BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	29/2/2024
Lab #	4
Aim	Write a program to implement SDT using Lex/Yacc
Objective	To develop and implement a Syntax-Directed Translation (SDT) system using Lex and Yacc, the primary goal is to create a parser that translates source code into a target format by defining lexical tokens and grammar rules. This experiment focuses on the application of compiler design principles, specifically in the realms of lexical analysis and parsing, with the ultimate aim of achieving efficient syntax-directed translation.
Theory	# Yacc ## Definition Yacc, which stands for "Yet Another Compiler Compiler," is a computer program that generates parser programs. It's designed to work with a given LALR(1) grammar. The term LALR refers to Look-Ahead Left-to-right Rightmost derivation, a type of context-free grammar. Yacc processes this grammar and outputs a C program. It relies on another tool called Lex, which is a lexical analyzer generator, to tokenize an input stream.[3] ## Parts of Yacc Program ### Definitions The definitions section is located at the top of the Yacc input file. This section includes header files or information about regular definitions or tokens. Token definitions use the `%token` directive, and any C-specific code is placed within `%{` and `%}`. For instance, you might see something like `%token ID` to define a token, or `{% #include <stdio.h> %}` to include a standard input/output header file.[2]

Rules

The rules section is enclosed between %% symbols. This section defines the actions that Yacc should take when scanning tokens. Actions in C are placed between curly brackets {}. These actions can be anything from printing output to executing complex logic.

Auxiliary Routines

Auxiliary routines include any functions that are required in the rules section. These are written in regular C syntax. This section also includes the `main()` function, where the `yyparse()` function is always called. The `yyparse()` function reads tokens, performs actions, and returns to the main function when it reaches the end of the file or encounters an error. It returns 0 if parsing was successful and 1 if it was unsuccessful.[3]

How Yacc Works

Translation Rules

Each translation rule input to YACC has a string specification that resembles a production of a grammar. It consists of a nonterminal on the left-hand side (LHS) and a few alternatives on the right-hand side (RHS). We refer to this string specification as a production.[2]

Parser Generation and Operation

YACC generates an LALR(1) parser for a language L from these productions. This parser operates as a bottom-up parser. The operation of the parser is as follows:

- For a shift action, the parser invokes the scanner to obtain the next token and continues the parse by using that token.
- While performing a reduced action in accordance with a production, it performs the semantic action associated with that production.

Semantic Actions and Intermediate Representation

The semantic actions associated with productions are used to build an intermediate representation or target code. The process is as follows:

- Every nonterminal symbol in the parser has an attribute.
- The semantic action associated with a production can access attributes of nonterminal symbols used in that production. A symbol $n in the semantic action, where n is an integer, designates the attribute of the nonterminal

symbol in the RHS of the production. The symbol $$ designates the attribute of the LHS nonterminal symbol of the production.

- The semantic action uses the values of these attributes for building the intermediate representation or target code.[2]

Steps for Compiling YACC Program

1. Write lex program in a file file.l and yacc in a file file.y[1]
2. Open Terminal and Navigate to the Directory where you have saved the files.
3. type lex file.l
4. type yacc file.y
5. type cc lex.yy.c y.tab.h -ll
6. type ./a.out [1]

Implementation / Code

sdt_lex.l

```
%{
#include "y.tab.h"
%}

%%
N { return N; }
S { return S; }
E { return E; }
W { return W; }
\n { return 0; }
. { return yytext[0]; }
%%

int yywrap(void) {
    return 1;
}
```

sdt_yacc.y

```
%{
#include <stdio.h>
int yylex();
```

```
void yyerror(const char *s);
int xpos = 0;
int ypos = 0;
%}

%token N S E W

%%

input: moves { printf("Final position is (%d, %d)\n", xpos, ypos); }
     | moves error { printf("Invalid string\n"); }
     ;

moves: moves move { }
     | move { }
     ;

move: N { ypos++; }
    | S { ypos--; }
    | E { xpos++; }
    | W { xpos--; }
    ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "%s\n", s);
}

int main() {
    printf("Enter input string: ");
    yyparse();
    return 0;
}
```

Output

```
adwait@spit:~/Documents/SPCC$ flex sdt_lex.l
adwait@spit:~/Documents/SPCC$ yacc -d sdt_yacc.y
adwait@spit:~/Documents/SPCC$ gcc lex.yy.c y.tab.c -o sdt_parser -ll
adwait@spit:~/Documents/SPCC$ ./sdt_parser
Enter input string: NNNNEEEEWWWWWSSS
Final position is (-1, 1)
adwait@spit:~/Documents/SPCC$ ./sdt_parser
Enter input string: NNNEEEWWS
Final position is (1, 2)
adwait@spit:~/Documents/SPCC$ ./sdt_parser
Enter input string: SSSWWW
Final position is (-3, -3)
adwait@spit:~/Documents/SPCC$ ./sdt_parser
Enter input string: NEEENWW
Final position is (1, 2)
adwait@spit:~/Documents/SPCC$
```

Conclusion

In this study, we delved into the application of syntax-directed translation through the utilization of Lex and Yacc in processing movement commands for determining a concluding position. The experiment offered a hands-on insight into the collaboration between lexical analyzers and parsers to comprehend and execute instructions according to grammar rules. Additionally, it underscored the significance of effective error handling, highlighting the necessity for parsing strategies that can handle errors robustly.

References

[1] *Example program for the lex and yacc programs*. (n.d.). Www.ibm.com. https://www.ibm.com/docs/en/aix/7.1?topic=information-example-program-lex-yacc-programs

[2] *Lecture 6: YACC and Syntax Directed Translation*. (n.d.). Retrieved February 29, 2024, from https://cs.gmu.edu/~white/CS540/Slides/Semantic/CS540-2-lecture6.pdf

[3] *Syntax Directed Translation in Compiler Design*. (2017, January 5). GeeksforGeeks. https://www.geeksforgeeks.org/syntax-directed-translation-in-compiler-design/