for lex:

flex [name of file].l

gcc lex.yy.c

a.exe

for lex and bison

flex e1.l

bison -dy e1.y

gcc lex.yy.c y.tab.c

A.exe

\*\*KIT:

%{

#include<stdio.h>

#include<stdlib.h>

int keyword=0;

int identifier=0;

int integer=0;

int float\_number=0;

int operator\_count=0;

%}

%%

"while"|"if"|"else"|"return"|"int"|"char"|"float" { keyword++; printf("Keyword : %s\n", yytext); }

[a-zA-Z\_][a-zA-Z0-9\_]\* { identifier++; printf("Identifier : %s\n", yytext); }

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

[0-9]\*"."[0-9]+ { float\_number++; printf("\t float : %s\n", yytext); }

"<="|"=="|"="|"++"|"-"|"\*"|"+" { operator\_count++; printf("\t operator : %s\n", yytext); }

.;

%%

int main()

{

yyin = fopen("C:\\Users\\ANUJ BELGAONKAR\\Desktop\\Practical1\\e1text.txt", "r");

yylex();

return 0;

}

int yywrap()

{

return 1;

}

KIT.txt:

#include<stdio.h>

int main(){

int a;

a = 5;

print("%d", a);

int b;

b = a+5;

print("%d", b);

return 0;

}

\*\*CGPA Company:

%{

#include <stdio.h>

#include <string.h>

int tcs = 0, infosys = 0, wipro = 0, accenture = 0, informatica = 0;

int males = 0, females = 0, cse = 0, it = 0, ec = 0;

%}

%%

[ \t]+ ; // Skip whitespace

[0-9]+\.[0-9]+ { float cgpa = atof(yytext); if (cgpa < 10) printf("CGPA: %s\n", yytext); }

[0-9]{10} { printf("Mobile: %s\n", yytext); }

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

[a-zA-Z0-9.\_%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,} { printf("Mail id: %s\n", yytext); }

"TCS" { printf("Placed at: TCS\n"); tcs++; }

"Infosys" { printf("Placed at: Infosys\n"); infosys++; }

"Wipro" { printf("Placed at: Wipro\n"); wipro++; }

"Accenture" { printf("Placed at: Accenture\n"); accenture++; }

"Informatica" { printf("Placed at: Informatica\n"); informatica++; }

"Female" { printf("Gender: Female\n"); females++; }

"Male" { printf("Gender: Male\n"); males++; }

"CSE" { printf("Branch: CSE\n"); cse++; }

"IT" { printf("Branch: IT\n"); it++; }

"EC" { printf("Branch: EC\n"); ec++; }

[a-zA-Z]+ { printf("Name of student: %s\n", yytext); }

. ;

%%

int main(void)

{

yyin = fopen("e3text.txt", "r");

yylex();

fclose(yyin);

printf("\nNumber of students placed in each company:\n");

printf("TCS: %d, \nInfosys: %d ,\n Wipro: %d,\n Accenture: %d,\n Informatica: %d\n",

tcs, infosys, wipro, accenture, informatica);

printf("Number of male students: %d\n", males);

printf("Number of female students: %d\n", females);

printf("Number of CSE students: %d\n", cse);

printf("Number of IT students: %d\n", it);

printf("Number of EC students: %d\n", ec);

return 0;

}

int yywrap() { return 1; }

CGPA\_Company.txt:

Abc Infosys Female 9.4 CSE 612333 abc@rknec.edu 9000000000

Def Infosys Male 8.5 IT 558923 def@rknec.edu 8827291892

Ghi Wipro Female 9.0 EC 700000 ghi@rknec.edu 7152083912

Jkl Accenture Male 8.2 CSE 500000 jkl@rknec.edu 618957289

Mno Informatica Female 9.8 IT 827196 mno@rknec.edu 5972071564

Pqr TCS Male 8.9 EC 650000 pqr@rknec.edu 9289501748

Stu Infosys Female 9.5 CSE 75888 stu@rknec.edu 8291057289

\*\*No\_of\_lines / N0\_of\_questions:

%{

#include <stdio.h>

int wordCount = 0,

lineCount = 0,

smallCount = 0,

capitalCount = 0,

digitCount = 0,

specialCount = 0,

question\_count = 0;

%}

%%

(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/\d{4} { printf("Date of examination: %s\n", yytext); }

"Sem:"[[:space:]]\*[IiI|IiIiI|IiIiIiIiI|IiIiIiIiIiIi]+ { printf("Semester: %s\n", yytext); }

Question[0-9]+ { wordCount += yyleng; question\_count++; }

[a-z] { smallCount++; }

[A-Z] { capitalCount++; }

[0-9] { digitCount++; }

[^a-zA-Z0-9\n] { specialCount++; }

\n { lineCount++; }

%%

int main() {

FILE \*file = fopen("e2text.txt", "r");

yyin = file;

yylex();

fclose(file);

printf("Number of questions: %d\n", question\_count);

printf("Number of words: %d\n", wordCount);

printf("Number of lines: %d\n", lineCount);

printf("Number of small letters: %d\n", smallCount);

printf("Number of capital letters: %d\n", capitalCount);

printf("Number of digits: %d\n", digitCount);

printf("Number of special characters: %d\n", specialCount);

return 0;

}

int yywrap() {

return 1;

}

Number\_of\_lines.txt:

ABC College

21/2/2000 Sem: II

Question1 : What are the benefits of tree plantation?

Question2 : What is water pollution?

Question3 : What should be done to avoid road accidents?

Question4 : What are your view on noise pollution?

Question5 : Why should people adopt pets?

