CSE 222 HOMEWORK #9 (Part 1)

Orhan Aksoy

09104302

TABLE OF CONTENTS

List Of Figures

List Of Tables

LIST OF CHANGES

This document is based on the Homework #5. There are basically two changes in the code:

1. In order to improve the performance, the variable, label and function definitions are stored inside hash maps instead of linked lists.
2. In order to measure performance, a new function is added to the language: time. This function simply returns the number of milliseconds passed the last call to this function.
3. A new test program is added that is used for performance measurements.

The changes in the document is listed below:

Section	Change
1.1.14	A new function named ‘time’ is added to the interpreter. This function is used to measure time periods.
2.2.2	The expression ‘list of variables’ is replaced with ‘map of variables’
3	On the class diagram, two new functions are added to the GITLContext class: getLastTime and setLastTime.
3.2, 3.3, 3.4, 3.5.6, 3.7.2.3, 3.7.2.4, 3.7.2.5.1, 3.7.2.7.1, 3.7.2.7.2	The expression ‘list of variables’ is replaced with ‘map of variables’ The expression ‘list of lables’ is replaced with ‘map of labels’ The expression ‘list of functions’ is replaced with ‘map of functions’
4.8	New performance test is added to the tests section. This test also has an analysis section in which the comparision between the hash map and list is made.
4.8	An explanation section is added to the beginning for testing

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 be the input value from keyboard. An example usage is shown below:

- `call scan 0 b`

1.1.14. For the purpose of getting time intervals in milliseconds, the function ‘time’ shall be used. The first parameter shall not be used. The second parameter shall be returned by the call, and will have the value of the number of milliseconds that passed since the last call to this function. An example usage is shown below:

- `call time 0 t`

1.1.15. For calling locally defined functions, ‘call’ statement shall be used. The first parameter shall be the input parameter to the function, and the second parameter shall be the return value from the function. An example usage is shown below:

- `call customFunction inputParam resultValue`

1.1.16. The function definition is comprised of the function return type (which is ‘int’), function name, parameter type (which is ‘int) and parameter name.

- `int customFunction int parameter`

1.1.17. ‘`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`

1.1.18. '`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`

1.1.19. The comments within the code shall begin with '//'.

1.1.20. 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 encountered by the interpreter. 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 activation stack containing variables and program counter, and parameters being passed to the expression evaluators, labels, functions, 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.

2.2.1. Comments

No action will be taken for comment lines.

2.2.2. Variable Declaration / usage

The variable name –value pair is stored in the map of variables in the topmost activation frame of the activation stack in the context.

2.2.3. Function Definition / 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. Other than those, custom functions can be defined and called.

Function definitions will be comprised of function return type – which is always int, function name, parameter type – which is also always int, and parameter name. When a call is made, a new activation frame will be created and pushed on top of the activation stack, and a jump will be made to the line after the function definition. Upon function return, the topmost activation frame will be popped and a jump will be made to the next line making the function call.

2.2.4. 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.

2.2.5. 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.

This mechanism will also be used during function calls.

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:

