Concordia University

Department of Computer Science

and Software Engineering

Final Examination Sample

Course:	COMP 442/6421	**Date:**	
Instructor:	Dr. Joey Paquet	**Time:**	

NAME :	
STUDENT ID :	

<u>General instructions and information:</u>

- The grade value and estimated time to complete all sections is provided
- Duration is 180 minutes
- This is an open book examination
- You are allowed to write your answers either in English or French

<u>Grades and time distribution</u>

Section	Grading	Estimated time to complete
I. Miscellaneous	20%	25 min
II. Lexical Analysis	10%	25 min
III. Context-Free Grammars	15%	25 min
IV. Top-down Parsing	25%	45 min
V. Bottom-Up Parsing	30%	50 min
Total	100%	170 min

Section I: Miscellaneous [20%, 25 minutes]

A. **[10%]** Building a parser can be done in various ways. Compare the following parser designs by giving comparative advantages and drawbacks for each:

- Recursive descent top-down predictive parser
- Table-driven top-down predictive parser
- CLR parser
- SLR parser

B. **[10%]** Describe the roles of the following phases of a typical compiler: (1) lexical analyzer, (2) syntax analyzer, (3) semantic analyzer, and (4) code generator. Explain how the following statement is analyzed and transformed by each of these phases:

```
total = sum1 + 1000 * (a – f(a)); /* recomputed total*/
```

Section II: Lexical Analysis [10%, 25 minutes]

A. **[10%]** Design a DFA that specifies the lexical form of scientific notation numbers such as the one used in C. The lexical specification used to specify such numbers is given below in EBNF. Examples of valid character sequences are: ***1.008e-2***, ***+1***, ***-2.4e10***, ***0***, ***1.2***, and ***10e2***.

```
<num>     → [+|–]<int>[.<digit>*<nonzero>][e[+|–]<int>]
<int>     → <nonzero> [<digit>*] | 0
<digit>   → 0..9
<nonzero> → 1..9
```

Section III: Context-Free Grammars [15%, 25 minutes]

Consider the following grammar:

```
<bexp>    → <bexp> or <bterm> | <bterm>
<bterm>   → <bterm> and <bfactor> | <bfactor>
<bfactor> → not <bfactor> | (<bexp>) | T | F
```

A. [5%] Construct the rightmost derivation for the expression: **`T or not F and T or F`**.
B. [5%] Represent the rightmost derivation using a parse tree.
C. [5%] Eliminate left recursions in the above grammar. Show the procedure to achieve your result.

Section IV: Top-down Parsing [25%, 45 minutes]

Consider the following grammar `G=(T,N,S,R)`, where:

```
                T =  {id, ., [, expr, ]}
                N =  {<Factor>, <IdNest>, <IndiceList>}
                S =  <Factor>
                R =  {R1,R2,R3,R4,R5} where:

R1: <Factor>     → id <IndiceList> <IdNest>
R2: <IdNest>     → . id <IndiceList> <IdNest>
R3: <IdNest>     → ε
R4: <IndiceList> → [ expr ] <IndiceList>
R5: <IndiceList> → ε
```

A. **[10%]** Construct the FIRST and FOLLOW sets for this grammar.
B. **[5%]** Construct the LL(1) top-down predictive parsing table for this grammar.
C. **[10%]** Using your constructed table, construct a parse trace for input string "**id.id[expr].id**". Every step of the parse trace should show: (1) the content of the stack, (2) the next token in input, (3) the action taken by the parser, and (4) the derivation step that corresponds to the parsing step.

Section V: Bottom-Up Parsing [30%, 50 minutes]

Consider the following grammar `G=(T,N,S,R)` (same as in section IV), where:

```
                T =  {id, ., [, expr, ]}
                N =  {<Factor>, <IdNest>, <IndiceList>}
                S =  <Factor>
                R =  {R1,R2,R3,R4,R5} where:

R1: <Factor>     → id <IndiceList> <IdNest>
R2: <IdNest>     → . id <IndiceList> <IdNest>
R3: <IdNest>     → ε
R4: <IndiceList> → [ expr ] <IndiceList>
R5: <IndiceList> → ε
```