\*\*Remove\_Comments:

%{

%}

start \/\\*

end \\*\/

%%

\/\/[^\n]\* { printf("Single-line comment: %s\n", yytext); }

{start}[^]{end} { printf("Multiline comment: %s\n", yytext); }

.|\n { fprintf(yyout, "%s", yytext); }

%%

int main()

{

yyin = fopen("ep4read.txt", "r");

yyout = fopen("ep4write.txt", "w");

yylex();

fclose(yyin);

fclose(yyout);

return 0;

}

int yywrap() {

return 1;

}

Input.txt:

//gsvsgsv

int main(){

//wgsbewr

int a = 6;

int b = 7;

print("%d", a);

print("%d", a+b);

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 2:

\*\*arithmetic expression to postfix:

Lex:

%{

#include "y.tab.h"

%}

%%

[ \t]+

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

[a-zA-Z]+ { yylval.str = strdup(yytext); return ID; }

\n { return NL; }

. { return yytext[0]; }

%%

Bison:

%{

#include<stdio.h>

#include<stdlib.h>

int answer=0;

%}

%token NUMBER ID NL

%left '+' '-'

%left '\*' '/'

%right NEGATIVE

%%

stmt : exp NL {

printf("Valid expression & Answer: %d \n",$1);

exit(0);

}

|

exp1 NL {

printf("Valid Expression \nBut, Calculation Can Be Performed On Numbers\n");

exit(0);

}

;

exp : exp '+' exp {$$=$1+$3; printf("+");}

| exp '-' exp {$$=$1-$3; printf("-");}

| exp '' exp {$$=$1$3; printf("\*");}

| exp '/' exp {$$=$1/$3; printf("/");}

| '(' exp ')' {$$=$2;}

| NUMBER {$$=$1; printf("%d", $1);}

;

exp1 : exp1 '+' exp1 {printf("+");}

| exp1 '-' exp1 {printf("-");}

| exp1 '' exp1 {printf("");}

| exp1 '/' exp1 {printf("/");}

| '(' exp1 ')'

| ID { printf("%s",$1);}

;

%%

int yyerror(char \*msg)

{

printf("Invalid Expression \n");

exit(0);

}

int main()

{

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

yyparse();

}

Lex file :

%{

#include "y.tab.h"

%}

%%

[ \t]+

[0-9]+ { yylval.num = atoi(yytext); return NUMBER; }

[a-zA-Z]+ { yylval.str = strdup(yytext); return ID; }

\n { return NL; }

. { return yytext[0]; }

%%

\*\*Expression matching strings:

Lex file:

%{

#include "y.tab.h"

%}

%%

"a" { return A; }

"b" { return B; }

"c" { return C; }

. { return ERROR; }

\n { return 0; } /\* Return 0 for newline to terminate input \*/

%%

int yywrap() {

return 1;

}

Bison file:

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token A B C

%token ERROR

%%

start: s C { printf("String accepted\n"); }

| ERROR { printf("String rejected\n"); }

;

s: A s B

| A B

;

%%

int yyerror(char \*msg) {

printf("Syntax Error\n");

return 0;

}

int main() {

printf("Enter the string: ");

yyparse();

return 0;

}

\*\*[if else validation](https://github.com/abubakr1934/Compiler-Design-Practical/tree/main/A1/practical%202/if%20else%20validation)

Lex:

%{

#include <stdio.h>

#include "y.tab.h"

%}

%%

[\t\n] /\* ignore whitespace and newlines \*/

if { return IF; }

else { return ELSE; }

[a-zA-Z]+ { return IDs; }

[0-9]+ { return NUM; }

">=" { return GE; }

"==" { return EQ; }

"!=" { return NE; }

"||" { return OR; }

"&&" { return AND; }

. { return yytext[0]; }

%%

int yywrap() {

return 1;

}

Bison:

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token ID NUM IF ELSE LE GE EQ NE OR AND

%right "="

%left OR AND

%left '>' '<' LE GE EQ NE

%left '+' '-'

%left '\*' '/'

%right UMINUS

%left '!'

%%

S : ST {printf("Valid if statement\n"); exit(0);}

ST : IF '(' E ')' DEF ELSE DEF

;

DEF : '{' BODY '}'

| E';'

| ST

|

;

BODY : BODY BODY

| E ';'

| ST

|

;

E : ID '=' E

| E '+' E

| E '-' E

| E '\*' E

| E '/' E

| E '<' E

| E '>' E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

| E '+' '+'

| E '-' '-'

| ID

| NUM

;

E2 : E'<'E

| E'>'E

| E LE E

| E GE E

| E EQ E

| E NE E

| E OR E

| E AND E

;

%%

main() {

printf("Enter the if else statement to check :");

yyparse();

}

int yywrap(void)

{

return 1;

}

int yyerror(char \*mes) {

printf("Invalid if else statement\n");

return 0;

}

\*\*For loop:

Lex:

%{

#include <stdio.h>

#include "y.tab.h"

%}

alpha [A-Za-z]

digit [0-9]

%%

[\t \n]+

for return FOR;

{alpha}({alpha}|{digit})\* return ID;

{digit}+ return NUM;

";" return SEMI;

"=" return EQUALS;

"+" return PLUS;

"-" return MINUS;

"++" return INCREMENT;

"--" return DECREMENT;

"\*" return TIMES;

"/" return DIVIDE;

"(" return LPAREN;

")" return RPAREN;

"{" return LBRACE;

"}" return RBRACE;

"<=" return LE;

">=" return GE;

"==" return EQ;

"!=" return NE;

"," return COMMA;

. return yytext[0];

