**SPCC CODES**

1: Program to generate Three Address Code (TAC)  
  
def is\_operator(c):

return c in "+-\*/"

def precedence(op):

if op in ('+', '-'):

return 1

if op in ('\*', '/'):

return 2

return 0

def infix\_to\_postfix(expression):

stack = []

postfix = []

tokens = expression.split()

for token in tokens:

if token.isalnum(): # operand (e.g., a, b, c, d, x1)

postfix.append(token)

elif is\_operator(token):

while stack and precedence(stack[-1]) >= precedence(token):

postfix.append(stack.pop())

stack.append(token)

elif token == '(':

stack.append(token)

elif token == ')':

while stack and stack[-1] != '(':

postfix.append(stack.pop())

stack.pop() # pop '('

while stack:

postfix.append(stack.pop())

return postfix

def generate\_TAC(expression):

# assuming expression is like "a = b + c \* d"

lhs, rhs = expression.split('=')

lhs = lhs.strip()

rhs = rhs.strip()

postfix = infix\_to\_postfix(rhs)

stack = []

temp\_count = 1

tac = []

for token in postfix:

if token.isalnum():

stack.append(token)

elif is\_operator(token):

op2 = stack.pop()

op1 = stack.pop()

temp = f"t{temp\_count}"

tac.append(f"{temp} = {op1} {token} {op2}")

stack.append(temp)

temp\_count += 1

final\_result = stack.pop()

tac.append(f"{lhs} = {final\_result}")

return tac

# Example usage

expr = input("Enter an expression (e.g., a = b + c \* d): ")

tac\_code = generate\_TAC(expr)

print("\nThree Address Code:")

for line in tac\_code:

print(line)

2. Program to implement Multi-Pass Macroprocessor for the given Assembly

Language (Pass1).

import re

# Function to process macro definitions and macro calls

def process\_macros(source\_code):

macros = {} # Store macro definitions

mnt = {} # Macro Name Table (MNT)

mdt = [] # Macro Definition Table (MDT)

at = {} # Argument Table (AT)

current\_macro\_name = None

current\_params = []

current\_body = []

# Split the source code into lines

lines = source\_code.splitlines()

# Pass 1: Find macro definitions and macro calls

for line in lines:

line = line.strip()

# Detect macro definition (MACRO <name> <params>)

macro\_def = re.match(r"MACRO\s+(\w+)\s\*(.\*)", line)

if macro\_def:

# If we encounter a macro definition, save the previous macro if it exists

if current\_macro\_name:

macros[current\_macro\_name] = {'params': current\_params, 'body': current\_body}

mnt[current\_macro\_name] = len(mdt) # Add to MNT

mdt.extend(current\_body) # Add the body of the previous macro to MDT

current\_macro\_name = macro\_def.group(1)

current\_params = macro\_def.group(2).split(",") if macro\_def.group(2) else []

current\_body = []

continue

# Detect end of macro definition (MEND)

if line.startswith("MEND"):

# Save the macro and its body to MDT

if current\_macro\_name:

macros[current\_macro\_name] = {'params': current\_params, 'body': current\_body}

mnt[current\_macro\_name] = len(mdt) # Add to MNT

mdt.extend(current\_body) # Add the body of the current macro to MDT

current\_macro\_name = None

current\_params = []

current\_body = []

continue

# If we are inside a macro definition, add the line to the body

if current\_macro\_name:

current\_body.append(line)

continue

# Process macro calls (Replace with placeholders in the MDT)

macro\_call = re.match(r"(\w+)\s\*(.\*)", line)

if macro\_call:

macro\_name = macro\_call.group(1)

if macro\_name in macros:

params = macro\_call.group(2).split(",") if macro\_call.group(2) else []

at[macro\_name] = params # Add argument mapping to AT

# Replace macro call with placeholder

mdt.append(f"CALL {macro\_name}")

# Return all tables and processed code (MDT and MNT)

return macros, mnt, mdt, at

# Sample Assembly Code

source\_code = """

MACRO ADD X, Y

LOAD X

ADD Y

MEND

MACRO SUB A, B

LOAD A

SUB B

MEND

START

ADD 5, 3

SUB 7, 4

END

"""

# Process the source code

macros, mnt, mdt, at = process\_macros(source\_code)

# Output the results of Pass 1

print("\nMacro Definition Table (MDT):")

for line in mdt:

print(line)

print("\nMacro Name Table (MNT):")

for name, idx in mnt.items():

print(f"{name}: {idx}")

print("\nArgument Table (AT):")

for macro, args in at.items():

print(f"{macro}: {', '.join(args)}")

3. Program to implement the FIRST set for the given grammar

def compute\_first(symbol, grammar, first\_sets):

# If FIRST already computed, return it

if symbol in first\_sets:

return first\_sets[symbol]

first = set()

