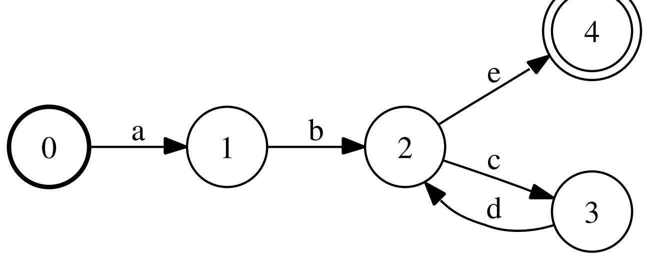
CMPT 379 - Sample Questions for Final Exam Preparation

(1) Regular and Context-free Grammars:

a. Consider the following DFA:



$$\begin{tikzcd} 0 \arrow[r, "a"] & 1 \arrow[r, "b"] & 2 \arrow[r, "e"] \arrow[r, bend right, "c"'] & 4 \\ & & & 3 \arrow[ul, bend left, "d"] \end{tikzcd}$$

Provide a regular expression for the regular language generated by this DFA.

b. You are given the following ordered list of token definitions:

TOKEN_A	$(ab)^*a$
TOKEN_B	$(ab)^*a(ca)^*$
TOKEN_C	$bab(bab)^*$
TOKEN_D	$a^*ba(ba)^*$

Provide the tokenized output for the input string *abacabababa* using the greedy longest match lexical analysis method.

c. The following CFG generates a regular language. Provide a regular expression that generates the same language as this CFG.

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow c \mid \epsilon \\ B &\rightarrow cbB \mid ca \end{aligned}$$

d. Are the following two context-free grammars equivalent? That is, do the two grammars generate the same language. Give a short precise reason for your answer.

G_1 :
$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow c \mid \epsilon \\ B &\rightarrow cbB \mid ca \end{aligned}$$

G_2 :
$$\begin{aligned} S &\rightarrow cAa \\ A &\rightarrow cB \mid B \\ B &\rightarrow bcB \mid \epsilon \end{aligned}$$

e. Each of the following C code fragments has a lexical or syntax error (or warning), or both. For each example, indicate if the compiler would generate a *lexical* and/or *syntax* error (or warning).

1. `void main() { char c = 'ab'; }`
2. `void main() { char a[] = "ab ; }`
3. `void main() { int x = 3 int y = 4; }`
4. `void main() { intx = 3; }`
5. `void main() { char a[] = "ab; printf("\n"); }`

(2) For alphabet Σ let us denote the *derivative* of a regular expression R with respect to a, where $a \in \Sigma$ as $\frac{dR}{da}$ or equivalently as $D_a[R]$. For any regular expression R, $D_a[R]$ is defined recursively using the following rules:

$$\begin{aligned} D_a[a] &= \epsilon && (1) \\ D_a[b] &= \phi, \text{ for } b \in \Sigma, b \neq a, \text{ or } b = \epsilon \text{ or } b = \phi && (2) \end{aligned}$$

If P and Q are regular expressions, then:

$$\begin{aligned} D_a[P \mid Q] &= D_a[P] \mid D_a[Q] && (3) \\ D_a[PQ] &= D_a[P]Q \mid \delta(P)D_a[Q] && (4) \\ D_a[P^*] &= D_a[P]P^* && (5) \end{aligned}$$

where $\delta(P) = \epsilon$, if $\epsilon \in P$ and $\delta(P) = \phi$, if $\epsilon \notin P$.
The empty set ϕ, is different from the empty string ϵ, and has the following properties:

$$\begin{aligned} \phi \mid R &= R \mid \phi &= R \\ \phi R &= R\phi &= \phi \end{aligned}$$

Also, we define derivative $D_s[R]$ of regular expression R with respect to a sequence of symbols $s = a_1, a_2, \ldots, a_r$ as:

$$D_s[R] = D_{a_1,\ldots,a_r}[R] = D_{a_r}[D_{a_1,\ldots,a_{r-1}}[R]]$$

A sequence s of zero length is written as: $D_\epsilon[R] = R$.
The intuition behind the notion of a derivative of a regular expression R with respect to symbol a is that it provides us with a regular expression R' such that the language of R', $L_{R'} = \{y \mid ay \in L_R\}$.
Let $R = (0 \mid 1)^*1$, the derivative $D_a[R]$ for any symbol $a \in \Sigma$, where $\Sigma = \{0, 1\}$ is:

$$\begin{aligned} D_a[R] &= D_a[(0 \mid 1)^*1] \\ &= D_a[(0 \mid 1)^*]\,1 \mid D_a[1], \text{ since } \epsilon \in (0 \mid 1)^* \quad \text{using (4)} \\ &= D_a[0 \mid 1]\,(0 \mid 1)^*\,1 \mid D_a[1] \quad \text{using (5)} \\ &= (D_a[0] \mid D_a[1])\,(0 \mid 1)^*\,1 \mid D_a[1] \quad \text{using (3)} \end{aligned}$$

