Simon Fraser University
School of Computing Science

CMPT 383

Assignment 2

Due date: October 25, 2005

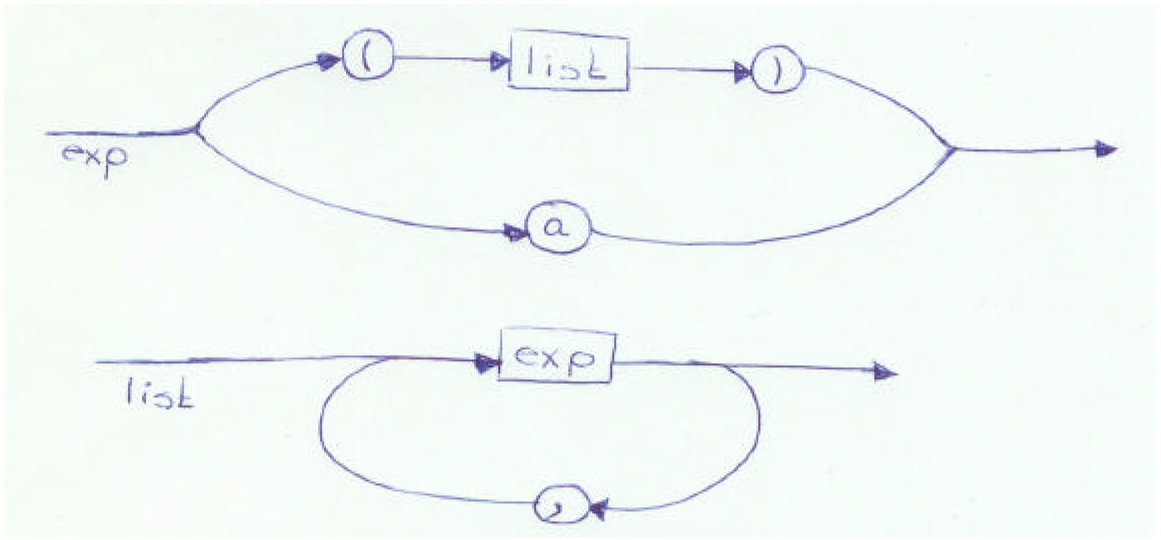
1) **(5 marks)** Problem 13 (page 172).
   `<S> ::= a <S> b | ab`

2) Given the following BNF:

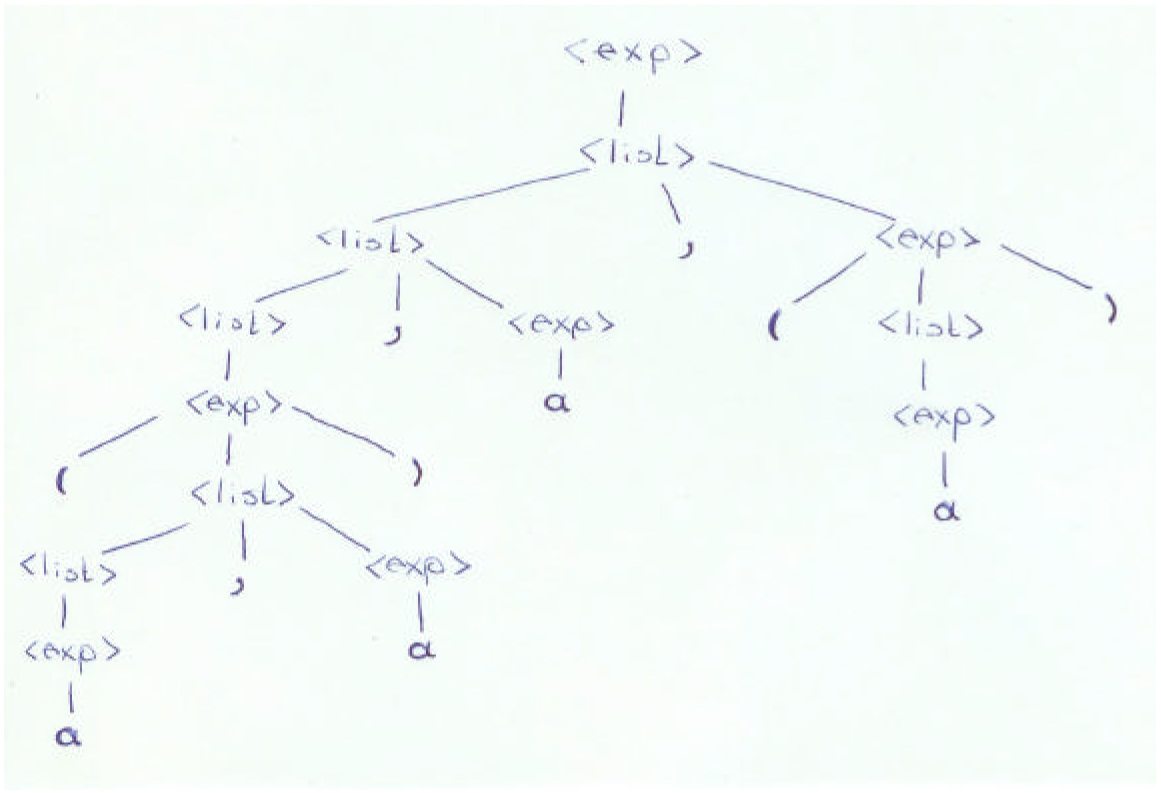
```
<exp> ::= ( <list> ) | a
<list> ::= <list>, <exp> | <exp>
```

