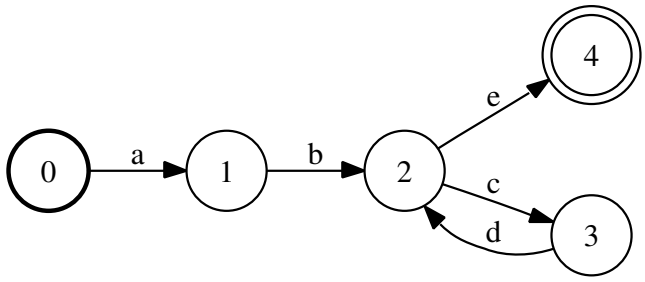
CMPT 379 - Sample Questions for Final Exam Preparation

(1) Regular and Context-free Grammars:

a. Consider the following DFA:



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

Answer: `ab(cd)*e`

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 *abacababababa* using the greedy longest match lexical analysis method.

Answer:

First match:

`TOKEN_A`	*aba*
`TOKEN_B`	*abaca*
`TOKEN_C`	no match
`TOKEN_D`	*aba*

Second match:

`TOKEN_A`	no match
`TOKEN_B`	no match
`TOKEN_C`	*bab*
`TOKEN_D`	*bababa*

Output:

`TOKEN_B`	*abaca*
`TOKEN_D`	*bababa*

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

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

Answer:
`c?(cb)*ca`

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 :

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

Answer:
`c?(cb)*ca`
is equivalent to
`cc?(bc)*a`

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'; }`

   Answer: lexical error/warning: multi-character character constant

2. `void main() { char a[] = "ab ; }`

   Answer:
   lexical error: missing terminating ” character
   syntax error: parse error at end of input

3. `void main() { int x = 3 int y = 4; }`

   Answer: syntax error: parse error before ’int’

4. `void main() { intx = 3; }`

   Answer: syntax error (symbol table error): ’intx’ undeclared

5. `void main() { char a[] = "ab; printf("\n"); }`

   Answer:
   syntax error: parse error before ’n’
   lexical error: missing terminating ” character

(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:

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

If P and Q are regular expressions, then:

$$D_a[P \mid Q] = D_a[P] \mid D_a[Q] \quad (3)$$
$$D_a[PQ] = D_a[P]Q \mid \delta(P)D_a[Q] \quad (4)$$
$$D_a[P^*] = D_a[P]P^* \quad (5)$$

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:

$$\phi \mid R = R \mid \phi = R$$
$$\phi R = R\phi = \phi$$

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. $D_1[R]$

Answer:

$$D_1[R] = (\phi \mid \epsilon)(0 \mid 1)^*1 \mid \epsilon = (0 \mid 1)^*1 \mid \epsilon = R \mid \epsilon$$

b. $D_{10}[R]$

Answer:

$$D_{10}[R] = D_0[D_1[R]] = D_0[R \mid \epsilon] = D_0[R] \mid D_0[\epsilon] = R \mid \phi = R$$

Note that:

$$D_{10}[R] = D_0[R] = D_\epsilon[R]$$

c. $D_{11}[R]$

Answer:

$$D_{11}[R] = D_1[D_1[R]] = D_1[R \mid \epsilon] = D_1[R] \mid D_1[\epsilon] = D_1[R] \mid \phi = D_1[R] = R \mid \epsilon$$

Note that:

$$D_{11}[R] = D_1[R]$$

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

Answer: For R, the number of derivatives is finite. Any additional symbols added to the end of strings 10 or 11 will not result in any new derivatives. A surprising link between derivatives and DFAs can be made: Let $D_\epsilon[R]$ be the start state q_0 in our DFA, on symbol 0 we take a transition to $D_0[R]$ which we have already determined to be equal to $D_\epsilon[R]$ and hence the transition leads to q_0. On symbol 1 we take a transition to $D_1[R]$ which is a new state q_1. Following this construction, we have a transition on 0 from q_1 to state corresponding to $D_{10}[R]$ which is q_0. Similarly, we also infer a transition from q_1 to $D_{11}[R] = q_1$ on symbol 1. If $\epsilon \in D_a[R]$ then the state associated with $D_a[R]$ is a final state. In practice, we can add a special symbol # as the end of input marker in R in order to determine the final state.

