# Design and Implementation of a Compiler
# **Final Exam**
# **Spring 2012**

## 1. True or False

a. _____ A DFA is more efficient to use for parsing than a NFA

b. _____ A DFA is more efficient to use for parsing than a recursive descent parser

c. _____ Every sentence of a context free language is a sentential form of the language

d. _____ DFA's are more efficient to create from regular expressions than NFA's

e.______ Any ambiguous grammar will produce conflicts in a LALR(1) parser

f. ______ A DFA has at most one edge leaving each state

g. ______ LL parsers cannot tolerate right-recursive grammar rules

h. ______ LR parsers cannot tolerate right-recursive grammar rules

i. ______ Every regular language is a context free language

j. ______ In an unambiguous grammar, each sentence has exactly one derivation starting from the start symbol

k. ______ A right-sentential form must end with a terminal symbol

l. _______ A left-sentential form must begin with a nonterminal symbol

m. ______ Equivalent grammars define the same language

n. ______ LR(1) grammars are more powerful than LL(1) grammars

o. ______ SLR(1) grammars are more powerful than SLR(0) grammars

p. ______ Yacc generates LL(1) parsers

q. ______ The *scope* of a variable is only the range of statements over which it is visible

r. ______ Scopes may be nested

s. ______ Does there exist a grammar that is LR(1) and is LALR(2) and is SLR(3), but which is not LALR(1)?

r. ______ Does there exist a grammar that is LR(1) and is LALR(2) and is SLR(3), but which is not LALR(1) and is not SLR(2)?

u. ______ The scope of a global variable is the entire range of program statements

v. ______ The lifetime of a static-declared local function variable is the entire range of program execution cycles

w. ______ If LR(x) is defined to mean the set of all languages that can be representable by an LR grammar with x look-ahead, then the following condition holds: $LR(0) \subset LR(1) \subset \ldots \subset LR(k) \subset LR(k+1)$

x. ______ The back end converts the standardized IR into machine code for some target machine.

y. ______ Proving the ambiguity of a general BNF grammar can be done by giving an example.

z. ______ Proving the non-ambiguity of a general BNF grammar is an NP-complete problem

aa. _____ Yacc can prove the non-ambiguity of a grammar without requiring an NP-complete algorithm

bb. _____ Yacc can handle ambiguity because it is an LALR(1) parser

cc. ______ An item in this form: "$A \rightarrow .X_1X_2\ldots X_j$" represents a prediction

dd. ______ LR(1) grammars exist for virtually all programming language grammars

**2.**

**Consider the following code:**

```
LD R0, y
ADD R0, R0, z
ST x, R0
LD R0, b
ADD R0, R0, c
ST a, R0
LD R0, a
ADD R0, R0, e
ST d, R0
```

**And consider the following peephole rules**:

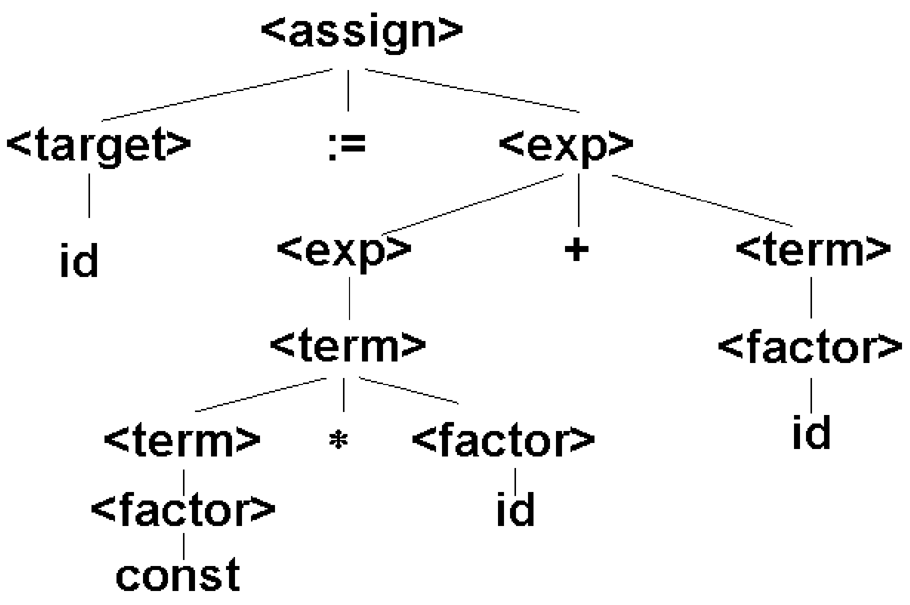
```
MUL R, R, #2→  SHL R, #1

ST addr, R  →    ST addr, R
ST addr, R

ST addr, R  →    ST addr, R
LD R, addr
```

**Write the final code, after peephole optimization.**

**3. Consider the following tree:**



a. Is it a parse tree or an abstract syntax tree?

b. circle handle in this tree

c. If it is a parse tree, draw the AST. Or, if it as an AST, draw the parse tree

**4a**. C-style comments cannot be nested; they begin with a **`/*`** and continue through the next occurrence of a **`*/`**.