   a) Write **(4 marks)** EBNF rules and **(5 marks)** syntax diagrams for the language.

```
<exp> ::= ( <list> ) | a
<list> ::= <exp> {, <exp>}
```



   b) **(4 marks)** Draw the parse tree for `((a,a),a,(a))`.

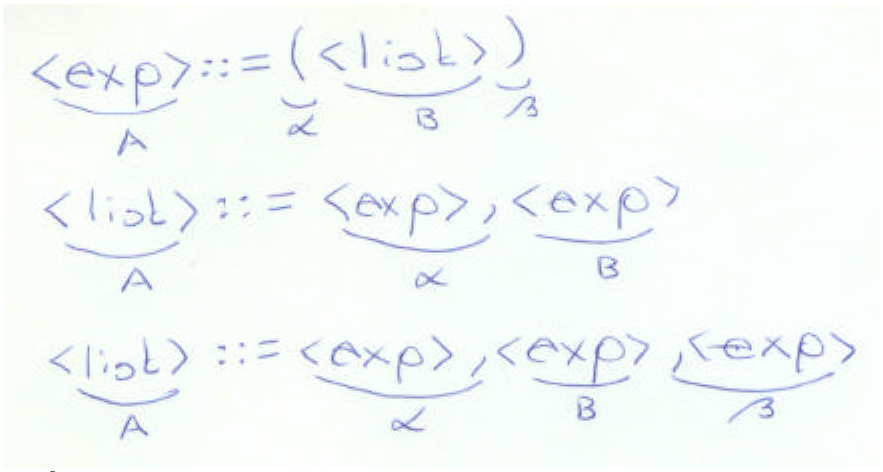


c) Compute the **(3 marks)** FIRST and **(5 marks)** FOLLOW sets for the nonterminals of the grammar.

```
Using EBNF:
FIRST(<exp>) = FIRST(() È FIRST(a) = {(,a}
FIRST(<list>) = FIRST(<exp>) = {(,a}

FOLLOW(<exp>) = {$} È FOLLOW(<list>) È FIRST(,<exp>) = {$,),,}
FOLLOW(<list>) = FIRST()) + {)}
```

Nonterminal	**FIRST**	**FOLLOW**
<exp>	{ (, a }	{ $,), , }
<list>	{ (, a }	{) }



$$\underbrace{\langle exp\rangle}_{A} ::= \underbrace{(}_{\alpha}\underbrace{\langle list\rangle}_{B}\underbrace{)}_{\beta}$$

$$\underbrace{\langle list\rangle}_{A} ::= \underbrace{\langle exp\rangle,}_{\alpha}\underbrace{\langle exp\rangle}_{B}$$

$$\underbrace{\langle list\rangle}_{A} ::= \underbrace{\langle exp\rangle,}_{\alpha}\underbrace{\langle exp\rangle}_{B}\underbrace{,\langle exp\rangle}_{\beta}$$

