Javier

Doug Carroll, Jeb Johnson, Isidro Perez, Jason Rice

Grammar

Program --> StatementLst

StatementLst --> Statement | Statement StatementLst

Statement --> “new_line” | Assignment | If | Loop

Assignment --> Identifier “=” Expression

If --> “if” Condition “{“ StatementLst “}” | If Else

Else --> “else” “{“ StatementLst “}”

Grammar Cont.

Condition --> Expr “==” Expr | Expr “>=” Expr | Expr “<=” Expr | Expr “>” Expr | Expr “<” Expr | Expr “!=” Expr | Condition “and” Condition | Condition “or” Condition

Loop --> “loop” Loop-Assignment “{“ StatementLst “}”

Loop-Assignment --> Identifier “=” Iterator

Expression --> Expression “+” Expression | Expression “-” Expression | Expression “*” Expression | Expression “/” Expression | Expression “%” Expression | “(“ Expression “)” | Number | Identifier

Grammar Cont.

Iterator --> Expr “,” Expr

Identifier --> Letter | Letter Number | Letter Identifier

Number --> Digit | Digit Number

Digit --> ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

Letter --> ‘a’..’z’ | ‘A’..’Z’

Syntax

- Operators: + - * / %
- Conditional: == != >= <= > < and or
- Assignment: someVariable = someValue
- If statements
  - If someConditionalExpression {Statements}
  - Else {Statements}
- Loops
  - Loop i = someExpression, someExpression2 {Statements}
  - Similar to a for loop, loops from value someExpression to someExpression2
- Terminating statements
  - Semi-colon
  - New line

Factorial in Javier

```
#/
    factorial of x, change the value of x to get its factorial
    if x is less than 0 you will get -1
/#

x = 8;
factorial = 1;
tmp = x;

if x >= 0 {
    loop i = 1, x {
        factorial = factorial * tmp;
        tmp = tmp - 1;
    }
} else {
    factorial = -1;
}

print factorial;
```

Lexical Analysis

- Input: Javier code file as a String.
- Output: ArrayList of Token objects.
  - The ArrayList of token objects includes every token within the code.
  - Spaces and commented code are removed at this stage.
    - Single line comment: #
    - Multi-line comment: #/ *comment here* /#
  - New lines are left in to separate statements.
- Tokens have a type and a value.
  - Example: = will have type Assignment and value =
  - (assignment, value)

Validation

Validation was straightforward and

Followed the grammar definitions.

One example is valid iterator.

```
/*
==========================================================================================
Checks if an iterator is two valid expressions
==========================================================================================
*/

private boolean is_valid_iterator() {

    String[] temp1, temp2;

    statement_index++;
    temp1 = get_code_segment(GrammarDefs.COMMA).clone();
    statement_index++;
    temp2 = get_code_segment(GrammarDefs.OPEN_BRACKET).clone();

    return is_valid_expression(temp1) && is_valid_expression(temp2);
}
```

```
/*
============================================================
Determines if token sequence is a valid mathematical expression
============================================================
*/

private boolean is_valid_expression(String[] token) {

    boolean flag = true;
    String prev_token = token[0];
    int i = 1;

    //checks expression for invalid combinations
    while (flag && i < token.length) {

        if (is_numerical_token(prev_token) && !is_valid_numerical_successor(token[i])) {

            flag = false;

        } else if (is_operation(prev_token) && !is_valid_operator_successor(token[i])) {

            flag = false;

        } else if (is_open_paren_token(prev_token) && !is_valid_open_par_successor(token[i])) {

            flag = false;

        } else if (is_closed_paren_token(prev_token) && !is_valid_closed_par_successor(token[i])) {

            flag = false;

        }

        prev_token = token[i];
        i++;
    }

    return flag;
}
```

Each statement is made up of small definitions. To make sure each piece was in the correct place switch cases statements were used.

```
/*
=====================================================
checks if token is allowed to follow an operator
=====================================================
*/

private boolean is_valid_operator_successor(String token) {

    boolean flag = false;

    switch (token) {

        case GrammarDefs.OPEN_PAREN:
            flag = true;
            break;

        case GrammarDefs.CLOSED_PAREN:
            flag = true;
            break;

        case GrammarDefs.IDENTIFIER:
            flag = true;
            break;

        case GrammarDefs.WHOLE_NUMBER:
            flag = true;
            break;

        default:
            break;
    }

    return flag;
}
```

Parser

- Breaks up tokens from the lexical analysis into a hashmap of labels, starting with the main program label if any code exists that is blocked off such as an if statement or loop then the statements are put into an label to be referenced later.

Example:

```
program = [

 x = 8 ;
 factorial = 1 ;
 tmp = x ;

 if x >= 0 { label-1 } else { label-3 }

 print factorial ;
 ]
```

```
label-1 = [
 loop i = 1 , x { label-2 }
 ]
```

```
label-3 = [
 factorial = -1 ;
 ]
```

Parser Cont.

- The parser then takes the hashmap of labels and checks that each label has correct grammar. It does this using the grammar rules that have been previously defined erasing and checking that the code follows the patterns laid out. This check happens on top of the Validation to check for any errors the validation may have missed. If an error is found the program will exit with a return code 0, if not the hashmap will be returned and passed to the Intermediate code.

Intermediate Code

- Breaks parsed code down into individual operations.
- Operations
  - ArrayList of Operations.
  - Operations have a type, a variable, and two values.
  - Operations converted to assembly.
- Order of operations
  - Done at this stage.
  - Breaks mathematical and conditional expressions down to Operations in correct order.
  - Users $t temporary variables within expressions
  - 4+5
    - add, $t1, 4, 5

Intermediate Code

- Related to MIPS Assembly
  - Jump returns are used: jr
  - Jump is used: jump someLabel
  - Program terminates on: end
- Labels represent individual code blocks
  - Always end in a jump return

Intermediate Code - Factorial

```
add,x,0,8
add,factorial,0,1
add,tmp,0,x
gtq,$t0,x,0
if,$t0,label-1,label-3
print,factorial
end
label-3
add,factorial,0,-1
jr
label-2
multi,$t1,factorial,tmp
add,factorial,0,$t1
sub,$t2,tmp,1
add,tmp,0,$t2
jr
label-1
add,i,0,1
looplabel0
ltq,$l0,i,x
lif,$l0,label-2,leaveloop0
add,i,i,1
jump,looplabel0
leaveloop0
jr
```

```
#/
    factorial of x, change the value of x to get its factorial
    if x is less than 0 you will get -1
/#

x = 8;
factorial = 1;
tmp = x;

if x >= 0 {
    loop i = 1, x {
        factorial = factorial * tmp;
        tmp = tmp - 1;
    }
} else {
    factorial = -1;
}

print factorial;
```

Runtime

```
JavierRuntime runtime = new JavierRuntime();

runtime.run(pathName);
```

- Reads the intermediate code file
- Creates symbol table
- Manipulates data
- Throws errors
- Ends

Five Given Example Programs

- Factorial: calculate the factorial of x
- Fizz-Buzz: classic fizz-buzz problem
- Pemdas: an exercise in order of operations and comparisons
- Reverse-Number: reverse a number
- Triangle-Number: print the triangle of a number

Thank You For Watching!