%%

Bison:

%{

#include <stdio.h>

#include <stdlib.h>

%}

%token FOR ID NUM SEMI EQUALS PLUS MINUS INCREMENT DECREMENT

TIMES DIVIDE LPAREN RPAREN LBRACE RBRACE LE GE EQ NE COMMA

%right EQUALS

%left PLUS MINUS

%left TIMES DIVIDE

%left INCREMENT DECREMENT

%left LE GE EQ NE

%left LPAREN RPAREN

%nonassoc LBRACE RBRACE

%%

Program : Statement { printf("Valid for statement\n"); exit(0); }

;

Statement : ForLoop

;

ForLoop : FOR LPAREN ForInit SEMI ForCond SEMI ForIncr RPAREN LBRACE

LoopBody RBRACE

;

ForInit : ID EQUALS exp

|

;

ForCond : exp

|

;

ForIncr : exp

|

;

LoopBody : Statement

| LoopBody Statement

|

;

exp : exp PLUS exp

| exp MINUS exp

| exp TIMES exp

| exp DIVIDE exp

| exp LE exp

| exp GE exp

| exp EQ exp

| exp NE exp

| ID

| NUM

| ID INCREMENT

| ID DECREMENT

;

%%

int main() {

printf("Enter the for loop statement to check :\n");

yyparse();

return 0;

}

int yywrap(void) {;

return 1;

}

int yyerror(char \*s) {

printf("Invalid for statement\n");

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 3:

\*\*\* First ,Follow,parsing table :

first = {}

follow = {}

productions = {

'A': ['SB', 'B'],

'S': ['a', 'Bc','ε'],

'B': ['b', 'd'],

}

first = {}

def calculate\_first(symbol):

if symbol in first:

return first[symbol]

first\_set = set()

for production in productions[symbol]:

if production[0].islower():

first\_set.add(production[0])

elif production[0].isupper():

first\_set |= calculate\_first(production[0])

if 'ε' in first\_set:

first\_set |= calculate\_first(production[1:])

first\_set.discard('ε')

first[symbol] = first\_set

return first\_set

for symbol in productions.keys():

calculate\_first(symbol)

for symbol, first\_set in first.items():

print("FIRST({}): {}".format(symbol, first\_set))

follow['A'] = {'$'}

start\_symbol = 'A'

follow[start\_symbol] = {'$'}

from collections import defaultdict

# from tabulate import tabulate

class Grammar:

def \_\_init\_\_(self, productions):

self.productions = productions

self.start\_symbol = list(self.productions.keys())[0]

self.first = self.calculate\_first()

self.follow = self.calculate\_follow()

def calculate\_first(self):

first = defaultdict(set)

for nt, rhs in reversed(self.productions.items()):

for alt in rhs:

for s in alt:

if s.isupper():

if 'ε' not in first[s]:

first[nt].update(first[s])

if 'ε' in first[nt]:

first[nt].remove('ε')

break

first[nt].update(first[s])

elif s != 'ε':

first[nt].add(s)

if 'ε' in first[nt]:

first[nt].remove('ε')

break

else:

first[nt].add('ε')

return first

def calculate\_follow(self):

follow = defaultdict(set)

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

while True:

old\_follow = dict(follow)

for nt, rhs in self.productions.items():

for alt in rhs:

for i, s in enumerate(alt):

if s.isupper(): # Nonterminal

if i+1 < len(alt):

follow[s].update(self.first[alt[i+1]])

if 'ε' in self.first[alt[i+1]]:

follow[s].update(follow[nt])

follow[s].discard('ε')

else:

follow[s].update(follow[nt])

if old\_follow == follow:

break

return follow

def print\_parsing\_table(self):

terminals = set()

non\_terminals = set()

for nt, rhs in self.productions.items():

non\_terminals.add(nt)

for alt in rhs:

for s in alt:

if not s.isupper() and s != 'ε':

terminals.add(s)

terminals = sorted(list(terminals))

non\_terminals = sorted(list(non\_terminals))

print(f"Non terminals: {non\_terminals}")

print(f"Terminals: {terminals}")

matrix = []

for nt in non\_terminals:

row = []

for t in terminals:

if t in self.first[nt]:

for prod in self.productions[nt]:

if prod[0] == t:

row.append(f"{nt} -> {prod}")

break

else:

row.append(' ')

else:

row.append(' ')

matrix.append(row)

# table = tabulate(matrix, headers=terminals, showindex=non\_terminals, tablefmt='grid')

# print(table)

def print\_sets(self):

print("FIRST sets:")

for nt, first\_set in self.first.items():

print(f"{nt}: {sorted(first\_set)}")

print("\nFOLLOW sets:")

for nt, follow\_set in self.follow.items():

print(f"{nt}: {sorted(follow\_set)}")

def main():

productions = {

'A': ['SB', 'B'],

'S': ['a', 'Bc','ε'],

'B': ['b', 'd'],

}

grammar = Grammar(productions)

grammar.print\_sets()

grammar.print\_parsing\_table()

if \_\_name\_\_ == "\_\_main\_\_":

main()

