CS 04 706(P) Compiler Lab Manual

List of Experiments

1.Write a lex program to identify the tokens of C Language.

2.Write a lex program to count & List the tokens of C Language.

3.Write a Parser in YACC for a given grammar.

4.Wrte a peraser in YACC for while & do while loop statements in C.

5.Parser for 'for' statements.

6.Write a desktop calculator in YACC.

7.Write a Yacc Program to convert intfix expressions to post fix.

8.Implement a symbol table in YACC.

9. Implement a type checker in YACC.

10.Generate the three address code for the sentences in the following grammar.
E -> E or E | E and E | not E|(E) | id | id relop id

11.Generate the three address code for control flow statements in C

LAB 1
(Lexical Analyzer Tool Lex)

Program structure:
Lex program is divided into 3 parts seperated by %%
as shown below

I Part

%%

II Part

%%

III Part

I Part is Definitionds and inclusion of header files,
II Part is Patterns & corresponding Actions.
III part is usually blank , and it is the place for auxiliary procedures if used in Action part.
Actions can be any C statements.

Example:

```
%{
***include header files if any****
%}

Definitions

%%
Patterns        {Actions}


%%
Auxiliary procedures
```

Lexical Analyzer generator Lex generates lexical analyzer if the specification of tokens is given as regular expressions. In the definition part we can define names corresponding to regular expression patterns. Eg: iden [A-Za-z[[A-Za-z0-9]*
num [0-9]+
In the II Part Patterns can be written seperated by | (pipe symbol).
Strings can be wrtten inside " " & defined names can be written inside {}.
Eg: "if"| "else"|"while" {Action}
{iden} {Action}
{num} {Action}

Notes:

[ ] specify a character class
\ is escape character
–shows a character range
. matches every character except newline
yyteminate() is a function used to terminate the lexical analyzis
–By default input is taken from keyboard,else assign the input descriptor to yyin.
yylex() is the function which calls the lexical analyzer.
Every matched lexeme is stored in a character array named yytext & yylen stores its length.
–We can associate values with tokens returned from lex program when a pattern is matched by assigning the value to the variable yylval(which should be defined in the first part of lex program as integer or character pointer.)

PROGRAM 1:
Lex pgm to identify the tokens in a C program.
Algorithm:
1Define names for the patterns corresponding to tokens in the first section.
2.Match the patterns using defined names (which should be written inside { }) and other patterns .
3.Print the token matched in the action part.

PROGRAM2:
Lex Program to count the tokens in a C program file.
Algorithm:
1 a)Define count variables and initialized to 0(inside%{ %})
b)Define names for the patterns corresponding to tokens in the first section.

2.Match the patterns using defined names (which should be written inside { }) and other patterns .
3.Increment the corresponding count in the action part.
4.Print the count variables when an exit condition is meet.

Questions:
1.What happened if the identifiers are matched before keywords?
2.Can we attach the lexeme corresponding to a token matched while we return tokens to YACC? If yes How?
3.What is the need of input buffering in Lexical Analysis Phase?
4.What is a regular expression?
5.What is meant by (i)Lexeme(ii)Pattern(iii)Token
6.What is the use of variables yytext ,yylval & the function yyteminate()?
7.What is yylex()?

LAB II:
Parser Tool YACC

Introduction
The unix utility yacc (Yet Another Compiler Compiler) parses a stream of token, typically generated by lex, according to a user-specified grammar.

Structure of a yacc file

A yacc file looks much like a lex file:

...definitions...
%%
...rules...
%%
...code...

In the example just saw, all three sections are present:

Definitions section

There are three things that can go in the definitions section:

C code *Any code between %{ and %} is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.*
definitions *The definitions section of a lex file was concerned with characters; in yacc this is tokens. These token definitions are written to a .h file when yacc compiles this file.*
associativity rules *These handle associativity and priority of operators.definitions*

Lex Yacc interaction

Conceptually, lex parses a file of characters and outputs a stream of tokens; yacc accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.
If lex program is supplying a tokenizer, the yacc program will repeatedly call the yylex routine. The lex rules will probably function by calling return everytime they have parsed a token.