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

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

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

Answer: The LR(0) states for G are shown below:

I_0 :
$S' \rightarrow \bullet S$
$S \rightarrow \bullet iS\,eS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_1 :
$S' \rightarrow S \bullet$

I_2 :
$S \rightarrow i \bullet S\,eS$
$S \rightarrow i \bullet S$
$S \rightarrow \bullet iS\,eS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_3
$S \rightarrow a \bullet$

I_4 :
$S \rightarrow iS \bullet eS$
$S \rightarrow iS \bullet$

I_5 :
$S \rightarrow iS\,e \bullet S$
$S \rightarrow \bullet iS\,eS$
$S \rightarrow \bullet iS$
$S \rightarrow \bullet a$

I_6 :
$S \rightarrow iS\,eS \bullet$

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

Answer:

state	action				goto
	i	*e*	*a*	$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r(4)		r(4)	
4		s5		r(3)	
5	s2		s3		6
6		r(2)		r(2)	

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

Answer: There is a shift/reduce conflict in state 4: s5 vs. r(3). Resolve conflict by always performing s5 when e is the next symbol. This strategy ensures that e is attached to the closest preceding i.

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.

Answer:

	action	stack	input
1:	s0	0	*iiaea*\$
2:	s2	02	*iaea*\$
3:	s2	022	*aea*\$
4:	s3	0223	*ea*\$
5:	r(4) = $S \rightarrow a$; goto(2, S) = 4	0224	*ea*\$
6:	s5; conflict resolution	02245	*a*\$
7:	s3	022453	\$
8:	r(4) = $S \rightarrow a$; goto(5, S) = 6	022456	\$
9:	r(2) = $S \rightarrow iS\,eS$; goto(2, S) = 4	024	\$
10:	r(3) = $S \rightarrow iS$; goto(0, S) = 1	01	\$
11:	acc	01	\$

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?

Answer: G' is the unambiguous version of G.

(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).

Answer:

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

FIRST(aSa) intersected with FIRST(aa) is non-empty.

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

Answer:

$$S' \rightarrow S, S \rightarrow aSb \mid \epsilon$$

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

Answer:

$$S' \rightarrow S, S \rightarrow AaAb \mid BbBa, A \rightarrow \epsilon, B \rightarrow \epsilon$$

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

Answer:

$$S \rightarrow abc \mid abd$$

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

Answer:

$$\begin{aligned} S' \quad &\rightarrow \quad S \text{`=' } \mathbf{id} \mid S \text{`(' } \mathbf{id} \text{ `)'} \\ S \quad &\rightarrow \quad S \text{`.' } \mathbf{id} \mid \epsilon \end{aligned}$$

$$S \rightarrow cAaAb \mid cBbBa, A \rightarrow \epsilon, B \rightarrow \epsilon$$

$$S \rightarrow L = R \mid R, R \rightarrow R* \mid L, L \rightarrow \mathbf{id}$$

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

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

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 $S.val$.

Answer:

```
(S                    # val = (2^2 + 2^0) + (2^-1 + 2^-3) = 5.625
   (L                 # in = (0, 1);  val = (2^2 + 2^0, 4)
     (L               # in = (1, 1);  val = (2^2, 3)
       (L             # in = (2, 1);  val = (2^2, 2)
         (B 1))       # in = 2;  val = 2^2
       (B 0))         # in = 1;  val = 0
     (B 1))           # in = 0;  val = 2^0
   .
   (L                 # in = (0, -1);  val = (2^-1 + 2^-3, 4)
     (L               # in = (1, -1);  val = (2^-1, 3)
       (L             # in = (2, -1);  val = (2^-1, 2)
         (B 1))       # in = -1;  val = 2^-1
       (B 0))         # in = -2;  val = 0
     (B 1)))          # in = -3;  val = 2^-3
```

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.

