GKV

PROGRAMMING LANGUAGE

Reference Manual

Prepared By: Team 9

- **Gowtham G Nayak**
- **Kallol Chatterjee**
- **Vimarsh Deo**

How to run the program?

To generate intermediate code:

Import the Compiler project into eclipse. Change HIGH_LEVEL_LANGUAGE_FILE_PATH INTERMEDIATE_LANGUAGE_FILE_PATH variables in Constants.java file define the file path where HLL and Intermediate Code exist. Also while running main method in java pass the filename of the HLL or IC as an argument.

To run the IC in runtime:

Import the Runtime project into eclipse. Change the INTERMEDIATE_LANGUAGE_FILE_PATH to point to a directory where IC files are stored. Pass the filename of the intermediate code which you want to execute as an argument to main method.

Entire source code can be found on github:

Root folder: https://github.com/KallolChatterjee/SER-502
Grammar: https://github.com/KallolChatterjee/SER-502/tree/master/GKV/Grammar
Compiler: https://github.com/KallolChatterjee/SER-502/tree/master/GKV/Compiler
Runtime: https://github.com/KallolChatterjee/SER-502/tree/master/GKV/Runtime

Sample Programs - High Leve lLanguage:

https://github.com/KallolChatterjee/SER-502/tree/master/GKV/Sample-Programs/High-Level-Programs

Sample Programs - Intermediate Code:

https://github.com/KallolChatterjee/SER-502/tree/master/GKV/Sample-Programs/Intermediate-Code

Demo on YouTube:

https://www.youtube.com/watch?v=EHAWrs_t4nY&feature=youtu.be

Tools Used:

- ANTLR 4.5
- Eclipse Luna SP2

Lexer File: GKVLexer.java
Parser File: GKVParser.java

High Level Programs & Corresponding Intermediate Code

1. Print Hello World

```
show "Hello World";
```

- ‘ show ’ keyword is used to print the text on screen.
- ‘ ; ’ marks the end of every statement in our high level language

```
Intermediate Code:

PRINT "Hello World"
```

- The print keyword is generated for show in intermediate language

2. Addition

```
integer a;
integer b;
integer c;
a = 10;
b = 20;
c = a + b;
show c;
```

- We use datatype ‘integer’ to declare variables a,b and c.
- Then we initialize variables to some values
- We use ‘ + ’ to add two variables and use the ‘ = ’ operator to assign the computed value to c
- Lastly , we use show keyword to print result on screen
- Like ‘ + ’ operator, we have other mathematical operators : ‘ - ’ , ’ * ’ , ’ / ’ which are minus operator, multiplication operator and division operator respectively

Intermediate Code:

```
DECLINT A
DECLINT B
DECLINT C
PUSH 10
SET A
PUSH 20
SET B
PUSH A
PUSH B
ADD
SET C
PRINT C
```

- Using DECLINT we are declaring integers
- PUSH is used to push the value to the stack
- SET is used to pop a value out of the stack and assign it to the identifier associated
- ADD is used to pop two values from the stack and add those two and again push the result in the stack

3. CheckIFEqual

```
integer a;
integer b;
integer c;
a = 10;
b = 20;
if (a equalTo b) then {
       show "A is equal to B";
} else {
       show "B is not equal to A";
}
```

- Firstly we declare integer variables a,b and c
- Then we initialize a and b to some values
- Then we do decision making using ‘ if ’ keyword.
- Inside the if statement we are comparing two variables using comparison operator ‘equalTo’ followed by keyword ‘ then ’
- Similar to above comparison operator, we also have other comparison operators : lessThan', 'greaterThan' , 'lessThanOrEqualTo' , 'greaterThanOrEqualTo' and 'notEqualTo'
- If condition stands true, It will execute the set of statements in the first block after ‘ then ’
- If condition is false, it will go to the ‘ else ‘ part and execute the block of statements after else
- Each block of statements is enclosed within opening and closing parenthesis.

Intermediate Code:

```
DECLINT A
DECLINT B
DECLINT C
PUSH 10
SET A
PUSH 20
SET B
PUSH A
PUSH B
EQL
JMPIFFALSE LABEL
START
SCOPESTART
PRINT "A is equal to B"
SCOPEEND
```

```
JMP ELSEEND
END
LABEL: ELSESTART
SCOPESTART
PRINT "B is not equal to A"
SCOPEEND
ELSEEND
```

- JMPIFFALSE pops a value from stack checks if it's Zero.
- Then jump to the label mentioned.
- SCOPESTART is to start a scope. This directs a compiler to start a new local symbol table.
- SCOPEEND is to end a scope. This directs the compiler to end the current scope and move to parent scope.
- EQL pops two values from the stack and checks if they are equal. If they are then it pushes 1to stack else pushes 0 to the stack.

