LAB 7 – 8

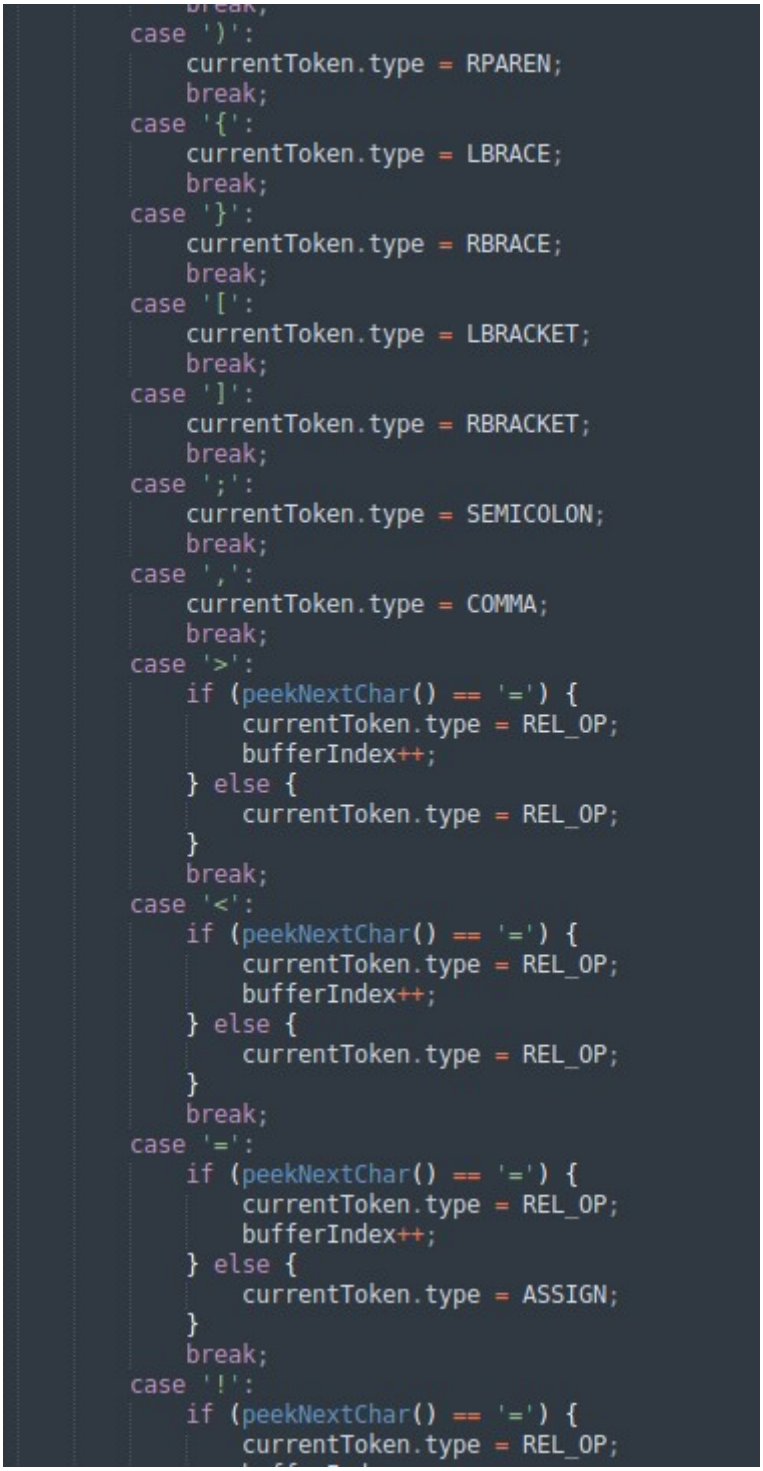
NAME: AYANIKA PAUL
ROLL NO. 22
REG NO. 220905128
D1

lab7) For given subset of grammar 7.1, design RD parser with appropriate error messages with expected
character and row and column number.
Lab8) Design the recursive descent parser to parse C program with variable declaration and decision statements
with error reporting of grammar 7.1.

CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef enum {
    MAIN, INT, CHAR, ID, NUM, LPAREN, RPAREN, LBRACE, RBRACE, LBRACKET, RBRACKET,SEMICOLON, COMMA, ASSIGN, IF, ELSE, REL_OP, ADD_OP, MUL_OP,END_OF_FILE,
    CHAR_LITERAL
} TokenType;

typedef struct Token {
    TokenType type;
    char* value;
    int row;
    int col;
} Token;

char* buffer;
int bufferIndex = 0;
int row = 1;
int col = 1;
void getNextToken(void);
void match(TokenType type);
void initTokenizer(char* buf);
Token currentToken;

char getNextChar() {
    if (bufferIndex >= strlen(buffer)) {
        return EOF;
    }
    char c = buffer[bufferIndex++];

    if (c == '\n') {
        row++;
        col = 1;
    } else {
        col++;
    }
    return c;
}
char peekNextChar() {
    if (bufferIndex >= strlen(buffer)) {
        return EOF;
    }
    return buffer[bufferIndex];
}
void getNextToken() {
    char c = getNextChar();
    while (isspace(c)) {
```