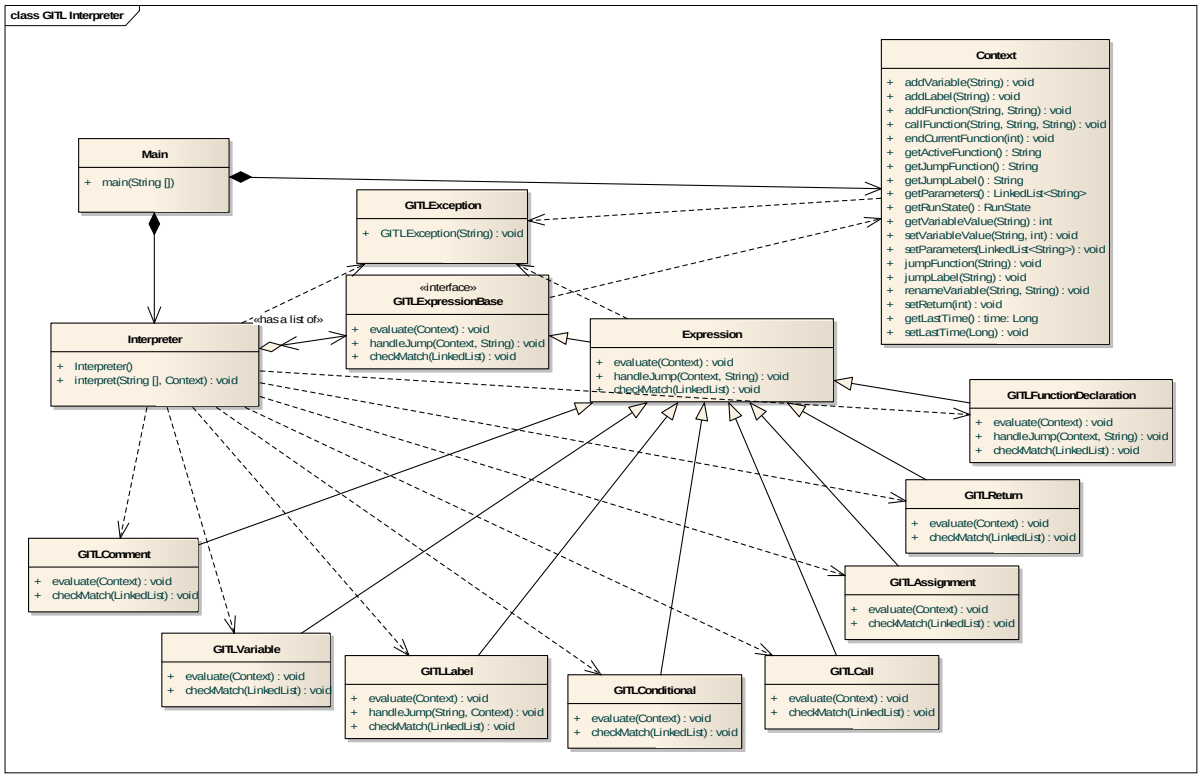

Figure 1 GITL Interpreter Class Diagram

3.1. Expression recognition

The Interpreter class is the class that handles the list of code lines. The 'interpret' method goes through all the code lines and gets them processed by the corresponding GITLExpression classes.

In order for the interpreter to recognize the correct expression class, each of the expression classes' 'checkMatch' method will be called by the interpreter. In this method, the expression classes will check for the keyword(s) and number of parameters provided by the parser, and then returns the matching status to the interpreter. When the interpreter receives a positive result from the checkMatch method, it gets the current line processed by that expression.

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:

- The activation stack – A stack of activation frames. The topmost frame is the active frame containing the current function name, variable map, the return address and return parameter. Refer to section 3.4.
- Current Line – The line number of the expression being interpreted.
- Exit Status – Set by the 'return' expression to tell the parser to quit.
- Current Label/Function Jump Target – Set by the conditional jump or function call expression when the target label/function has not been interpreted before. When this target is a nonempty string, the interpreter looks for a label or function definition only, rather than executing lines.
- A hashmap of functions – A map of function names mapped to function objects, each of which contain line number and parameter list of a function.
- A hashmap of Labels – A map of labels that have been interpreted to their line numbers.
- 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 **Error! Reference source not found.**. 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 label map 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. Activation Stack and Function Calls

The activation stack is comprised of activation frames. An activation frame is a temporary storage that the expressions use during interpretation within a function. Every function call creates a new activation frame to be used by the expressions within the called function. During creation of the activation frame, the following information is stored:

- The function name – The name of the function being called
- The return line – The line number at which the interpreter will jump to after interpretation of the called function.
- The return parameter – The name of the parameter used for function return value.
- The call parameter – The parameter being passed to the function. This parameter is stored in the variable list of the active frame as a regular variable.

Every time a function call expression is interpreted, this created activation frame is pushed on top of the activation stack stored in the context. From that point on, the expression evaluators use the information from the active frame on top of the stack. When the function return expression is interpreted, this frame is popped from the stack, the return value is put in the return variable and the current line value is restored to the line following the corresponding call expression.

The branching mechanism used during function calls is exactly same as the conditional jumps (Refer to section 3.3). If the called function is in the map of functions of the context, a jump is made to the known address. Otherwise, the parser is put into the JUMP state, in which the interpreter will seek the called function through *handleJump* methods.