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 4

\*\*\*First Follow ,pasing table ,LL1 table, string validation:

def first(string):

first\_ = set()

if string in non\_terminals:

alternatives = productions\_dict[string]

for alternative in alternatives:

first\_2 = first(alternative)

first\_ = first\_ | first\_2

elif string in terminals:

first\_ = {string}

elif string == '' or string == '#':

first\_ = {'#'}

else:

first\_2 = first(string[0])

if '#' in first\_2:

i = 1

while '#' in first\_2:

first\_ = first\_ | (first\_2 - {'#'})

if string[i:] in terminals:

first\_ = first\_ | {string[i:]}

break

elif string[i:] == '':

first\_ = first\_ | {'#'}

break

first\_2 = first(string[i:])

first\_ = first\_ | first\_2 - {'#'}

i += 1

else:

first\_ = first\_ | first\_2

return first\_

def follow(nT):

follow\_ = set()

prods = productions\_dict.items()

if nT == starting\_symbol:

follow\_ = follow\_ | {'$'}

for nt, rhs in prods:

for alt in rhs:

for char in alt:

if char == nT:

following\_str = alt[alt.index(char) + 1:]

if following\_str == '':

if nt == nT:

continue

else:

follow\_ = follow\_ | follow(nt)

else:

follow\_2 = first(following\_str)

if '#' in follow\_2:

follow\_ = follow\_ | follow\_2 - {'#'}

follow\_ = follow\_ | follow(nt)

else:

follow\_ = follow\_ | follow\_2

return follow\_

terminals = ['$', 'a', 'b', 'p', 'c']

non\_terminals = ['S', 'A', 'B', 'C']

starting\_symbol = 'S'

productions = ['S->A|BC', 'A->a|b', 'B->p|#', 'C->c']

productions\_dict = {}

for nT in non\_terminals:

productions\_dict[nT] = []

for production in productions:

nonterm\_to\_prod = production.split("->")

alternatives = nonterm\_to\_prod[1].split("|")

for alternative in alternatives:

productions\_dict[nonterm\_to\_prod[0]].append(alternative)

FIRST = {}

FOLLOW = {}

for non\_terminal in non\_terminals:

FIRST[non\_terminal] = set()

for non\_terminal in non\_terminals:

FOLLOW[non\_terminal] = set()

for non\_terminal in non\_terminals:

FIRST[non\_terminal] = FIRST[non\_terminal] | first(non\_terminal)

FOLLOW[starting\_symbol] = FOLLOW[starting\_symbol] | {'$'}

for non\_terminal in non\_terminals:

FOLLOW[non\_terminal] = FOLLOW[non\_terminal] | follow(non\_terminal)

print("{: ^20}{: ^20}{: ^20}".format('Non Terminals', 'First', 'Follow'))

for non\_terminal in non\_terminals:

print("{: ^20}{: ^20}{: ^20}".format(non\_terminal, str(FIRST[non\_terminal]), str(FOLLOW[non\_terminal])))

def generate\_LL1\_table(productions):

ll1\_table = {}

for non\_terminal in non\_terminals:

ll1\_table[non\_terminal] = {}

for terminal in terminals:

ll1\_table[non\_terminal][terminal] = []

for production in productions:

non\_terminal, production\_body = production.split("->")

production\_body = production\_body.split("|")

first\_set = first(production\_body[0])

for terminal in first\_set:

ll1\_table[non\_terminal][terminal].append(production\_body[0])

if '#' in first\_set:

follow\_set = follow(non\_terminal)

for terminal in follow\_set:

ll1\_table[non\_terminal][terminal].append(production\_body[0])

return ll1\_table

ll1\_table = generate\_LL1\_table(productions)

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

for non\_terminal in non\_terminals:

print(non\_terminal, ll1\_table[non\_terminal])

def validate\_string(input\_string):

stack = ["$", starting\_symbol]

buffer = list(input\_string)

buffer.append("$")

print("Stack\t\tBuffer\t\tAction")

print("-----\t\t------\t\t------")

while True:

print(f"{' '.join(stack)}\t\t{' '.join(buffer)}\t\t", end="")

top = stack[-1]

front = buffer[0]

if top == front == "$":

print("String accepted")

break

elif top in terminals:

if top == front:

stack.pop()

buffer.pop(0)

print("Match")

else:

print("String rejected")

break

elif top in non\_terminals:

if front in FIRST[top]:

production = productions\_dict[top][0]

stack.pop()

if production != "#":

stack.extend(list(reversed(production)))

print(f"Expand using {top} -> {production}")

elif "#" in FIRST[top] and buffer[0] in FOLLOW[top]:

stack.pop()

stack.append("#")

print(f"Expand using {top} -> #")

else:

print("String rejected")

break

else:

print("Error: Invalid entry in the stack")

break

validate\_string("a$")

print()

validate\_string("apc$")

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 5

\*\*\*If-else-sdts:

def generate\_3\_address\_code(input\_construct):

lines = input\_construct.split('\n')

code = []

address = 1

temp\_count = 1

for line in lines:

line = line.strip()

if line.startswith("if"):

conditions = line.split('(')[1].split('&&')

condition1 = conditions[0].strip()

condition2 = conditions[1].strip().split(')')[0].strip()

code.append(f"{address}) if {condition1} goto {address + 2}")

code.append(f"{address + 1}) goto {address + 9}")

code.append(f"{address + 2}) if {condition2} goto {address + 4}")

code.append(f"{address + 3}) goto {address + 9}")

address += 4

elif line.startswith("else"):

code.append(f"{address}) goto {address + 8}")

address += 1

elif line.startswith("{") or line.startswith("}"):

continue

else:

operands = line.split('=')

result = operands[0].strip()

expression = operands[1].strip().replace(';', '')

temp\_var = f"T{temp\_count}"

code.append(f"{address}) {temp\_var} = {expression}")

code.append(f"{address + 1}) {result} = {temp\_var}")

temp\_count += 1

address += 2

code.append(f"{address}) END")

return code

# Input your if-else construct here

input\_construct = """if (a<5 && b>c)

{

c= b+d;

d= i+j;

