Complier Lab (CSE4112)

Assignment #2 Generating Three-address Code for Flow-of-Control Statements

Assignment Details: We now have added Boolean/logical operators and Conditional statements to our grammar [marked in Red]. Your task is to build upon the solution to assignment#1 to include these new constructs and generate 3AC for a code segment.

A. Syntax

Note: The syntax for expressions are written in BNF like format. A | indicates a choice, a pair of curly brackets { and } indicates zero, one or more repetitions, and a pair of squared brackets [and] indicates zero or one repetition.

Ident ::= Letter { Letter | Digit } .

Letter ::= *letters "A" to "Z"* | *"a" to "z"* .

Integer ::= Digit { Digit } | "0x" HexDigit { HexDigit } .

Digit ::= *digits "0" to "9"* .

HexDigit ::= Digit | *letters "a" to "f"* | *letters "A" to "F"*.

NoSignFactor ::= Ident | Integer | "(" Expr ")" .

Factor ::= ["+" | "-"] NoSignFactor .

Term ::= Factor { ("*" | "/" | "&&") Factor } .

SimpleExpr ::= Term { ("+" | "-" | "||") Term } .

Expr ::= SimpleExpr [("==" | "!=" | "<" | "<=" | ">" | ">=") SimpleExpr] .

Assignment ::= Ident "=" Expr ";" .

IfStmt ::= "if" "(" Expr ")" StmtBlock ["else" StmtBlock] .

WhileStmt ::= "while" "(" Expr ")" StmtBlock .

Stmt ::= Assignment | ifStmt | WhileStmt .

StmtBlock ::= "{" { Stmt } "}" .

B. Three-address Code

Three-address code (often abbreviated to TAC or 3AC) is an **intermediate code** used by optimizing compilers to aid in the implementation of code-improving transformations. Each 3AC instruction has at most **three** operands. Below are examples of different types of 3AC. **In this assignment you will need to use 3AC types 1-6**.

Types of 3-address codes

1. Binary Operations
```
x = x + y
x = x * y
```

2. Unary Operation
```
x = -y
```

3. Assignment/Move
```
x = y
```

4. Jump/Goto/Unconditional Branch
```
goto label_3
```

5. Label
```
label_3:
```

6. Conditional Jumps/Branches
```
    if x < y then goto label_5
```
Possible conditions: < , > , <=, >=, !=, ==

7. Array Accessing
```
x[i] = y
y = x[j]
```

8. Function Call
```
param x
param y
call foo
result x
```

9. Pointer Manipulations
```
x = &y
y = *x
*x = y
```

C. Implementation:

You will add to the LEX/FLEX and YACC/Bison specifications that you have in assignment#1.
In order to generate 3ACs this time you cannot print the 3AC lines as soon the parser does a reduce operation. In order to generate 3AC you will have to:

(i) Work with Boolean operators. One way to handle them is to generate code that simply evaluates it just like arithmetic expressions. You may assume, any non-zero value of an expression is TRUE (i.e., an expression that results in Zero is FALSE). However, simply generating code that evaluates the Boolean expression is not the only way to handle them. For extra credit implement the translation scheme outlined in section 6.6.3 in the textbook.

(ii) You will find that printing the 3AC at the time of reduce will be problematic in the case of conditional statements. Rather the 3AC codes corresponding to expressions and statements need to be stored (e.g., in Quads) so that you can rearrange them.

Sample Input

```
{ a= 5;

while (a + 1){

        a = - b *c ;

}

}
```

Sample Output

```
a=5
label_1:
t1 = a +1
if t1 == 0 goto label_2
t2 = - b
t3 = t2 * c
a = t3
goto label_1
label_2:
```

Assignment Due: **06/02 and 08/02 for the respective groups**. If for some reason the classes are cancelled we will setup a different day in the week for lab.