When a call is made to a function, the parameter being passed to the function is stored in the activation frame of the called function. However, while interpretation of the 'call' expression, the interpreter does not know the real variable name of the call parameter defined in the function definition expression. As a result, the parameter is stored with a name '__callParam". When the interpreter finds the function definition line, it renames this variable with the real parameter name to be used by the expressions within that function.

3.5. 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.

3.5.1. 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.

3.5.2. Function Call

There are two default functions that can be called: print and scan. Also, locally defined functions can be called. The GITLCall class handles these calls. It uses the getVariableValue method from the active frame to retrieve a variable's value if it is being passed to the called function. If the function is 'print', this value is printed on the console. If the function is 'scan', the interpreter reads a value from the console and uses the setVariableValue method to set the variable value in the active frame.

If the called function is a locally defined function, the interpreter makes the call using the mechanism explained in section 3.4.

3.5.3. 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.

3.5.4. 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.

3.5.5. Function Return

The return expression (GITLREturn) calls the 'endCurrentFunction' method from the context. This method, in turn, pops the topmost frame from the activation stack. If the activation stack has no more elements after this pop operation, this means that 'main' is ending, so the return value is stored and the exit flag is set to tell the interpreter to stop processing this program.

If the activation stack is not empty, then, the return line number is read from the popped frame and the current line number is set to this number. Also, the return value is assigned to the return variable.

3.5.6. Function/Label definitions

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

3.6. 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.

3.7. Algorithms

3.7.1. 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;
}
```

3.7.2. Interpretation Algorithms (Evaluate methods)

3.7.2.1. Variable Declaration

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

3.7.2.2. 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 topmost activation frame in the context.
- Evaluate the postfix expression
- Put the result into the variable in the topmost activation frame in the context.

3.7.2.3. 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
- If the function is a locally defined function
    - Get the return parameter name from the tokens
    - Get the passed parameter from the tokens. If it is a variable, not a constant, get its value from the topmost activation frame in the context.
    - Call the 'callFunction' method of the Context. This method in turn:
        - Creates a new activation frame, storing the function name, current line, and return parameter name. Then push it on top of the activation stack.
        - Adds a new variable named '__callParam' to this new activation frame. Sets its value to the passed value to the function.
        - Searches for the function in the function map. If found, makes the jump, renames the "__callParam" to the real parameter name.
        - If not found, sets parser state to JUMP and start searching for this function. (Refer to section 3.4 for further information)

3.7.2.4. 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 map 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 map, set the parser state to JUMP and start searching for this label. (Refer to section 3.3 for further information)

3.7.2.5. Function Definition

3.7.2.5.1. The ‘handleJump’ operation

- 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 map in the context.
- Check if this is the function the interpreter is seeking. If so, reset the interpreter state form JUMP.

3.7.2.5.2. The ‘evaluate' operation

- Raise an exception – The evaluate method should never be called for function definitions. They should be encountered only when the interpreter is in JUMP state explained in section 3.4.

3.7.2.6. Function Termination

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

3.7.2.7. Label definition

3.7.2.7.1. The ‘handleJump’ operation

- Get the parameter from the context. This single parameter is the name of the label.
- Add the label to the map of labels (together with its line number) in the context.
- If the current jump target is this label, reset the interpreter state from JUMP.

3.7.2.7.2. The ‘evaluate operation

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

3.7.2.8. 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 topmost activation frame in the context..
- Pass this value to the caller using the mechanism explained in section 3.5.5

4. Testing

In order to test the application, the following sources are necessary:

The sources:

- Main.java, GITLInterpreter.java, GITLContext, GITLExpressions.java, GITLException
-

The GITL Test PRograms:

- arithmetic.gitl, console.gitl, errors.gitl, functioncall.gitl, iterfact.gitl, keywords.gitl, myprogrec.gitl, testhash.gitl

Before testing, the sources should be compiled using

javac Main.java

To run the application, use

java Main <name of the GITL test program>

4.1. Keywords

A test program named ‘keywords’ is used for testing recognition of all keywords.
The expected result is seeing ‘15’ as the program output.