Answer:

Rules	Syntax-directed definition
$S \rightarrow L$	\$1.in = (0, 1); \$0.val = \$1.val.first;
$S \rightarrow L\ .\ L$	\$1.in = (0, 1); \$3.in = (0, -1); \$0.val = (\$1.val.first + \$3.val.first, 0);
$L \rightarrow B\ L'$	\$1.in = \$0.in.second $*$ \$2.val.second; \$2.in = (\$0.in.first+1, \$0.in.second); \$0.val = (\$2.val.first + \$1.val , \$2.val.second + \$0.in.second);
$L' \rightarrow B\ L'$	\$1.in = \$0.in.second $*$ \$2.val.second; \$2.in = (\$0.in.first+1, \$0.in.second); \$0.val = (\$2.val.first + \$1.val, \$2.val.second + \$0.in.second);
$L' \rightarrow \epsilon$	\$0.val = (\$0.in.second < 0) ? (0, \$0.in.first) : (0,0);
$B \rightarrow 0$	\$0.val = 0;
$B \rightarrow 1$	\$0.val = $2^{0.in}$;

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.

Answer:

```
(S                    # val = (2^2 + 2^0) + (2^-1 + 2^-3) = 5.625
   (L                 # in = (0, 1);  val = (2^2 + 2^0, 3)
     (B 1)            # in = 2;  val = 2^2
     (L'              # in = (1, 1);  val = (2^0, 2)
        (B 0)         # in = 1;  val = 0
        (L'           # in = (2, 1);  val = (2^0, 1)
           (B 1)      # in = 0;  val = 2^0
           (L' eps)))) # in = (3, 1);  val = (0, 0)
   .
   (L                 # in = (0, -1);  val = (2^-1 + 2^-3, 0)
     (B 1)            # in = -1;  val = 2^-1
     (L'              # in = (1, -1);  val = (2^-3, 1)
        (B 0)         # in = -2;  val = 0
        (L'           # in = (2, -1);  val = (2^-3, 2)
           (B 1)      # in = -3;  val = 2^-3
           (L' eps)))) # in = (3, -1);  val = (0, 3)
```

(6) Consider the following expression grammar.

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

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 $1/E$

Answer: The following solution also simplifies the expressions. The simplification is not required but it is nice to have.

Production	Semantic Rule
$E \rightarrow E$ **'+'** T	dv := simplify(+, 1.dv, 3.dv); ex := '1.ex + 2.ex';
$E \rightarrow T$	dv := 1.dv; ex := 1.ex;

$T \rightarrow T$ **'*'** F	t1 := simplify(*, 1.ex, 3.dv);
	t2 := simplify(*, 3.ex, 1.dv);
	dv := simplify(+, t1, t2); ex := '1.ex * 2.ex';

$T \rightarrow F$	dv := 1.dv; ex := 1.ex;
$F \rightarrow$ `exp` **'('** E **')'**	dv := simplify(*, 3.dv, 'exp(3.ex)'); ex := 'exp(3.ex)';
$F \rightarrow$ `ln` **'('** E **')'**	dv := simplify(*, 3.dv, '1 / 3.ex'); ex := 'ln(3.ex)';
$F \rightarrow$ **'-'** F	0.dv := '− 1.dv'; 0.ex = '− 1.ex';
$F \rightarrow$ **'x'**	0.dv = 1; 0.ex = x;
$F \rightarrow$ `c`	0.dv := 0; 0.ex := c.lexval;

Pseudo-code to simplify an expression:

```
string simplify (string op, string a, string b)
{
  if (isInteger(a) and isInteger(b)) {
    if (op eq '+') return string(int(a) + int(b));
    if (op eq '*') return string(int(a) * int(b));
  }
  if (op eq '+') {
    if (a eq '0') return b;
    if (b eq '0') return a;
  }
  if ((op eq '*') and ((a eq '0') or (b eq '0'))) return '0';
  return 'a op b';
}
```

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)`

Answer:

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

```
(E                    # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
  (T                  # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
    (F                # dv = 2 * exp(2 * x + 4), ex = exp(2 * x + 4)
      exp
      \(
      (E              # dv = 2 + 0 = 2, ex = 2 * x + 4
         (E           # dv = 2, ex = 2 * x
           (T         # dv = 2 * 1 + x * 0 = 2, ex = 2 * x
             (T       # dv = 0, ex = 2
               (F 2)) # dv = 0, ex = 2
             *
             (F x)))  # dv = 1, ex = x
         +
         (T           # dv = 0, ex = 4
           (F 4)))    # dv = 0, ex = 4
      \))))
```

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.