}

else

{

d= a+ b;

k= x+y;

}"""

three\_address\_code = generate\_3\_address\_code(input\_construct)

for instruction in three\_address\_code:

print(instruction)

\*\*for loop sdts:

# Write a program to generate three address code for the given language construct using SDTS (only for for loop).

# read for loop from the for\_loop.txt file

def read\_for\_loop():

with open("for\_loop.txt", "r") as file:

return file.read()

class ParseTreeNode:

def \_\_init\_\_(self, data,attr=None):

self.data = data

self.children = []

self.parent = None

self.attributes = attr

def add\_child(self, child):

self.children.append(child)

def \_\_str\_\_(self, level=0):

ret = "\t"\*level+repr(self.data)+"\n"

for child in self.children:

ret += child.\_\_str\_\_(level+1)

return ret

def get\_tokens(for\_loop):

index\_of\_start\_of\_initialization = for\_loop.index("(")

index\_of\_end\_of\_initialization = for\_loop.index(")")

initialization = for\_loop[index\_of\_start\_of\_initialization+1:index\_of\_end\_of\_initialization]

intialization,condition,increment = initialization.split(";")

# simplify the increment

if increment[-2:] == "++":

increment = increment[:-2] + f"={increment[0]}+1"

elif increment[-2:] == "--":

increment = increment[:-2] + f"={increment[0]}-1"

statement = for\_loop[for\_loop.index("{")+1:for\_loop.index("}")]

statements = statement.split(";")

statements = list(map(lambda statement:statement.strip(),statements))

statements = list(filter(lambda statement:statement!="",statements))

return intialization,condition,increment,statements

def generate\_parse\_tree(for\_loop):

initialization,condition,increment,statements = get\_tokens(for\_loop)

root = ParseTreeNode("For Loop","root")

# add intialization

ini = ParseTreeNode(initialization, "intialization")

expr,assignment = ini.data.split("=")

ini.add\_child(ParseTreeNode(expr))

ini.add\_child(ParseTreeNode("="))

ini.add\_child(ParseTreeNode(assignment))

root.add\_child(ini)

root.add\_child(ParseTreeNode('M','incr\_addr'))

# add condition

con = ParseTreeNode(condition,"condition")

opr = None

for i in con.data:

if i in ['<','>','=']:

opr = i

break

expr1,expr2 = con.data.split(opr)

con.add\_child(ParseTreeNode(expr1,"id"))

con.add\_child(ParseTreeNode(opr,"relop"))

con.add\_child(ParseTreeNode(expr2,"id"))

root.add\_child(con)

root.add\_child(ParseTreeNode('M','incr\_addr'))

# add increment

inc = ParseTreeNode(increment,"increment")

expr,assignment = inc.data.split("=")

opr=None

for i in assignment:

if i in ['+','-','\*','/']:

opr = i

break

assign = ParseTreeNode(assignment,"assignment")

nested\_expr1,nested\_expr2 = assignment.split(opr)

inc.add\_child(ParseTreeNode(expr,"id"))

inc.add\_child(ParseTreeNode("=", "assign"))

assign.add\_child(ParseTreeNode(nested\_expr1,"id"))

assign.add\_child(ParseTreeNode(opr,"op"))

assign.add\_child(ParseTreeNode(nested\_expr2,"id"))

inc.add\_child(assign)

root.add\_child(inc)

root.add\_child(ParseTreeNode('M','incr\_addr'))

for statement in statements:

statement\_node = ParseTreeNode(statement,"statement")

expr,assignment = statement\_node.data.split("=")

opr=None

for i in assignment:

if i in ['+','-','\*','/']:

opr = i

break

statement\_node.add\_child(ParseTreeNode(expr,"id"))

statement\_node.add\_child(ParseTreeNode("=", "assign"))

assign = ParseTreeNode(assignment,"assignment")

nested\_expr1,nested\_expr2 = assignment.split(opr)

assign.add\_child(ParseTreeNode(nested\_expr1,"id"))

assign.add\_child(ParseTreeNode(opr,"op"))

assign.add\_child(ParseTreeNode(nested\_expr2,"id"))

statement\_node.add\_child(assign)

root.add\_child(statement\_node)

return root

# inorder traversal of parse tree, and generate three address code

address = {"A":1}

def generate\_three\_address\_code(node: ParseTreeNode, statements, three\_address\_code):

if len(node.children) == 0:

return three\_address\_code

for child in node.children:

three\_address\_code=generate\_three\_address\_code(child, statements,three\_address\_code)

if node.attributes == "intialization":

three\_address\_code.append(f"{address['A']}. {node.children[0].data} = {node.children[2].data}")

address['A'] += 1

elif node.attributes == "condition":

three\_address\_code.append(f"{address['A']}. if {node.children[0].data} {node.children[1].data} {node.children[2].data} goto {address['A'] + 4}")

address['A'] += 1

three\_address\_code.append(f"{address['A']}. goto {address['A'] + len(statements) + 4}")

address['A'] += 1

elif node.attributes == "increment":

three\_address\_code.append(f"{address['A']}. {node.children[0].data} = {node.children[2].children[0].data} {node.children[2].children[1].data} {node.children[2].children[2].data}")

address['A'] += 1

three\_address\_code.append(f"{address['A']}. goto {2}")

address['A'] += 1

elif node.attributes == "statement":

three\_address\_code.append(f"{address['A']}. {node.children[0].data} = {node.children[2].children[0].data} {node.children[2].children[1].data} {node.children[2].children[2].data}")

address['A'] += 1

elif node.attributes == "root":

three\_address\_code.append(f"{address['A']}. goto {4}")

address['A'] += 1

return three\_address\_code

if \_\_name\_\_ == "\_\_main\_\_":

for\_loop = read\_for\_loop()

print("For Loop: ")

print(for\_loop)

parse\_tree = generate\_parse\_tree(for\_loop)

print("Parse Tree: ")

print(parse\_tree)

