Assignment- 1

COMPILER CONSTRUCTION

Submitted by:	Numan Ali
Roll no:	20021519-137
Submitted to:	Mam. Saliha Zahoor
Section:	BSCS-B-20
Date:	11 Nov,2023

Department of Computer Science

Q1: Explain Lexical Analyzer Generator, Structure of Lex program, Sample Lex programs. Also explain other Compiler Construction Tools.

Lex is a tool that creates lexical analyzers by converting a set of regular expressions from an input file into a C-based finite state machine. The resulting program, when executed, becomes a functional lexical analyzer. This analyzer processes input character streams and outputs corresponding token streams. The generation process can utilize either NFA or DFA, with DFA being commonly favored for efficiency in lex program implementation.

Types of token as following

1. Identifier
2. Keyword
3. Operator
4. Constants
5. Special symbol(@, $, #)

Structure of a Lex Program:

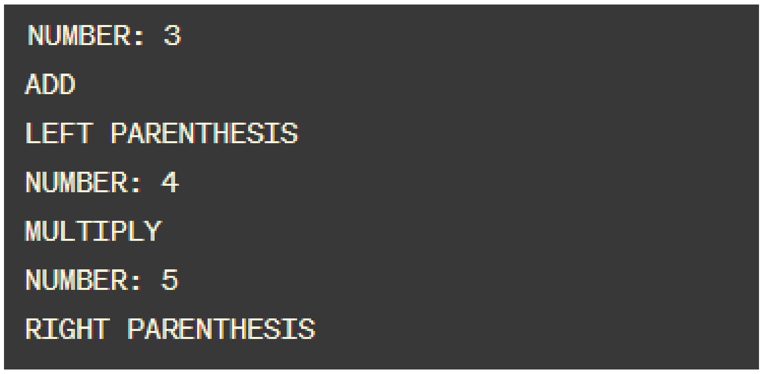
1. **Definitions Section:**
   This section contains a set of rules, each consisting of a regular expression and an associated action.
2. **Rules Section:**
   In this section, you define regular expressions and associate them with token names. These regular expressions describe the patterns that the lexical analyzer will recognize.
3. **C Code Section:**
   This section contains a set of rules, each consisting of a regular expression and an associated action. When a pattern matching the regular expression is found in the input, the corresponding action is executed.

Program Example:

```
%{
#include <stdio.h>
%}
DIGIT [0-9]
WS    [\t\n]
%%
{DIGIT}+ { printf("NUMBER: %s\n", yytext); }
{WS}     { /* Ignore whitespace */ }
"+"        { printf("ADD\n"); }
"-"        { printf("SUBTRACT\n"); }
"*"        { printf("MULTIPLY\n"); }
"/"        { printf("DIVIDE\n"); }
"("        { printf("LEFT PARENTHESIS\n"); }
")"        { printf("RIGHT PARENTHESIS\n"); }

.          { printf("INVALID CHARACTER: %s\n", yytext); }

%%

int main() {
    yylex(); // Start lexical analysis
    return 0;
}
```

Output:

For input: 3 + (4 * 5)



```
NUMBER: 3
ADD
LEFT PARENTHESIS
NUMBER: 4
MULTIPLY
NUMBER: 5
RIGHT PARENTHESIS
```

Other Compilers:

- **Flex:**
  Purpose: A faster alternative to Lex for generating lexical analyzers
  Usage: Works similarly to Lex but offers improvements in terms of speed and functionality
- **Bison:**
  Purpose: A counterpart to Yacc for generating parsers.
  Usage: Takes a context-free grammar and produces a parser capable of understanding the syntactic structure of the source code.
- **LLVM:**
  Purpose: A collection of modular and reusable compiler and toolchain technologies.
  Usage: Provides infrastructure for building compilers and related tools, including optimization passes and code generation.
- **ANTLR:**
- **Purpose:** A powerful parser generator that supports multiple target languages.
  Usage: Allows the specification of grammars for languages and generates parsers in various programming languages

Q2. Write a program of identification of tokens. Your program will take a .txt file that contains the source code of C language as an input and gives the sequence of tokens an output.

Program:

```
%{
#include <stdio.h>
%}
%%
int         { printf("<keyword, int>\n"); }
if          { printf("<keyword, if>\n"); }
else        { printf("<keyword, else>\n"); }
for         { printf("<keyword, for>\n"); }
while       { printf("<keyword, while>\n"); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("<id, %s>\n", yytext); }
[0-9]+      { printf("<number, %s>\n", yytext); }
"=="        { printf("<comparison, ==>\n"); }
"="         { printf("<assignment, =>\n"); }
"("         { printf("<parenthesis, (>\n"); }
")"         { printf("<parenthesis, )>\n"); }
"{"         { printf("<brace, {>\n"); }
"}"         { printf("<brace, }>\n"); }
";"         { printf("<semicolon, ;>\n"); }


[ \t\n]     ; // Ignore whitespace
```

Numan Ali
20021519-137

```
.           { printf("Invalid token: %s\n", yytext); }


%%


int main() {

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

    yylex(); // Start lexical analysis

    fclose(yyin);

    return 0;

}
```

Steps For Runing:

- Save the file with name Lexer with extension .l
- Generate the lexer using Flex. It will create a new file with extension lex.yy.c
- Compile the generated C file (e.g., **lex.yy.c**)

Output:

```
<keyword, int>

<id, main>

<parenthesis, (>

<parenthesis, )>

<brace, {>

<keyword, int>

<id, a>

<assignment, =>

<number, 5>

<semicolon, ;>

<keyword, int>
```

Numan Ali
20021519-137

```
<id, b>
<assignment, =>
<number, 10>
<semicolon, ;>
<keyword, if>
<parenthesis, (>
<id, a>
<comparison, ==>
<id, b>
<parenthesis, )>
<brace, {>
<id, printf>
<parenthesis, (>
<string_literal, "Equal\n">
<parenthesis, )>
<semicolon, ;>
<brace, }>
<keyword, else>
<brace, {>
<id, printf>
<parenthesis, (>
<string_literal, "Not Equal\n">
<parenthesis, )>
<semicolon, ;>
<brace, }>
<return_keyword, return>
<number, 0>
<semicolon, ;>
```

Numan Ali
20021519-137

<brace, }>

Q3. Explain Recursive Descent parser and Predictive Parser with the help of examples.

Recursive Descent Parser:

- **Approach:** Recursive Descent Parsing involves writing a set of recursive procedures to recognize the productions of a grammar.
- **Implementation:** Each non-terminal in the grammar corresponds to a parsing function. The parsing functions call each other recursively based on the production rules.
- **Backtracking:** Recursive Descent Parsers may involve backtracking when there are multiple alternatives for a production.
- **Predictive Parsing:** If a Recursive Descent Parser can predict the correct production without backtracking, it is also called a Predictive Parser.

Example:

String: r e a d

Grammer

S→r X d | r Z d

X→ o a | e a

Z→a i

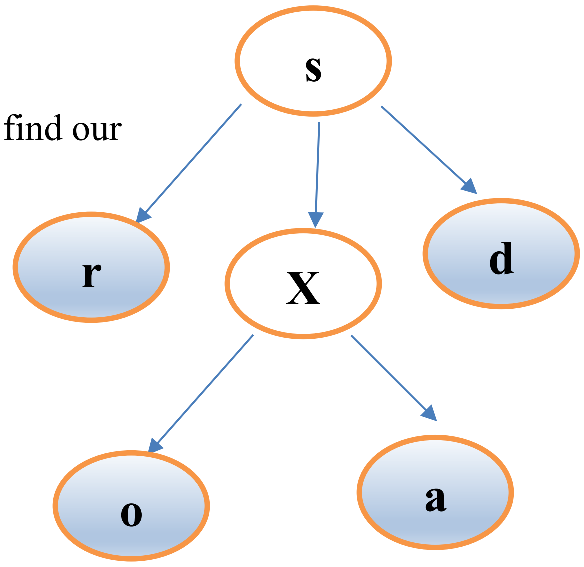
Step 1: By choosing this

approach we have answer **r o a d** didn't find our

actual string output (r e a d)

so, we again perform backtracking

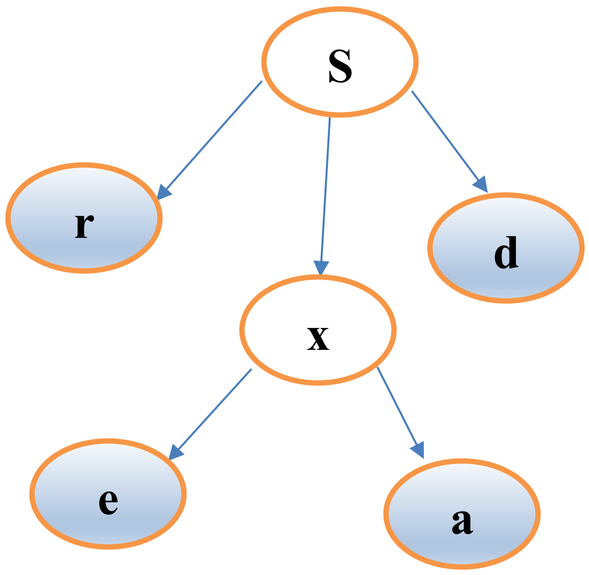
for desired outcome.



Step 2: Backtracking

Now we get the actual string through the parser.

r e a d



Predictive Parser / LL1 Parser:

A Predictive Parser, specifically in the context of LL parsing, is a top-down parser that makes unambiguous parsing decisions without backtracking. It uses a parsing table derived from the grammar, operating in a deterministic manner with a lookahead of one symbol (LL(1)).

Example :

Grammer:

1. S -> aABb

2. A -> c | ε

3. B -> d | ε

First Sets:

- **First(S):** {a}
- **First(A):** {c, ε}
- **First(B):** {d, ε}

Follow Sets:

- **Follow(S):** {$, b}
- **Follow(A):** {b, d, $}
- **Follow(B):** {b, $}

	a	**b**	**c**	**d**	**$**
S	S -> aABb				
A		A -> ε	A -> c		A -> ε
B		B -> ε		B -> d	B -> ε

The input string is accepted, and the parsing steps demonstrate the LL(1) parsing process for the given grammar.

Numan Ali
20021519-137