Answer: next, true, false.

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

Answer: Yes.

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

Answer:

Rules	Syntax-directed definition
$B \rightarrow$ **true**	\$0.code = "goto" \$0.true;
$B \rightarrow$ **false**	\$0.code = "goto" \$0.false;

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

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

Answer:

```
L3:           # label(begin)
    goto L4   # 3.code from B -> true
L4:           # label(true)
    assign    # 5.code from S -> assign
    goto L3   # instr
L2:           # 1.code + label(firstStmt)
    assign    # 2.code from S -> assign
L1:           # P -> S
```

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

Answer:
begin = newlabel();
instr = "goto" begin;
check = $2.next = newlabel();
true = $5.true = newlabel();
$5.false = $0.next;
$0.code = label(begin) + $2.code + label(check) + $5.code + label(true) + instr;

f. Provide the code generation output for:

`do assign assign while (false)`

Answer:

```
L2:             # label(begin)
    assign      # 2.code from S -> S S
L4: assign
L3:             # label(check)
    goto L1     # 5.code from B -> false
L5: goto L2     # label(true) + instr
L1:             # P -> S
```

(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))`

Answer:

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

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

Answer:

$$\begin{array}{rcl} S' & \Rightarrow & S \Rightarrow (N,C) \Rightarrow (x,C) \Rightarrow (x,S) \Rightarrow (x,(N,C)) \Rightarrow (x,(y,C)) \Rightarrow (x,(y,S)) \\ & \Rightarrow & (x,(y,T)) \Rightarrow (x,(y,b)) \end{array}$$

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

Answer:

$$\begin{array}{rcl} S' & \Rightarrow & S \Rightarrow (N,C) \Rightarrow (N,S) \Rightarrow (N,(N,C)) \Rightarrow (N,(N,S)) \Rightarrow (N,(N,T)) \\ & \Rightarrow & (N,(N,b)) \Rightarrow (N,(y,b)) \Rightarrow (x,(y,b)) \end{array}$$

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

Answer:

```
(S' (S \(
       (N x)
       ,
       (C (S \(
              (N y)
              ,
              (C (S (T b)))
              \)))
       \)))
```

Notice that the tree constructed using the leftmost derivation is identical to the tree constructed using the rightmost derivation

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

Answer: parse trees with non-terminals N and terminals T

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

Answer:

$$C \to C, S \mid S \Rightarrow C \to SC', C' \to, SC' \mid \epsilon \tag{1}$$

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

Answer: Yes, it is LL(1) because all 3 conditions are satisfied: there is no $A \to u \mid v$ where FIRST(u) intersected with FIRST(v) is nonempty, for $S' \to S \mid \epsilon$, S does not derive the empty string and FOLLOW(S') = {\$} intersected with FIRST(S) = $\{a, b, c, (\}$ is the empty set.

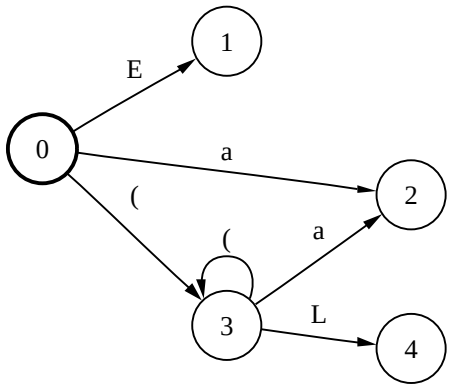
(9) Consider the augmented CFG G with S' as the start symbol:

$$
\begin{aligned}
S' &\rightarrow E && (2)\\
E &\rightarrow a && (3)\\
E &\rightarrow (\ L\ ) && (4)\\
L &\rightarrow \epsilon && (5)\\
L &\rightarrow E\ L && (6)
\end{aligned}
$$

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:

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

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:

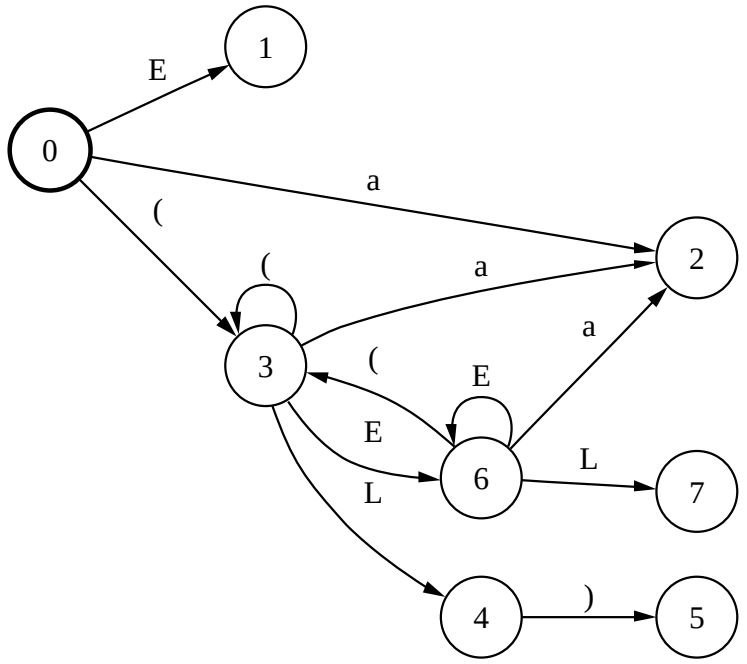
a. Add the remaining three itemsets (use the numbers 5, 6, 7 for the new itemsets as indicated by entries in the action/goto table), and then

Answer:

$$
\begin{array}{lllll}
5: & E & \rightarrow & (L)\bullet & \\
 & & & & \\
6: & L & \rightarrow & E\bullet\ L & \\
 & L & \rightarrow & \epsilon\bullet & \\
 & L & \rightarrow & \bullet\ EL & \\
 & E & \rightarrow & \bullet\ a & \\
 & E & \rightarrow & \bullet\ (L) & \\
 & & & & \\
7: & L & \rightarrow & EL\bullet &
\end{array}
$$

b. Complete the LR(0) automaton by adding the three new states including all the necessary transitions, and then

Answer:



c. Complete the action/goto table entries (remember to indicate error entries), and finally

Answer:

	(	)	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	reduce(3)	reduce(3)	reduce(3)	reduce(3)		
6	shift 3 reduce(4)	error reduce(4)	shift 2 reduce(4)	error reduce(4)	goto 6	goto 7
7	reduce(5)	reduce(5)	reduce(5)	reduce(5)		

d. Indicate whether the grammar G is a LR(0) grammar

Answer: nope. conflicts all over the place.

(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.

Answer: No. FIRST(aSa) intersected with FIRST(aa) is non-empty.

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.

Answer: No. While the grammar is unambiguous, consider the potential viable prefix a^*b^*aaaa and the potential viable prefix a^*b^*aa – the same prefix can lead to a shift on a or a reduce using $S \rightarrow aa$ on the handle a^*b^*aa.

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.

Answer: No. It now includes ϵ in the language.

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

Answer: No, it is still not an LL(1) grammar because the intersection of FIRST(aSa) and FOLLOW(S) = $\{a, b, \$\}$ is non-empty.

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.

Answer: No, it is still not an LL(1) grammar because the intersection of FIRST(aSa) and FOLLOW(S) = $\{a, b, \$\}$ is non-empty.

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

Answer: No to all. Ambiguous.

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

Answer: Not LL(1), FIRST(A) $\cap$ FIRST(B) is not disjoint. Is LR(0) because the only case to where a conflict might occur is when items $S \rightarrow \bullet A$ and $S \rightarrow \bullet B$ predict two items: $A \rightarrow \bullet dAd$ and $B \rightarrow \bullet dBd$, but the successor on A and B ensure that item $A \rightarrow dAd\bullet$ has to be in a different state from $B \rightarrow dBd\bullet$. Since grammar is LR(0) it is also SLR(1) and LR(1).