# If terminal, FIRST is itself

if not symbol.isupper():

first.add(symbol)

return first

for production in grammar[symbol]:

# If production is epsilon

if production == "ε":

first.add("ε")

else:

i = 0

while i < len(production):

sym = production[i]

# Get FIRST of current symbol

sym\_first = compute\_first(sym, grammar, first\_sets)

# Add all except epsilon

first.update(sym\_first - {"ε"})

# If ε not in FIRST(sym), stop

if "ε" not in sym\_first:

break

i += 1

# If ε in all symbols, add ε

if i == len(production):

first.add("ε")

# Cache result

first\_sets[symbol] = first

return first

grammar = {

"E": ["TE'"],

"E'": ["+TE'", "ε"],

"T": ["FT'"],

"T'": ["\*FT'", "ε"],

"F": ["(E)", "id"]

}

first\_sets = {}

for non\_terminal in grammar:

compute\_first(non\_terminal, grammar, first\_sets)

print("\nFIRST sets:")

for nt, first in first\_sets.items():

print(f"FIRST({nt}) = {{ {', '.join(sorted(first))} }}")

4. Program to implement the FOLLOW set for the given grammar

def compute\_follow(grammar, first\_sets, non\_terminals, start\_symbol):

follow\_sets = {non\_terminal: set() for non\_terminal in grammar}

follow\_sets[start\_symbol].add('$')

updated = True

while updated:

updated = False

for non\_terminal, productions in grammar.items():

for production in productions:

for i, symbol in enumerate(production):

if symbol in non\_terminals:

if i + 1 < len(production):

beta = production[i + 1:]

for first\_symbol in first\_sets[beta[0]]:

if first\_symbol != 'ε':

if first\_symbol not in follow\_sets[symbol]:

follow\_sets[symbol].add(first\_symbol)

updated = True

if 'ε' in compute\_first\_of\_sequence(beta, first\_sets):

for follow\_b in follow\_sets[non\_terminal]:

if follow\_b not in follow\_sets[symbol]:

follow\_sets[symbol].add(follow\_b)

updated = True

elif i == len(production) - 1:

for follow\_b in follow\_sets[non\_terminal]:

if follow\_b not in follow\_sets[symbol]:

follow\_sets[symbol].add(follow\_b)

updated = True

return follow\_sets

def compute\_first\_of\_sequence(symbols, first\_sets):

first = set()

for symbol in symbols:

first.update(first\_sets[symbol] - {'ε'})

if 'ε' not in first\_sets[symbol]:

return first

first.add('ε')

return first

if \_\_name\_\_ == '\_\_main\_\_':

grammar\_input = {

'S': [['A', 'B']],

'A': [['a'], ['ε']],

'B': [['b']]

}

first\_input = {

'S': {'a', 'b'},

'A': {'a', 'ε'},

'B': {'b'},

'a': {'a'},

'b': {'b'}

}

non\_terminals\_input = {'S', 'A', 'B'}

start\_symbol\_input = 'S'

follow\_result = compute\_follow(grammar\_input, first\_input, non\_terminals\_input, start\_symbol\_input)

print("FOLLOW sets:")

for non\_terminal, follow\_set in follow\_result.items():

print(f"FOLLOW({non\_terminal}) = {follow\_set}")

print("\n--- Another Example ---")

grammar\_input\_2 = {

'E': [['T', 'E\'']],

'E\'': [['+', 'T', 'E\''], ['ε']],

'T': [['F', 'T\'']],

'T\'': [['\*', 'F', 'T\''], ['ε']],

'F': [['(', 'E', ')'], ['id']]

}

first\_input\_2 = {

'E': {'(', 'id'},

'E\'': {'+', 'ε'},

'T': {'(', 'id'},

'T\'': {'\*', 'ε'},

'F': {'(', 'id'},

'+': {'+'},

'\*': {'\*'},

'(': {'('},

')': {')'},

'id': {'id'}

}

non\_terminals\_input\_2 = {'E', 'E\'', 'T', 'T\'', 'F'}

start\_symbol\_input\_2 = 'E'

follow\_result\_2 = compute\_follow(grammar\_input\_2, first\_input\_2, non\_terminals\_input\_2, start\_symbol\_input\_2)

print("FOLLOW sets:")

for non\_terminal, follow\_set in follow\_result\_2.items():

print(f"FOLLOW({non\_terminal}) = {follow\_set}")

5. Program to implement LL(1) parser

FIRST and FOLLOW will be given, Parsing table is expected

# Given Grammar

grammar = {

"E": ["T E'"],

"E'": ["+ T E'", "ε"],

"T": ["F T'"],

"T'": ["\* F T'", "ε"],

"F": ["( E )", "id"]

