CSE 222 HOMEWORK #4

Orhan Aksoy

09104302

TABLE OF CONTENTS

1. Problem Description

An interpreter needs to be developed that can execute small programs written in GITL (Gebze Institute of Technology Language). The program will be coded and saved in a file with a specific extension, and the interpreter will read and execute the code from the file.

1.1. Requirements

1.1.1. The GITL programs shall be read and executed from files with extension .GITL provided as the first argument to the application.

1.1.2. The GITL programs shall have a main function that should take an integer and return an integer.

1.1.3. Each line of the program shall include only one statement.

1.1.4. The variable declarations shall be done as below:

- `int a`
- `int myVar`

1.1.5. The variable declarations shall be able to exist anywhere in the program code.

1.1.6. The type of the variables shall be `int`.

1.1.7. The separator between the tokens in the code lines shall be a space character.

1.1.8. The arithmetic expression shall be defined as below:

- `a = a * 2`
- `b = c / ( 33 – 4 )`

1.1.9. The operators in the arithmetic expressions shall be =, -, +, *, / and)

1.1.10. The operator precedence shall be implemented as below:

1	(,)	Parenthesis
2	/, *	Division and multiplication
3	+, -	Addition and subtraction

1.1.11. Labels shall be defined as below:

- `Label lab1`
- `B = label l2`

1.1.12. For the purpose of printing variables and constants, the function ‘print’ shall be used. The first parameter shall be the output parameter. Second parameter shall not be used. Example usage is shown below:

- `call print a res`
- `call print 123 res`

1.1.13. For the purpose of reading from keyboard, the function ‘scan’ shall be used. The first parameter shall not be used. The second parameter shall not the input value from keyboard. An example usage is shown below:

- `call scan 0 b`

1.1.14. ‘`if`’ statement shall be used for conditional branching. If the first parameter after the ‘`if`’ statement is greater than 0, the program will go to the label provided as the second parameter after the ‘`if`’ statement. An example usage is shown below:

- `if a label 2`

1.1.15. ‘`return`’ statement shall be used for returning from the current function. The first parameter after the ‘`return`’ statement shall be the return value. Example usage is shown below:

- `return 1`
- `return a`

1.1.16. The comments within the code shall begin with ‘//’.

1.1.17. The syntax errors and runtime errors shall be reported by the application during execution. The reported error shall include the line number. Example error reports are shown below:

- Variable not declared
- Invalid expression

2. Analysis

Since the application is an interpreter, it will read only one line at a time and process it. The processing of each line can be divided into two phases: Parsing the input line, and interpretation and output generation.

The input to the parsing phase is the input line from the file. The output is an expression object representing this input string. The interpretation/output generation phase uses the expression from the previous phase and executes it using the current context.

Two kinds of errors are generated during the execution of each line: Syntax errors and runtime errors. However, since this application will act as an interpreter rather than a compiler, the lines will be processed when they are met. The consequence of this fact is that the compiler and runtime errors will both be caught during the program run.

During the interpretation of each line, the following tasks will be handled:

2.1. Parsing

The tokens in each line will be separated by a space character. So, Java class 'StringTokenizer' is a convenient tool for tokenizing the input line. A parser class will tokenize each input line and check for each possible expression for a match.

Each valid expression will have its own syntax. So, expressions will themselves have the information of how they're going to be recognized. In order to match the correct expression, the parser will ask each expression if the input token list (representing the current line) matches it. The expressions will also validate the number of parameters during expression checking and report errors when necessary.

2.2. Interpretation

During the interpretation of each expression, the current context is going to be provided to the expression evaluator. The context is storage for variables, parameters, program counter, labels, current parsing state, etc. that each expression can use – either read or write.

The interpretation of each expression type is different. They're listed below.

2.3. Comments

No action will be taken for comment lines.

2.3.1. Variable Declaration / usage

The variable name –value pair is added to the list of variables in the context.

2.3.2. Function Call

Variables from the context and constants will be able to be printed out using the ‘print’ function call and user input will be captured using the ‘scan’ function call.

