

Umm Al-Qura University

Computer Science & Artificial Intelligent

College Computer Science Department



COMPILER CONSTRUCTION CS2341

PROJECT GROUP ID: 11

Appendix

Name	Part	ID
ليان صالح القرشي	Report and research	44511056
وجن العتيبي	Part 1 task II	445005620
ريتاج خالد القرشي	Part 2 task II	44512037
ريم الشهراني	Part 1 task III	445000375
سارة سلطان المطرفي	Report and research	445005027

Introduction

Description:

This project seeks to provide a practical understanding of the main components of compiler constructing by implementing a lexical analyzer (LEX) and a syntax analyzer (YACC). These tools are crucial in the early stages of compilation because they break down source code into understandable tokens and check their syntactic structure by using a specific grammar. In addition, a **symbol table** is used to store and track identifiers and keywords discovered during lexical analysis.

Structure of the Report:

1. **Lexical Analysis:** This part presents research findings on LEX, its syntax, and the implementation of the lexical analyzer, including code, input, and output screenshots.
2. **Parsing:** This section covers YACC research, including its structure and integration with LEX, as well as the syntax analyzer implementation and associated screenshots.
3. **Summary:** Reflections on project problems, and lessons learned throughout the completion of the project
4. **Citations / References** of all resources used to complete the project

Lexical Analysis

1) What is LEX?

Lex is a tool or program that creates a lexical analyzer and helps us perform the task of lexical analysis (It converts characters stream into tokens).

2) How does LEX work?

The working of lex in compiler design as a lexical analysis takes place in multiple steps. Firstly, we create a file that describes the generation of the lex analyzer. This file is written in Lex language and has a .l extension. The lex compiler converts this program into a C file called lex.yy.c. The C compiler then runs this C file, and it is compiled into file. This file is our working Lexical Analyzer which will produce the stream of tokens based on the input text.

3) What is the syntax of LEX code?

Declarations (The declarations include declarations of variables.)

%%

Translation rules (These rules consist of Pattern and Action.)

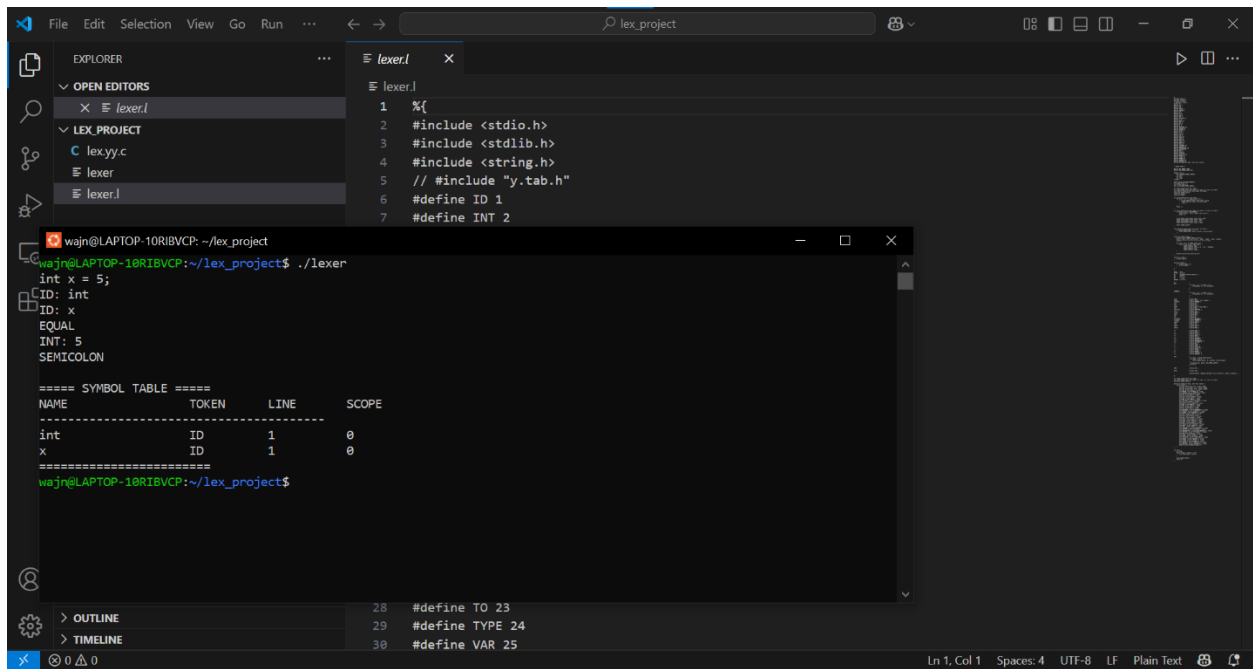
%%

Auxiliary procedures (The Auxiliary section holds auxiliary functions used in the actions.)

4) Can you use an IDE for LEX development?

Yes, it's possible. While LEX is typically used from the command line, it can be integrated with IDEs that support C programming, such as Code::Blocks, Eclipse CDT, or Visual Studio Code. These IDEs help with writing (.l) files, syntax highlighting, and compiling the generated C code. Some configuration may be needed to run LEX and compile the resulting files.