}

# Given FIRST sets

first\_sets = {

"E": {"(", "id"},

"E'": {"+", "ε"},

"T": {"(", "id"},

"T'": {"\*", "ε"},

"F": {"(", "id"}

}

# Given FOLLOW sets

follow\_sets = {

"E": {")", "$"},

"E'": {")", "$"},

"T": {"+", ")", "$"},

"T'": {"+", ")", "$"},

"F": {"\*", "+", ")", "$"}

}

# Terminals and Non-terminals

non\_terminals = list(grammar.keys())

terminals = {"id", "+", "\*", "(", ")", "$"}

# Initialize parsing table

parsing\_table = {nt: {t: "" for t in terminals} for nt in non\_terminals}

# Helper: get FIRST of a string of symbols

def get\_first\_of\_string(symbols):

first = set()

for symbol in symbols.split():

if symbol not in first\_sets:

first.add(symbol) # terminal

break

first |= (first\_sets[symbol] - {"ε"})

if "ε" not in first\_sets[symbol]:

break

else:

first.add("ε")

return first

# Fill the LL(1) parsing table

for head in grammar:

for production in grammar[head]:

first = get\_first\_of\_string(production)

for terminal in (first - {"ε"}):

parsing\_table[head][terminal] = production

if "ε" in first:

for follow\_symbol in follow\_sets[head]:

parsing\_table[head][follow\_symbol] = "ε"

# Print the parsing table

print("LL(1) Parsing Table:")

print(f"{'':<5}", end='')

for t in sorted(terminals):

print(f"{t:<10}", end='')

print()

for nt in non\_terminals:

print(f"{nt:<5}", end='')

for t in sorted(terminals):

prod = parsing\_table[nt][t]

print(f"{prod:<10}", end='')

print()

6. Program to remove Left Recursion from the given grammar

Grammar will be given, modified grammar without left recursion is expected

def remove\_left\_recursion(grammar):

new\_grammar = {}

for non\_terminal in grammar:

alpha = [] # left recursive productions (A → Aα)

beta = [] # non-left recursive productions (A → β)

for production in grammar[non\_terminal]:

if production.startswith(non\_terminal):

alpha.append(production[len(non\_terminal):]) # Remove the NT from the left

else:

beta.append(production)

# If left recursion exists

if alpha:

new\_nt = non\_terminal + "'" # A' as new NT

new\_grammar[non\_terminal] = [b + new\_nt for b in beta]

new\_grammar[new\_nt] = [a + new\_nt for a in alpha]

new\_grammar[new\_nt].append("ε") # add epsilon

else:

new\_grammar[non\_terminal] = grammar[non\_terminal]

return new\_grammar

# Example grammar with left recursion

grammar = {

"A": ["Aa", "b"],

"B": ["Bc", "Bd", "e"]

}

print("Original Grammar:")

for nt in grammar:

print(f"{nt} -> {' | '.join(grammar[nt])}")

# Remove Left Recursion

updated\_grammar = remove\_left\_recursion(grammar)

print("\nGrammar after removing Left Recursion:")

for nt in updated\_grammar:

print(f"{nt} -> {' | '.join(updated\_grammar[nt])}")

7. Program to develop an Optimized Code.

Sample Code will be given, optimized code is expected

def constant\_folding(code):

optimized = []

replacements = {}

for line in code:

parts = line.split('=')

if len(parts) != 2:

optimized.append(line)

continue

lhs = parts[0].strip()

rhs = parts[1].strip()

try:

result = eval(rhs)

replacements[lhs] = str(result)

optimized.append(f"{lhs} = {result}")

except:

# Replace known constants in the RHS

for var, val in replacements.items():

rhs = rhs.replace(var, val)

optimized.append(f"{lhs} = {rhs}")

return optimized

def common\_subexpression\_elimination(code):

expressions = {}

optimized = []

for line in code:

if '=' not in line:

optimized.append(line)

continue

lhs, rhs = [x.strip() for x in line.split('=')]

# Skip replacement if RHS is a constant value

try:

eval(rhs)

# it's a constant, don't replace with a variable

optimized.append(f"{lhs} = {rhs}")

continue

except:

pass

if rhs in expressions:

optimized.append(f"{lhs} = {expressions[rhs]}")

else:

expressions[rhs] = lhs

optimized.append(f"{lhs} = {rhs}")

return optimized

def optimize\_code(code):

print("\nStep 1: Constant Folding")

code = constant\_folding(code)

for c in code:

print(c)

print("\nStep 2: Common Subexpression Elimination")

code = common\_subexpression\_elimination(code)

for c in code:

print(c)

return code

# 🔸 Sample input code

sample\_code = [

"t1 = 4 + 5",

"t2 = a + b",

"t3 = 4 + 5",

"t4 = t1 + t2",

"t5 = t3 + t2",

"t6 = t1 + t2",

"t7 = 2 \* 3"

]

print("Original Code:")

for line in sample\_code:

print(line)