```
// GITL test program - keywords
// keywords.gitl
int main int p
int c
int a
a = 3 + 2 * 5
// This is a comment
c = a + 2
call print c result
if c thelabel
label thelabel
return a
end
```

4.2. Syntax Errors

A test program named ‘errors’ is used for testing recognition of all keywords. To make the test, the test program is run. After getting the first error output, the error is fixed and the program is run again. This is repeated until no errors are reported by the interpreter.

```
1 - // GITL test program - error reports
2 - // error.gitl
3 - int main int p
4 - int c
5 - int c
6 - d = 3
7 - d = 3 + 2 *
8 - call
```

The first error reported by the interpreter will be at line 5: Duplicate variable. Rename it with ‘c1’.

The next error will be at line 6: Variable not found. Add its declaration between line 5 and 6. The new code is:

```
1 - // GITL test program - error reports
2 - // error.gitl
3 - int main int p
4 - int c
5 - int c1
6 - int d
7 - d = 3
8 - d = 3 + 2 *
9 - call
```

The next error will be at line 8: Invalid expression. Remove the ‘*’ symbol.

The next error will be at line 9: Invalid statement. Remove this line.

After removing the last line 9, the last error will be at line 9, too: Function ‘main’ not terminated. Add an ‘end’ statement at the end.

4.3. Arithmetic Expressions

A test program named ‘arithmetic.gitl’ is used for testing arithmetic expression calculation.

```
1- // GITL test program - arithmetic expressions
2- // arithmetic.gitl
3- int main int p
4- int c
5- int b
6- b = 3
7- c = b + 5 * 8 + 2 * 2
8- call print c res
9- c = b * 5 + 8 * 2 + 1
10- call print c res
11- c = b * ( 5 + 8 ) + ( 2 + 1 )
12- call print c res
13- end
```

The expected result from the application is seeing three results as program output: 47, 32 and 42.

Now, add the following statement between lines 12 and 13. This statement should overflow the integer data type, and the interpreter will report this error.

```
c = 1000000000 * 3
```

4.4. Console I/O

A test program named ‘console.gitl’ is used for testing console input/output.
When the application runs, it waits for the user to enter an integer. It then prints out the input value.

```
// GITL test program - read console
// readconsole.gitl
int main int p4
int aa
call scan 0 aa
call print aa 0
return aa
end
```

4.5. Conditional Expressions

A test program named ‘iterfact.gitl’ is used for testing conditional expressions. This program waits for the user to enter an integer, and then calculates the factorial of the input integer, and prints it on the console output.

```
// GITL program for iterative factorial calculation
// iterfact.GITL
int main int a
   // define input var
   int input

   // Scan the input from user
   // we assume user enters a positive int.
   call scan 0 input

   // result will hold the factorial value
   int result
   result = 1

   // Start of the loop
   label lab1

   result = result * input
   input = input - 1

   // check to see if we reached the end
   if input lab1

   // The factorial is calculated, print result
   int dummy
   call print result dummy

   // return no errors
   return 0
end
```

4.6. Function Calls

A test program named ‘functioncall.gitl’ is used for testing conditional expressions. This program makes a function call in main. The called function, in turn, makes three more calls: The first one defined before itself, the second between this function and main, and the last one after main.

```
1- // GITL test program - function calls
2- // functioncall.gitl
3-
4- int function2 int param
5-     int result
6-     param = 100 + param
7-     return param
8- end
9-
10- int function1 int param
11-     int result
12-     call print param result
13-     call function2 param result
14-     call print result result
15-     call function3 param result
16-     call print result result
17-     call function4 param result
18-     call print result result
19-     return param
20- end
21

22- int function3 int param
23-     int result
24-     param = 200 + param
25-     return param
26- end
27-
28- int main int arg
29-     int c
30-     int result
31-     c = 4
32-     call function1 c result
33- end
34-
35- int function4 int param
36-      int result
37-      param = 300 + param
38-      return param
39- end
```

The expected result from the application is seeing three results as program output: 4, 104, 204 and 304. Each of which are evidences that the corresponding function is called

Now, add a new function call between the lines 32 and 33:

```
33- call function5 c result
```

This statement will generate an error saying: Function not found: function5

4.7. Recursive Calls - The homework sample

A test program named ‘myprogrec.gitl’ is used for testing the recursive calls. The program waits for the user to enter an integer. Then, it calculates the factorial of this number making recursive calls, and outputs the result.

