JAL LANGUAGE REFERENCE

HARISH MALIK

NIKHIL KUMAR

SHUJIAN KE

1 INTRODUCTION

This reference manual describes the JAL programming language. It is built using JAVA and makes use of ANTLR to generate the parser and Lexer. JAL is a functional language and takes inspiration from C language. It is statically typed and supports “int” and “bool” datatypes. It uses static typing and performs lazy evaluation. The program begins its execution inside the main function. It supports if-else construct, while statements and an inbuilt stack data structure.

2 PROGRAM COMPILATION AND EXECUTION

Below are the steps to successfully compile and run JAL programs.

1. Download the JAL.jar and extract it.
2. The contents after extraction will have the structure as given below :
3. The Source Code directory includes all the source code for the project.
4. Write a program with .jal extension.
5. Compile it by running the command java –jar JALCompile.jar <filename.jal>
6. This will generate the .jalclass file which is the intermediate code.
7. To execute the intermediate code, run the command java –jar JALRuntime.jar <filename.jalclass>
8. The doc directory includes the documents for the language.
9. The sample programs directory includes all the sample codes.

3 IMPORTANT LINKS

1. Github link for the repository : https://github.com/hmalik108/JAL
2. Youtube link for the video : https://www.youtube.com/watch?v=NUwhkTQGbek

4 SAMPLE PROGRAMS

1. **ADD TWO NUMBERS**

```
func int add(int a, int b):
       int c;
       c = a + b;
       return c;
end_func

func int main():
       println(add(100,200));
       return 0;
       end_func
```

Intermediate Code:

```
.start method add paramCount: 2
store b
store a
load a loc:0
load b loc:1
add
store c loc:2
load c loc:2
return
.end method add
.start method main paramCount: 0
push 100
push 200
.invoke add paramsCount: 2
println
push 0
return
.end method main
```

Explanation

a. .start and .end indicate the start and end of function body
b. Store command is used to store the value of the variable from stack into symbol table.
c. Load command is used to move the value of the variable from the symbol table onto stack.
d. Return is used to return the top of the stack
e. Push is used to push constants onto the stack
f. Add commands pops two constants from the stack and pushes back the result.

2. FINDING MAX OF TWO NUMBERS

```
func int max(int a, int b):
       int c;
       if(a<b):
              c = b;
       else:
              c = a;
       end_if
       return c;
end_func


func int main():
       println(max(15,24));
       return 0;
end_func
```

Intermediate Code:

```
.start method max paramCount: 2
store b
store a
load a loc:0
load b loc:1
less_than
branch_if: if_true: 0
load b loc:1
store c loc:2
branch_if: goto endIf: 0
branch_if: if_not_true: 0
load a loc:0
store c loc:2
branch_if: endIf: 0
load c loc:2
return
.end method max
.start method main paramCount: 0
push 15
push 24
.invoke max paramsCount: 2
println
push 0
return
.end method main
```

Explanation

a. Less_than pops two constants from the stack and evaluates if the condition is true or not.
b. If the condition is true then the statements after if_true label are executed.
c. If the condition is false then the statements after the if_not_true are executed.
d. End_if label can be used to move to the end of the if-else statements.
e. Unique Identifier indicated against label is used for nested if-else.

3. SUM TO FIRST N NATURAL NUMBERS

```
func int sum(int a):
       int c;
       c = 1;
       int sum;
```

```
        sum = 0;
        while(c<=a):
                sum = sum + c;
                c = c + 1;
        end_while
        return sum;
end_func


func int main():
        println(sum(10));
        return 0;
end_func
```

Intermediate Code:

```
        .start method sum paramCount: 1
        store a
        push 1
        store c loc:1
        push 0
        store sum loc:2
        while: 0
        load c loc:1
        load a loc:0
        less_than_or_equal
        branch_loop: if_true: 0
        load sum loc:2
        load c loc:1
        add
        store sum loc:2
        load c loc:1
        push 1
        add
        store c loc:1
        end_while: 0
        load sum loc:2
        return
        .end method sum
        .start method main paramCount: 0
        push 10
        .invoke sum paramsCount: 1
        println
        push 0
        return
        .end method main
```

Explanation:

a. The while tag is used to indicate that a while loop follows.
b. The condition is then evaluated.
c. The part of code between the while and end_while tags are then executed as long as the condition is met.