The output / input



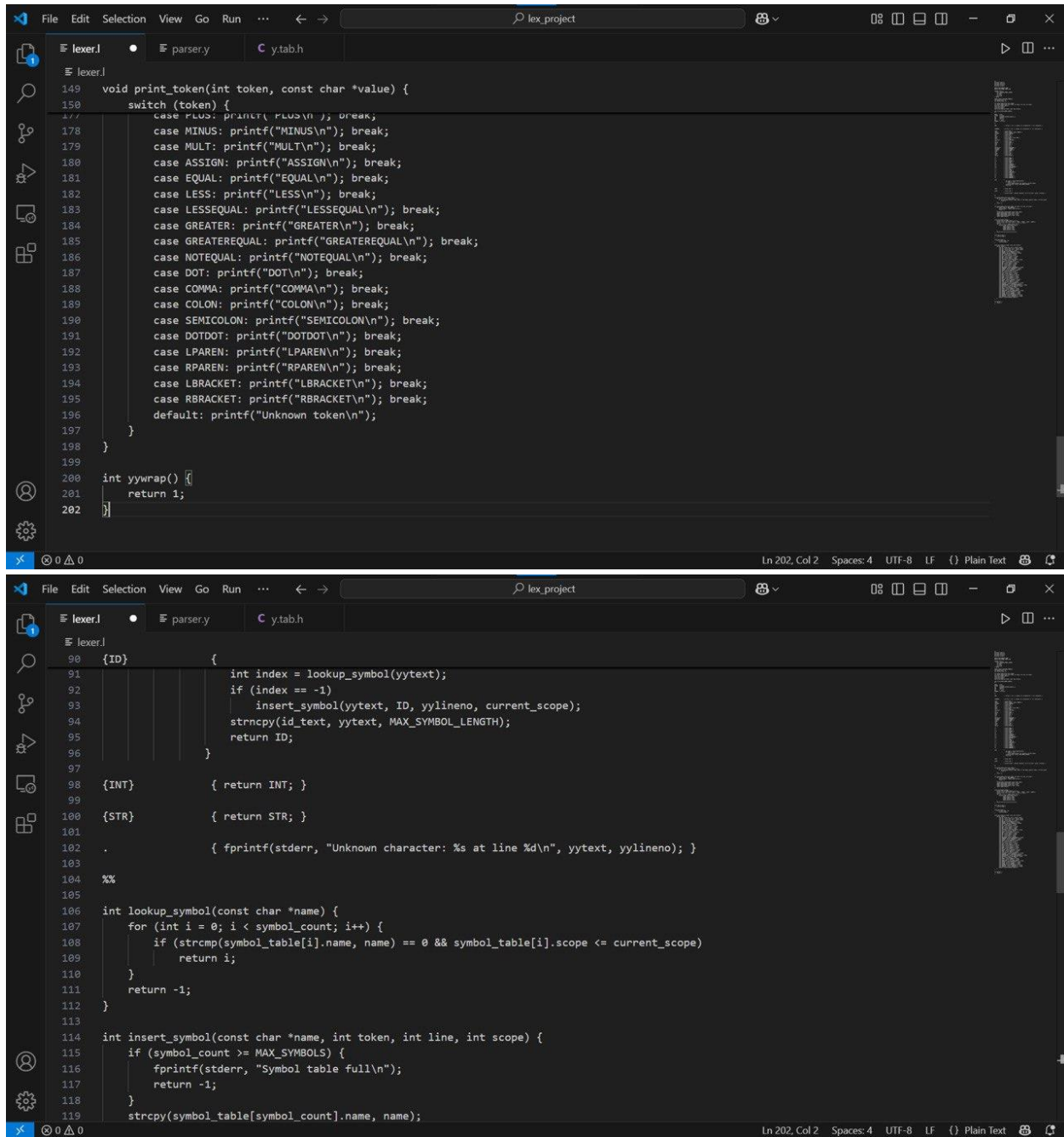
```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  // #include "y.tab.h"
6  #define ID 1
7  #define INT 2
```

```
wajn@LAPTOP-10RIBVCP: ~/lex_project
wajn@LAPTOP-10RIBVCP:~/lex_project$ ./lexer
int x = 5;
ID: int
ID: x
EQUAL
INT: 5
SEMICOLON

===== SYMBOL TABLE =====
NAME          TOKEN    LINE    SCOPE
-----
int           ID        1        0
x             ID        1        0
=====
wajn@LAPTOP-10RIBVCP:~/lex_project$
```

```
> OUTLINE 28 #define TO 23
> TIMELINE 29 #define TYPE 24
30 #define VAR 25
```