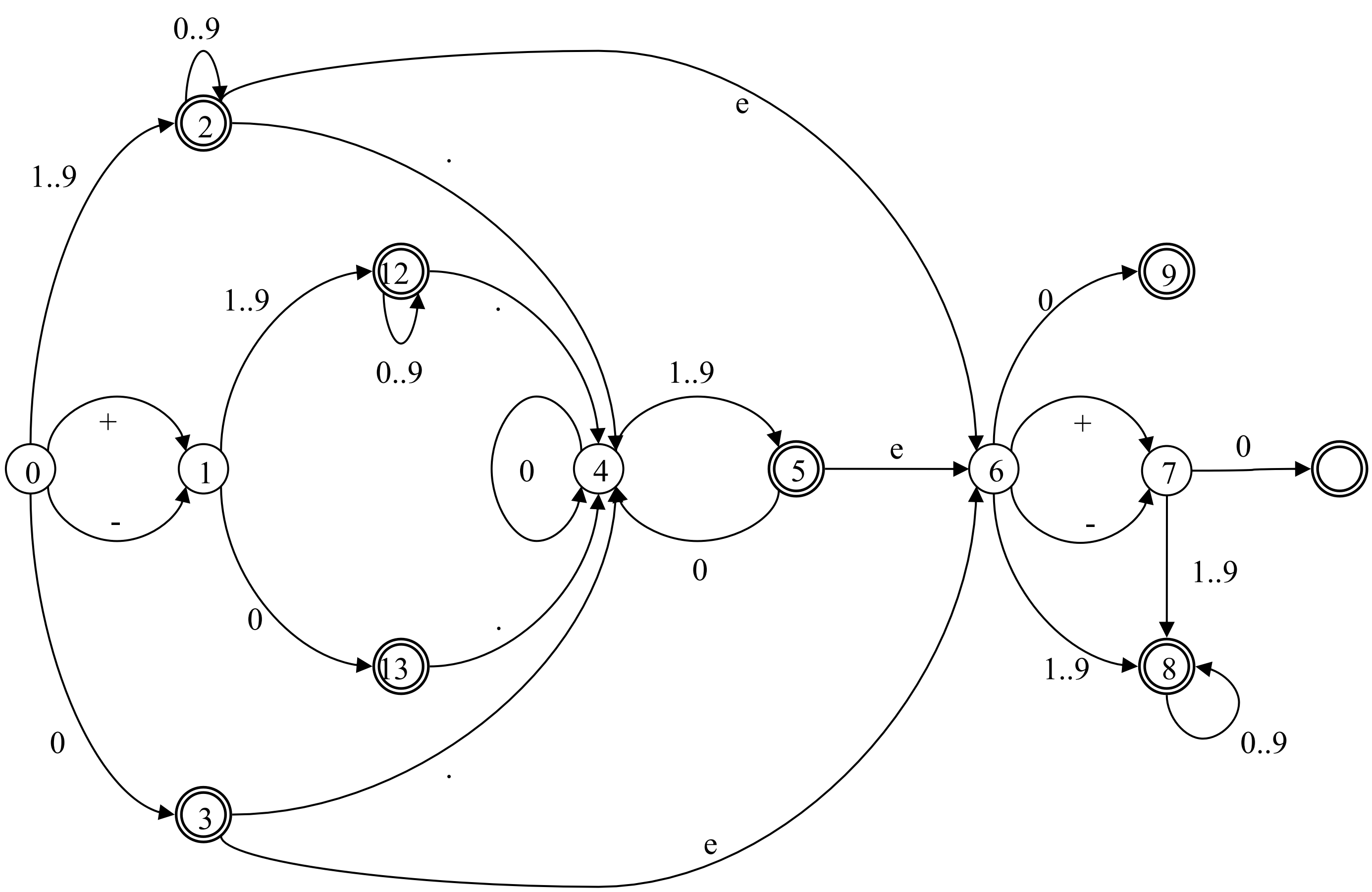
A. **[10%]** Construct the Simple (SLR) item sets that correspond to this grammar.
B. **[5%]** Draw a DFA that corresponds to the items sets found in your answer V.A.
C. **[5%]** Construct the LR(1) bottom-up parsing table for this grammar.
D. **[10%]** Using your constructed table, construct a parse trace for the input string "**id.id[expr].id**". Every step of the parse trace should show: (1) the content of the stack, (2) the next token in input, (3) the action taken by the parser, and (4) the derivation step that corresponds to the parsing step.

Partial Solution

Section II: Lexical Analysis [10%, 25 minutes]

A. **[10%]** Design a DFA that specifies the lexical form of scientific notation numbers such as the one used in C. The lexical specification used to specify such numbers is given below in EBNF. Examples of valid character sequences are: 1.008e-2, +1, -2.4e10, 0, 1.2, and 10e2.