Putting in $a = 0$, we get:

$$D_0[R] = (\epsilon \mid \phi)(0 \mid 1)^*1 \mid \phi \quad \text{using (1) and (2)}$$

This expressions can be simplified with identities for ϵ and ϕ to get:

$$D_0[R] = (0 \mid 1)^*1 = R$$

We know from our previous definition that $D_\epsilon[R] = R = D_0[R]$. Provide the simplified expressions for the following derivatives of $R = (0 \mid 1)^*1$:

a. $D_1[R]$

b. $D_{10}[R]$

c. $D_{11}[R]$

d. For R is the number of derivatives is finite or infinite?

(3) LR(0) Parsing: Consider the following context-free grammar G.

$$\begin{aligned} S' &\rightarrow S && (1)\\ S &\rightarrow iSeS && (2)\\ S &\rightarrow iS && (3)\\ S &\rightarrow a && (4) \end{aligned}$$

a. Provide the LR(0) itemsets for grammar G. (you do not need to draw the automata)

b. Provide the LR parsing table based on the above LR(0) states.

c. State if there are any conflicts in the parsing table and provide a method for resolving the conflicts, if any.

d. Provide the parsing actions taken on the input *iiaea* using your solution to question (3c). Show the remaining input and the contents of the stack for each step.

e. Consider the following grammar G':

$$\begin{aligned} S' &\rightarrow S\\ S &\rightarrow M \mid U\\ M &\rightarrow iMeM \mid a\\ U &\rightarrow iS \mid iMeU \end{aligned}$$

What is the relationship of this grammar to G above?

(4) Provide a grammar that satisfies all conditions in each question below. You have to provide a brief reason as to why the grammar satisfies each condition.

a. Provide a grammar G_0 that is not LL(1) or LL(2).

b. Provide a grammar G_1 that is not LR(0), is SLR(1), is LL(1) and is LR(1).

c. Provide a grammar G_2 that is not LR(0), is not SLR(1), is LL(1) and is LR(1).

d. Provide a grammar G_3 that is not LL(1), is LL(3) and is LR(0).

e. Provide a grammar G_4 that is not LR(0), is not SLR(1), is not LL(1), and is LR(1)

(5) Let the synthesized attribute *val* give the decimal floating point value of the binary number generated by S in the following grammar.

$$\begin{aligned} S &\rightarrow L \,.\, L \mid L \\ L &\rightarrow L\, B \mid B \\ B &\rightarrow \texttt{0} \mid \texttt{1} \end{aligned}$$

For example, on input `101.101`, the integer part of the number is

$$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

and the fractional part is

$$1 \times \frac{1}{2^1} + 0 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} = \frac{5}{8}$$

providing the value of $5\frac{5}{8} = 5.625$ for the synthesized attribute $S.val$.
Consider the following attribute grammar for a syntax-directed definition which determines $S.val$ for all strings in the language.

Rules	Syntax-directed definition
$S \rightarrow L$	\$1.in = (0, 1); # inherited attr *in* is a tuple: (first, second) \$0.val = \$1.val.first; # synthesized attr *val* for *L* is also a tuple

$S \rightarrow L\,.\,L$	\$1.in = (0, 1);
	\$3.in = (0, −1);
	\$0.val = \$1.val.first + \$3.val.first;

$L \rightarrow L\,B$	\$1.in = (\$0.in.first+1 , \$0.in.second);
	\$2.in = (\$0.in.second < 0) ? −(\$1.val.second+1) : \$0.in.first;
	\$0.val = (\$1.val.first + \$2.val , \$1.val.second+1);

$L \rightarrow B$	\$1.in = (\$0.in.second < 0) ? −1 : \$0.in.first;
	\$0.val = (\$1.val, 2);

$B \rightarrow 0$	\$0.val = 0;
$B \rightarrow 1$	\$0.val = $2^{0.in}$;

a. Draw the parse tree for the input `101.101` and decorate the nodes with the inherited and synthesized attributes needed to determine the value of $S.val$.

b. Provide a new attribute grammar where you have eliminated left recursion from the grammar. You *must* eliminate left recursion in the following way: for a left recursive rule schema of the type $A \rightarrow A\alpha \mid \beta$ convert it to $A \rightarrow \beta A', A' \rightarrow \alpha A' \mid \epsilon$.