\_,\_,\_,statements = get\_tokens(for\_loop)

three\_address\_code=generate\_three\_address\_code(parse\_tree,statements=statements,three\_address\_code=[])

print("Three Address Code: ")

for code in three\_address\_code:

print(code)

\*\*\*while sdts.py

from typing import List

import pandas as pd

class Expression:

def \_init\_(self, operator=None, left\_expr=None, right\_expr=None, value=None):

self.operator = operator

self.left\_expr = left\_expr

self.right\_expr = right\_expr

self.value = value

def \_str\_(self):

return (

f"{self.operator=} | {self.left\_expr=} | {self.right\_expr=} | {self.value=}"

)

def generate\_3addr\_code(self):

if self.operator is None:

return self.value

elif self.operator == "+":

return f"T{self.value}=T{self.left\_expr.generate\_3addr\_code()}+T{self.right\_expr.generate\_3addr\_code()}"

elif self.operator == "-":

return f"T{self.value}=T{self.left\_expr.generate\_3addr\_code()}-T{self.right\_expr.generate\_3addr\_code()}"

elif self.operator == "\*":

return f"T{self.value}=T{self.left\_expr.generate\_3addr\_code()}\*T{self.right\_expr.generate\_3addr\_code()}"

elif self.operator == "/":

return f"T{self.value}=T{self.left\_expr.generate\_3addr\_code()}/T{self.right\_expr.generate\_3addr\_code()}"

elif self.operator == "<":

return f"if (T{self.left\_expr.generate\_3addr\_code()}<T{self.right\_expr.generate\_3addr\_code()}) goto {self.value}"

def build(string, start, end):

if start >= end:

return Expression(value=int(string[start:end]))

operator\_indices = [i for i in range(start, end) if string[i] in "+-\*/<"]

if not operator\_indices: # no operator found

return Expression(value=(string[start:end]))

op\_index = min(operator\_indices) # choose the first operator

root = Expression(operator=string[op\_index])

root.left\_expr = build(string, start, op\_index)

root.right\_expr = build(string, op\_index + 1, end)

return root

# Example usage:

lines = []

with open("input.txt", "r") as f:

lines = f.readlines()

expression\_string = lines[0].strip()

start = expression\_string.index("(") + 1

end = expression\_string.index(")")

root = build(expression\_string, start, end)

# output = ""

# print(root.generate\_3addr\_code())

def inorder(root):

# global output

if root is None:

return

# output += "("

inorder(root.left\_expr)

print(root)

# output += root.value

inorder(root.right\_expr)

# output += ")"

OPERATORS = ["+", "-", "\*", "/", "<", ">", "="]

def go\_get\_tokens(s):

d = {}

for i in range(len(s)):

if s[i] in OPERATORS:

if s[i] not in d:

d[s[i]] = []

d[s[i]].append((s[:i], s[i + 1 :]))

if "=" in d:

return ["=", d["="]]

elif "/" in d:

return ["/", d["="]]

elif "\*" in d:

return ["", d[""]]

elif "+" in d:

return ["+", d["+"]]

elif "-" in d:

return ["-", d["-"]]

class Statement:

def \_init\_(self, L):

self.list\_of\_statements = L

class Assignment:

def \_init\_(self, e1, e2):

self.operator = "="

self.left\_expr = e1

self.right\_expr = e2

def \_str\_(self):

return f"{self.operator=} | {self.left\_expr=} | {self.right\_expr=}"

def parse\_statements(statements):

assigns = []

for i in statements:

tokens = go\_get\_tokens(i)

op, l = tokens

left = build(l[0][0], 0, len(l[0][0]))

right = build(l[0][1], 0, len(l[0][1]))

assignment = Assignment(left, right)

assigns.append(assignment)

return assigns

# inorder(root)

# print(output)

statemenst = []

lines = [\_.strip() for \_ in lines]

open = lines.index("{") + 1

end = lines.index("}")

# print(lines[open:end])

assignments = parse\_statements(lines[open:end])

# print(assignments)

# for i in assignments:

# print("\*\*\* Statement \*\*\*\*")

# print()

# print("Left")

# inorder(i.left\_expr)

# print(f"Operator: {i.operator}")

# print("Right")

# inorder(i.right\_expr)

# print()

#

statement\_object = Statement(assignments)

class WhileExpression:

def \_init\_(self, expressions, statement):

self.t1 = "while"

self.m1 = None

self.expression = expressions

self.statements = statement

while\_expression = WhileExpression(root, statement\_object)

print(while\_expression)

three\_address\_codes = []

def statement\_tac(statement):

variables = 1

for s in statement:

# print(s)

left\_val = s.left\_expr.value

right\_val = ""

right\_val += (

s.right\_expr.left\_expr.value

+ s.right\_expr.operator

+ s.right\_expr.right\_expr.value

)

right\_val = right\_val.strip(";")

# print(left\_val)

# print(right\_val)

three\_address\_codes.append(f"T{variables} = {right\_val}")

three\_address\_codes.append(f"{left\_val} = T{variables}")

variables += 1

START = 0

def expression\_tac(expression):

exp = ""