<num>	→	[**+**|**–**]<int>[**.**<digit>*<nonzero>][**e**[**+**|**–**]<int>]
<int>	→	<nonzero> [<digit>*] | **0**
<digit>	→	**0..9**
<nonzero>	→	**1..9**



Section III: Context-Free Grammars [15%, 25 minutes]

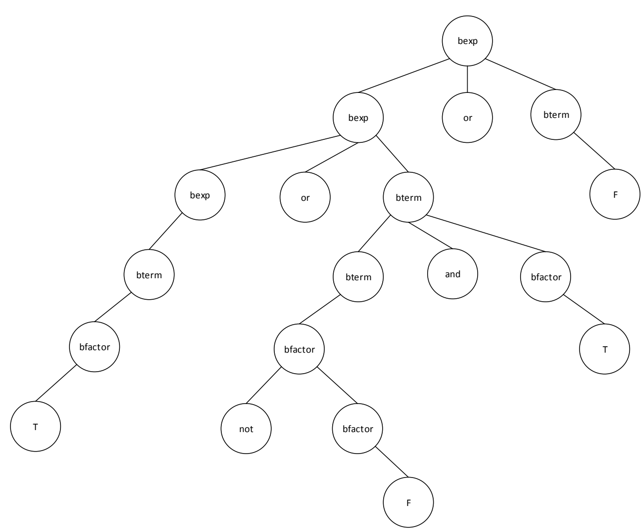
Consider the following grammar:

<bexp>	→	<bexp> **or** <bterm> | <bterm>
<bterm>	→	<bterm> **and** <bfactor> | <bfactor>
<bfactor>	→	**not** <bfactor> | (<bexp>) | **T** | **F**

A. [5%] Construct the *rightmost* derivation for the expression: **T or not F and T or F**.

<bexp>	⇒	<bexp> or <bterm>	<bexp> → <bexp> or <bterm>
	⇒	<bexp> or <bfactor>	<bterm> → <bfactor>
	⇒	<bexp> or F	<bfactor> → F
	⇒	<bexp> or <bterm> or F	<bexp> → <bexp> or <bterm>
	⇒	<bexp> or <bterm> and <bfactor> or F	<bterm> → <bterm> and <bfactor>
	⇒	<bexp> or <bterm> and T or F	<bfactor> → T
	⇒	<bexp> or <bfactor> and T or F	<bterm> → <bfactor>
	⇒	<bexp> or not <bfactor> and T or F	<bfactor> → not <bfactor>
	⇒	<bexp> or not F and T or F	<bfactor> → F
	⇒	<bterm> or not F and T or F	<bexp> → <bterm>
	⇒	<bfactor> or not F and T or F	<bterm> → <bfactor>
	⇒	T or not F and T or F	<bfactor> → T

B. [5%] Represent the rightmost derivation using a parse tree.



C. [5%] Revise the above grammar to eliminate left recursions. Show the procedure to achieve your result.

<bexp> → <bexp> or <bterm>	A → Aα	<bterm> → <bterm> and <bfactor>
<bexp> → <bterm>	A → β	<bterm> → <bfactor>
<bexp> → <bterm><bexp’>	A → βA’	<bterm> → <bfactor><bterm’>
<bexpr’> → ε | or <bterm><bexp’>	A’ → ε | αA’	<bterm> → and <bfactor><bterm’>

Section III: Top-down Parsing [20%, 45 minutes]

Consider the following grammar `G=(T,N,S,R)`, where:

```
                        T  =  {id, ., [, expr, ]}
                        N  =  {<Factor>, <IdNest>, <IndiceList>}
                        S  =  <Factor>
                        R  =  {R1,R2,R3,R4,R5} where:

R1: <Factor>          →  id <IndiceList> <IdNest>
R2: <IdNest>          →  . id <IndiceList> <IdNest>
R3: <IdNest>          →  ε
R4: <IndiceList>      →  [ expr ] <IndiceList>
R5: <IndiceList>      →  ε
```

