is not a variable.
- 45 * u + ( - Z ) //no unary operator is allowed.
( ( y + 7 ) * z   //unbalanced parenthesis.
Sum = J + I
//sum is not a valid variable (variables consist of  one letter only).
[ x + y ]         //invalid token ]
```

Here is a sample program:

```
read x
y = 3
z = x * x + y * y
print z
print ( z + x ) / 2
```

This is an expected output of executing your program:

```
Enter an integer number for variable x: 2
13
7
```

Your interpreter

Your interpreter should take the input program file as a command line argument. Given an input program file your program must read the statements one by one (each statement is in a line) and execute them. In case there is an error in a statement, the program should print the statement and the an appropriate message about the error, and then terminate the execution of the input program.

There are two possible types of errors:

1. **Syntax error:** if a statement is invalid (violates the rules of the Espresso language) it has a syntax error. For example if the infix expression is invalid (all examples we mentioned above), it is a syntax error, or if a variable name is invalid (such as x2), or you found an invalid operator or character (such as !, ^, &, ...). For these type of errors it's enough that your program prints "syntax error" along with the line number and the statement that contains the error.
2. **Undefined variable error:** If a statement tries to access the value of a variable that is not initialized before, the your program must print an undefined variable error message. For instance if the first statement of a program is

"`Z = x + 3`" your program should print:

```
Line 1. Z = x + 3
error: variable x is not defined.
```

The following shows some examples:

Sample program

```
read x
x = x ^ 2
```

```
print x
```

Expected output:

```
Enter an integer number for variable x: 25
Line 2. x = x ^ 2
Syntax error.
```

Sample program2

```
x = 24
read y
print x + ( -y )
```

Expected output:

```
Enter an integer number for variable y: 4
Line 3. print x + ( -y )
Syntax error.
```

Sample program3

```
read p
read q
d = p * p + q * q
print d
d = c * d
print d
```

Expected output:

```
Enter an integer number for variable p: 2
Enter an integer number for variable q: 4
20
Line 5. d = c * d
error: variable c is not defined.
```

Guideline:

Handling Variables[1]:

There are different ways to handle variables. Here is a suggestion: you can define a class `Variable` in your program to store the value for each variable. The class variable has two field member: value which store the integer value and `initialized` which is a boolean variable (like a flag) that indicates whether the variable has been initialized or not (defined before or not).

```
class Variable {
    private int value;
    private boolean initialized;

    public Variable(){
          initialized = false;
    }
    public void setValue(int v){
          this.value = v;
          this.initialized = true;
    }
    public int getValue() throws UndefinedVariableException {
          if (!initialized)
                throw new UndefinedVariableException();
          else
                return value;
    }
}
```

Note that `getValue()` throws an exception if we try to get the value of an un-initialized (undefined) variable.
Now you can create an array of variables as follows:

```
Variable [] variable_table;
for (int i = 0; i < 'z'; i++) {
     variable_table[ i ] = new Variable();
}
```

[1] You may use Java HashTable or HashMap if you like. I do not put any constraint on it. But as we have not discussed hash tables in class, I recommend you to use the Variable class.

To set a new variable with name `'F'` we can write:

```
variable_table['F'].setValue(10);
```

and to get the value of variable 'F' you can write:

```
char c = 'F'
try {
  int value = variable_table[c].getValue();
   System.out.println("variable " +c +" is: " + value);
} catch (UndefinedVariableException E) {
   //the variable has not defined yet. Print an error message and exit
}
```

Expressions:

There are different ways to evaluate expressions. You may:

1. Write a method to convert the infix expression to postfix and another method to evaluate the postfix expression. Both algorithms are explained in the lectures.
2. Write your own algorithm to evaluate infix expressions.

The advantage of the first method is that if there's any error in the infix expression it can be detected while you are converting the infix expression to postfix.

Handling errors:

There are two types of errors as we discussed but they may occur and be captured and handled in different steps.

- The first level of error checking is when we read each statement and check if which type of statement it is (there are three different types of statements in Espresso language, input, output or assignment). Errors captured here are syntax errors
- The second level of error checking can be done while you are converting the infix expression to postfix (syntax errors like invalid variable name, unbalanced parentheses, invalid operator, invalid token…)
- The third level of error checking can be done when you are evaluating the postfix expression (all syntax errors have been handled so far, so this would be undefined variable error).

Note1: Make sure that you keep the line number so that you can report the error message properly.

Note2: If you are not converting infix to postfix, the second and third steps will be done while you are evaluating the infix expression.

Input program file

The interpreter must get its input filename as command line argument. If you are using the eclipse compiler go to the Run menu and click on "Run Configuration" (if you are using Windows: then click on "Java Application" on the left bar). Select the "Arguments" tab and type the name of the input file in the Program arguments box. If you are using the command line java compiler, type the name of the input file after the main class. For instance if the input program is **`test.esp`** then type **`java ESPInterpreter test.esp`** in the linux prompt**.** This way the arg[0] value in the main method below will be the string **`"test.esp".`**

```
public static void main(String arg[]){
  //arg[0] is the first argument which should be the name of the input program.
      ...
}
```

Important Remarks

It's very important that you make sure your program compiles without any problem (you may get 0 if your program doesn't compile)

Submission

Create a directory called src and copy all your source files and any readme file in it, zip it and submit it to the blackboard.