   Hint: Solving question (5c) at the same time may be helpful.

c. Draw the parse tree for the input `101.101` and decorate the nodes with the inherited and synthesized attributes using your new attribute grammar without left recursion.

(6) Consider the following expression grammar.

$$
\begin{array}{rcl}
E & \rightarrow & E \text{ `}\texttt{+}\text{'} \; T \\
 & | & T \\
T & \rightarrow & T \text{ `}\texttt{*}\text{'} \; F \\
 & | & F \\
F & \rightarrow & \texttt{exp} \text{ `}\texttt{(}\text{'} \; E \text{ `}\texttt{)}\text{'} \\
 & | & \texttt{ln} \text{ `}\texttt{(}\text{'} \; E \text{ `}\texttt{)}\text{'} \\
 & | & \text{`}\texttt{-}\text{'} \; F \\
 & | & \text{`}\texttt{x}\text{'} \\
 & | & \texttt{c}
\end{array}
$$

We assume a lexical analyzer that provides the tokens we need. For instance, `c` is an integer constant token. Note that `exp` is the exponential function, i.e. `exp(x)` is e^x and `ln` is the natural logarithm, i.e. `ln(x)` is $ln(x)$ also written as $log_e(x)$.

a. Provide a L-attributed syntax directed definition that computes the derivative of an input expression. Explain each attribute used in your attribute grammar.

D[input string]	output string = derivative(input string)
$D[c]$	0
$D[x]$	1
$D[x + c]$	1
$D[E_1 + E_2]$	$D[E_1] + D[E_2]$
$D[-E]$	$-D[E]$
$D[c * E]$	$c * D[E]$
$D[E_1 * E_2]$	$E_1 * D[E_2] + E_2 * D[E_1]$
$D[exp(x)]$	$exp(x)$
$D[ln(x)]$	$1/x$
$D[f(E)]$	$D[E] * f'(E)$, f' is the derivative of f
	if $f(E)$ is $exp(E)$, $f'(E)$ is $exp(E)$
	if $f(E)$ is $ln(E)$, $f'(E)$ is $1/E$

b. Using your syntax-directed definition provide the derivative for the input string shown below. Provide the parse tree for the input string and the attribute values at each node in the tree.

```
exp(2 * x + 4)
```

c. *(optional; no marks)* Extend your syntax-directed definition so that it can handle second derivatives, and third derivatives. For example, the derivative of `x * x + 2 * x` will be `x + x + 2`, and the second derivative (derivative of the derivative) will be `2` and the third derivative will be `0`.

(7) The following attribute grammar implements code-generation for a fragment of a programming language.

Rules	Syntax-directed definition

$P \rightarrow S$	\$1.next = newlabel();
	\$0.code = \$1.code + label(\$1.next);

$S \rightarrow$ **assign**	\$0.code = ”assign”;

$S \rightarrow$ **while ‘(’** B **‘)’** S	begin = \$5.next = newlabel();
	true = \$3.true = newlabel();
	\$3.false = \$0.next;
	instr = ”goto” begin;
	\$0.code = label(begin) + \$3.code + label(true) + \$5.code + instr;

$S \rightarrow S\ S$	firstStmt = \$1.next = newlabel();
	\$2.next = \$0.next;
	\$0.code = \$1.code + label(firstStmt) + \$2.code;

$B \rightarrow$ **true**	?
$B \rightarrow$ **false**	?

We assume that the parser interprets the statement concatenation rule $S \rightarrow S\ S$ unambiguously as being right-associative, and that the while statement takes precedence over statement concatenation.

The operator + simply concatenates three-address instructions and labels. We assume that *newlabel()* creates a new label each time it is called (returning `L1`, `L2`, ...), and that *label(L)* attaches label *L* to the next three-address instruction that is generated, e.g. *label(L1)* would result in `L1:` generated in the output.

a. The attribute **code** in the above syntax directed definition is a *synthesized* attribute. List all the *inherited* attributes.

b. Is the syntax-directed definition *L-attributed*?

c. Add the syntax directed definition for the rules $B \rightarrow$ **true** and $B \rightarrow$ **false** .

d. Provide the output three-address instructions for the input:

```
while (true) assign assign
```

e. We add a new rule to the grammar: $S \rightarrow$ **do** S **while ‘(’** B **‘)’** . Provide the syntax directed definition for this new rule.

f. Provide the code generation output for:

```
do assign assign while (false)
```

(8) Consider the CFG G with S' as the start symbol:

$$\begin{array}{rcl} S' & \rightarrow & S \mid \epsilon \\ S & \rightarrow & T \mid (N , C) \\ C & \rightarrow & C , S \mid S \\ T & \rightarrow & a \mid b \mid c \\ N & \rightarrow & x \mid y \mid z \end{array}$$

a. For each of the following strings, write down `true` if the string is in the language $L(G)$ generated by G, `false` otherwise.