# print(expression.left\_expr)

# print(expression.right\_expr)

exp += (

expression.left\_expr.value + expression.operator + expression.right\_expr.value

)

three\_address\_codes.append(f"if ({exp}) then goto :- {START + 2}")

three\_address\_codes.append("goto :- ")

expression\_tac(while\_expression.expression)

statement\_tac(while\_expression.statements.list\_of\_statements)

three\_address\_codes.append(f"goto :- {START}")

END = len(three\_address\_codes)

three\_address\_codes[1] = f"goto :- {START + END}"

three\_address\_codes.append("END")

# print(three\_address\_codes)

# for s in three\_address\_codes:

# print(s)

d = {"TAC": three\_address\_codes}

df = pd.DataFrame(d)

s = "10\*5-4/5"

r2 = build(s, 0, len(s))

print(df)

inorder(r2)

\*\*\*do-while.py:

class ThreeAddressGenerator:

def \_\_init\_\_(self):

self.temp\_count = 1

self.code = []

def new\_temp(self):

temp = f"T{self.temp\_count}"

self.temp\_count += 1

return temp

def generate\_code(self, condition, true\_statements, false\_statements):

code = []

code.append("START")

for statement in true\_statements:

code.extend(self.parse\_statement(statement))

code.append(f"if {condition} goto {len(code) + 2}")

code.append(f"goto {len(code) + 3}")

for statement in false\_statements:

code.extend(self.parse\_statement(statement))

code.append("END")

return code

def parse\_statement(self, statement):

code = []

tokens = statement.split()

if tokens[0] == 'if':

code.append(f"if {''.join(tokens[1:])} goto {len(code) + 2}")

code.append("goto X")

elif tokens[0] == 'goto':

code.append(f"{tokens[0]} {tokens[1]}")

elif len(tokens) >= 3 and tokens[1] == '=':

temp = self.new\_temp()

code.append(f"{temp}={''.join(tokens[2:])}")

code.append(f"{tokens[0]}={temp}")

else:

code.append(statement)

return code

if \_\_name\_\_ == "\_\_main\_\_":

generator = ThreeAddressGenerator()

condition = "(x>10)"

true\_statements = [

"y = x \* 2",

"z = y - 5"

]

false\_statements = [

"z = x + 5",

"y = z \* 2"

]

code = generator.generate\_code(condition, true\_statements, false\_statements)

for line in code:

print(line)

-\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 6

\*\*\*local optimization.py

from typing import List

import re

l: List[int] = []

with open("input.txt", "r") as f:

k = f.readlines()

l.append(k)

# print(k)

l = [i.strip() for i in l[0]]

# print(l)

OPERATORS = ["+", "-", "/", "\*", "<", ">"]

def eliminate\_dead\_code(l):

lhs = []

rhs = []

combined = []

for i in l:

l, r = i.split("=")

lhs.append(l.strip())

rhs.append(r.strip())

combined.append([l.strip(), r.strip()])

return combined

combined = eliminate\_dead\_code(l)

print(combined)

print("Original")

for \_ in combined:

print(f"{\_[0]} = {\_[1]}")

print()

new\_combined = []

for i in range(len(combined)):

a = combined[i][0]

flag = 0

for j in range(len(combined)):

if i == j:

continue

k = combined[j][1]

for k\_ in k:

if k\_ == a:

flag = 1

break

if flag or i == len(combined) - 1:

new\_combined.append(combined[i])

# print(new\_combined)

print("After removing dead code")

for \_ in new\_combined:

print(f"{\_[0]} = {\_[1]}")

print()

props = []

def evaluate\_expression(expr):

output = ""

i = 0

while i < len(expr):

if expr[i] == " ":

i += 1

continue

if expr[i].isalpha():

s = ""

j = i

while j < len(expr) and expr[j].isalpha():

output += expr[j]

j += 1

i = j

elif expr[i] in "+-/\*":

output += " " + expr[i] + " "

i += 1

else:

s = ""

while i < len(expr) and not expr[i].isalpha():

s += expr[i]

i += 1

if s[-1] in "+-/\*":

las = s[-1]

s = s.strip("+/-\*")

ans = eval(s)

output += str(ans)

output += " " + las + " "

else:

ans = eval(s)

output += str(ans)

i += 1

# pattern = re.compile("[a-zA-Z]")

# result = re.sub(pattern, "0", expr)

# print(result)

# print(eval(result))

return output

# print(f"Ans:- {evaluate\_expression("x+2+3")}")

def constant\_propogation(new\_combined):

for i in range(len(new\_combined)):

l, r = new\_combined[i][0], new\_combined[i][1]

flag = 0

flag = 0

for j in range(len(new\_combined)):

string = ""

if i == j:

continue

for k in range(len(new\_combined[j][1])):

if new\_combined[j][1][k] == l:

flag = 1

string += r

else:

string += new\_combined[j][1][k]

if flag:

new\_combined[j][1] = string

props.append(new\_combined[j])

else:

props.append(new\_combined[j])

kk = {}

ops = []

ops1 = []

for c in new\_combined:

if c[1] not in kk:

kk[c[1]] = c[0]

ops1.append(c)

continue

else:

if kk[c[1]] not in ops:

# print("Hell:- " + kk[c[1]], c[0])

ops1.append([c[0], kk[c[1]]])

c[0] = kk[c[1]]

else:

ops1.append(c)

ops.append(c[0])

print("Common subexpression elimination")

for i in ops1:

print(f"{i[0]} = {i[1]}")

print()

constant\_propogation(ops1)

props = [tuple(i) for i in props]

d = {k: 1 for k in props}

# print(props)

print("\nPropogation")

props = [i for i in d.keys()]

# print(props)

lhss = l[-1].split("=")[0].strip()

