Complier Lab (CSE4112)

Assignment #1 Generating Three-address Code for Assignment Statements

Assignment Details: Below is the (partial) syntax for block of assignment statements for the programming language we will implement. It is very similar to the grammar that you worked on in assignment #0. The new features are (i) there are sequences of digits and sequence of hex digits and both are recognized as integers, (ii) we have unary minus operator (iii) we can have a list of assignment statements within "{" and "}" instead of a single assignment statement. You are to generate intermediate code (i.e., Three-address code) for the assignment statements as an output of your compiler. **Please read the instructions carefully.**

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 .

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

Stmt ::= Assignment .

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 only need to use 3AC for binary operations, Unary operations, and Move**.

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 have to write a LEX/FLEX and YACC/Bison specifications in this assignment. In YACC or Bison you have to put down the corresponding CFG productions of each line of the syntax description. You will have to decide which symbols/variables in the language specification you want to define in the lexical analyzer and return to the parser, and which ones you want to be matched in YACC/Bison. For example, you may choose to recognize symbols such as *ident*, *Digit*, *HexDigit* in LEX, have the operators "+", "-" etc matched in the parser.

In assignment # 0, the input expressions were evaluated. Here, instead of evaluating the expressions, you will generate 3AC for a block of assignment statements. See the sample input and output below. In the previous assignment, the attribute for the variables in the grammar were integers which held the value of the sub-expression rooted at each node of the parse tree and actions corresponding to each reduce operation evaluated it. In order to generate 3ACs

(i) you will need to print the appropriate 3AC in each reduce action i.e., action for each production.

(ii) And pass the temporary variable that holds the value of the sub-expression rooted at each node to the variable on the left hand side of the production. (

iii) Also you have to work with identifiers (*ident*) which can be treated similarly to the integers for this assignment. However, you **do not** have to implement a symbol table.

You are encouraged to draw the parse tree for the sample inputs and work out your translation scheme as done in the lab class for assignment #0.

Sample Input # 1

```
{ a = - b *c ; }
```

Sample output # 1

```
t1 = - b
t2 = t1 * c
a = t2
```

Sample Input # 2

```
{ a = - 5 * 0x4f ;
b = a * b - c; }
```

Sample output # 2

```
t1 = - 5
t2 = t1 *  0x4f
a = t2
t3 = a * b
t4 = t3 - c
b = t4
```

Assignment Due: 21/02 and 23/02 for the respective groups. Next week in lab I would like to see your progress in completing this assignment.