Viva Questions

(Lex, Flex) — FastFlex

Lex – lexical Analyzer
YACC → Parser [syntax Analyzer + Semantic Analyzer]

Q Order of creation of output files
(Lex + YACC)

Lex only:

lex src pgm

① file.l → [Lex Compiler] → lex.yy.c

② lex.yy.c → [C Compiler] → a.out

③ I/p stream → [a.out] → Sequence of tokens

① When we write [lex <filename.l>] command, the lex compiler generates a scanner (Lexical Analyzer) which gets saved as lex.yy.c

② Whenever we write the command [cc lex.yy.c -ll]
C-compiler ←
compile the scanner & grab main from lex library (ll)

③ **./a.out** : Run the scanner & take i/p from console (std input)

Page No. | Date

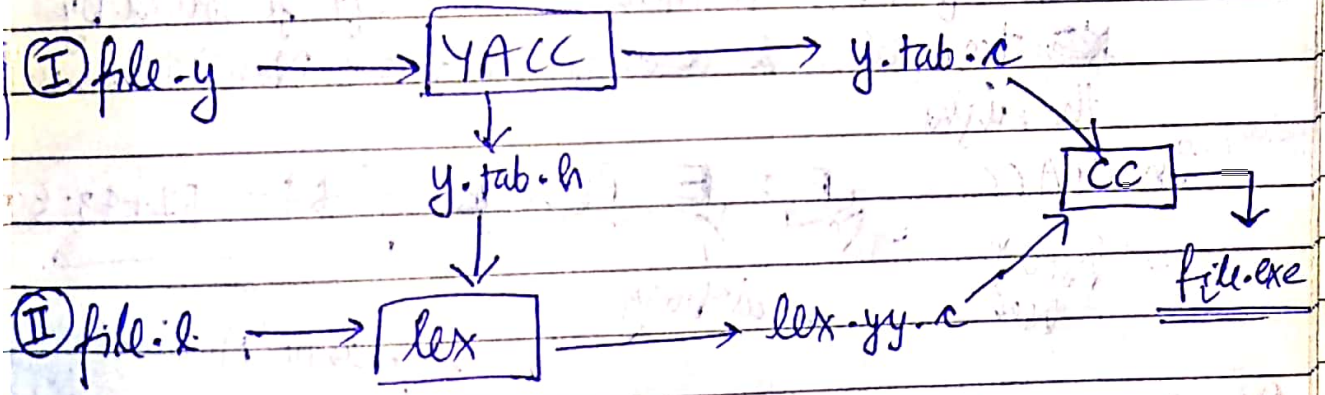
⇒ Input/Output translator using Yacc. (Yet Another Compiler Compiler)

① Yacc specification file.y → [YACC compiler] → y.tab.c

② y.tab.c → [C Compiler] → a.out ★

③ i/p → [a.out] → output.

⇒ Process of building a parser with YACC & lex



CC ⇒ C-compiler.

Note: YACC translates a given CFG (<file.y>) into a C-implementation (y.tab.c) of a corresponding Pushdown Automata (finite state m/c with a stack).

* YACC Production consist of (i) prodⁿ head & prodⁿ body and (ii) action part. This is the SDD we have studied in class for Semantic Analysis. Hence, YACC performs both syntax as well as semantic Analysis

Q Semantic Rule

SDT → Grammar + Semantic Rules

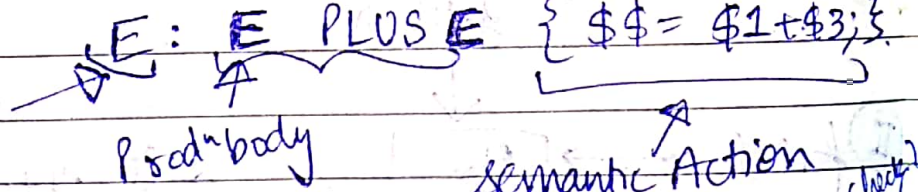
- A SDD specifies the values of attributes by associating semantic rules with the grammar production.
- Every symbol in the grammar's body will have 0 or more attributes associated with it.
- Value of these attributes are evaluated by the semantic rules associated with the grammar.

Q Production.

↳ helps to define a grammar / CFG

↳ consist of Non Terminal at the left followed by terminal(s) & zero or more Non Terminals on the right

In YACC



Q Semantic Action — what action is to be taken when body of a particular prodn's rule is matched with the input string & reduction takes place. The action part consist of one (or more) C-statements enclosed within '{' & '}'

Q What Parser YACC uses internally ? → LALR(1)

Q YACC is Bottom-up or Top-down. Explain.