2.3.3. Assignment / Infix expression evaluation

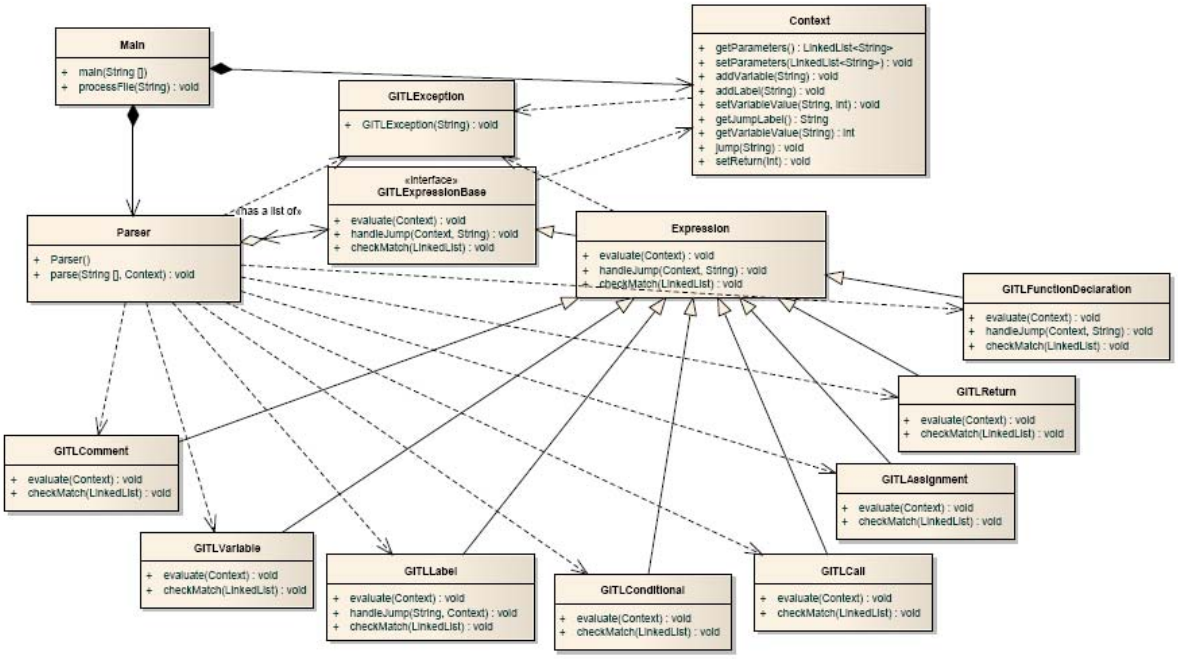
When an assignment operator is used, the right side of the expression will be handled as an infix expression. Variables used in the expressions will be replaced with their actual values from the context. The result is stored in one of the variables in the context.

2.3.4. Branching

Labels will be stored in the context. If the label that a condition expression wants to jump to is not stored in the context during its interpretation, this means that, that label is further down in the code. In this case, the parser will proceed until the corresponding label without actually executing anything. This will be accomplished by changing the state of the parser from RUN state to JUMPING state.

3. Design and Implementation

Considering the solution explained in the *Analysis* section, the class diagram is created as show below. The GOF pattern "Interpreter" is used as the basis of this structure:



3.1. Parsing

The Parser class is the class that handles the list of code lines. It has the 'parse' function for this purpose that goes through all the code lines and gets them processed by the corresponding GITLExpression classes.

In order for the parser to recognize the correct expression class, each of the expression classes' 'checkMatch' method will be called by the parser. In this method, the expression classes will check for the keyword(s) and number of parameters provided by the parser.

3.2. The context

During the interpretation of the code lines, the parser and the expression classes need a storage area where they can read/write important information. This storage area is called the *Context*. A class named Context handles this task. The stored information in the context is:

- Active Function – The name of the current function that is being interpreted.
- Current Line – The line number of the expression being interpreted.
- Exit Status – Set by the 'return' expression to tell the parser to quit.
- Current Jump Target – Set by the conditional jump expression when the target label has not been interpreted. When this target is a nonempty string, the parser looks for a label only, rather than executing lines.
- A list of functions – A list of function objects, each of which contain the name, line number and parameter list of a function.
- A list of Labels – A list of labels that have been interpreted together with their line numbers.
- A list of Variables – Name – value pairs of variables.
- Return value – Set by the return expression.

3.3. Parser States and Branching

During the interpretation of each code line, the parser can be in one of two states: RUN state and JUMP state. RUN state is the normal state in which each expression's *evaluate* function is called. This is explained in section 3.4. During the RUN state, if a condition expression tries to jump to a label that has not been interpreted before (that is, the label does not exist in the list of labels in the context), this means that the label is going to be searched further down the code. This is accomplished by setting the parser state to JUMP, in which the parser calls the *handleJump* method of the expressions, rather than their *evaluate* methods.

The *handleJump* method in the base Expression class has an empty body. Only two expressions override this method: The label expression and the function declaration expression. Both of these expressions check for the name of the current jump target with their own names, and restore the state to RUN if it matches.

3.4. Expression evaluation

During the normal RUN state of the parser, after a match is found in the key list, the parser calls the *evaluate* function of the corresponding expression object. During this evaluation, the expression object needs the current context in which it is running. For this purpose, the parser presents the current context to the expression object during *evaluate* function call.

3.4.1. Variable declaration/usage

The context provides 3 functions for this purpose: addVariable, getVariableValue, setVariableValue. In all expressions where a variable value is used, the 'getVariableValue' method is used to replace the variable with its value within the expression. The

GITLVariable class uses the ‘addVariable’ method to create the variable. The assignment operator calls the ‘setVariableValue’ function after arithmetic evaluation.