The code



```

149 void print_token(int token, const char *value) {
150     switch (token) {
151         case PLUS: printf("PLUS\n"); break;
152         case MINUS: printf("MINUS\n"); break;
153         case MULT: printf("MULT\n"); break;
154         case ASSIGN: printf("ASSIGN\n"); break;
155         case EQUAL: printf("EQUAL\n"); break;
156         case LESS: printf("LESS\n"); break;
157         case LESSEQUAL: printf("LESSEQUAL\n"); break;
158         case GREATER: printf("GREATER\n"); break;
159         case GREATEREQUAL: printf("GREATEREQUAL\n"); break;
160         case NOTEQUAL: printf("NOTEQUAL\n"); break;
161         case DOT: printf("DOT\n"); break;
162         case COMMA: printf("COMMA\n"); break;
163         case COLON: printf("COLON\n"); break;
164         case SEMICOLON: printf("SEMICOLON\n"); break;
165         case DOTDOT: printf("DOTDOT\n"); break;
166         case LPAREN: printf("LPAREN\n"); break;
167         case RPAREN: printf("RPAREN\n"); break;
168         case LBRACKET: printf("LBRACKET\n"); break;
169         case RBRACKET: printf("RBRACKET\n"); break;
170         default: printf("Unknown token\n");
171     }
172 }
173
174 int yywrap() {
175     return 1;
176 }

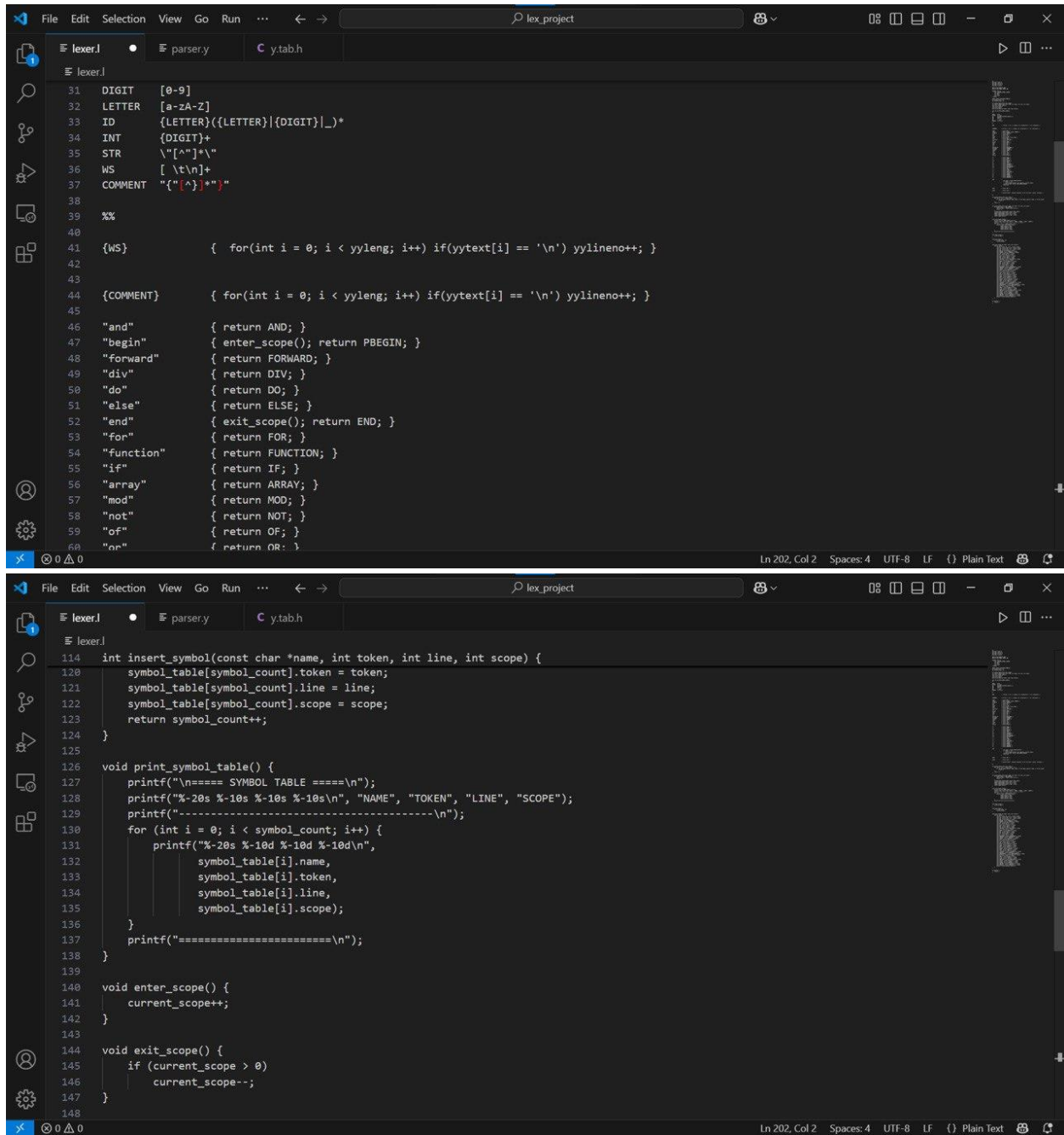
```

```

90 {ID} {
91     int index = lookup_symbol(yytext);
92     if (index == -1)
93         insert_symbol(yytext, ID, yylineno, current_scope);
94     strncpy(id_text, yytext, MAX_SYMBOL_LENGTH);
95     return ID;
96 }
97
98 {INT} { return INT; }
99
100 {STR} { return STR; }
101
102 . { fprintf(stderr, "Unknown character: %s at line %d\n", yytext, yylineno); }
103
104 %%
105
106 int lookup_symbol(const char *name) {
107     for (int i = 0; i < symbol_count; i++) {
108         if (strcmp(symbol_table[i].name, name) == 0 && symbol_table[i].scope <= current_scope)
109             return i;
110     }
111     return -1;
112 }
113
114 int insert_symbol(const char *name, int token, int line, int scope) {
115     if (symbol_count >= MAX_SYMBOLS) {
116         fprintf(stderr, "Symbol table full\n");
117         return -1;
118     }
119     strcpy(symbol_table[symbol_count].name, name);

```

Compiler Construction



The image displays two screenshots of a code editor, likely Visual Studio Code, showing the implementation of a compiler construction project. The editor has a dark theme and a sidebar on the left with icons for Explorer, Search, Source Control, and Run and Debug. The top toolbar includes File, Edit, Selection, View, Go, Run, and other standard editor actions. The project name 'lex_project' is visible in the top right corner.

The first screenshot shows the `lexer.l` file, which defines lexical tokens and their corresponding actions. The tokens are defined as follows:

- `DIGIT`: `[0-9]`
- `LETTER`: `[a-zA-Z]`
- `ID`: `{LETTER}({LETTER}|{DIGIT}|_)*`
- `INT`: `{DIGIT}+`
- `STR`: `\"[^\"]+\"`
- `WS`: `[\t\n]+`
- `COMMENT`: `\"{\"[^\"]+\"}`

The actions are defined in the `%%` section:

- `{WS}`: `{ for(int i = 0; i < yytext[i]; i++) if(yytext[i] == '\\n') yylineno++; }`
- `{COMMENT}`: `{ for(int i = 0; i < yytext[i]; i++) if(yytext[i] == '\\n') yylineno++; }`
- `"and"`: `{ return AND; }`
- `"begin"`: `{ enter_scope(); return PBEGIN; }`
- `"forward"`: `{ return FORWARD; }`
- `"div"`: `{ return DIV; }`
- `"do"`: `{ return DO; }`
- `"else"`: `{ return ELSE; }`
- `"end"`: `{ exit_scope(); return END; }`
- `"for"`: `{ return FOR; }`
- `"function"`: `{ return FUNCTION; }`
- `"if"`: `{ return IF; }`
- `"array"`: `{ return ARRAY; }`
- `"mod"`: `{ return MOD; }`
- `"not"`: `{ return NOT; }`
- `"of"`: `{ return OF; }`
- `"or"`: `{ return OR; }`

The second screenshot shows the `y.tab.h` file, which contains the declarations for the symbol table and the functions used in the parser. The declarations are as follows:

```
int insert_symbol(const char *name, int token, int line, int scope) {
    symbol_table[symbol_count].token = token;
    symbol_table[symbol_count].line = line;
    symbol_table[symbol_count].scope = scope;
    return symbol_count++;
}

void print_symbol_table() {
    printf("\\n===== SYMBOL TABLE =====\\n");
    printf("%-20s %-10s %-10s %-10s\\n", "NAME", "TOKEN", "LINE", "SCOPE");
    printf("-----\\n");
    for (int i = 0; i < symbol_count; i++) {
        printf("%-20s %-10d %-10d %-10d\\n",
            symbol_table[i].name,
            symbol_table[i].token,
            symbol_table[i].line,
            symbol_table[i].scope);
    }
    printf("=====\\n");
}

void enter_scope() {
    current_scope++;
}

void exit_scope() {
    if (current_scope > 0)
        current_scope--;
}
```

Compiler Construction

The image displays two screenshots of a Visual Studio Code editor window, showing the implementation of a compiler's front-end components. The top screenshot shows the `lexer.l` file, which defines a set of lexical rules. The bottom screenshot shows the `parser.y` file, which defines the parser's actions and shift-reduce logic.