The shared header file of return codes

If lex is to return tokens that yacc will process, they have to agree on what tokens there are. This is done as follows.
• The yacc file will have token definitions
%token NUMBER
in the definitions section.
• When the yacc file is translated with yacc -d, a header file y.tab.h is created that has definitions like
#define NUMBER 258
This file can then be included in both the lex and yacc program.

• The lex file can then call return NUMBER, and the yacc program can match on this token.
The return codes that are defined from %TOKEN definitions typically start at around 258, so that single characters can simply be returned as their integer value:

```
/* in the lex program */
[0-9]+ {return NUMBER}
[-+*/] {return *yytext}
```

```
/* in the yacc program */
sum : TERMS '+' TERM
```

Rules section

The rules section contains the grammar of the language we want to parse. This looks like

```
name1 : THING something OTHERTHING {action}
        | othersomething THING {other action}
name2 : .....
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.
The terminal symbols get matched with return codes from the lex tokenizer.

User code section

The minimal main program is

```
int main()
{
yyparse();
return 0;
}
```

Extensions to more ambitious programs should be self-evident.
In addition to the main program, the code section will usually also contain subroutines, to be used either in the yacc or the lex program.

PROGRAM 1:
Write a Parser in YACC for a given grammar.

Algorithm:
YACC code:

1. Define tokens corresponding to Terminals in the given grammar.
2. Write the grammar in the YACC format.
3. Define main & call yyparse inside that.
4. Define yywrap & yyerror functions.

LEX code:

1. Include y.tab.h to get the token definitions in the YACC file.
2.Define patterns corresponding to the tokens.
3.Return tokens when the patterns are matched.

Compilation Order:

1.First Translate Yacc file *.y using the command yacc -d *.y.
Now 2 files are generated. y.tab.c & y.tab.h.
2.Translate LEX file *.l using the command LEX *.l.
Now a c file named lex.yy.c is generated.
3.Compile the 2 C files generated using the command gcc lex.yy.c y.tab.c
Now the executable file a.out is generated.
4.Execute the pgm using the command . /a.out

PROGRAM 2:

Write a parser in YACC for do while statements in C.
1.Find the CFG for do while loop .
YACC code:

1.Define tokens corresponding to Terminals in the grammar.
5.Write the grammar in the YACC format.
6.Define main & call yyparse inside that.
7.Define yywrap & yyerror functions.

LEX code:

1. Include y.tab.h to get the token definitions in the YACC file.
2.Define patterns corresponding to the tokens.
3.Return tokens when the patterns are matched.

PROGRAM 3:
Write a Calcutator in YACC.

1.Find the CFG for arithmetic expressions.
YACC code:

1.Define tokens corresponding to Terminals in the grammar.
2.Write the grammar in the YACC format.
3.Evaluate the arithmetic expressions using the values attached with the Number token.
($$ indicate the top of the value stack. $1,$2,..indicates the values attached with the symbols(1st ,2nd respectively) in the right side of a production.)
4.Define main & call yyparse inside that.
5.Define yywrap & yyerror functions.

LEX code:

1. Include y.tab.h to get the token definitions in the YACC file.
2.DEfine the variable yylval to attach values with tokens.
2.Define patterns corresponding to the tokens.
3.Return tokens when the patterns are matched.Attach values with number token by assigning values to yylval.

Questions:

1. What is the difference between topdown & Bottomup Parsing? Name 2 topdown & bottomup Parsers.
2. What is left recursion?What is the need of eliminating left recursion in topdown parsing? How it is eliminated?
3. What are the 3 types of LR parsers? Which LR parser have the least number of states?
4. Which type of parser is generated by YACC?
5. What is meant by shift reduce & reduce reduce conflict?
6. What is the purpose of y.tab.h? When it is generated?
   7. What is the difference between parse tree & syntax tree?
   8.What is meant by an ambiguis grammar?

LAB 5,6:
SYMBOL TABLE

PROGRAM 1:
Implement a symbol table in YACC.

Algorithm

Symbol table: A data structure used by a compiler to keep

track of semantics of variables.

- Data type.
- When is used: scope.

  The effective context where a name is valid.
- Where it is stored: storage address.

Writing symbol table in YACC

1. Define Tokens:
UOP (Unary Operator)

COOP
(Conditional Operator)
AROP
(Binary Arithmetic Operators)
IF

ELSE

DO

FOR

WHILE

ASOP(Assignment OPerator)

OPBRACES
(Opening Braces)
CLBRACES
(Closing Braces)
INT

FLOAT

CHAR

DOUBLE

EXIT

START(“Start
” to indicate the starting of the program code)
NUMBER

LABEL
(variables)
SEMI(Semi column)

2. Define 2 dimensional character arrays to store the symbol table fields

char varn[100][10] :Variable Name
char vartype[100][10] :Variable Type)
char varval[100][10] :Variable Value

3. Define symbol table routines:

check**(name):** check whether a name is currently in the symbol table or not.
print Undeclared if not present
display()
: Display the symbol Table entries

****