For example, consider the string: "*/*/    /*  /*/*/" So, where does the comment begin and end? Well it goes form the first "/*" (which is here: */*/    /*  /*/*/), up to the first "*/" that comes after it (which is here: */*/    /*    /*/*/).    So, even though there was another "/*" inside (which was here:    */*/    /*    /*/*/), yet there is not nesting, so the second "*/" (which was here: */*/    /*    /*/*/) is NOT part of the comment.

**Using the syntax of lex, write down a regular expression that correctly matches to C-style comments; or explain clearly why it cannot be done with such a regular expression.**

**b.** Pascal comments **can** be nested; they begin with the sequence of two characters: **`(*`** and continue through the matching occurrence of the sequence of two characters **`*)`**.

For one example, the following string "*(*)    (*  )*)*)" is a "*" followed by two nested comments: "*(*)    (*  )*)*)".

**Using the syntax of lex, write down a regular expression that correctly matches to Pascal-style comments; or explain clearly why it cannot be done with such a regular expression.**

**5**. Consider the following program:

```
int w=1, x=1,y=1,z=1;

void main()
{    int x=2;y=2;

     void f2(int w, int z)
     {    int x=3;z=3;

          void f3()
          {    int w=4;z=4;
               printf( “%d%d%d%d\n”,w,x,y,z);
          }//end of f3()

          void f4()
          {    int w=5;
               {    int w =6;
                    z=2;
               }
               printf( “%d%d%d%d\n”,w,x,y,z);
          } //end of f4()

          // body of f2() begins here
          f3();
          f4();
          printf( “%d%d%d%d\n”,w,x,y,z);
          z = 8;
     } //end of f2()

     // body of main() begins here
     printf( “%d%d%d%d\n”,w,x,y,z);
     f2(w,z);
     printf( “%d%d%d%d\n”,w,x,y,z);
} //end of main()
```

**a. List the output that you would expect to see when this program is run.**

**b. Draw the contents of the stack (in the way described in the lecture, and with the stack growing upwards) at the point where f4() is about to execute its return. In your drawing, it should be clear where each function’s activation record (ie, stack frame) is, and it should be clear where each variable is.**

Draw your stack in this space

6. Consider the following sets of items. Uppercase letters are nonterminals; lowercase letters are terminals.

`s0:` $\{A \rightarrow \bullet B, \quad B \rightarrow a \bullet A, \quad C \rightarrow Da \bullet cB, \quad D \rightarrow Da \bullet cB\}$

`s1:` $\{A \rightarrow B \bullet, \quad B \rightarrow aB \bullet A, \quad C \rightarrow D \bullet cB, \quad D \rightarrow B \bullet \}$

`s2:` $\{E \rightarrow F \bullet E, \quad E \rightarrow xyF \bullet, \quad C \rightarrow F \bullet cB, \quad D \rightarrow yF \bullet, \quad E \rightarrow \bullet x\}$

`s3:` $\{E \rightarrow F \bullet F, \quad E \rightarrow xyF \bullet F, \quad C \rightarrow F \bullet cB, \quad D \rightarrow yF \bullet, \quad F \rightarrow \bullet Ex\}$

`s4:` $\{E \rightarrow \bullet F, \quad E \rightarrow xyF \bullet x, \quad C \rightarrow F \bullet cB, \quad D \rightarrow yF \bullet, \quad F \rightarrow \bullet Ex\}$

a. Which of those sets *cannot* be a LR(0) parser state obtained from a CFG using the canonical LR(0) construction?

b. For those sets above that can be LR(0) parser states: With no lookahead, which states will have shift-reduce conflicts?

c. For those sets above that can be LR(0) parser states: With no lookahead, which states will have reduce-reduce conflicts?

d. No SLR(1) parser state ever has a “shift-shift” conflict. With reference to the basic ideas behind what LR parser states are for, briefly but clearly explain why.