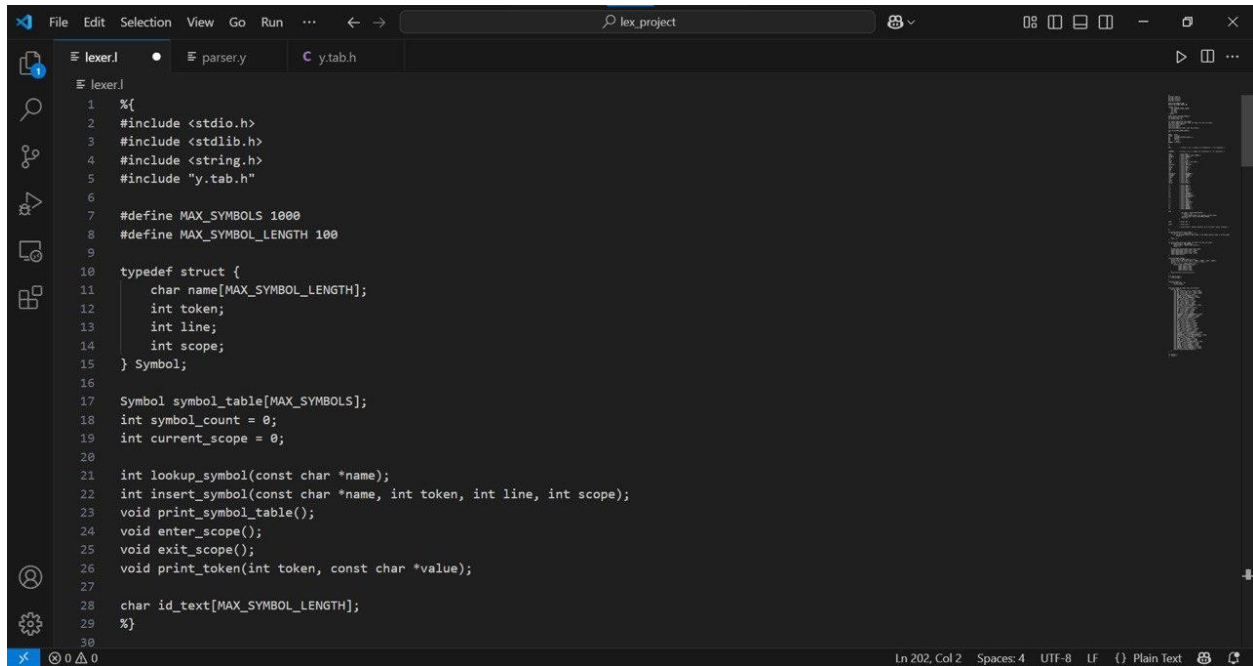
lexer.l

```
61 "procedure" { return PROCEDURE; }
62 "program"   { return PROGRAM; }
63 "record"    { return RECORD; }
64 "then"      { return THEN; }
65 "to"        { return TO; }
66 "type"      { return TYPE; }
67 "var"       { return VAR; }
68 "while"     { return WHILE; }
69
70 "+"         { return PLUS; }
71 "-"         { return MINUS; }
72 "*"         { return MULT; }
73 ":@"       { return ASSIGN; }
74 "="         { return EQUAL; }
75 "<"         { return LESS; }
76 "<="        { return LESSEQUAL; }
77 ">"         { return GREATER; }
78 ">="        { return GREATEREQUAL; }
79 "<>"        { return NOTEQUAL; }
80 "."         { return DOT; }
81 ","         { return COMMA; }
82 ":"         { return COLON; }
83 ";"         { return SEMICOLON; }
84 "."         { return DOTDOT; }
85 "("         { return LPAREN; }
86 ")"         { return RPAREN; }
87 "["         { return LBRACKET; }
88 "]"         { return RBRACKET; }
89
90 {ID}        {
```

parser.y

```
149 void print_token(int token, const char *value) {
150     switch (token) {
151         case ID: printf("ID: %s\n", value); break;
152         case INT: printf("INT: %s\n", value); break;
153         case STR: printf("STR: %s\n", value); break;
154         case AND: printf("AND\n"); break;
155         case PBEGIN: printf("BEGIN\n"); break;
156         case FORWARD: printf("FORWARD\n"); break;
157         case DIV: printf("DIV\n"); break;
158         case DO: printf("DO\n"); break;
159         case ELSE: printf("ELSE\n"); break;
160         case END: printf("END\n"); break;
161         case FOR: printf("FOR\n"); break;
162         case FUNCTION: printf("FUNCTION\n"); break;
163         case IF: printf("IF\n"); break;
164         case ARRAY: printf("ARRAY\n"); break;
165         case MOD: printf("MOD\n"); break;
166         case NOT: printf("NOT\n"); break;
167         case OF: printf("OF\n"); break;
168         case OR: printf("OR\n"); break;
169         case PROCEDURE: printf("PROCEDURE\n"); break;
170         case PROGRAM: printf("PROGRAM\n"); break;
171         case RECORD: printf("RECORD\n"); break;
172         case THEN: printf("THEN\n"); break;
173         case TO: printf("TO\n"); break;
174         case TYPE: printf("TYPE\n"); break;
175         case VAR: printf("VAR\n"); break;
176         case WHILE: printf("WHILE\n"); break;
177         case PLUS: printf("PLUS\n"); break;
178         case MINUS: printf("MINUS\n"); break;
```


Compiler Construction



```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "y.tab.h"
6
7  #define MAX_SYMBOLS 1000
8  #define MAX_SYMBOL_LENGTH 100
9
10 typedef struct {
11     char name[MAX_SYMBOL_LENGTH];
12     int token;
13     int line;
14     int scope;
15 } Symbol;
16
17 Symbol symbol_table[MAX_SYMBOLS];
18 int symbol_count = 0;
19 int current_scope = 0;
20
21 int lookup_symbol(const char *name);
22 int insert_symbol(const char *name, int token, int line, int scope);
23 void print_symbol_table();
24 void enter_scope();
25 void exit_scope();
26 void print_token(int token, const char *value);
27
28 char id_text[MAX_SYMBOL_LENGTH];
29 %}
```

C: > Users > sarah > Downloads > Telegram Desktop > test.pas

```
1  program TestProgram(input, output);
2  begin
3      x := 10;
4      y := 20;
5      if x + y > 25 then
6          z := x + y;
7  end.
```

Syntax Analyzer

1) What is YACC?