Using BNF (eliminate left recursion):
```
<exp> ::= (<list>) | a
<list> ::= <exp> <list'>
<list'> ::= , <exp> <list'> | Î

FIRST(<exp>) = FIRST(() È FIRST(a) = {(,a}
FIRST(<list>) = FIRST(<exp>) = {(,a}
FIRST(<list'>) = FIRST(,)È {Î} = {,,Î}

FOLLOW(<exp>)={$} È FOLLOW(<list'>) È FIRST(<list'>) = {$,,,)}
FOLLOW(<list>) = FIRST()) + {)}
FOLLOW(<list'>) = FOLLOW(<list>) = {)}
```

Nonterminal	FIRST	FOLLOW
<exp>	{ (, a }	{ $,), , }
<list>	{ (, a }	{) }
<list'>	{ ,, ∊ }	{) }

d) **(10 marks)** Construct the predictive parsing table.

Using EBNF:

Nonterminal	a	(	)	,	$
<exp>	<exp> ::= a	<exp ::= (<list>)			
<list>	<list> ::= <exp> {,<exp>}	<list> ::= <exp> {,<exp>}			

Using BNF:

Nonter	a	(	)	,	$
<exp>	<exp> ::= a	<exp ::= (<list>)			
<list>	<list> ::= <exp> {,<exp>}	<list> ::= <exp> {,<exp>}			
<list'>			<list'>::= ∊	<list'> ::= ,<exp><list'>	<list'>::= ∊

e) **(5 marks)** Show the sequence of moves (trace) for `((a,a),a,(a))`.

Using EBNF:

STACK	INPUT	OUTPUT
$ <exp>	((a,a),a,(a)) $	

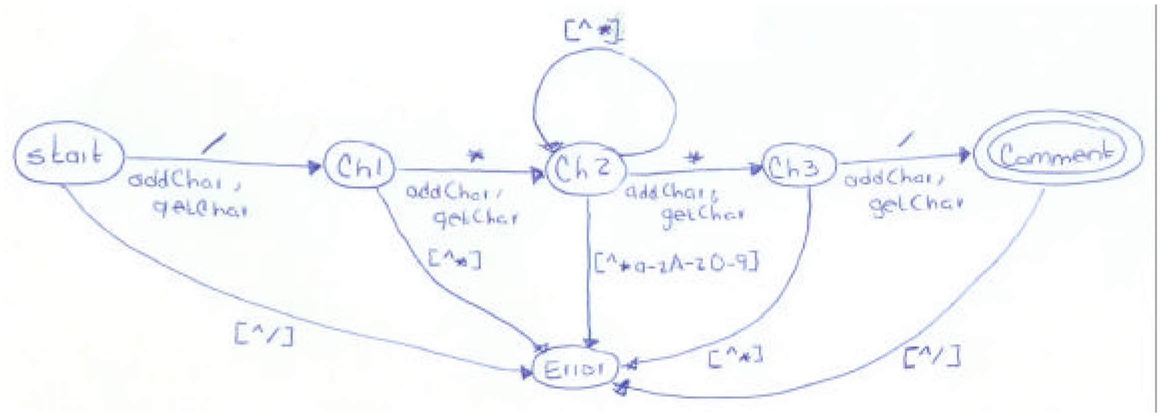
$) <list> (	((a,a),a,(a)) $	<exp> ::= (<list>)
$) <list>	(a,a),a,(a)) $	
$) {<exp,} <exp>	(a,a),a,(a)) $	<list> ::= <exp> {,<exp>)
$) {<exp,}) <list> (	(a,a),a,(a)) $	<exp> ::= (<list>)
$) {<exp,}) <list>	a,a),a,(a)) $	
$) {<exp>,}) {<exp>,} <exp>	a,a),a,(a)) $	<list> ::= <exp> {,<exp>}
$) {<exp>,}) {<exp>,} a	a,a),a,(a)) $	<exp> ::= a
$) {<exp>,}) {<exp>,}	,a),a,(a)) $	
$) {<exp>,}) <exp>	a),a,(a)) $	
$) {<exp>,}) a	a),a,(a)) $	<exp> ::= a
$) {<exp>,})	),a,(a)) $	
$) {<exp>,}	,a,(a)) $	
$) {<exp>,}) <exp>	a,(a)) $	
$) {<exp>,}) a	a,(a)) $	<exp> ::= a
$) {<exp>,})	,(a)) $	
$) <exp>	(a)) $	
$)) <list> (	(a)) $	<exp> ::= (<list>)
$)) <list>	a)) $	
$)) <exp>	a)) $	<list> ::= <exp>
$)) a	a)) $	<exp> ::= a
$))	)) $	
$)	) $	
$	$	

