Experiment #2 Generating Three-address Code for Assignment Statements

Assignment Details:

People, even programmers, can use language with very little a priori knowledge. We all generalize from examples, guess intentions, and succeed in communicating long before the schools and textbooks round out our native tongue.

The language that we will use in this class is named X; it is inspired by the language used by Edsger Dijkstra in his book A Discipline of Programming. You can find the informal reference manual of the programming language X with some example programs **here.** In this assignment you are to generate 3AC for code segments of language X. Please read the instructions carefully.

A. Grammar of Language X

You can find the CFG of the programming language X **here**. In this assignment you are to generate intermediate code (i.e., Three-address code) for only the **assignment statements (CFG 9 and its corresponding definitions)** as an output of your compiler. For this assignment we will not have subprogram calls. Therefore the part of the grammar that pertains to this assignment is

```
assignment → vars := exprs;

vars → id  | vars , id

exprs → expr | exprs , expr

expr → disjunction

disjunction → conjunction | disjunction or conjunction

conjunction → negation | conjunction & negation

negation → relation |  ~ relation

relation → sum
             | sum <  sum
             | sum <= sum
             | sum = sum
             | sum ~= sum
             | sum>= sum
             | sum>  sum
```

sum → term | - term | sum + term | sum - term

term → factor | term * factor | term / factor

factor → true | false | integer | real | id | (expr)

Here, *id* is the token for identifiers that must start with a letter followed by only letter and digits. *integer* is any sequence of one or more digits and *real* is any sequence of digits that contain a decimal point (there should be at least one digit on both sides of the decimal point). *true* and *false* are tokens for the literals 'true' and 'false'.

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:

Complier Lab (CSE4112)

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 should choose to recognize tokens such as *Integer, real, id* in LEX, and have the operators "+", "-" etc. matched in the parser. In experiment#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. However in this assignment you will have to use them differently.

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 (*id*) which can be treated similarly as the integers for this assignment.

However, you **do not** have to implement a symbol table for this assignment. 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 experiment #0.

Sample Input # 1
 	
new := old+sgn/step;
	t1 = sgn
	t2 = t1 / step
	t3 = old + t2
	new = t3

Sample Input # 2
 	
np, x := trial*trial, 2+3;
	t1=trail
	t2=t1*trail
	t3 = 2 + 3
	np = t2
	x = t3

Sample Input # 3
 	
y:= (a+b)<c;
	t1=a
	t2=t1+b
	t3 = t2<c
	y = t3

Sample Input # 4
 	
x, z := ~(m&n), (m<b | m<c)
	t1=m
	t2 = t1 & n
	t3 = ~t2
	x =t3
	t4 = t1 < b
	t5 = t1 < c
	t6 = t4 | t5
	z = t6

Assignment Due: 21/02, 11:59pm (no further extension will be entertained)