**VIVA QUESTIONS**

**command instructions in yacc**

**Method for lex**

lex pgm.l

gcc lex.yy.c

./a.out.

If output file name is not provided, the default output file is named a.out

Compiling Yacc programs

yacc file.y -d

lex file.l

gcc lex.yy.c y.tab.c -w

./a.out

**What is yacc?**

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

**what is the structure of a yacc file?**

A yacc file looks much like a lex file:

...definitions...

%%

...rules...

%%

...code...

**what are definitions?**

All code between %{ and %} is copied to the beginning of the resulting C file.

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.

what are rules?

A number of combinations of pattern and action: if the action is more than a single. command it needs to be in braces.

**what is code ?**

This can be very elaborate, but the main ingredient is the call to yylex, the lexical analyser. If the code segment is left out, a default main is used which only calls yylex.

**Explain how Lex and Yacc interact?**

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 your 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. We will now see the way lex returns information in such a way that yacc can use it for parsing.

**Explain 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

**WHAT DOES LEX PARSE DO?**

lex parse can return a symbol that is put on top of the stack, so that yacc can access it. This symbol is returned in the variable yylval. By default, this is defined as an int, so the lex program would have

extern int llval;

%%

[0-9]+ {llval=atoi(yytext); return NUMBER;}

**how to return double values?**

If more than just integers need to be returned, the specifications in the yacc code become more complicated. Suppose we want to return double values, and integer indices in a table. The following three actions are needed.

1. The possible return values need to be stated:

%union {int ival; double dval;}

2. These types need to be connected to the possible return tokens:

%token <ival> INDEX

%token <dval> NUMBER

3. The types of non-terminals need to be given:

%type <dval> expr

%type <dval> mulex

%type <dval> term

The generated .h file will now have

#define INDEX 258

#define NUMBER 259

typedef union {int ival; double dval;} YYSTYPE;

extern YYSTYPE yylval;

**Regular expressions**

Many Unix utilities have regular expressions of some sort, but unfortunately they don’t all have the same power. Here are the basics:

. Match any character except newlines. \n A newline character.

\t A tab character.

ˆ The beginning of the line.

$ The end of the line.

<expr>\* Zero or more occurences of the expression. <expr>+ One or more occurences of the expression. (<expr1>|<expr2>) One expression of another. [<set>] A set of character or ranges, such as [a-zA-Z]. [ˆ<set>] The complement of the set, for instance [ˆ \t].

If the lex program is to be used on its own, this section will contain a main program. If you leave this section empty you will get the default main:

int main() {

yylex();

return 0; }

where yylex is the parser that is built from the rules.