It automatically generates the LALR(1) parsers from formal grammar specifications. YACC plays an important role in compiler and interpreter development since it provides a means to specify the grammar of a language and to produce parsers that either interpret or compile code written in that language.

2) How does YACC work?

YACC takes grammar rules written in a specific syntax and uses them to generate C source code for a parser. This parser can then be compiled into an executable program that reads input, checks if it follows the grammar, and performs actions based on the structure of the input.

3) How does it integrate with LEX?

YACC works together with LEX by receiving the stream of tokens that LEX generates during lexical analysis. LEX scans the input text, recognizes patterns, and sends tokens to YACC through the `yylex()` function. After that, YACC uses its grammar rules to analyze the structure of these tokens and performs the appropriate actions. This collaboration allows both tools to function as the front-end of a compiler, where LEX focuses on tokenizing the input and YACC focuses on parsing it according to the language grammar.

4) What is the structure of a YACC program?

A YACC program is divided into three main sections, separated by `%%`. The first section is the **declarations section**, where tokens, data types, and header files are defined. The second section contains the **grammar rules**, where each rule defines a production and its associated C action code. The third section includes **auxiliary functions**, such as `main()`, which support the parser's functionality. This structure allows YACC to generate a syntax analyzer based on the defined grammar.

The output\ Input

```
wajn@LAPTOP-10R1BVCP: ~/lex_project
wajn@LAPTOP-10R1BVCP:~/lex_project$ ./parser test.pas
Starting parser...
Recognized identifier: input
Recognized identifier list with: output
  at line 1aracter:
  at line 2aracter:
Recognized integer in expression
Recognized assignment statement
Recognized statement
  at line 3aracter:
Recognized integer in expression
Recognized assignment statement
Recognized multiple statements
  at line 4aracter:
Recognized identifier in expression
Recognized identifier in expression
Recognized integer in expression
Recognized greater-than expression
Recognized addition expression
  at line 5aracter:
Recognized identifier in expression
Recognized identifier in expression
Recognized addition expression
Recognized assignment statement
Recognized if statement
Recognized multiple statements
  at line 6aracter:
Parsing complete.
wajn@LAPTOP-10R1BVCP:~/lex_project$
```

The Grammer code

The image displays two screenshots of a code editor (Visual Studio Code) showing the grammar rules for a compiler project. The editor has a dark theme and a sidebar on the left with icons for Explorer, Search, Source Control, and Run and Debug. The top screenshot shows the first part of the grammar file, and the bottom screenshot shows the second part.

Top Screenshot Grammar Rules:

```

1  Program : PROGRAM ID LPAREN IdentifierList RPAREN SEMICOLON
2          TypeDefinitions
3          VariableDeclarations
4          SubprogramDeclarations
5          CompoundStatement
6          DOT;
7
8  TypeDefinitions :
9          /* empty */
10         | TYPE TypeDefinition SEMICOLON
11         | TypeDefinitions TypeDefinition SEMICOLON
12         ;
13
14  TypeDefinition : ID EQUAL Type;
15
16  VariableDeclarations :
17         /* empty */
18         | VAR VariableDeclaration SEMICOLON
19         | VariableDeclarations VariableDeclaration SEMICOLON
20         ;
21
22  VariableDeclaration : IdentifierList COLON Type;
23
24  SubprogramDeclarations :
25         /* empty */
26         | SubprogramDeclaration SEMICOLON
27         | SubprogramDeclarations SubprogramDeclaration SEMICOLON
28         ;
29
30  SubprogramDeclaration : ProcedureDeclaration

```

Bottom Screenshot Grammar Rules:

```

30  SubprogramDeclaration : ProcedureDeclaration
31         | FunctionDeclaration;
32
33  ProcedureDeclaration : PROCEDURE ID LPAREN FormalParameterList RPAREN SEMICOLON ProcedureBody;
34
35  FunctionDeclaration : FUNCTION ID LPAREN FormalParameterList RPAREN COLON ResultType SEMICOLON ProcedureBody;
36
37  ProcedureBody : Block
38         | FORWARD;
39
40  FormalParameterList :
41         /* empty */
42         | IdentifierList COLON Type
43         | FormalParameterList SEMICOLON IdentifierList COLON Type;
44
45  Block : VariableDeclarations CompoundStatement;
46
47  CompoundStatement : PBEGIN StatementSequence END;
48
49  StatementSequence : Statement
50         | StatementSequence SEMICOLON Statement;
51
52  Statement : SimpleStatement
53         | StructuredStatement;
54
55  SimpleStatement :
56         /* empty */
57         | AssignmentStatement
58         | ProcedureStatement;

```

Compiler Construction

The image displays two screenshots of a VS Code editor window showing a grammar file named 'Grammar.txt' for a compiler construction project. The editor has a dark theme and a sidebar on the left with icons for Explorer, Search, Run and Debug, and Extensions. The top toolbar includes File, Edit, Selection, View, Go, Run, and a search bar.