```
void check(char *ptr)

{

int i,j;

for(i=0;i<=count;i++)

{

j=1;
```

```
if(strcmp(varn[i],ptr)==0)
    {
      j=0;
      break;
    }
}
    if(j==1)
   {
          printf("Undeclared variable %s\n",ptr);
          exit(0);
   }
}
 ************************************************
void display()
{
int i;
printf("SYMBOL TABLE\nVariable Name\tData Type\n");
for(i=0;i<=count;i++)
printf("\n%s\t%s",varn[i],vartype[i]);
}
```

4. Define Yacc Grammar Rules

```
start:
START OPBRACES decls stmts CLBRACES EXIT {printf("Program OK\n");display(); exit(0);};



decls:
  INT {strcpy(vartype[++count],yytext);}  LABEL {strcpy(varn[count],yytext);} eqoptn SEMI decls

   | FLOAT {strcpy(vartype[++count],yytext);}  LABEL {strcpy(varn[count],yytext);} eqoptn SEMI
decls

   |CHAR {strcpy(vartype[++count],yytext);}  LABEL {strcpy(varn[count],yytext);} eqoptn SEMI
decls

   |DOUBLE {strcpy(vartype[++count],yytext);}  LABEL {strcpy(varn[count],yytext);} eqoptn
SEMI decls

      |

      ;


eqoptn:
ASOP optn

       |

       ;


stmts:
stmts1

      |

      ;

stmts1:
LABEL {check(yytext);}ASOP NUMBER SEMI;


optn:
LABEL  {check(yytext);}

      | NUMBER
```

Questions:
1.Which is the most effective method to implement a symbol table?
2. In which phase of the compiler symbol table is used?
3. What are the commonly used fields in symbol table?

LAB 7,8:
Type Checker

PROGRAM 1:
Implement a Type Checker in YACC.

Semantic Rules for Type Checking

P → D;

D → D1, D

D -> D1

D1 → T id addvar(id.value,T.type)

T → char T.type = char

T → integer T.type = integer

T → T1* T.type = pointer(T1.type)

S → id = E if lookup(id).type != E.type print error

S → if E then S1 if E.type != boolean print error

E → id E.type = lookup(id)

E → E1 relop E2 if E1 & E2 boolean E.type = boolean else print error

E → E1 op E2 if (E1.type = E2.type) E.type=E1.type

addvar(value, type) : Add values to the symbol Table
lookup(id) : Search in the symbol Table

.

Questions:

1. What is the difference between static & dynamic type checking?

LAB 9,10:(Generation of intermediate code)

PROGRAM 1:
Generate the three address code for the sentences in the following g
rammar.
E -> E **or** E | E **and** E | **not** E|(E) | id | id **relop** id

Examples:
input: A OR B AND NOT C
output: t1 = not C
t2 = B AND t1
t3 = A OR t2

PROGRAM 2
Generate the three address code for control flow statements in C

Examples:
input: If A < B AND C < D

A = C * 10 + D
output:

t1 = A < B
t2 = C < D
t3 = t1 AND t2
t3 goto true1
goto endif1
true1: t4 = c*10
t5 = t4 * D
A = t5
endif1:

input:
While a<b do
if c < d then x = y + 2
else x = y – 2

output:
0 t1 = A < B
1 t1 goto 3
2 goto 11
3 t2 = c < d
4 t2 goto 8
5 t3 = y-2
6 x = t3
7 goto 10
8 t4 = y + 2
9 x = t4
10 goto 0
11.

Questions:

1. What is meant by triples & quadriples?
2. What are the different types of intermediate representations?
3. What is code optimization?