# CSL302: Compiler Design

Mid Sem Examination Solutions (2025-26-M Semester)

**Max. Points:** 100 **Duration:** 2 hours

**September 19, 2025**

**Instructions:**

1. All the questions are compulsory
2. Don't ask any unnecessary questions during the exam. If you make any assumptions, write them down clearly.

<u>**Question-1:**</u> **[20 Points]**

Write a lexical analyzer that takes an input file, cleans the input file as per the following rules, and produces an output file which is cleaned.

a. Replaces all the tab spaces, more than one contiguous space into a single space.
b. Converts the amount listed in different foreign currencies ($, €) to Indian currency (₹). You can use the following conversion.
   i. $1 = ₹90
   ii. €1 = ₹100
c. Deletes unknown values having NULL, UNKNOWN, NOT AVAILABLE, and NA in the input file
d. Make all the following date formats to the consistent format DD-MM-YYYY
   i. YYYY-DD-MM to DD-MM-YYYY
   ii. DD/MM/YYYY to DD-MM-YYYY
e. Convert the words containing all the capital letters to smaller letters, except the following words.
   NASA, ISRO, AI, ML, SQL, CPU, GPU, DBMS, ID

You need not write the complete lex code, but you should write the code corresponding to the rules sections along with any functions you may be requiring.

**Solution:**

```
%{
        #include <stdio.h>
        #include <string.h>
%}
```

```
%%

/* --- Whitespace cleanup --- */ 2 Marks
[\t]+                        { printf(" "); }
"  "("  ")+                  { printf(" "); }

/* --- Currency conversion --- */  6 Marks (3 each for $ and €)
"\$"[0-9]+(\.[0-9]+)?   {                         /* 3 marks */
                             double val = atof(yytext+1);
                              printf("₹%.2f", val * 90);
                        }
"€"[0-9]+(\.[0-9]+)?    {                         /* 3 marks */
                             double val = atof(yytext+3);
                             printf("₹%.2f", val * 100);
                        }

/* --- Remove unwanted values --- */ 2 Marks (0.5 each)
"NULL"               { }
"UNKNOWN"                { }
"NOT AVAILABLE"          { }
"NA"                 { }

/* --- Date conversions --- */ 6 Marks
[0-9]{4}-[0-9]{2}-[0-9]{2}  {                     /* 3 marks */
                        int y,d,m;
                        sscanf(yytext, "%d-%d-%d", &y, &d, &m);
                        printf("%02d-%02d-%04d", d, m, y);
                    }

[0-9]{2}/[0-9]{2}/[0-9]{4}  {                     /* 3 marks */
                        int d,m,y;
                        sscanf(yytext, "%d/%d/%d", &d, &m, &y);
                        printf("%02d-%02d-%04d", d, m, y);
                    }

/* --- Uppercase word conversion --- */ 3 Marks
[A-Z]+              {
                    if (strcmp(yytext,"NASA")==0 || strcmp(yytext,"ISRO")==0 ||
                        strcmp(yytext,"AI")==0   || strcmp(yytext,"ML")==0   ||
                        strcmp(yytext,"SQL")==0  || strcmp(yytext,"CPU")==0  ||
                        strcmp(yytext,"GPU")==0  || strcmp(yytext,"DBMS")==0 ||
                        strcmp(yytext,"ID")==0) {
                        printf("%s", yytext);
                    } else {
                        for(int i=0; yytext[i]; i++) yytext[i] = tolower(yytext[i]);
                        printf("%s", yytext);
                    }
                }
```

**/* --- Default: copy input --- */ 1 Marks**

```
.|\n                {    printf("%s", yytext);  }

%%

int main() {
    yylex();
    return 0;
}
```

**Question-2:** **[7 Points]**

1. Show that the following grammar is ambiguous with an example string with two left most or two right most derivations.
   $$S \rightarrow S\ S \mid (S) \mid \varepsilon$$
2. Rewrite the above grammar to make it unambiguous. Take the same example as in the first part, and show a unique left-most or right-most derivation.

**Solution:**

**Part-1: 4 Marks**

Example input: ()

Left-most derivation-1: $S \rightarrow SS \rightarrow S$ (Expanded first S with ε) $\rightarrow (S) \rightarrow ()$

Left-most derivation-2: $S \rightarrow SS \rightarrow (S)S \rightarrow ()S \rightarrow ()$

**Part-2:**

New grammar: **2 Marks.**

$S \rightarrow (S)\ S \mid \varepsilon$

For the same input string ()

Left-most derivation-1: $S \rightarrow (S)S \rightarrow ()S \rightarrow ()$ **1 Mark.**

**Question-3:** **[20 Points]**

Write a CFG for a mini assembly language with the following features.