   1. `y`
   2. `c`
   3. `(x)`
   4. `(x,y)`
   5. `(z,a,b,a,b,c)`
   6. `(x,a,(y,b),c)`
   7. `(x,(y,a),(z,b))`
   8. `(x,(x,(x,(x,a))`

b. Write down the leftmost derivation for the string `(x,(y,b))`

c. Write down the rightmost derivation for the string `(x,(y,b))`

d. Draw the parse tree produced by G for the string `(x,(y,b))`

e. Write down a concise description in English of the language $L(G)$.

f. Eliminate left-recursion from the grammar G to create a new grammar G'.

g. Is G' an LL(1) grammar? Justify your answer with a precise but brief reason.

(9) Consider the augmented CFG G with S' as the start symbol:

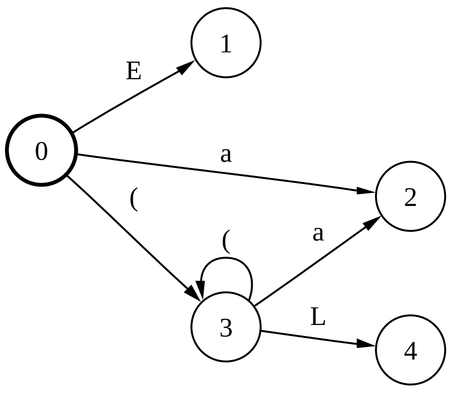
$$\begin{array}{rcl} S' & \rightarrow & E \quad (1)\\ E & \rightarrow & a \quad (2)\\ E & \rightarrow & (\,L\,) \quad (3)\\ L & \rightarrow & \epsilon \quad (4)\\ L & \rightarrow & E\,L \quad (5)\end{array}$$

A certain lazy professor starts writing the LR(0) itemsets for G in order to create an action/goto table for LR parsing. After writing down five itemsets, the associated LR(0) automaton and the action/goto table, the professor is too tired to continue. You have to finish the job.

The incomplete list of LR(0) itemsets is shown below:

$$\begin{array}{llcl} 0: & S' & \rightarrow & \bullet E\\ & E & \rightarrow & \bullet a\\ & E & \rightarrow & \bullet (L)\\ \\ 1: & S' & \rightarrow & E\bullet\\ \\ 2: & E & \rightarrow & a\bullet \end{array}
\qquad
\begin{array}{llcl} 3: & E & \rightarrow & (\bullet L)\\ & L & \rightarrow & \epsilon\bullet\\ & L & \rightarrow & \bullet EL\\ & E & \rightarrow & \bullet a\\ & E & \rightarrow & \bullet (L)\\ \\ 4: & E & \rightarrow & (L\bullet) \end{array}$$

The incomplete LR(0) automaton constructed for the above itemsets is shown below:



The incomplete LR(0) action/goto table created from the automaton is shown below:

	(	)	a	$	E	L
0	shift 3	error	shift 2	error	goto 1	
1	error	error	error	accept		
2	reduce(2)	reduce(2)	reduce(2)	reduce(2)		
3	shift 3 reduce(4)	reduce(4)	shift 2 reduce(4)	reduce(4)	goto 6	goto 4
4	error	shift 5	error	error		
5						
6						
7						

Your task for this question has four components:

a. Add the remaining three itemsets (use the numbers 5, 6, 7 for the new itemsets as indicated by entries in the action/goto table), and then
b. Complete the LR(0) automaton by adding the three new states including all the necessary transitions, and then
c. Complete the action/goto table entries (remember to indicate error entries), and finally
d. Indicate whether the grammar *G* is a LR(0) grammar

(10) Consider the following grammar G:

$$S \rightarrow aSa \mid bSb \mid aa \mid bb$$

a. Is the CFG G an LL(1) grammar? Provide a reason for your answer.

b. Based on inspecting the set of possible viable prefixes for grammar G, is G an LR(1) grammar? Provide an example viable prefix or a comparison between two candidate viable prefixes to support your answer.

c. Consider a slightly modified version of grammar G. Let's call it G':

$$S \rightarrow aSa \mid bSb \mid \epsilon$$

Does this modified grammar G' generate the same language as the original grammar G? Provide a reason for your answer.

d. Is G' an LL(1) grammar? Provide a reason for your answer.

e. Consider a slightly modified version of grammar G. Let's call it G':

$$S \rightarrow aSa \mid bSb \mid \epsilon$$

Is G' an LL(1) grammar? Provide a reason for your answer.