4. NESTED IF-ELSE STATEMENT

```
func int bavaisgreat():
       int a;
       a = 4;
       int b;
       b = 5;
       if( a < b):
              if(b < 4):
                     println(a+b);
              else:
                     println(a);
              end_if
       else:
              println(a * a+b);
       end_if

       return a;

end_func

func int main():
       println(bavaisgreat());
       return 0;
end_func
```

Intermediate Code:

```
.start method bavaisgreat paramCount: 0
push 4
store a loc:0
push 5
store b loc:1
load a loc:0
load b loc:1
less_than
branch_if: if_true: 1
load b loc:1
push 4
less_than
branch_if: if_true: 0
load a loc:0
load b loc:1
add
```

```
println
branch_if: goto endIf: 0
branch_if: if_not_true: 0
load a loc:0
println
branch_if: endIf: 0
branch_if: goto endIf: 1
branch_if: if_not_true: 1
load a loc:0
load a loc:0
mul
load b loc:1
add
println
branch_if: endIf: 1
load a loc:0
return
.end method bavaisgreat
.start method main paramCount: 0
.invoke bavaisgreat paramsCount: 0
println
push 0
return
.end method main
```

Explanation

a. The unique identifier is used to differentiate the if-else.
b. This is helpful for nested if-else constructs

5. STACK

```
func int main():
       stack s;
       s.push(10);
       s.push(20);
       s.push(30);
       println(s.top());
       s.pop();
       println(s.top());
       s.pop();
       return 0;
end_func
```

Intermediate Code :

```
.start method main paramCount: 0
stack s
push_stack s 10
push_stack s 20
push_stack s 30
top_stack s
println
pop_stack s
top_stack s
println
pop_stack s
push 0
return
.end method main
```

Explanation:

a. Stack s indicates that a stack has been declared
b. Push_stack indicates a push operation from the built-in data structure.
c. Pop_stack indicates a pop operation from the built-in data structure.
d. Top_stack indicates a top operation from the built-in data structure.

6. FACTORIAL OF A NUMBER – RECURSIVE

```
func int fact(int n):
   int result;
   if(n>1):
              result = fact(n-1) * n;
       else:
              result = 1;
       end_if
       return result;
end_func

func int main():
       println(fact(5));
       return 0;
end_func
```

Intermediate:

```
.start method fact paramCount: 1
store n
load n loc:0
push 1
greater_than
branch_if: if_true: 0
load n loc:0
push 1
sub
```

```
.invoke fact paramsCount: 1
load n loc:0
mul
store result loc:1
branch_if: goto endIf: 0
branch_if: if_not_true: 0
push 1
store result loc:1
branch_if: endIf: 0
load result loc:1
return
.end method fact
.start method main paramCount: 0
push 5
.invoke fact paramsCount: 1
println
push 0
return
.end method main
```

Explanation:

a. paramCount indicates the number of parameters passed to the function.
b. Recursion is handled at runtime

7. SCOPES

```
func int randomNumber():
   int i;
   i = 4;
   return i;
end_func

func int main():
       int i;
       i = 42;
       println(randomNumber());
       println(i);
       return 0;
end_func
```

Intermediate:

```
.start method randomNumber paramCount: 0
push 4
store i loc:0
load i loc:0
return
```

```
.end method randomNumber
.start method main paramCount: 0
push 42
store i loc:0
.invoke randomNumber paramsCount: 0
println
load i loc:0
println
push 0
return
.end method main
```

Explanation:

a. Scoping is managed using symbol table.