```
void getNextToken() {
    char c = getNextChar();
    while (isspace(c)) {
        c = getNextChar();
    }
    if (c == EOF) {
        currentToken.type = END_OF_FILE;
        return;
    }
    if (isdigit(c)) {
        currentToken.type = NUM;
        currentToken.value = malloc(10);
        currentToken.value[0] = c;
        int i = 1;
        while (isdigit(c = getNextChar())) {
            currentToken.value[i++] = c;
        }
        bufferIndex--;
        currentToken.value[i] = '\0';
        return;
    }
    if (isalpha(c)) {
        currentToken.type = ID;
        currentToken.value = malloc(10);
        currentToken.value[0] = c;
        int i = 1;
        while (isalnum(c = getNextChar())) {
            currentToken.value[i++] = c;
        }
        bufferIndex--;
        currentToken.value[i] = '\0';
        if (strcmp(currentToken.value, "main") == 0) {
            currentToken.type = MAIN;
        } else if (strcmp(currentToken.value, "int") == 0) {
            currentToken.type = INT;
        } else if (strcmp(currentToken.value, "char") == 0) {
            currentToken.type = CHAR;
        } else if (strcmp(currentToken.value, "if") == 0) {
            currentToken.type = IF;
        } else if (strcmp(currentToken.value, "else") == 0) {
            currentToken.type = ELSE;
        }
        return;
    }
    switch (c) {
        case '(':
            currentToken.type = LPAREN;
            break;
```



```
            if (peekNextChar() == '=') {
                currentToken.type = REL_OP;
                bufferIndex++;
            } else {
                printf("Error at line %d, column %d: Invalid character '!'\n", row, col);
                exit(1);
            }
            break;
        case '\'':
            c = getNextChar();
            if (c == '\\') {
                c = getNextChar();
                if (c == 'n' || c == 't' || c == 'r' || c == '\\' || c == '\'' || c == '\"') {
                    char escapedChar[3] = {'\\', c, '\0'};
                    currentToken.value = strdup(escapedChar);
                } else {
                    printf("Error at line %d, column %d: Invalid escape sequence\n", row, col);
                    exit(1);
                }
            } else if (c != '\'' && c != EOF) {
                char charLiteral[2] = {c, '\0'};
                currentToken.value = strdup(charLiteral);
            } else {
                printf("Error at line %d, column %d: Invalid or incomplete character literal\n", row, col);
                exit(1);
            }
            if (getNextChar() != '\'') {
                printf("Error at line %d, column %d: Missing closing single quote\n", row, col);
                exit(1);
            }
            currentToken.type = NUM;
            return;
        default:
            printf("Invalid character: %c\n", c);
            exit(1);
    }
}
void match(TokenType type) {
    if (currentToken.type != type) {
        printf("Error at line %d, column %d: Expected %d, but found %d\n", row, col, type, currentToken.type);
        printf("Token value: %s\n", currentToken.value);
        exit(1);
    }
    getNextToken();
}
void program() {
    if (currentToken.type != MAIN) {
```

```
    if (currentToken.type != MAIN) {
        printf("Error at line %d, column %d: Expected 'main'\n", row, col);
        exit(1);
    }
    match(MAIN);
    match(LPAREN);
    match(RPAREN);
    match(LBRACE);
    declarations();
    statement_list();
    if (currentToken.type != RBRACE) {
        printf("Error at line %d, column %d: Expected '}'\n", row, col);
        exit(1);
    }
    match(RBRACE);
    printf("Parsing successful!\n");
}
void declarations() {
    while (currentToken.type == INT || currentToken.type == CHAR) {
        if (currentToken.type == INT) {
            match(INT);
        } else {
            match(CHAR);
        }
        identifier_list();
        if (currentToken.type != SEMICOLON) {
            printf("Error at line %d, column %d: Expected ';'\n", row, col);
            exit(1);
        }
        match(SEMICOLON);
    }
}
void identifier_list() {
    if (currentToken.type != ID) {
        printf("Error at line %d, column %d: Expected identifier\n", row, col);
        exit(1);
    }
    match(ID);
    if (currentToken.type == LBRACKET) {
        match(LBRACKET);
        if (currentToken.type == NUM) {
            match(NUM);
        } else {
            printf("Error at line %d, column %d: Expected array size\n", row, col);
            exit(1);
        }
        match(RBRACKET);
    }
    while (currentToken.type == COMMA) {
```

```
    while (currentToken.type == COMMA) {
        match(COMMA);
        if (currentToken.type != ID) {
            printf("Error at line %d, column %d: Expected identifier\n", row, col);
            exit(1);
        }
        match(ID);
        if (currentToken.type == LBRACKET) {
            match(LBRACKET);

            if (currentToken.type == NUM) {
                match(NUM);
            } else {
                printf("Error at line %d, column %d: Expected array size\n", row, col);
                exit(1);
            }
            match(RBRACKET);
        }
    }
}
void statement_list() {
    while (currentToken.type == ID || currentToken.type == IF) {
        statement();
    }
}
void statement() {
    if (currentToken.type == ID) {
        assign_stat();

        if (currentToken.type != SEMICOLON) {
            printf("Error at line %d, column %d: Expected ';'\n", row, col);
            exit(1);
        }
        match(SEMICOLON);
    } else if (currentToken.type == IF) {
        decision_stat();
    }
}
void assign_stat() {
    if (currentToken.type == ID) {
        match(ID);

        if (currentToken.type == LBRACKET) {
            match(LBRACKET);

            if (currentToken.type == NUM) {
                match(NUM);
            } else {
                printf("Error at line %d, column %d: Expected array index\n", row, col);
```

```
            if (currentToken.type == NUM) {
                match(NUM);
            } else {
                printf("Error at line %d, column %d: Expected array index\n", row, col);
                exit(1);
            }

            match(RBRACKET);
        }
        if (currentToken.type != ASSIGN) {
            printf("Error at line %d, column %d: Expected '='\n", row, col);
            exit(1);
        }
        match(ASSIGN);
        expn();
    } else {
        printf("Error at line %d, column %d: Expected identifier\n", row, col);
        exit(1);
    }
}
void expn() {
    simple_expn();

    if (currentToken.type == REL_OP) {
        eprime();
    }
}
void eprime() {
    match(REL_OP);
    simple_expn();
}
void simple_expn() {
    term();
    while (currentToken.type == ADD_OP) {
        seprime();
    }
}
void seprime() {
    match(ADD_OP);
    term();
}
void term() {
    factor();
    while (currentToken.type == MUL_OP) {
        tprime();
    }
}
void tprime() {
    match(MUL_OP);
```

```
void tprime() {
    match(MUL_OP);
    factor();
}
void factor() {
    if (currentToken.type == ID) {
        match(ID);

        if (currentToken.type == LBRACKET) {
            match(LBRACKET);

            expn();

            match(RBRACKET);
        }
    } else if (currentToken.type == NUM || currentToken.type == CHAR_LITERAL) {
        match(currentToken.type);
    } else {
        printf("Error at line %d, column %d: Expected identifier or number\n", row, col);
        exit(1);
    }
}
void decision_stat() {
    match(IF);
    match(LPAREN);
    expn();
    match(RPAREN);
    match(LBRACE);

    statement_list();

    if (currentToken.type != RBRACE) {
        printf("Error at line %d, column %d: Expected '}'\n", row, col);
        exit(1);
    }
    match(RBRACE);

    dprime();
}
void dprime() {
    if (currentToken.type == ELSE) {
        match(ELSE);
        match(LBRACE);

        statement_list();

        if (currentToken.type != RBRACE) {
```

```
        if (currentToken.type != RBRACE) {
            printf("Error at line %d, column %d: Expected '}'\n", row, col);
            exit(1);
        }
        match(RBRACE);
    }
}
void initTokenizer(char* buf) {
    buffer = buf;
    bufferIndex = 0;
    row = 1;
    col = 1;
    getNextToken();
}
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s filename\n", argv[0]);
        return 1;
    }
    FILE* file = fopen(argv[1], "r");
    if (!file) {
        printf("Could not open file %s\n", argv[1]);
        return 1;
    }
    fseek(file, 0, SEEK_END);
    long fileSize = ftell(file);
    rewind(file);
    char* inputBuffer = malloc(fileSize + 1);
    if (!inputBuffer) {
        printf("Memory allocation failed\n");
        fclose(file);
        return 1;
    }
    size_t bytesRead = fread(inputBuffer, 1, fileSize, file);
    inputBuffer[bytesRead] = '\0';
    fclose(file);
    initTokenizer(inputBuffer);
    program();
    free(inputBuffer);
    return 0;
}
```

INPUT:

test1.c

q1.c | test1.c

```
main() {
    int x, y[10];
    char z;

    x = 5;

    if (x > 10) {
        y[0] = 20;
    } else {
        z = 'a';
    }
}
```

OUTPUT:

```
student@oslab-02:~/220905128/lab8$ ./a.out test1.c
Parsing successful!
```

WRONG INPUT:

err1.c

err1.c

```
main() {
    int x[10];
    x[a] = 20;
}
```

```
student@oslab-02:~/220905128/lab8$ ./a.out err1.c
Error at line 3, column 10: Expected array index
```

LAB7 Inputs:

err1.c — lab7

```
main(){
    int a = 5, b ;
    a = a + 5;
    }
```

err2.c

```
main(){
```

OUTPUT:

```
student@oslab-02:~/220905128/lab7$ ./a.out err1.c
Error at line 2, column 11: Expected ';'
student@oslab-02:~/220905128/lab7$ ./a.out err2.c
Error at line 3, column 1: Expected '}'
```