3.4.2. Function Call

There are two functions that can be called: print and scan. The GITLCall class handles these calls. It uses the getVariableValue method from the context to retrieve a variable’s value if it is being printed. It uses the setVariableValue method to set the variable value if the scan function is used.

3.4.3. Assignment / Infix expression evaluation

An assignment operation is recognized from the ‘=’ operator. The GITLAssignment class first converts the right side of the assignment statement into postfix form. During this conversion, the variables are replaced with their values from the Context. Then, the postfix expression is evaluated and stored in a variable using the setVariableValue method of the Context.

3.4.4. Conditional Jump

For this purpose, the GITLConditional class is used. When an ‘if’ statement is met, the variable following it is checked. If it is nonzero,

a) The last parameter of the if statement is the label to jump to. This label is searched in the Context to see if it has been stored by a ‘Label’ expression before. If so, a jump is made to that label – That is, the current line value is set to the current line value of the label in the context.
b) IF the label is not found in the Context, the parser is set to JUMP state, in which the label is found by the parser through handleJump methods of the expressions. This is explained in section 3.3.

3.4.5. Function Return

The return expression (GITLREturn) puts the return value in the Context and sets the exit flag (exitStatus member of the Context).

3.4.6. Function/Label declarations

The label expression (GITLLabel) and the function definition expression (GITLFunctionDefinition) adds entries to label/function lists in the context for further reference – by a conditional jump or a call.

3.5. Error Reports

GITLException class is extended from Exception class to use for catching parsing/interpretation errors. The parse function is surrounded by the try – catch block, reporting the error string before leaving the function.

3.6. Algorithms

3.6.1. Parsing Algorithm

The algorithm for the parse function is as follows:

```
try {
        reset lineCounter;
        Set the jump target to "main" and the parser state to "JUMP"
        for each code line
                tokenize the line
                for each possible expression
                        Get the expression to check itself for a match.
                        IF a match is found,
                                Put the current token list (the parameters) in the context.
                                If current state is JUMP
                                        call the expression object's handleJump method
                                If current state is RUN
                                        call the expression object's evaluate method
                If no expression matches the current line
                        Generate exception
                Check for the exit status from the context. If true, break the loop.
                Increment the line counter.
} Catch (GITLException e) {
        Report error from e;
}
```

3.6.2. Interpretation Algorithms (Evaluate methods)

3.6.2.1. Variable Declaration

- Get the parameters from the context. (Only one parameter, which is the variable name)
- Add this variable to the context.

3.6.2.2. Assignment

- Get the parameters from the context. (The name of the variable being assigned, and the list of tokens representing the infix expression)

- Convert the infix expression to its postfix form. During this conversion, replace variables with their values from the context
- Evaluate the postfix expression
- Put the result into the variable in the context.

3.6.2.3. Function call

- Get the parameters from the context. (Three parameters: function name and its input and output)
- If the function name is print
  - If the input parameter is a variable, retrieve it from the context
  - Print this value on the console
- If the function name is scan
  - Get the value from the user
  - Store it in the input variable in the context

3.6.2.4. Conditional Jump

- Get the parameters from the context. (The condition parameter and the jump label)
- If the condition parameter is a variable, retrieve it from the context.
- Check if the condition parameter is nonzero. If nonzero
  - Check the list of labels in the context. If found, set the lineCounter of the parser to this location.
  - If this label is not found in the label list, set the parser state to JUMP and start searching for this label. (Refer to section 3.3 for further information)

3.6.2.5. Function Definition

- Get the parameters from the context. (For the function definition case, there are 3 parameters: Function name, parameter type (int) and parameter variable.
- Add the function to the function list in the context, and set the active function to this function name in the context.

3.6.2.6. Function Termination

- Reset the active function to empty string in the context.

3.6.2.7. Label definition

- Get the parameter from the context. This single parameter is the name of the label.
- Add the label to the list of labels (together with its line number) in the context.

3.6.2.8. Function Return

- Get the parameter from the context. This single parameter is value to be returned.
- If the value to be returned is a variable, retrieve its value from the context.
- Set the `function return value` to this value in the context.

4. Testing

For testing purpose, 6 functions are written in Main: Test1_Keywords, Test2_Conditional, Test3_ReadConsole, Test4_Arithmetic, Test5_Error.

Test1_Keywords function contain all expressions in the code. It is verified that this code is interpreted without errors.

Test2_Conditional tests the if statement and jump to a label functionality.

Test3_ReadConsole tests the “call scan” function.

Test4_Arithmetic tests infix->postfix conversion and postfix evaluation for various examples.

Test5_Error tests for interpreter errors. In order to use it, the errors should be corrected one by one and the output is observed.

Also, the sample program ‘factorial’ is also executed and the correct result was observed.