A. [10%] Construct the FIRST and FOLLOW sets for this grammar.

FIRST(Factor) = { id }
FIRST(IdNest) = { . }
FIRST(IndiceList) = { (}

FOLLOW(S) = { $ }
FOLLOW(A) = { $ }
FOLLOW(B) = { $: }

B. [5%] Construct the top-down predictive parsing table for this grammar.

	$	]	expr	[	id	.
Factor					R1	
IdNest	R3					R2
IndiceList	R5			R4		R5

C. [5%] Using your constructed table, construct a parse trace for the input string ***id.id[expr].id.***

Stack	input	action	Derivation steps
$-F	id.id[expr].id$	R1 (F→ id IL IDN)	F ⇒ id IL IDN
$IDN-IL-id	id.id[expr].id$		⇒ id IL IDN
$IDN-IL	.id[expr].id$	R5 (IL→ ε)	⇒ id IDN
$IDN	.id[expr].id$	R2 (IDN→ . id IL IDN)	⇒ id . id IL IDN
$ IDN-IL-id -.	.id[expr].id$		⇒ id . id IL IDN
$ IDN-IL-id	id[expr].id$		⇒ id . id IL IDN
$ IDN-IL	[expr].id$	R4 (IL→[expr] IL)	⇒ id . id [expr] IL IDN
$ IDN-IL-]-expr -[	[expr].id$		⇒ id . id [expr] IL IDN
$ IDN-IL-]-expr	expr].id$		⇒ id . id [expr] IL IDN
$ IDN-IL-]	].id$		⇒ id . id [expr] IL IDN
$ IDN-IL	.id$	R5 (IL→ ε)	⇒ id . id [expr] IDN
$ IDN	.id$	R2 (IDN→ . id IL IDN)	⇒ id . id [expr] . id IL IDN
$ IDN-IL-id -.	.id$		⇒ id . id [expr] . id IL IDN
$ IDN-IL-id	id$		⇒ id . id [expr] . id IL IDN
$ IDN-IL	$	R5 (IL→ ε)	⇒ id . id [expr] . id IDN
$ IDN	$	R3 (IDN→ ε)	⇒ id . id [expr] . id
$	$	Acc	

Section IV: Bottom-Up Parsing [25%, 50 minutes]

Consider the following grammar `G=(T,N,S,R)`, where:

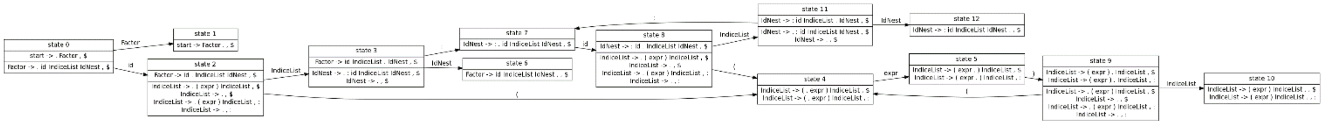
```
T =  {id, ., [, expr, ]}
N =  {<Factor>, <IdNest>, <IndiceList>}
S =  <Factor>
R =  {R1,R2,R3,R4,R5} where:

R1: <Factor>        →  id <IndiceList> <IdNest>
R2: <IdNest>        →  . id <IndiceList> <IdNest>
R3: <IdNest>        →  ε
R4: <IndiceList>    →  [ expr ] <IndiceList>
R5: <IndiceList>    →  ε
```