Top Screenshot (Lines 86-115):

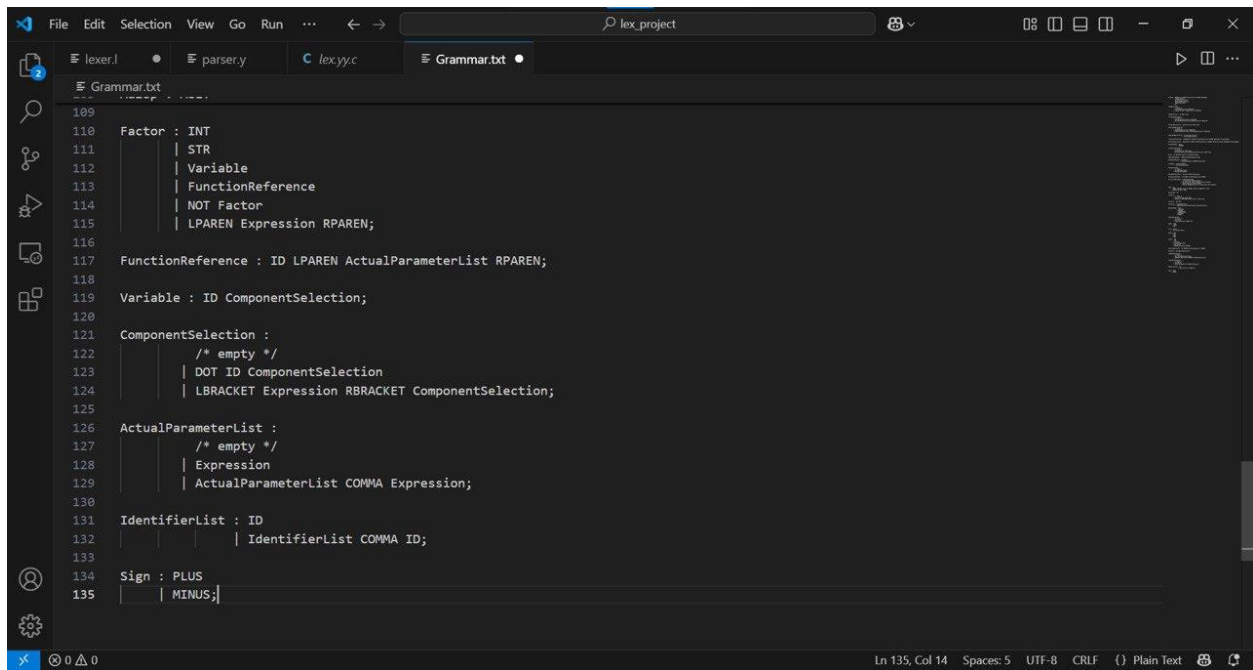
```
86 RelationalOp : LESS
87               | LESSEQUAL
88               | GREATER
89               | GREATEREQUAL
90               | NOTEQUAL
91               | EQUAL;
92
93 SimpleExpression :
94                 /* empty */
95                 | Sign Term
96                 | SimpleExpression AddOp Term;
97
98 AddOp : PLUS
99       | MINUS
100      | OR;
101
102 Term : Factor
103      | Term MulOp Factor;
104
105 MulOp : MULT
106       | DIV
107       | MOD
108       | AND;
109
110 Factor : INT
111        | STR
112        | Variable
113        | FunctionReference
114        | NOT Factor
115        | LPAREN Expression RPAREN;
```

Bottom Screenshot (Lines 59-88):

```
59
60 AssignmentStatement : Variable ASSIGN Expression;
61
62 ProcedureStatement : ID LPAREN ActualParameterList RPAREN;
63
64 StructuredStatement : CompoundStatement
65                     | IF Expression THEN Statement
66                     | IF Expression THEN Statement ELSE Statement
67                     | WHILE Expression DO Statement
68                     | FOR ID ASSIGN Expression TO Expression DO Statement;
69
70 Type : ID
71      | ARRAY LBRACKET Constant DOTDOT Constant RBRACKET OF Type
72      | RECORD FieldList END;
73
74 ResultType : ID;
75
76 FieldList :
77           /* empty */
78           | IdentifierList COLON Type
79           | FieldList SEMICOLON IdentifierList COLON Type;
80
81 Constant : Sign INT;
82
83 Expression : SimpleExpression
84            | SimpleExpression RelationalOp SimpleExpression;
85
86 RelationalOp : LESS
87              | LESSEQUAL
88              | GREATER
```

The status bar at the bottom of both screenshots indicates 'Ln 135, Col 14', 'Spaces: 5', 'UTF-8', 'CRLF', and 'Plain Text'.

Compiler Construction



```
109
110 Factor : INT
111         | STR
112         | Variable
113         | FunctionReference
114         | NOT Factor
115         | LPAREN Expression RPAREN;
116
117 FunctionReference : ID LPAREN ActualParameterList RPAREN;
118
119 Variable : ID ComponentSelection;
120
121 ComponentSelection :
122         /* empty */
123         | DOT ID ComponentSelection
124         | LBRACKET Expression RBRACKET ComponentSelection;
125
126 ActualParameterList :
127         /* empty */
128         | Expression
129         | ActualParameterList COMMA Expression;
130
131 IdentifierList : ID
132                | IdentifierList COMMA ID;
133
134 Sign : PLUS
135       | MINUS;
```