7. Consider the following augmented grammar, and partially filled SLR(1) tables. Fill in the tables for states 0 and 5. (All other states have already been filled in appropriately.)

| | action | | | goto | |
|---|---|---|---|---|---|
| | a | b | $ | S | A |
| 0 | | | | | |
| 1 | s4 | | acc | | |
| 2 | r1 | | r1 | | |
| 3 | r3 | | r3 | | |
| 4 | | s3 | | | 5 |
| 5 | | | | | |

```
S' ::= S

(1)   S ::= A

(2)   S ::= SaA

(3)   A ::= b
```

8. Consider the following SLR(1) table and grammar

| | action | | | | | | goto | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | acc | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

```
(1) E ::= E + T
(2) E ::= T
(3) T ::= T * F
(4) T ::= F
(5) F ::= (E)
(6) F ::= id
```

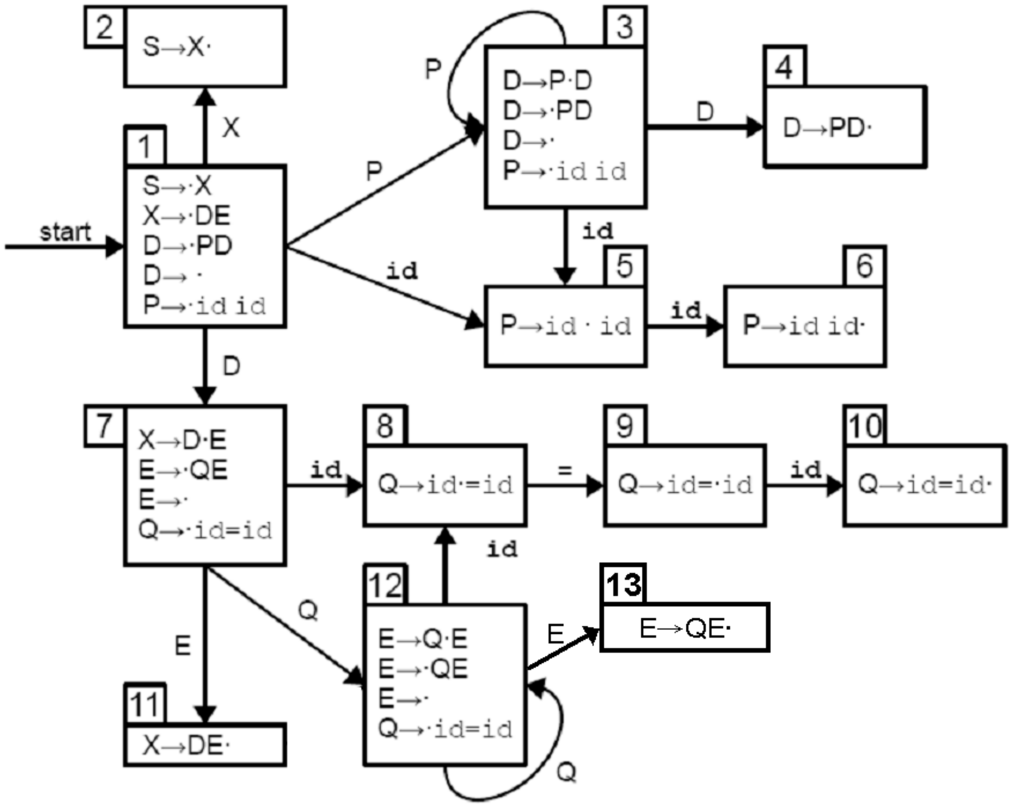
The input string to parse is “id + (id) * id”. Show the stack and input at each step of the parse. For each step, clearly indicate the action: for shifts, indicate what symbol was shifted and the new state, and for reductions, indicate which rule was used and the new state.

| stack | input | action |
|---|---|---|
|  | id + (id) * id $ |  |

9. In this problem, we will explore how several different LR parsing algorithms handle or fail to handle a particular grammar:

$S \rightarrow X$
$X \rightarrow DE$
$D \rightarrow PD$
$D \rightarrow \varepsilon$
$E \rightarrow QE$
$E \rightarrow \varepsilon$
$P \rightarrow \text{id id}$
$Q \rightarrow \text{id} = \text{id}$

Below is an LR(0) automaton for this language.
We've numbered the states for you.
The terminals in this grammar are:
id = $, where $ is the end-of-input symbol.



2 S→X·
1 S→·X X→·DE D→·PD D→· P→·id id
start
X
3 D→P·D D→·PD D→· P→·id id
P
4 D→PD·
D
id
5 P→id · id
6 P→id id·
7 X→D·E E→·QE E→· Q→·id=id
8 Q→id·=id
=
9 Q→id=·id
10 Q→id=id·
11 X→DE·
E
Q
12 E→Q·E E→·QE E→· Q→·id=id
13 E→QE·

a. This grammar is not LR(0). Why not?

b. Is this grammar SLR(1)? If so, why? If not, why not?

c. Create the augmented LALR(1) grammar for this language Showing your steps and showing the result.

d. Compute the FOLLOW sets for each nonterminal in the LALR(1)-augmented grammar. Show your result below (but the work is not necessary.

e. Is this grammar LALR(1)? If so, use your answers from (c) and (d) to prove why. If not, use your answers from (c) and (d) to prove why not. (Even if you already know for a fact whether this grammar is LALR(1) or not, you must use the results from the last two problems to back up your claim.)