4. Function

```
integer x;
integer y;
integer z;


x = 6;
y = 3;


function add uses integer a, integer b
returns integer {
       integer c ;
       c = a+b;
       return c;
}


function sub uses integer a, integer b
returns integer {
       integer c ;
       c = b - a;
       return c;
}


z = call add with x, y;
show z;
z = call sub with x, y;
show z;
```

- Firstly , we are declaring three variables x, y and z and assigning values to x and y.
- Then we are defining a function named add.
- We start by writing the keyword ‘ function ’ followed by function name , then the keyword ‘uses’, followed by parameters, then the keyword ‘ returns ’ followed by return datatype.
- Parameters in the function definition are written in the form of datatype followed by identifier name.
- Inside the add function , we are declaring another integer variable c, using ‘ + ’ operator to add the values of a and b and using ‘ = ’ operator to initialize the value in c.
- Finally we are using the keyword ‘ return ’ keyword to return the value c
- Similarly we are defining the sub function. The only difference is that here we use ‘ – ‘ operator to calculate the difference between the two variables.
- To invoke the add function we need to call the function using ‘ call ‘ keyword followed by function name , then the ‘ with ’ keyword followed by the parameters.
- Here we have initialized the result of the function to another variable z

Intermediate Code:

```
DECLINT X
DECLINT Y
DECLINT Z
PUSH 6
SET X
PUSH 3
SET Y
FUNCSTART ADD INTEGER
SCOPESTART
DECLINT A
DECLINT B
DECLINT C
PUSH A
PUSH B
ADD
SET C
PUSH C
RET C
SCOPEEND
FUNCEND
FUNCSTART ADD INTEGER
SCOPESTART
DECLINT A
DECLINT B
DECLINT C
PUSH B
PUSH A
SUB
SET C
PUSH C
```

```
RET C
SCOPEEND
FUNCEND
PUSH X
PUSH Y
CALL ADD 2
SET Z
PRINT Z
PUSH X
PUSH Y
CALL SUB 2
SET Z
PRINT Z
```

- CALL is a directive for runtime to create an activation record in the heap.
- RET pushes the return value to the stack and exists the activation record.
- This value can be popped and assigned to a variable as a return value by popping from the stack.

<u>5. Stack</u>

```
stack a;
a.push(3);
a.push(4);


integer b;
b = a.pop();
show b;
b = a.pop();
show b;
```

- We have defined a datatype stack and declaring a variable a of stack type
- We have defined push and pop operators for variables of type stack.
- We are first pushing values 3 and 4 on the stack and then popping out the values in integer b
- Finally we print those popped values using show

Intermeidate Code:

```
DECLSTACK A
A.PUSH 3
A.PUSH 4
DECLINT B
A.POP
SET B
PRINT B
A.POP
SET B
PRINT B
```

- A.POP will pop a value from the stack object.
- A.PUSH will push a value into the stack object.

6. While

```
integer i;
i = 0;


while (i lessThan 10) {
       i = i + 1;
       show i;
}


show "done";
```

- First we define a variable i of type integer and initialize it to zero
- Then we use while statement for iteration.
- In while statement, we have the keyword ‘ while ’ followed by the condition.
- If the condition is true, the sequence of statements in the following block executes.
- Lastly we print “done” using show keyword

Intermediate Code:

```
DECLINT I
PUSH 0
SET I
LOOPHEAD : PUSH I
PUSH 10
LT
JMPIFFALSE LABEL
START
SCOPESTART
PUSH I
PUSH 1
ADD
SET I
PRINT I
SCOPEEND
END
JMP LOOPHEAD
LABEL : PRINT "done"
```

- JMP is unconditional jump statement used to jump to a particular label.

7. Recursion

```
function fact uses integer a returns integer {
        if (a equalTo 1) then {
                return 1;
        } else {
                integer temp;
                temp = call fact with (a - 1);
                temp = a * temp;
                return temp;
        }
}

call fact with 3;
```

- In the above code, we display factorial of 3 using recursion.
- When call keyword is encountered, it calls the function fact and passes parameter 3 to it.
- In the first if statement, we are checking if value of a is 1, in which case it is returning 1
- Else it is calling the function fact again with parameter decreased by 1
- Then it is multiplying the parameter with the return value of the function called
- Finally it is returning the computed factorial.
- We have also implemented Fibonacci and GCD computation using recursion

Intermediate Code:

```
FUNCSTART FACT INTEGER
SCOPESTART
DECLINT A
PUSH A
PUSH 1
EQL
JMPIFFALSE LABEL
START
SCOPESTART
PUSH 1
RET 1
SCOPEEND
JMP ELSEEND
END
LABEL: ELSESTART
SCOPESTART
DECLINT TEMP
PUSH A
PUSH 1
SUB
CALL FACT 1
SET TEMP
PUSH A
PUSH TEMP
MULT
```

```
SET TEMP
PUSH TEMP
RET TEMP
SCOPEEND
ELSEEND
SCOPEEND
FUNCEND
PUSH 3
CALL FACT 1
```

- Recursive functions have a CALL statement inside their function body.
- This is directive to the compiler to suggest that it is a recursive function.
- A new activation record should be created on a stack without deleting the current activation record.