ఇండియన్ ఇన్స్టిట్యూట్ ఆఫ్ ఇన్ఫర్మేషన్ టెక్నాలజీ,
డిజైన్ అండ్ మాన్యుఫ్యాక్చరింగ్, కర్నూలు

भारतीय सूचना प्रौद्योगिकी, अभिकल्पना
एवं विनिर्माण संस्थान, कर्नूल

Department of Computer Science and Engineering, Dec 2024- April 2025

Compiler Design Practice (S6-B.Tech) -Assignment 5

Policies for Submission and Evaluation

You must submit your assignment in the moodle (Eduserver) course page, on or before the ubmission deadline. Also, ensure that your programs in the assignment must compile and execute without. During evaluation your uploaded programs will be checked. Failure to execute programs in the assignment without compilation errors may lead to zero marks for that program.

Your submission will also be tested for plagiarism, by automated tools. In case your code fails to pass the test, you will be straightaway awarded zero marks for this assignment and considered by the examiner for awarding **“F”** grade in the course. Detection of ANY malpractice regarding the lab course will also lead to awarding an **“F”** grade.

Last date for submitting : 20/03/2025 08:00 Hrs

Naming Conventions for Submission

Submit a single ZIP (.zip) file (do not submit in any other archived formats like .rar or .tar.gz). The name of this file must be ASSG<NUMBER>_<ROLLNO>_<FIRSTNAME>.zip. (eg: ASSG1_118cs0006_LAXMAN.zip). DO NOT add any other files (like temporary files, inputfiles, etc.) except your source code, into the zip archive. The source codes must be named as ASSG<NUMBER>_<ROLLNO>_<FIRSTNAME>_<PROGRAM-NO>.<extension>. (For example: ASSG1_118cs0006_LAXMAN_1.c). If there are multiple parts for a particular question, then name the source files for each part separately as in ASSG1_118cs0006_LAXMAN_1b.c.

If you do not conform to the above naming conventions, your submission might not be recognized by some automated tools, and hence will lead to a score of 0 for the submission. So, **make sure that you follow the naming conventions.**

Q 1:
Design and implement a **recursive descent parser** for a given **context-free grammar (CFG)**. The parser should handle a subset of arithmetic expressions, including addition, subtraction, multiplication, and division with proper operator precedence.

Grammar:
E → T + E | T - E | T
T → F * T | F / T | F
F → (E) | id

Sample Input:
(a + b) * c - d / e
Sample Output:
Valid Expression
Parse Tree:

```
   (-)
  /   \
 (*)   (/)
 / \   / \
a  (+) d   e
   / \
  b   c
```

Q2
Develop an **LL(1) parser** for a given **CFG**. Construct the **FIRST** and **FOLLOW** sets, generate the parsing table, and implement a predictive parsing algorithm.
Grammar:
E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

Sample Input:
id + id * id
Sample Output:
Parsing Steps:

Stack	Input	Action
E$	id+id*id$	Expand E → TE'
TE'$	id+id*id$	Expand T → FT'
FT'E'$	id+id*id$	Expand F → id
T'E'$	+id*id$	Expand E' → +TE'
+TE'$	+id*id$	Match +
TE'$	id*id$	Expand T → FT'
FT'E'$	id*id$	Expand F → id
T'E'$	*id$	Expand T' → *FT'
*FT'E'$	*id$	Match *
FT'E'$	id$	Expand F → id
T'E'$	ε	Reduce T' → ε
E'$	ε	Reduce E' → ε

Success: Input is valid.

Q3

Implement an **LR(0) parser** for a given **CFG**. Construct the **LR(0) automaton**, generate the parsing table, and implement the shift-reduce parsing algorithm.

Grammar:

S → CC
C → cC | d
Sample Input:
cccd

Sample Output:
Parsing Steps:
Stack | Input | Action

$ | cccd$ | Shift c
$c | ccd$ | Shift c
$cc | cd$ | Shift c
$ccc | d$ | Shift d
$cccd | $ | Reduce C → d
$cccC | $ | Reduce C → cC
$ccC | $ | Reduce C → cC
$C | $ | Reduce S → CC
$S | $ | Success

Q4

Construct an **SLR(1) parser** using **FOLLOW** sets for conflict resolution. Implement the shift-reduce parser based on the generated table.

Grammar:

E → E + T | T
T → T * F | F
F → (E) | id

Sample Input:
id + id * id

Sample Output:

Valid Expression
Shift-Reduce Steps:
Shift id
Reduce F → id
Reduce T → F
Shift +
Shift id
Reduce F → id
Reduce T → F
Shift *
Shift id

Reduce F → id
Reduce T → T * F
Reduce E → E + T
Success

Q5
Enhance an **SLR(1) parser** to construct an **LALR(1) parser** by merging similar states. Implement the parser for a subset of programming language constructs.

Grammar:

S → L = R | R
L → * R | id
R → L
Sample Input:
id = * id

Sample Output:
Parsing Steps:
Shift id
Reduce L → id
Shift =
Shift *
Shift id
Reduce L → id
Reduce R → L
Reduce L → * R
Reduce R → L
Reduce S → L = R
Success

Q6
Modify an **LL(1) or LR(1) parser** to include an **error recovery mechanism**. Implement panic mode recovery and phrase-level recovery techniques.

Grammar:

E → E + T | T
T → T * F | F
F → (E) | id

Sample Input:
id + * id
Sample Output:
Syntax Error at position 4: Unexpected token '*'
Error Recovery: Skipping '*'
Valid Parsing Resumed

Q7
Develop a parser using **YACC (Yet Another Compiler Compiler)** to evaluate arithmetic expressions involving +, -, *, / with proper precedence and associativity.
E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | number
Sample Input:
3 + 5 * 2

Sample Output:
Number: 3
Number: 5
Number: 2
Multiplication
Addition
Q8
Write a **YACC parser** for an **LL(1) grammar** that recognizes a simple assignment statement syntax.

Grammar:

S → id = E
E → E + T | T
T → T * F | F
F → id | number
Sample Input:
x = y + 3 * z
Sample Output:
Identifier: x
Assignment Operator: =
Identifier: y
Number: 3
Identifier: z
Multiplication
Addition

Q9
Develop a **YACC parser** for Boolean expressions using operators **AND, OR, NOT**.

Grammar:

E → E OR T | T
T → T AND F | F
F → NOT F | (E) | id
Sample Input:
a AND b OR NOT c
Sample Output:
Identifier: a
Identifier: b
AND operation
Identifier: c
NOT operation
OR operation

Q10
Write a **YACC parser** to recognize and validate **C-style if-else statements**.

Grammar:

S → if (E) S else S | if (E) S | statement
E → id < id | id > id | id == id
statement → id = id;
Sample Input:
if (x < y)
  z = x;
else
  z = y;
Sample Output:
If statement detected
Condition: x < y
Assignment: z = x
Else statement detected
Assignment: z = y