```
// GITL program for recursive factorial calculation
// myprogrec.GITL

// recursive factorial function
int factorial int n

   // base case
   if n lab1
   return 1

   // divide and conquer
   label lab1

   int res
   int t
   t = n - 1
   // recursive step
   call factorial t res
   res = n * res
   return res
end

int main int a
   // define input var
   int input

   // Scan the input from user
   // we assume user enters a positive
   call scan 0 input

   // result will hold the factorial va

   int result

   // call factorial
   call factorial input result

   // The factorial is calculated, prin
```

```
    int dummy
    call print result dummy

    // return no errors
    return 0
end
```

4.8. Performance Measurements

A test program named ‘testhash.gitl’ is used for testing the interpreter performance. The program makes calls to three functions repeatedly. Each of these functions have 50 variables defined in them. Afterwards, a few of these variables are used in expressions.

The variables in the first and second functions are defined in their alphabetical order. In the third function, the variables are defined in random order.

Each of these functions are called 250 consecutive times in the program. Besides, the whole code is repeated in an outer loop 100 times. Each of the 250-call periods are measured using the new ‘time’ function added to the language. The measured time periods are printed on the console. Between each pair of outer loop calculations, a separator number ‘9999999’ is printed out on the console. The output is collected and saved.

The pseudocode of the GITL program is shown below:

```
Loop for 100 times
        Start measuring time
        Loop for 250 times
                Call function test_random_variables
        Note the first time period elapsed
        Loop for 250 times
                Call function test_ordered_variables_1
        Note the second time period elapsed
        Loop for 250 times
                Call function test_ordered_variables_2
        Note the third time period elapsed.
```

The same test is also made with the GITLContext class without hashmap implementation. The outputs are also saved for comparison with the version with hashmap implementation.

The comparison of the outputs can be seen below. The values on the table are outputs from the program that represent the average number of milliseconds that passed during the execution of the related function 250 times consecutively:

	Variables in random order		Variables in alphabetical order (1)		Variables in alphabetical order (2)	
	Avg. Time	Std. Dev.	Avg. Time	Std. Dev.	Avg. Time	Std. Dev.
Without Hashmap	34.9	6.6	34.4	6.3	37.6	7.7
With Hashmap	22.2	7.8	23.2	7.9	21.5	7.6

Table 1 Comparison of Hashmap and List Implementations

The averages are obtained using 100 samples ignoring the first one. As seen on the table, the gained CPU time is around 40% by using the hashmap implementations.

However, the measured data have a very high variance. The standard deviations in each collected data is around 1/3 of the actual data. Although a reasonable comparison can be made between the versions with and without hashmap implementations, the three functions in each case cannot be compared among themselves because of this reason.

The reason behind the high variance could be the background processing that java does undeterministically and/or the dynamic CPU speed regulation in the test machine.

The test code is shown below:

```
// GITL test program - hash map performance
// testhash.gitl

int test_random_vars int p

int var_28
int var_16
int var_17

… total of 50 variables defined in random order


var_01 = 3
var_02 = 5
var_03 = 7
end

int test_ordered_vars1 int p

int var_01
int var_02

.. The 50 variables defined in alphabetical order

int var_50

var_01 = 3
var_02 = 5
var_03 = 7
```

```
end

int test_ordered_vars2 int p

int var_01
int var_02

.. The 50 variables defined in alphabetical order

int var_50

var_48 = 3
var_49 = 5
var_50 = 7

end

int main int p

int result

int loop2
int t1

int bigLoopCount
bigLoopCount = 100

label bigLoop

call time 1 t1
loop2 = 250
label loop2_1
call test_random_vars 4 result
loop2 = loop2 - 1
if loop2 loop2_1
call time 1 t1
call print t1 result

call time 1 t1
loop2 = 250
label loop2_2
call test_ordered_vars1 4 result
loop2 = loop2 - 1
if loop2 loop2_2
call time 1 t1
call print t1 result

call time 1 t1
loop2 = 250
label loop2_3
call test_ordered_vars2 4 result
loop2 = loop2 - 1
if loop2 loop2_3
call time 1 t1
```

```
call print t1 result

call print 9999999 result

bigLoopCount = bigLoopCount - 1
if bigLoopCount bigLoop


end
```