A. **[10%]** Construct the SLR item sets that correspond to this grammar.

States	Items	Transitions	table entries
State 0 : V[ε]	S →• Factor: $	State 1	Goto[0, Factor]:1
	Factor→• id IndiceList Idnest : $	State 2	Action[0, id]:Shift 2
State 1 : V[Factor]	S →Factor •: $	Accept	Action[1, $]:Acc
State 2 : V[id]	Factor→id • IndiceList Idnest : $	State 3	Goto[2, Factor]:3
	IndiceList→•[expr] IndiceList : .	State 4	Action[2, [ ]:Shift 4
	IndiceList→ ε • : .	Handle R5	Action[2, .]:Reduce R5
	IndiceList→•[expr] IndiceList : $	State 4	Action[2, [ ]:Shift 4
	IndiceList→ ε • : $	Handle R5	Action[2, $]:Reduce R5
State 3 : V[id IndiceList]	Factor→id IndiceList • Idnest : $	State 6	Goto[3, Idnest]:6
	IdNest→•.id IndiceList IdNest : $	State 7	Action[3, .] :Shift 7
	IdNest→ ε • : $	Handle R3	Action[3, $]:Reduce R3
State 4 : V[[ ]	IndiceList→ [• expr] IndiceList : .	State 5	Action[4, expr]:Shift 5
	IndiceList→ [• expr] IndiceList : $	State 5	Action[4, expr]:Shift 5
State 5 : V[[expr]	IndiceList→ [expr •] IndiceList : .	State 9	Action[5,]]:Shift 9
	IndiceList→ [expr •] IndiceList : $	State 9	Action[5,]]:Shift 9
State 6 : V[id IndiceList Idnest]	Factor→id IndiceList Idnest • : $	Handle R1	Action[6, $] :Reduce R1
State 7 : V[.]	IdNest→. • id IndiceList IdNest : $	State 8	Action[7, id]:Shift 8
State 8 : V[. id]	IdNest→. id • IndiceList IdNest : $	State 11	Goto[8, IndiceList]:11
	IndiceList→•[expr] IndiceList : .	State 4	Action[8, [ ]:Shift 4
	IndiceList→ ε • : .	Handle R5	Action[8, .]:Reduce R5
	IndiceList→•[expr] IndiceList : $	State 4	Action[8, [ ]:Shift 4
	IndiceList→ ε • : $	Handle R5	Action[8, $]:Reduce R5
State 9 : V[[expr]]	IndiceList→ [expr] • IndiceList : .	State 10	Goto[9, IndiceList]:10
	IndiceList→ [expr] • IndiceList : $	State 10	Goto[9, IndiceList]:10
	IndiceList→•[expr] IndiceList : .	State 4	Action[9, [ ]:Shift 4
	IndiceList→ ε • : .	Handle R5	Action[9, .]:Reduce 5
	IndiceList→•[expr] IndiceList : $	State 4	Action[9, [ ]:Shift 4
	IndiceList→ ε • : $	Handle R5	Action[9, $]:Reduce 5
State 10 : V[[expr] IndiceList]	IndiceList→ [expr] IndiceList • : .	Handle R4	Action[10, .]:Reduce R4
	IndiceList→ [expr] IndiceList • : $	Handle R4	Action[10, $]:Reduce R4
State 11 V[. id IndiceList]	IdNest→. id IndiceList • IdNest : $	State 12	Goto[11, IdNest]:12
	IdNest→•.id IndiceList IdNest : $	State 7	Action[11, .]:Shift 7
	IdNest→ ε • : $	Handle R3	Action[11,$]:Reduce R3

State 12 : V[. id IndiceList IdNest]	<u>IdNest→. id IndiceList IdNest • : $</u>	Handle R2	Action[12, $]:Reduce R2

B. **[5%]** Draw a DFA that corresponds to the items sets found in your answer IV.A.



C. **[5%]** Construct the bottom-up parsing table for this grammar.

state	$	]	expr	[	id	.	Factor	IdNest	IndiceList
0					S2		1		
1	Acc								
2	R5					R5			3
3	R3					S7		6	
4			S5						
5		S9							
6	R1								
7					S8				
8	R5			S4		R5			11
9	R5			S4		R5			10
10	R4					R4			
11	R3					S7		12	
12	R2								

D. **[5%]** Using your constructed table, construct a parse trace for the input string ***id.id[expr].id***.

Stack	input	Action	Derivation Steps
0	id.id[expr].id$	S2	id.id[expr].id
0-id-2	.id[expr].id$	R5	⇐ id IL.id[expr].id
0-id-2-IL-3	id.id[expr].id$	S7	⇐ id IL.id[expr].id
0-id-2-IL-3-.-7	id[expr].id$	S8	⇐ id IL.id[expr].id
0-id-2-IL-3-.-7-id-8	[expr].id$	S4	⇐ id IL.id[expr].id
0-id-2-IL-3-.-7-id-8-[-4	expr].id$	S5	⇐ id IL.id[expr].id
0-id-2-IL-3-.-7-id-8-[-4-ex-5	].id$	S9	⇐ id IL.id[expr].id
0-id-2-IL-3-.-7-id-8-[-4-ex-5-]-9	.id$	R5	⇐ id IL.id[expr] IL.id
0-id-2-IL-3-.-7-id-8-[-4-ex-5-]-9-IL-10	.id$	R4	⇐ id IL.id IL.id
0-id-2-IL-3-.-7-id-8-IL-11	.id$	S7	⇐ id IL.id IL.id
0-id-2-IL-3-.-7-id-8-IL-11-.-7	id$	S8	⇐ id IL.id IL.id
0-id-2-IL-3-.-7-id-8-IL-11-.-7-id-8	$	R5	⇐ id IL.id IL.id IL
0-id-2-IL-3-.-7-id-8-IL-11-.-7-id-8-IL-11	$	R3	⇐ id IL.id IL.id IL IDN
0-id-2-IL-3-.-7-id-8-IL-11-.-7-id-8-IL-11-IDN-12	$	R2	⇐ id IL.id IL IDN
0-id-2-IL-3-.-7-id-8-IL-11-IDN-12	$	R2	⇐ id IL IDN
0-id-2-IL-3-IDN-6	$	R3	⇐ Fac
0-Fac-1	$	Acc	