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

Answer: Is LL(1). FIRST(aA) $\cap$ FIRST(bB) is disjoint. Is LR(0) since the change in the grammar from the previous question cannot introduce any new conflicts in the LR(0) table and therefore the grammar is SLR(1) and also LR(1).

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

Answer: Is LL(1) because FIRST($AaAb$) $\cap$ FIRST($BbBa$) is disjoint, but not SLR(1) as FOLLOW(A) $\cap$ FOLLOW(B) is not disjoint. Since grammar is not SLR(1) it is also not LR(0). Grammar is LR(1) because the LR(1) items $[A \rightarrow \epsilon\bullet, a]$ and $[B \rightarrow \epsilon\bullet, b]$ which occur in the same itemset have distinct lookahead due to the LR(1) closure condition. The successor on A and B ensures that the two rules with S as left-hand side will appear in different itemsets and therefore no other conflicts can occur.

(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.

Answer:

```
B1:                 B4:                 B6:
  i = m-1             if i>=j goto B6      t11 = 4*i
  j = n                                    x = A[t11]
  t1 = 4 * n        B5:                    t12 = 4*i
  v = A[t1]           t6 = 4*i             t13 = 4*n
                      x = A[t6]            t14 = A[t13]
B2:                   t7 = 4*i             A[t12] =  t14
  i = i+1             t8 = 4*j             t15 = 4*n
  t2 = 4 * i          t9 = A[t8]           A[t15] = x
  t3 = A[t2]          A[t7] = t9
  if t3<v goto B2     t10 = 4*j          B1 --> B2 --> B3 --> B4 --> B5
                      A[t10] = x         B2 --> B2
B3:                   goto B2            B3 --> B3
  j = j-1                                B5 --> B2
  t4 = 4*j
  t5 = A[t4]
  if t5>v goto B3
```

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

Answer:

```
B5:
  t6 = 4*i
  x = A[t6]
  t8 = 4*j
  t9 = A[t8]
  A[t6] = t9
  A[t8] = x
  goto B2

B6:
  t11 = 4*i
  x = A[t11]
  t13 = 4*n
  t14 = A[t13]
  A[t11] = t14
  A[t13] = x
```

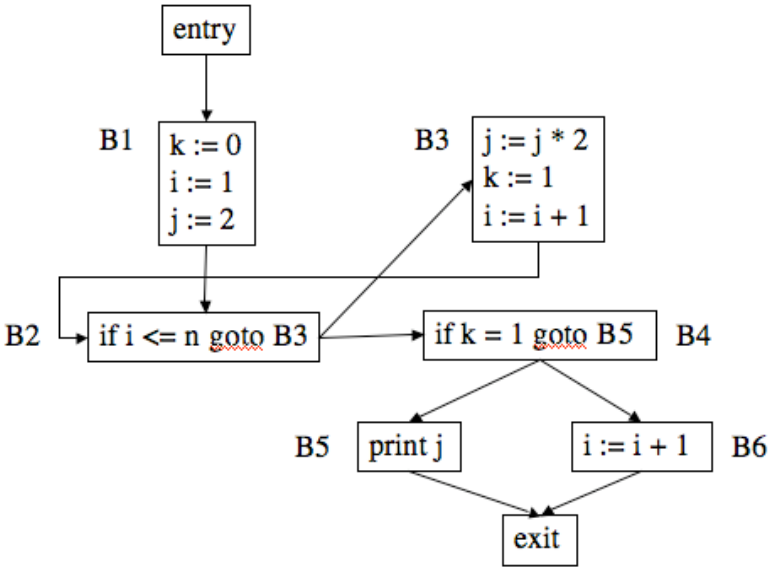
c. The instruction `t4 := 4*j` is repeatedly executed inside an inner loop (as can be seen in the control flow graph). Analyze the change in values in `t4` and reduce the strength of this instruction by replacing the multiplication operation with the cheaper operation (such as addition or subtraction). 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.

Answer:

```
B1:
  i = m-1
  j = n
  t1 = 4*n
  v = A[t1]
  t4 = 4*j

B3:
  j = j-1
  t4 = t4-4
  t5 = A[t4]
  if t5>v goto B3
```

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.



Answer: