Saivya Singh
220905730
CSE D
44

Lab 8 : RD Parser for Declaration Statements

Q1 . *Design the recursive descent parser to parse C program with variable declaration and decision statements*
with error reporting of grammar 7.1.

Code :

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

typedef enum {
    T_MAIN, T_INT, T_CHAR, T_IF, T_ELSE, T_ID, T_NUM, T_EQ, T_NEQ, T_LE,
T_GE,
    T_LT, T_GT, T_PLUS, T_MINUS, T_STAR, T_DIV, T_MOD, T_ASSIGN, T_SEMI,
    T_COMMA, T_LPAREN, T_RPAREN, T_LBRACE, T_RBRACE, T_EOF, T_UNKNOWN
} TokenType;

typedef struct {
    TokenType type;
    char lexeme[128];
    int line;
    int col;
} Token;

static Token currentToken;
static int g_line = 1;
static int g_col = 0;
static FILE *source;

void parseProgram();
void parseDeclarations();
void parseDataType();
void parseIdentifierList();
void parseStatementList();
void parseStatement();
void parseBlock();
void parseAssignStat();
void parseDecisionStat();
void parseDprime();
void parseExpn();
void parseEprime();
void parseRelop();
void parseSimpleExp();
void parseSeprime();
void parseAddop();
void parseTerm();
void parseTprime();
void parseMulop();
void parseFactor();

const char* tokenTypeName(TokenType t) {
    switch(t) {
        case T_MAIN: return "main";
        case T_INT: return "int";
        case T_CHAR: return "char";
        case T_IF: return "if";
        case T_ELSE: return "else";
        case T_ID: return "identifier";
        case T_NUM: return "number";
        case T_EQ: return "==";
        case T_NEQ: return "!=";
        case T_LE: return "<=";
        case T_GE: return ">=";
        case T_LT: return "<";
        case T_GT: return ">";
        case T_PLUS: return "+";
        case T_MINUS: return "-";
        case T_STAR: return "*";
        case T_DIV: return "/";
        case T_MOD: return "%";
        case T_ASSIGN: return "=";
        case T_SEMI: return ";";
        case T_COMMA: return ",";
        case T_LPAREN: return "(";
        case T_RPAREN: return ")";
        case T_LBRACE: return "{";
        case T_RBRACE: return "}";
        case T_EOF: return "EOF";
        default: return "UNKNOWN";
    }
}

void error(const char* expected) {
    fprintf(stderr, "Error at line %d, col %d: expected %s but found '%s' (token
type: %s)\n",
            currentToken.line, currentToken.col, expected, currentToken.lexeme,
tokenTypeName(currentToken.type));
    exit(EXIT_FAILURE);
}

int nextChar() {
    int c = fgetc(source);
    if(c == '\n') { g_line++; g_col = 0; } else { g_col++; }
    return c;
}
```

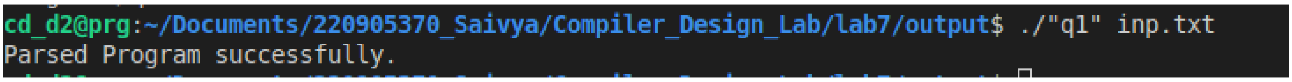
```
int peekChar() {
    int c = fgetc(source);
    ungetc(c, source);
    return c;
}

int skipWhitespace() {
    int c;
    do { c = nextChar(); } while(isspace(c));
    return c;
}

Token getNextToken() {
    Token token;
    int c;
    token.type = T_UNKNOWN;
    token.lexeme[0] = '\0';
    token.line = g_line;
    token.col = g_col;
    c = skipWhitespace();
    if(c == EOF) { token.type = T_EOF; strcpy(token.lexeme, "EOF"); return
token; }
    token.line = g_line;
    token.col = g_col;
    if(c == '(') { token.type = T_LPAREN; strcpy(token.lexeme, "("); return token;
}
    if(c == ')') { token.type = T_RPAREN; strcpy(token.lexeme, ")"); return token;
}
    if(c == '{') { token.type = T_LBRACE; strcpy(token.lexeme, "{"); return
token; }
    if(c == '}') { token.type = T_RBRACE; strcpy(token.lexeme, "}"); return
token; }
    if(c == ';') { token.type = T_SEMI; strcpy(token.lexeme, ";"); return token; }
    if(c == ',') { token.type = T_COMMA; strcpy(token.lexeme, ","); return
token; }
    if(c == '+') { token.type = T_PLUS; strcpy(token.lexeme, "+"); return
token; }
    if(c == '-') { token.type = T_MINUS; strcpy(token.lexeme, "-"); return
token; }
    if(c == '*') { token.type = T_STAR; strcpy(token.lexeme, "*"); return token; }
    if(c == '/') { token.type = T_DIV; strcpy(token.lexeme, "/"); return token; }
    if(c == '%') { token.type = T_MOD; strcpy(token.lexeme, "%"); return
token; }
    if(c == '=') {
        if(peekChar() == '=') { nextChar(); token.type = T_EQ;
strcpy(token.lexeme, "=="); }
        else { token.type = T_ASSIGN; strcpy(token.lexeme, "="); }
        return token;
    }
    if(c == '!') {
        if(peekChar() == '=') { nextChar(); token.type = T_NEQ;
strcpy(token.lexeme, "!="); return token; }
        else { token.type = T_UNKNOWN; token.lexeme[0] = '!'; token.lexeme[1]
= '\0'; return token; }
    }
    if(c == '<') {
        if(peekChar() == '=') { nextChar(); token.type = T_LE;
strcpy(token.lexeme, "<="); }
        else { token.type = T_LT; strcpy(token.lexeme, "<"); }
        return token;
    }
    if(c == '>') {
        if(peekChar() == '=') { nextChar(); token.type = T_GE;
strcpy(token.lexeme, ">="); }
        else { token.type = T_GT; strcpy(token.lexeme, ">"); }
        return token;
    }
    if(isalpha(c)) {
        char buffer[128];
        int i = 0;
        buffer[i++] = (char)c;
        while(isalnum(peekChar()) || peekChar() == '_') { c = nextChar();
buffer[i++] = (char)c; }
        buffer[i] = '\0';
        if(strcmp(buffer, "main") == 0) token.type = T_MAIN;
        else if(strcmp(buffer, "int") == 0) token.type = T_INT;
        else if(strcmp(buffer, "char") == 0) token.type = T_CHAR;
        else if(strcmp(buffer, "if") == 0) token.type = T_IF;
        else if(strcmp(buffer, "else") == 0) token.type = T_ELSE;
        else token.type = T_ID;
        strcpy(token.lexeme, buffer);
        return token;
    }
    if(isdigit(c)) {
        char buffer[128];
        int i = 0;
        buffer[i++] = (char)c;
        while(isdigit(peekChar())) { c = nextChar(); buffer[i++] = (char)c; }
        buffer[i] = '\0';
        token.type = T_NUM;
        strcpy(token.lexeme, buffer);
        return token;
    }
    token.type = T_UNKNOWN;
    token.lexeme[0] = (char)c;
    token.lexeme[1] = '\0';
    return token;
}

void advance() { currentToken = getNextToken(); }

void match(TokenType expected) {
    if(currentToken.type == expected) { advance(); }
    else { error(tokenTypeName(expected)); }
}

void parseProgram() {
    match(T_MAIN);
    match(T_LPAREN);
    match(T_RPAREN);
    match(T_LBRACE);
    parseDeclarations();
    parseStatementList();
    match(T_RBRACE);
    printf("Parsed Program successfully.\n");
}

void parseDeclarations() {
    if(currentToken.type == T_INT || currentToken.type == T_CHAR) {
        parseDataType();
        parseIdentifierList();
        match(T_SEMI);
        parseDeclarations();
    }
}

void parseDataType() {
    if(currentToken.type == T_INT) { match(T_INT); }
    else if(currentToken.type == T_CHAR) { match(T_CHAR); }
    else { error("int or char"); }
}

void parseIdentifierList() {
    if(currentToken.type == T_ID) {
        match(T_ID);
        if(currentToken.type == T_COMMA) { match(T_COMMA);
parseIdentifierList(); }
    } else { error("identifier"); }
}

void parseStatementList() {
    while(currentToken.type == T_ID || currentToken.type == T_IF ||
currentToken.type == T_LBRACE) {
        parseStatement();
    }
}

void parseStatement() {
    if(currentToken.type == T_ID) { parseAssignStat(); }
    else if(currentToken.type == T_IF) { parseDecisionStat(); }
    else if(currentToken.type == T_LBRACE) { parseBlock(); }
    else { error("statement (id, if, or block)"); }
}
```

```
void parseBlock() {
    match(T_LBRACE);
    parseStatementList();
    match(T_RBRACE);
}

void parseAssignStat() {
    match(T_ID);
    match(T_ASSIGN);
    parseExpn();
    match(T_SEMI);
}

void parseDecisionStat() {
    match(T_IF);
    match(T_LPAREN);
    parseExpn();
    match(T_RPAREN);
    parseStatement();
    parseDprime();
}

void parseDprime() {
    if(currentToken.type == T_ELSE) {
        match(T_ELSE);
        parseStatement();
    }
}

void parseExpn() {
    parseSimpleExp();
    parseEprime();
}

void parseEprime() {
    if(currentToken.type == T_EQ || currentToken.type == T_NEQ ||
       currentToken.type == T_LE || currentToken.type == T_GE ||
       currentToken.type == T_LT || currentToken.type == T_GT) {
        parseRelop();
        parseSimpleExp();
    }
}

void parseRelop() {
    if(currentToken.type == T_EQ || currentToken.type == T_NEQ ||
       currentToken.type == T_LE || currentToken.type == T_GE ||
       currentToken.type == T_LT || currentToken.type == T_GT) { advance(); }
    else { error("relational operator (==, !=, <=, >=, <, >)"); }
}

void parseSimpleExp() {
    parseTerm();
    parseSeprime();
}

void parseSeprime() {
    while(currentToken.type == T_PLUS || currentToken.type == T_MINUS) {
        parseAddop();
        parseTerm();
    }
}

void parseAddop() {
    if(currentToken.type == T_PLUS || currentToken.type == T_MINUS)
{ advance(); }
    else { error("add operator (+ or -)"); }
}

void parseTerm() {
    parseFactor();
    parseTprime();
}

void parseTprime() {
    while(currentToken.type == T_STAR || currentToken.type == T_DIV ||
currentToken.type == T_MOD) {
        parseMulop();
        parseFactor();
    }
}

void parseMulop() {
    if(currentToken.type == T_STAR || currentToken.type == T_DIV ||
currentToken.type == T_MOD) { advance(); }
    else { error("multiplicative operator (*, /, or %)"); }
}

void parseFactor() {
    if(currentToken.type == T_ID) { match(T_ID); }
    else if(currentToken.type == T_NUM) { match(T_NUM); }
    else { error("identifier or number"); }
}

int main(int argc, char* argv[]) {
    if(argc < 2) { printf("Usage: %s <source-file>\n", argv[0]); return 1; }
    source = fopen(argv[1], "r");
    if(!source) { perror("Error opening file"); return 1; }
    advance();
    parseProgram();
    if(currentToken.type != T_EOF) { error("EOF"); }
    fclose(source);
    return 0;
}
```

Input :

```
main() {

}
```

Output :

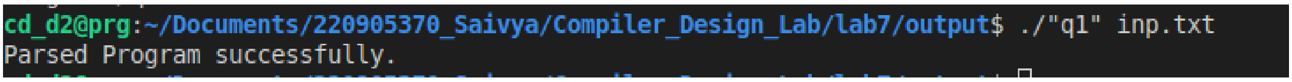


```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Parsed Program successfully.
```

Input :

```
main(){
    int a;
}
```

Output :

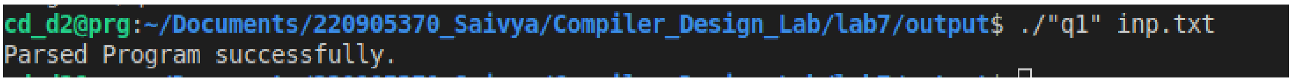


```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Parsed Program successfully.
```

Input :

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

Output :

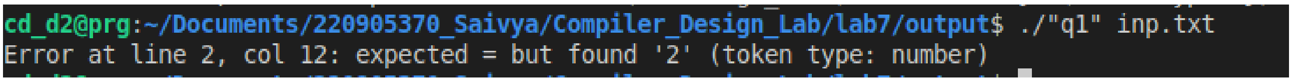


```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Parsed Program successfully.
```

Input :

```
main(){
    return 2;
}
```

Output :

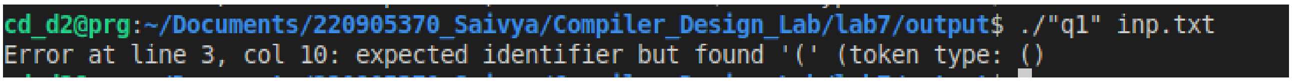


```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Error at line 2, col 12: expected = but found '2' (token type: number)
```

Input :

```
main(){
    int a,b;
    char (i);
    i = b+c;
    a=10;
    c=a;
}
```

Output :

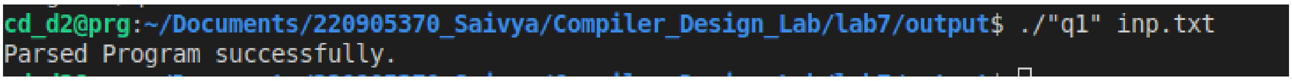


```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Error at line 3, col 10: expected identifier but found '(' (token type: ()
```

Input :

```
main() {
    int a, b;
    char c;
    a = 5;
    b = a - 3;
    if (a == b) {
        a = 10;
    }
    else {
        c = 2;
    }
}
```

Output :



```
cd_d2@prg:~/Documents/220905370_Saivya/Compiler_Design_Lab/lab7/output$ ./"q1" inp.txt
Parsed Program successfully.
```