Using BNF:

STACK	INPUT	OUTPUT
$ <exp>	((a,a),a,(a)) $	
$) <list> (	((a,a),a,(a)) $	<exp> ::= (<list>)
$) <list>	(a,a),a,(a)) $	
$) <list'><exp>	(a,a),a,(a)) $	<list> ::= <exp> <list'>
$) <list'>) <list> (	(a,a),a,(a)) $	<exp> ::= (<list>)
$) <list'>) <list>	a,a),a,(a)) $	
$) <list'>) <list'> <exp>	a,a),a,(a)) $	<list> ::= <exp> <list'>
$) <list'>) <list'> a	a,a),a,(a)) $	<exp> ::= a
$) <list'>) <list'>	,a),a,(a)) $	
$) <list'>) <list'> <exp> ,	,a),a,(a)) $	<list'> ::= , <exp> <list'>
$) <list'>) <list'> <exp>	a),a,(a)) $	
$) <list'>) <list'> a	a),a,(a)) $	<exp> ::= a
$) <list'>) <list'>	),a,(a)) $	
$) <list'>)	),a,(a)) $	<list'> ::= ∈
$) <list'>	,a,(a)) $	
$) <list'> <exp> ,	,a,(a)) $	<list'> ::= , <exp> <list'>
$) <list'> <exp>	a,(a)) $	
$) <list'> a	a,(a)) $	<exp> ::= a
$) <list'>	,(a)) $	
$) <list'> <exp> ,	,(a)) $	<list'> ::= , <exp> <list'>
$) <list'> <exp>	(a)) $	

$) <list’>) <list> (	(a)) $	<exp> ::= (<list>)
$) <list’>) <list>	a)) $	
$) <list’>) <list’> <exp>	a)) $	<list> ::= <exp> <list’>
$) <list’>) <list’> a	a)) $	<exp> ::= a
$) <list’>) <list’>	)) $	
$) <list’>)	)) $	<list’> ::= ∈
$) <list’>	) $	
$)	) $	<list’> ::= ∈
$	$	

3) **(5 marks)** Design a state diagram to recognize one form of the comments of the C-based programming languages, those that begin with /* and end with */



4) **(4 marks)** Problem 19 (page 172).

```
a) 2*(b-1)-1 > 0
   2*(b-1) > 1
   b-1 > 1/2
   b > 3/2
   {b > 3/2} 2*(b-1)-1 {a > 0}
b) (c+10)/3 > 6
   c+10 > 18
   c > 8
   {c > 8} (c+10)/3 {b > 6}
c) a+2*b-1 > 1
   a+2*b > 2
   2*b > 2-a         or    a > 2-2*b
   b > 1-a/2         or    a > 2(1-b)
   {b > 1-a/2} a+2*b-1 {a > 1} or
   {a > 2(1-b)} a+2*b-1 {a > 1}
d) 2*y+x-1 > 11
   2*y+x > 12
   2*y > 12-x  or    x > 12-2*y
   y > 6-x/2         or    x > 2(6-y)
   {y > 6-x/2} 2*y+x-1 {x > 11} or
   {x > 2(6-y)} 2*y+x-1 {x > 11}
```

5) **(3 marks)** Problem 23 (page 173).