(11) For each grammar below indicate whether or not it is an LL(1) grammar, an LR(0) grammar, an SLR(1) grammar and/or an LR(1) grammar. *For each grammar you have to provide four distinct yes/no answers*. Provide a short reason for each yes or no answer. Note that each grammar below has production rules separated by commas.

a. $S \rightarrow A \mid B \, , \; A \rightarrow c \mid dAd \, , \; B \rightarrow c \mid dBd$

b. $S \rightarrow A \mid B \, , \; A \rightarrow c \mid dAd \, , \; B \rightarrow e \mid dBd$

c. $S \rightarrow aA \mid bB \, , \; A \rightarrow c \mid dAd \, , \; B \rightarrow c \mid dBd$

d. $S \rightarrow AaAb \mid BbBa \, , \; A \rightarrow \epsilon \, , \; B \rightarrow \epsilon$

(12) Consider the following three-address code (TAC) program:

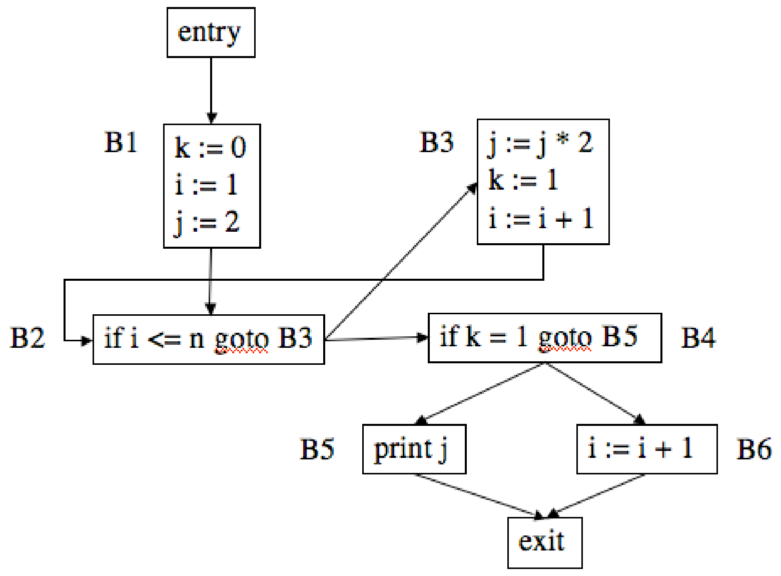
```
    i := m - 1
    j := n
    t1 := 4 * n
    v := A[ t1 ]
L1: i := i + 1
    t2 := 4 * i
    t3 := A[ t2 ]
    if t3 < v goto L1
L2: j := j - 1
    t4 := 4 * j
    t5 := A[ t4 ]
    if t5 > v goto L2
    if i >= j goto L3
    t6 := 4 * i
    x := A[ t6 ]
    t7 := 4 * i
    t8 := 4 * j
    t9 := A[ t8 ]
    A[ t7 ] := t9
    t10 := 4 * j
    A[ t10 ] := x
    goto L1
L3: t11 := 4 * i
    x := A[ t11 ]
    t12 := 4 * i
    t13 := 4 * n
    t14 := A[ t13 ]
    A[ t12 ] := t14
    t15 := 4 * n
    A[ t15 ] := x
```

a. Construct the control flowgraph for the above TAC program.

b. Perform *local* common subexpression elimination (i.e. only eliminate common subexpressions within each basic block) and provide the revised control flowgraph.

c. The instruction `t4 := 4*j` is repeatedly executed inside an inner loop (as can be seen in the control flow graph). Analyze the change in values in `t4` and reduce the strength of this instruction by replacing the multiplication operation with the cheaper operation (such as addition or subtraction). In order to do this, you can add new instructions using expensive operations like multiplication as long as these instructions are added outside the inner loop.

d. For the following flowgraph construct the flowgraph in minimal Static Single Assignment (SSA) form. A *minimal* SSA form has no redundant static variable definitions.


entry
B1
k := 0
i := 1
j := 2
B3
j := j * 2
k := 1
i := i + 1
B2
if i <= n goto B3
if k = 1 goto B5
B4
B5
print j
i := i + 1
B6
exit