# 🔧 Optimize

final\_code = optimize\_code(sample\_code)

print("\n🔹 Final Optimized Code:")

for line in final\_code:

print(line)

8. Program to implement Lexical Analyzer

import re

# Define token categories

keywords = {"int", "float", "char", "if", "else", "while", "for", "return", "void"}

operators = {'+', '-', '\*', '/', '=', '==', '!=', '<', '>', '<=', '>=', '&&', '||'}

separators = {'(', ')', '{', '}', ';', ','}

# Classify each token

def classify\_token(token):

if token in keywords:

return "Keyword"

elif token in operators:

return "Operator"

elif token in separators:

return "Separator"

elif re.fullmatch(r'\d+(\.\d+)?', token):

return "Literal"

elif re.fullmatch(r'[a-zA-Z\_]\w\*', token):

return "Identifier"

else:

return "Unknown"

# Lexical Analyzer

def lexical\_analyzer(code):

# Improved token pattern: float, int, words, multi-char ops, single-char ops/seps

tokens = re.findall(r'\d+\.\d+|\d+|==|!=|<=|>=|[a-zA-Z\_]\w\*|[^\s]', code)

token\_types = {

"Keyword": set(),

"Identifier": set(),

"Operator": set(),

"Literal": set(),

"Separator": set(),

"Unknown": set()

}

for token in tokens:

category = classify\_token(token)

token\_types[category].add(token)

# Print grouped results

print("=== Lexical Analyzer Output (Grouped by Type) ===\n")

for category in ["Keyword", "Identifier", "Operator", "Literal", "Separator", "Unknown"]:

if token\_types[category]:

print(f"{category}s: {', '.join(sorted(token\_types[category]))}")

# 🔸 Sample Code

sample\_code = """

int a = 5;

float b = 3.14;

if (a > b) {

a = a + b;

}

"""

lexical\_analyzer(sample\_code)

9. LEX Program to check whether input character is vowel or consonants.

%{

#include <stdio.h>

int yywrap(void){

return 1;

}

%}

%%

[aeiouAEIOU] { printf("'%s' is a VOWEL\n", yytext); }

[a-zA-Z] { printf("'%s' is a CONSONANT\n", yytext); }

.|\n { /\* Ignore any other characters \*/ }

%%

int main() {

printf("Enter a character: ");

yylex();

return 0;

}

Save the code in a file named vowel\_consonant.l

lex vowel\_consonant.l

gcc lex.yy.c -o vowel\_consonant

./vowel\_consonant

10. LEX Program to check whether the string is a word or a number.

%{

#include <stdio.h>

int yywrap(void);

%}

%%

[0-9]+(\.[0-9]+)? { printf("'%s' is a NUMBER\n", yytext); }

[a-zA-Z]+ { printf("'%s' is a WORD\n", yytext); }

.|\n { /\* Ignore other characters \*/ }

%%

int yywrap(void) {

return 1;

}

int main() {

printf("Enter input: ");

yylex();

return 0;

}

11. EX Program to count the number of lines, words and characters in a text.

%{

#include <stdio.h>

int line\_count = 0;

int word\_count = 0;

int char\_count = 0;

int yywrap(void);

%}

%%

\n { line\_count++; char\_count++; }

[ \t]+ { char\_count += yyleng; } // Spaces and tabs (word separators)

[a-zA-Z0-9]+ { word\_count++; char\_count += yyleng; } // Words (letters/numbers)

. { char\_count += yyleng; } // All other characters

%%

int yywrap(void) {

return 1;

}

int main() {

printf("Enter text (Ctrl+D to end input):\n");

yylex();

printf("\nLines: %d\nWords: %d\nCharacters: %d\n", line\_count, word\_count, char\_count);

return 0;

}

12. LEX Program to implement Lexical Analyzer

%{

#include <stdio.h>

#include <string.h>

char \*keywords[] = {

"int", "float", "char", "double", "if", "else", "while", "for", "return", "void"

};

int isKeyword(char \*str) {

for(int i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++) {

if(strcmp(str, keywords[i]) == 0)

return 1;

}

return 0;

}

%}

%%

[ \t\n] ; // Ignore whitespace

"=="|"!="|"<="|">="|"="|"<"|">" { printf("Relational Operator: %s\n", yytext); }

"\*"|"+"|"-"|"/" { printf("Arithmetic Operator: %s\n", yytext); }

[0-9]+\.[0-9]+ { printf("Float Literal: %s\n", yytext); }

[0-9]+ { printf("Integer Literal: %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* {

if (isKeyword(yytext))

printf("Keyword: %s\n", yytext);

else

printf("Identifier: %s\n", yytext);

}

. { printf("Special Symbol: %s\n", yytext); }

%%

int main() {

printf("Enter the input:\n");

yylex();

return 0;

}

int yywrap() {

return 1;

}