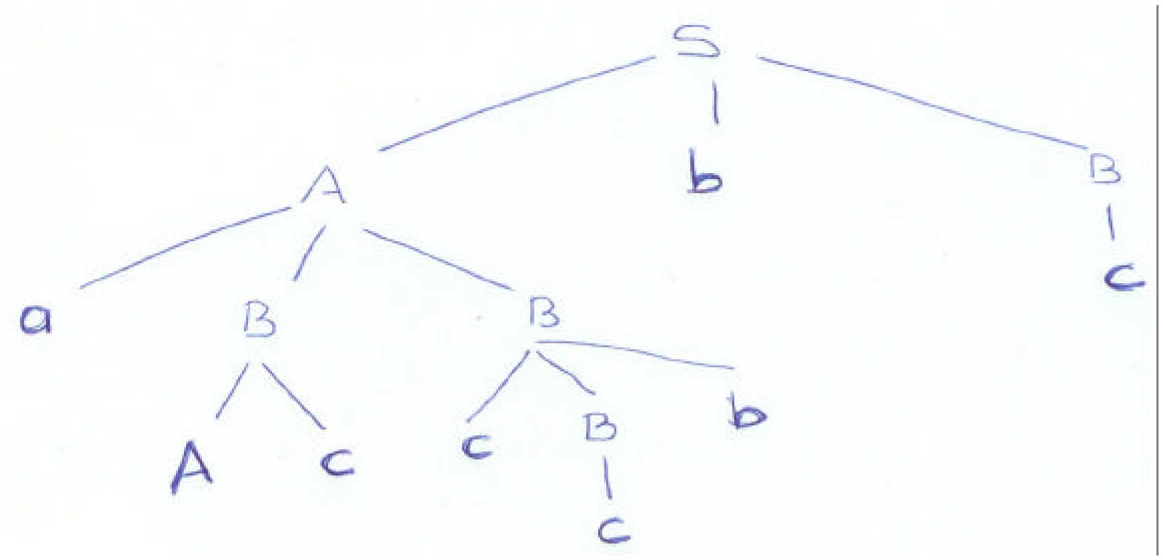
1. **Syntax rule: <assign> → <var> = <expr>**
   Semantic rule: <expr>.expected_type ← <var>.actual_type

2. **Syntax rule: <expr> → <var>[2]+<var>[3]**
   Semantic rule: <expr>.actual_type ← <var>[2].actual_type
   Predicate: <var>[2].actual_type==<var>[3].actual_type
3. **Syntax rule: <expr> → <var>**
   Semantic rule: <expr>.actual_type ← <var>.actual_type
4. **Syntax rule: <var> → A | B | C**
   Semantic rule: <var>.actual_type ← look-up(<var>.string)

6) Problem 6 (page 203).