**Tokens:** ALU_OP, REG, JUMP, JNZ, HLT_OP, LABEL, CONST

1. An assembly program contains zero or more statements.
2. Each statement can be one of the following:
   a. Arithmetic statement
   b. Conditional statement
   c. Halt statement
3. The arithmetic statement starts with ALU_OP followed by 3 operands, which are separated by commas. The first operands should be REG, the second and third operand can be REG or CONST. Examples are
   a. ALU_OP REG, CONST, REG
   b. ALU_OP REG, REG, REG
4. Conditional statements start with JUMP or JNZ. If the statement starts with a JUMP then it is followed by LABEL. If it starts with JNZ, it should have two operands REG, LABEL. The following are two valid examples.
   a. JUMP LABEL
   b. JNZ REG LABEL
5. Halt statement starts with HLT_OP and does not have any operands.
6. Each statement can be optionally prefixed with **LABEL:**

## Solution:

---

<program> → <statements>

<statements> → <statement> <statements> 4 Marks
| ε

---

<statement> → <opt_label> <arithmetic_stmt>
| <opt_label> <conditional_stmt> 4 Marks
| <opt_label> <halt_stmt>

---

<opt_label> → LABEL:
| ε 2 Marks

---

<arithmetic_stmt> → ALU_OP REG ,<operand> , <operand>

<operand> → REG 4 Marks
| CONST

---

<conditional_stmt> → JUMP LABEL
| JNZ REG LABEL 4 Marks

---

<halt_stmt> → HLT_OP 2 Marks

---

**Question-4:** **[25 Points]**

Consider the following grammar:

A → A x B | C y
B → z A y
C → x A | ε

where {A, B, C} is the set of nonterminal symbols, A is the start symbol, and {x, y, z} is the set of terminal symbols.

(a) Compute the FIRST and FOLLOW sets. Is the above grammar LL(1)? Justify using the parser table.
(b) If the grammar is not LL(1), rewrite the grammar to make it LL(1)
(c) Reconstruct the FIRST, FOLLOW sets, and the LL(1) parser table on the revised grammar and show that the grammar is LL(1).

**Answer**:

**Section(a):** **[10 Marks] 6 Marks for First and Follow sets, 2.5 Marks for parser table, 1.5 marks for the justification**

First(A) = { x, y}
First(B) = { z}
First(C) = { x, ε}
Follow(A) = { $, x, y}
Follow(B) = { $, x, y}
Follow(C) = { y}

No, the given grammar is **not LL(1)**, as it contains a Left-recursive production for A:

**A** → **A** x B | C y

Justification using the Parser Table:

1. A → A x B
2. A → C y
3. B → z A y
4. C → x A
5. C → ε

| | **x** | **y** | **z** | **$** |
|---|---|---|---|---|
| **A** | (1) A → A x B<br>(2) A → C y<br>*Conflict* | (1) A → A x B<br>(2) A → C y<br>*Conflict* | | |
| **B** | | | (3) B → z A y | |
| **C** | (4) C → x A | (5) C → ε | | |

**Section(b): [3 Marks]**

In order to make it LL(1) we have to make it **Left-recursion free.**

A → A x B | C y
B → z A y
C → x A | ε

will become:
A → C y A'
A' → x B A' | ε
B → z A y
C → x A | ε

**Section(c): [12 Marks] 8 Marks for First and Follow sets, 4 Marks for parser table.**

First(A) = { x, y}
First(A') = { x, ε}
First(B) = { z}
First(C) = { x, ε}
Follow(A) = { $, y}
Follow(A') = { $, y}
Follow(B) = { $, x, y}
Follow(C) = { y}

Updated Parser Table will be:

1. A → C y A'
2. A' → x B A'
3. A' → ε
4. B → z A y
5. C → x A
6. C → ε

| | x | y | z | $ |
|---|---|---|---|---|
| **A** | (1) A → C y A’ | (1) A → C y A’ | | |
| **A’** | (2) A’ → x B A’ | (3) A’ → ε | | (3) A’ → ε |
| **B** | | | (4) B → z A y | |
| **C** | (5) C → x A | (6) C → ε | | |