Ln 135, Col 14 Spaces: 5 UTF-8 CRLF Plain Text

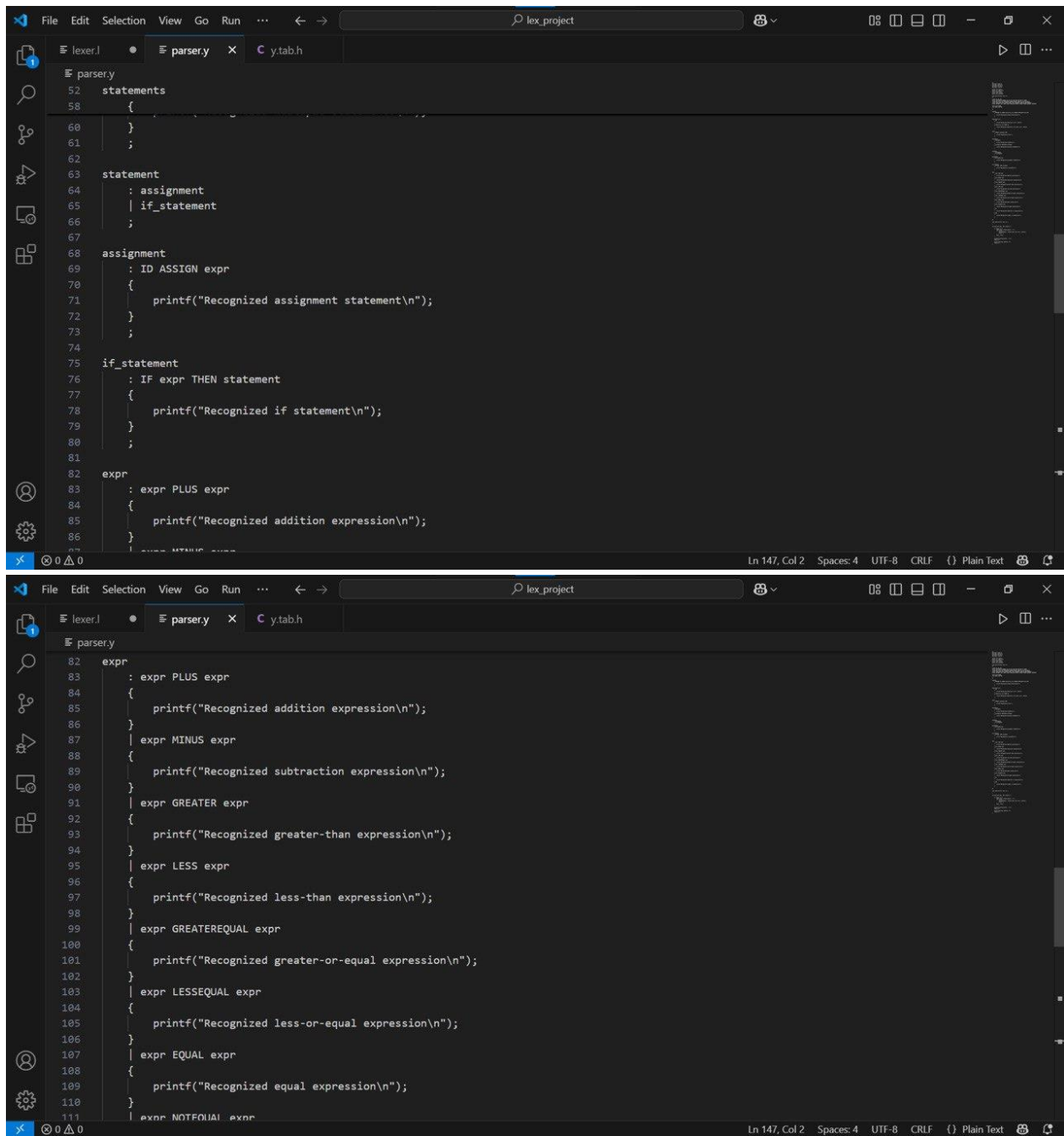
The Parser code

```

1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include "y.tab.h"
6
7  extern int yylex();
8  extern int yyparse();
9  extern FILE *yyin;
10 extern char *yytext;
11 extern int yylineno;
12
13 void yyerror(const char *s);
14 %}
15
16 %token ID INT STR
17 %token AND PBEGIN FORWARD DIV DO ELSE END FOR FUNCTION IF ARRAY
18 %token MOD NOT OF OR PROCEDURE PROGRAM RECORD THEN TO TYPE VAR WHILE
19 %token PLUS MINUS MULT ASSIGN EQUAL LESS LESSEQUAL GREATER GREATEREQUAL
20 %token NOTEQUAL DOT COMMA COLON SEMICOLON DOTDOT LPAREN RPAREN LBRACKET RBRACKET
21
22 %left PLUS MINUS
23 %left MULT DIV MOD
24
25 %%
26
27 program
28 : PROGRAM ID LPAREN identifier_list RPAREN SEMICOLON block DOT
29 {
30     printf("Recognized program declaration\n");
31 }
32 ;
33
34 identifier_list
35 : ID
36 {
37     printf("Recognized identifier: %s\n", yytext);
38 }
39 | identifier_list COMMA ID
40 {
41     printf("Recognized identifier list with: %s\n", yytext);
42 }
43 ;
44
45 block
46 : PBEGIN statements END
47 {
48     printf("Recognized block\n");
49 }
50 ;
51
52 statements
53 : statement
54 {
55     printf("Recognized statement\n");
56 }
57 | statements SEMICOLON statement
58 {

```

Compiler Construction



The image displays two screenshots of a Visual Studio Code editor window, showing the implementation of a parser in C. The editor is open to a file named `parser.y` within a project named `lex_project`. The code is written in C and uses a recursive descent parsing approach.

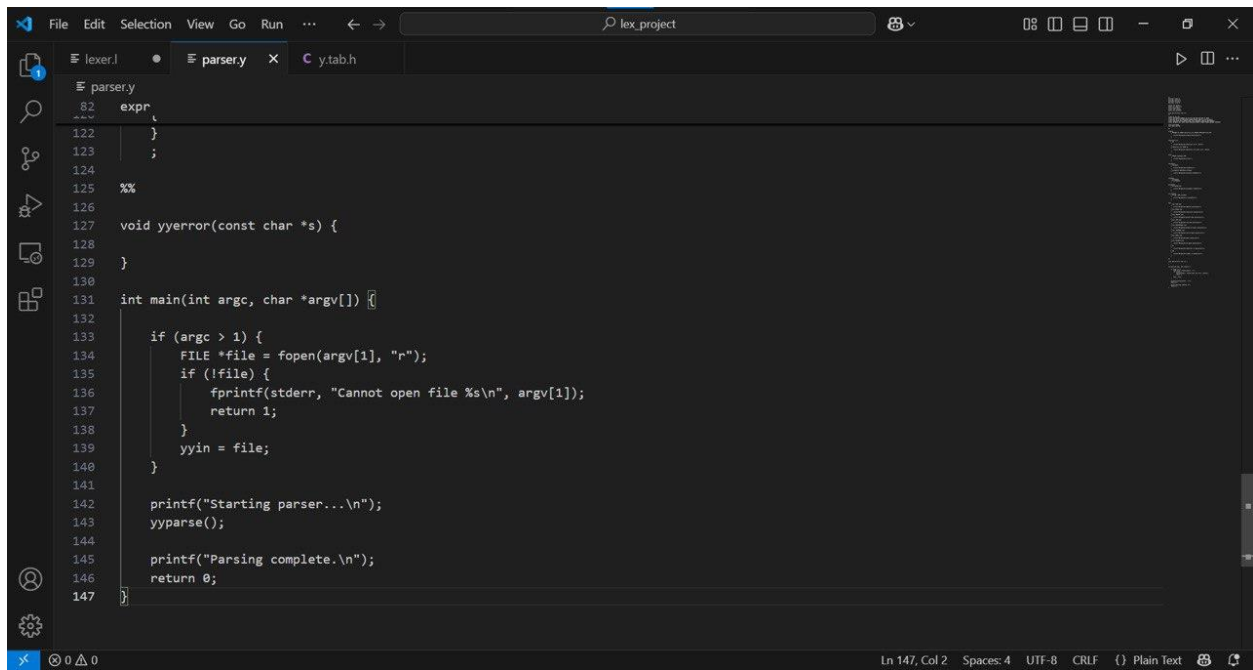
Top Screenshot: Shows the `statements` and `statement` rules. The `statements` rule is a list of `statement` tokens. The `statement` rule is a union of `assignment` and `if_statement`. The `assignment` rule is `ID ASSIGN expr`, and the `if_statement` rule is `IF expr THEN statement`. Both rules include a `printf` statement for debugging.

```
52 statements
53 {
54     statement
55 }
56
57
58 statement
59 : assignment
60 | if_statement
61 ;
62
63 assignment
64 : ID ASSIGN expr
65 {
66     printf("Recognized assignment statement\n");
67 }
68 ;
69
70 if_statement
71 : IF expr THEN statement
72 {
73     printf("Recognized if statement\n");
74 }
75 ;
76
77 expr
78 : expr PLUS expr
79 {
80     printf("Recognized addition expression\n");
81 }
82 ;
```