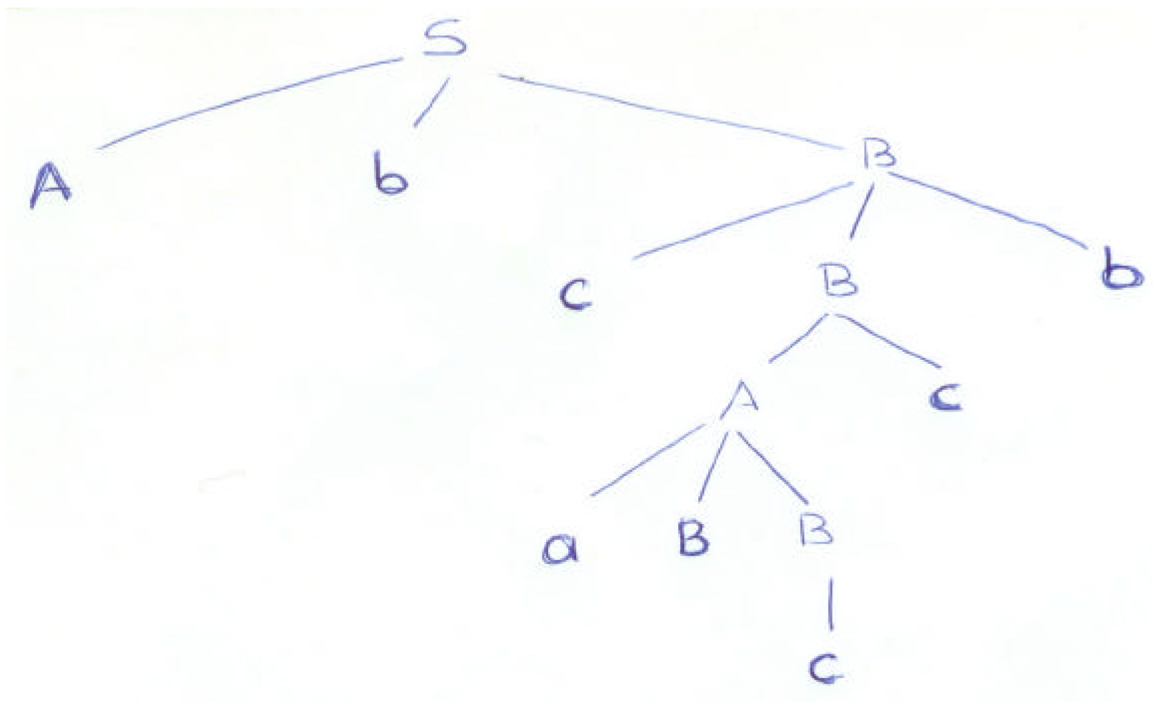
   a) (5 marks)

   Phrases: `aAcccbbc, aAcccb, Ac, ccB, c, c`
   Simple Phrases: `Ac, c, c`
   Handle: Ac

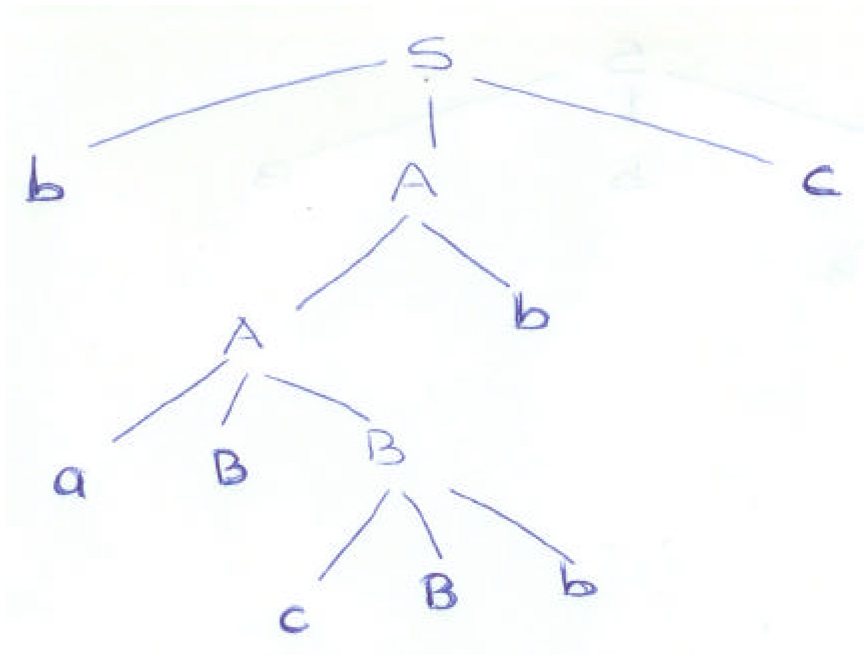


   b) (3 marks)

   Phrases: `AbcaBccb, caBccb, aBccb, aBc, c`
   Simple Phrases: `c`
   Handle: c



c) **(3 marks)**
Phrases: **baBcBbbc, aBcBbb, aBcBb, cBb**
Simple Phrases: **cBb**
Handle: cBb



7) **(5 marks)** Is it possible to have a language without any reserved words? Discuss.
Indeed, Fortran is a language without reserved words, so it is certainly possible for a language to have no reserved words. The problem is that all language constructs become essentially context dependent, and a parser cannot decide which construct is applicable

without help from the symbol table and semantic analyzer. This enormously complicates the parsing process, and context-free grammar techniques cannot be used. For this reason, the trend has been toward greater use of reserved words and less use of predefined identifiers.