**<u>Question-5:</u>** **[28 Points]**

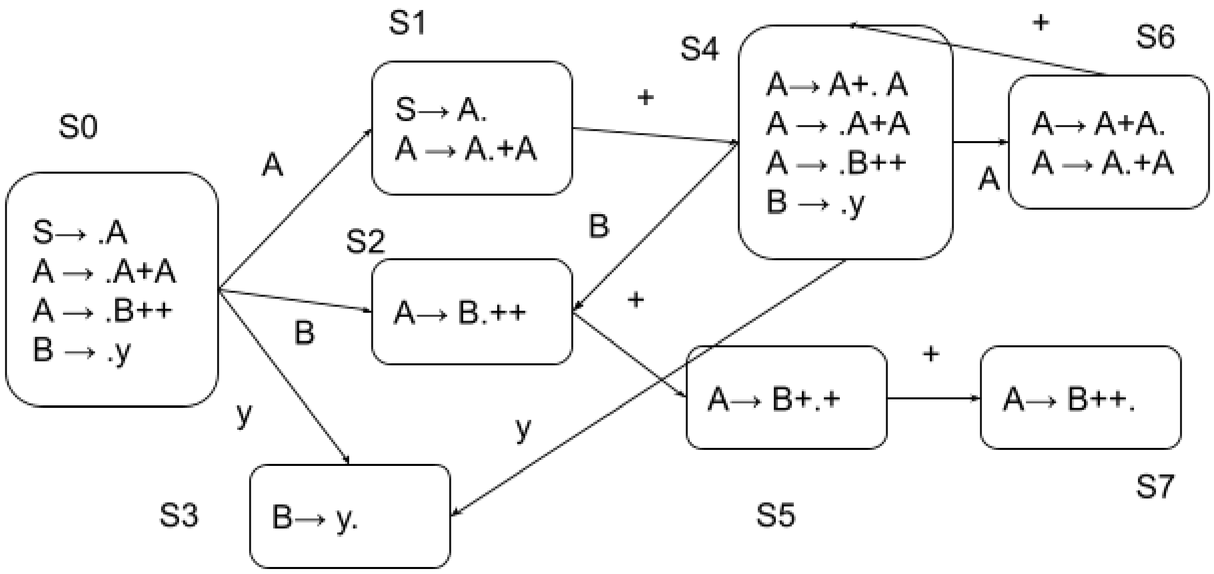
Consider the following grammar:

(1) S → A
(2) A → A+A
(3) A → B++
(4) B → y

(a) Is the grammar LR(0)? Justify your answer by showing DFA and the LR(0) parser table.
(b) Trace the input string y+++y++ using the LR(0) parser table, and check if it can be accepted by the LR(0) algorithm. If not, show the step where the conflict occurs.
(c) Repeat the above both Part-(a) and Part-(b) for the SLR algorithm.

Part-(a): **12 Marks (8 Marks for DFA and 4 Marks for parser table, only reductions are counted in the table.).**

DFA:



LR(0) Parser table

| **State** | **+** | **y** | **$** | **A** | **B** |
|---|---|---|---|---|---|
| **S0** |  | S3 |  | S1 | S2 |
| **S1** | S4/r1 | r1 | r1 |  |  |
| **S2** | S5 |  |  |  |  |
| **S3** | r4 | r4 | r4 |  |  |
| **S4** |  | S3 |  | S6 | S2 |
| **S5** | S7 |  |  |  |  |
| **S6** | S4/r2 | r2 | r2 |  |  |
| **S7** | r3 | r3 | r3 |  |  |

The grammar is not LR(0) as there are conflicts.

(b): Input string: **4 Marks: Each row carries 0.5 Marks and 1 Marks for the conflict**

y+++y++

| **State** | **Stack** | **Token Stream** | **Action Taken** |
|---|---|---|---|
| S0 | | y+++y++$ | Shift y |
| S3 | y | +++y++$ | Reduce B→ y |
| S2 | B | +++y++$ | Shift + |
| S5 | B+ | ++y++$ | Shift + |
| S7 | B++ | +y++$ | Reduce A→ B++ |
| S1 | A | +y++$ | Conflict |

Part-(C): DFA is same as part-(a)

Follow(S): {$}
Follow(A): {$, +}
Follow(B): {+}

SLR Parser table (**4 Marks, only reductions are counted.**)

| **State** | **+** | **y** | **$** | **A** | **B** |
|---|---|---|---|---|---|
| **S0** | | S3 | | S1 | S2 |
| **S1** | S4 | | r1 | | |
| **S2** | S5 | | | | |
| **S3** | r4 | | | | |
| **S4** | | S3 | | S6 | S5 |
| **S5** | S7 | | | | |
| **S6** | S4/r2 | | r2 | | |
| **S7** | r3 | | r3 | | |

The grammar is not SLR as there are conflicts.

Input parsing: (**8 Marks**: Each row carries 0.5 Marks and 2 Marks for the conflict)

| **State** | **Stack** | **Token Stream** | **Action Taken** |
|---|---|---|---|
| S0 | | y+++y++$ | Shift y |
| S3 | y | +++y++$ | Reduce B→ y |
| S2 | B | +++y++$ | Shift + |
| S5 | B+ | ++y++$ | Shift + |
| S7 | B++ | +y++$ | Reduce A→ B++ |
| S1 | A | +y++$ | Shift + |
| S4 | A+ | y++$ | Shift y |
| S3 | A+y | ++$ | Reduce B→ y |
| S2 | A+B | ++$ | Shift + |
| S5 | A+B+ | +$ | Shift + |
| S7 | A+B++ | $ | Reduce A→ B++ |
| S6 | A+A | $ | Reduce A→ A+A |
| S1 | A | $ | Reduce S→ A |