Bottom Screenshot: Shows the `expr` rule, which is a union of various arithmetic and comparison expressions. The rules include `expr PLUS expr`, `expr MINUS expr`, `expr GREATER expr`, `expr GREATEREQUAL expr`, `expr LESSEQUAL expr`, `expr EQUAL expr`, and `expr NOTFOUALLI expr`. Each rule includes a `printf` statement for debugging.

```
82 expr
83 : expr PLUS expr
84 {
85     printf("Recognized addition expression\n");
86 }
87 | expr MINUS expr
88 {
89     printf("Recognized subtraction expression\n");
90 }
91 | expr GREATER expr
92 {
93     printf("Recognized greater-than expression\n");
94 }
95 | expr LESS expr
96 {
97     printf("Recognized less-than expression\n");
98 }
99 | expr GREATEREQUAL expr
100 {
101     printf("Recognized greater-or-equal expression\n");
102 }
103 | expr LESSEQUAL expr
104 {
105     printf("Recognized less-or-equal expression\n");
106 }
107 | expr EQUAL expr
108 {
109     printf("Recognized equal expression\n");
110 }
111 | expr NOTFOUALLI expr
```


Compiler Construction



```
File Edit Selection View Go Run ... lex_project
parsery
82  expr
122  }
123  ;
124
125  %%
126
127  void yyerror(const char *s) {
128
129  }
130
131  int main(int argc, char *argv[]) {
132
133      if (argc > 1) {
134          FILE *file = fopen(argv[1], "r");
135          if (!file) {
136              fprintf(stderr, "Cannot open file %s\n", argv[1]);
137              return 1;
138          }
139          yyin = file;
140      }
141
142      printf("Starting parser...\n");
143      yyparse();
144
145      printf("Parsing complete.\n");
146      return 0;
147  }
```

Ln 147, Col 2 Spaces: 4 UTF-8 CRLF Plain Text

Test Cases Output

Test case 1:

1) Production rules heads:

Program
CompoundStatement
StatementSequence
AssignmentStatement

2) Symbol Table:

Name	Type
x	variable

Test case 2 :

1) Production rules heads:

Program
VariableDeclaration
FunctionDeclaration
CompoundStatement

2) Symbol Table:

Name	Type
x	variable
y	variable
add	function

Test case 3 :

1) Production rules heads:

Program
SubprogramDeclarations
ProcedureDeclaration
Block
CompoundStatement
StatementSequence
AssignmentStatement
CompoundStatement
StatementSequence
ProcedureStatement

2) Symbol Table:

Name	Type
test3	program
printSum	procedure
a	parameter
b	parameter
result	variable

Test case 4 :

1) Production rules heads:

Program
SubprogramDeclarations
FunctionDeclaration
Block
CompoundStatement
StatementSequence
AssignmentStatement
CompoundStatement
StatementSequence
AssignmentStatement

2) Symbol Table:

Name	Type
test4	program
square	function
n	parameter
result	variable

How To Run The Code

LEXER:

```
mkdir lex_project
```

```
cd lex_project
```

```
code.
```

```
flex lexer.l
```

```
gcc lex.yy.c -o lexer.exe -lfl
```

```
gcc lex.yy.c -o lexer -lfl
```

```
./ lexer
```

```
--> input
```

Symbol Table:

```
cd lex_project
```

```
ls
```

```
(File name)
```

```
flex lexer.l
```

```
gcc lex.yy.c -o lexer -lfl
```

```
./lexer
```

```
touch test.pas
```

```
./ lexer < test.pas
```

```
--> input
```

Parser:

```
cd lex_project
```

```
ls
```

```
(File name)
```

```
yacc -d parser.y
```

```
flex lexer.l
```

```
gcc -o parser y.tab.c lex.yy.c -lfl
```

```
./parser test.pas
```

```
--> input
```

Summary

- Challenges:
 - **General Debugging** : Recompiling frequently, tracing down ambiguous errors, and determining which part (lexer vs parser) was broken required a great deal of mental effort.
 - **Disk Space Issues**: Errors like no space left on device wasted time debugging non-code-related problems.
 - **Token Mismatch**: Token names in the lexer had to *exactly* match those declared in the parser. Which took long to deal with.

References

Lex & Yacc book :<https://www.naukri.com/code360/library/lexical-analysis-in-compiler-design>

<https://www.geeksforgeeks.org/introduction-to-yacc/><https://www.geeksforgeeks.org/introduction-to-yacc/>

<https://norasandler.com/2017/11/29/Write-a-Compiler.html>

<https://austinhoenley.com/blog/teenytinycompiler1.html>

<https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf>

<https://www.ibm.com/sa-ar>

note: Throughout the research and writing process, we would like to appreciate the usage of OpenAI's ChatGPT as an additional tool for **understanding** course material, and analyzing difficult subjects.