# print(lhss)

left\_expr = None

for i in props:

if i[0] == lhss:

left\_expr = i

break

# print(left\_expr)

final\_exp = left\_expr[1]

final\_eval = evaluate\_expression(final\_exp)

print(f"{left\_expr[0]} = {final\_eval}")

Input.txt:

a = 1

b = 2

c = b + 3

d = x

f = b + 3

e = x + c + f

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Practical 7

\*\*\*gen kill in out

import pandas as pd

blocks = []

with open("input.txt", "r") as f:

k = f.readlines()

k.append("\n")

# print(k)

temp = []

for i in k:

if i == "\n":

blocks.append(temp)

temp = []

else:

temp.append(i)

for t in blocks:

for y in range(len(t)):

t[y] = t[y].strip("\n")

class Expression:

def \_\_init\_\_(self, number, expression):

self.number = number

self.expression = expression

def \_\_str\_\_(self):

return f"{self.number}) {self.expression}"

class Blocks:

def \_\_init\_\_(self, block\_number, list\_of\_Expressions):

self.lof\_exprs = list\_of\_Expressions

self.block\_number = block\_number

def \_\_str\_\_(self):

output = f"Printing block number: {self.block\_number}\n"

for i in self.lof\_exprs:

k = i

output += str(k) + "\n"

output += "\n"

return output

expressions = []

blocks\_class = []

number = 1

for i in range(len(blocks)):

exprs = []

for j in range(len(blocks[i])):

exprs.append(Expression(number, blocks[i][j]))

number += 1

expressions.append(exprs)

blocks\_class.append(Blocks(i + 1, exprs))

DAG = {k: [] for k in blocks\_class}

DAG[blocks\_class[0]].extend([blocks\_class[1]])

DAG[blocks\_class[1]].extend([blocks\_class[2], blocks\_class[3]])

DAG[blocks\_class[2]].extend([blocks\_class[3], blocks\_class[4]])

DAG[blocks\_class[3]].extend([blocks\_class[1], blocks\_class[5]])

DAG[blocks\_class[4]].extend([blocks\_class[2]])

DAG[blocks\_class[5]] = []

OUTPUT = {}

OUTPUT["Blocks"] = [f"B{i.block\_number}" for i in blocks\_class]

OUTPUT["GEN"] = [0] \* len(blocks\_class)

OUTPUT["KILL"] = [0] \* len(blocks\_class)

OUTPUT["Predecessor"] = [0] \* len(blocks\_class)

OUTPUT["IN"] = [set() for \_ in range(len(blocks\_class))]

OUTPUT["OUT"] = [0] \* len(blocks\_class)

# print(OUTPUT)

gen\_array = []

for i in range(len(blocks\_class)):

temp\_arr = []

for j in blocks\_class[i].lof\_exprs:

temp\_arr.append(j.number)

gen\_array.append(temp\_arr)

# print(gen\_array)

OUTPUT["GEN"] = [set(i) for i in gen\_array]

kill\_arr = []

for i in range(len(blocks\_class)):

l\_expr = blocks\_class[i].lof\_exprs

lhs = []

temp\_kill = []

for j in l\_expr:

e = j.expression.split("=")

lhs.append(e[0].strip())

for j in range(len(blocks\_class)):

if i == j:

continue

lexpr = blocks\_class[j].lof\_exprs

for k in lexpr:

t = k.expression.split("=")

if t[0].strip() in lhs:

temp\_kill.append(k.number)

kill\_arr.append(temp\_kill)

# print(kill\_arr)

OUTPUT["KILL"] = [set(i) for i in kill\_arr]

precedence\_arr = []

for i in range(len(blocks\_class)):

curr = blocks\_class[i]

temp\_prec = []

for j in range(len(blocks\_class)):

if i == j:

continue

for it in DAG[blocks\_class[j]]:

if it.block\_number == curr.block\_number:

temp\_prec.append(blocks\_class[j].block\_number)

precedence\_arr.append(temp\_prec)

OUTPUT["Predecessor"] = precedence\_arr

OUTPUT["OUT"] = [set(i) for i in OUTPUT["GEN"]]

df = pd.DataFrame(OUTPUT)

print(f"Iteration: {0}")

print(df)

print()

iteration = 1

change = True

while change:

change = False

# prev\_state = OUTPUT

for i in range(len(blocks\_class)):

precs = OUTPUT["Predecessor"][i]

temp\_set = set()

for p in precs:

temp\_set = temp\_set.union(OUTPUT["OUT"][p - 1])

# OUTPUT["IN"][i] = OUTPUT["IN"][i].union(OUTPUT["OUT"][p - 1])

# print(OUTPUT["OUT"][p - 1])

OUTPUT["IN"][i] = temp\_set

# print(f"Block: {i+1} and Set: {temp\_set}")

oldotp = OUTPUT["OUT"][i]

OUTPUT["OUT"][i] = (OUTPUT["IN"][i] - OUTPUT["KILL"][i]).union(OUTPUT["GEN"][i])

if oldotp != OUTPUT["OUT"][i]:

change = True

if change:

print(f"Iteration: {iteration}")

dff = pd.DataFrame(OUTPUT)

print(dff)

print()

iteration += 1

if not change:

break

# if OUTPUT == prev\_state:

# break

# print(OUTPUT["IN"])

print(f"Iteration: {iteration}")

dff = pd.DataFrame(OUTPUT)

print(dff)

Input.txt:

b = 1

c = 2

a = b + c

d = a - b

d = c + d

c = b + c

e = a - b

d = b + c

e = e + 1

b = c \* d

c = b - d