8) **(5 marks)** The syntax of the `if` statement in Ada is as follows:

```
<if_statement> ::= if <condition> then <sequence_of_statements>
                 { elsif <condition> then <sequence_of_statements> }
                 [ else <sequence_of_statements>
                 end if;
```

Any number of `elsif` clauses are allowed to avoid deeply nested `if` statements. The keyword `elsif` might be considered aesthetically repugnant. What would be the consequences be of replacing `elsif` with the two keywords `else if`?

If the single word `elsif` is replace by the two keywords `else if` then this last word `if` could be interpreted in two different ways: (a) another option of the previous if-statement or (b) a new if-statement. This problem adds ambiguity to the language and consequently decreases readability and writability.

9) **(12 marks)** Problem 14 (page 248).

Main: X, Y, Z
Sub1: Y, Z, A
Sub2: Z, A, B
Sub3: X, A, W

a)

Main	**X**	**Y**	**Z**			
Sub1		**Y**	**Z**	**A**		
Sub2			**Z**	**A**	**B**	
Sub3	**X**			**A**		**W**

Visible Variables: X, A, W from Sub3; Z, B from Sub2; Y from Sub1

b)

Main	**X**	**Y**	**Z**			
Sub1		**Y**	**Z**	**A**		
Sub3	**X**			**A**		**W**

Visible Variables: X, A, W from Sub3; Y, Z from Sub1

c)

Main	**X**	**Y**	**Z**			
Sub2			**Z**	**A**	**B**	
Sub3	**X**			**A**		**W**
Sub1		**Y**	**Z**	**A**		

Visible Variables: Y, Z, A from Sub1; X, W from Sub3; B from Sub2

d)

Main	**X**	**Y**	**Z**			
Sub3	**X**			**A**		**W**
Sub1		**Y**	**Z**	**A**		

Visible Variables: Y, Z, A from Sub1; X, W from Sub3

e)

Main	**X**	**Y**	**Z**			
Sub1		**Y**	**Z**	**A**		
Sub3	**X**			**A**		**W**
Sub2			**Z**	**A**	**B**	

Visible Variables: Z, A, B from Sub2; X, W from Sub3, Y from Sub1

f)

Main	**X**	**Y**	**Z**			
Sub3	**X**			**A**		**W**

Sub2 Z A B

Sub1 Y Z A

Visible Variables: Y, Z, A from Sub1; B from Sub2; X, W from Sub3

10) **(9 marks)** Problem 4 (page 249).

11) **(5 marks)** Problem 5 (page 249), only C++ and Java.

12) **(10 marks)** Describe the scopes of the declarations in the following C program. How would the scopes change using dynamic instead of static scoping? What does the program print in each case?

```
int a, b;

int p(void) {
   int a, p;

   a = 0;  b = 1;  p = 2;
   return p; }

void print(void) {
   printf("%d\n%d\n", a, b); }

void q(void) {
   int b;
   a = 3;  b = 4;
   print(); }

void main(void) {
   a = p();
   q(); }
```

STATIC

Main declares a, b (global scope)
p() declares a, p (local scope); b from Main can be accessed
q() declares b; a from Main can be accessed.
print() does not declare any variable; a from Main and b from q can be accessed.

	a (Main)	**a (p)**	**b (Main)**	**b (q)**	**p**
Main	2				
p		0	1		2
q	3			4	

The program prints:
3
1

DYNAMIC

Main declares a, b (global scope)
p() declares a, p (local scope); b from Main can be accessed

q() declares b; a from Main can be accessed.
print() does not declare any variable; a,b from Main can be accessed.

	a (Main)	**a (p)**	**b (Main)**	**b (q)**	**p**
Main	2				
p		0	1		2
q	3			4	

The program prints:

```
3
4
```