Ans Bottom-up. ∵ given a string, we are trying to reach the start symbol by following a series of reduce and shift moves until we reach the source symbol (Top). Hence dirn of evaluation is Bottom to up.

Page No. Date

Also YACC can handle left recursive grammar (exclusive feature of Bottom-up Parsing)

Q Types of conflicts in lex & YACC + how are they resolved

* Conflict Resolution in lex
→ Prefer a longer prefix to a shorter prefix
→ If longest prefix matches 2 or more pattern, prefer the pattern listed first in the lex program.

As YACC uses LALR(1) as its parsing method ∴ 2 types of conflicts possible → Shift-Reduce (S-R) & Reduce-Reduce (R-R).

Conflict resolution by YACC
In SR conflict → Shift is preferred
In RR conflict → First reduce is preferred

Q Algo used by lexical Analyser (Read theory)
Ans Regular expression algo

Q Types of Parser
- Top down - Recursive Descent, LL(1) (Non-Recursive Descent)
- Bottom up - operator precedence, LR parsers (LR(0), SLR(1), CLR(1), LALR(1)) ↑ YACC

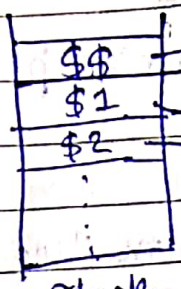
Q Semantic Rule + Expression is called → Syntax Directed Defn (SDD)

#All Questions related to program syntax / reserved words

Here
- %left ⟹ to indicate left associativity for any token
- %right ⟹ " " right " " " " " (also implies token is an operator (+/-/×/÷))

$$ in a semantic action refers to the attribute value associated with the non-terminal in the head of prodn, while $i (i=1,2,...) refers to the value associated with i^{th} grammar symbol of the body. The semantic action is performed whenever a reduction is performed. ∴ semantic action computes a value for $$ in terms of $i's

y.tab.c → C file uses a stack for parsing the expression entered as follows:-



→ Top of stack (TOS)
→ element below TOS
→ element 2 places below TOS
& so on.

Lex

* The lex command uses the rules & ~~defined~~ actions contained in file to generate a program, lex.yy.c which can be compiled (using cc command. This program can then receive the input, break the i/p into logical pieces defined by the rules & run the pgm fragments contained in the actions in file

* lex command stores the yylex() in a file named lex.yy.c

* yylex() analyzes the i/p stream using a pgm structure called Finite state machine. (⇔ Chi done → Transition state diagram for accepting/rejecting a string)

Lex Predefined variables

Name	Use
int yylex(void)	call to invoke lexer, returns tokens
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	(lexeme value) / value associated with token.
int yywrap()	wrapup; return 1 if input exhausted, return 0 if end of i/p not reached.
FILE *yyout; *yyin	output file ; i/p file resp.
INITIAL	initial start condition.
BEGIN	condition switch start condition
ECHO	write matched string.

* When yylex() matches a string in the i/p stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

Note: lex uses fixed names for intermediate & o/p files ⇒ you can have only one lex-generated program in a given directory.

YACC

* yacc -d, <file.y> produces the <y.tab.h> file in the directory. y.tab.h contains the #define statements that associate the yacc assigned token codes with the token names. This allows the source files other than y.tab.c to access the token codes by including this header file.
(We are able to use the %token ——, defined in YACC file, in our lex file because of #include "y.tab.h")

Yacc grammar can be ambiguous; specified precedence rules are used to break ambiguities.

y.tab.c file must be compiled with a C-Compiler to produce a yyparse() function. This function must be loaded with a yylex lexical analyzer function, as well as main() and yyerror(). The lex command is used for creating lexical analyzer usable by yacc.

* %start ↑ eg %start S
symbol Name

It indicates the highest level prodn rule to be reduced, i.e. the rule where parser can consider its work done & terminate.
If this definition is not included, the parser uses the first production rule as the start symbol for the Grammar.

%type defines each symbol as a data type 'type', to resolve ambiguities. If this construct is present, yacc performs type checking & otherwise assumes all symbols to be of type integer.

* The specially defined token error matches any unrecognized sequence of input. This token causes the parser to invoke the yyerror() function.

%union <union-def> : defines the yylval global variable as a union, where <union-def> is a standard C definition in the format

{ type member; [type member; ...] }

At least one member should be an int. Any valid C data type can be defined, including structures. When you run yacc with the -d option, the definition of yylval is placed in the <y.tab.h> file & can be refered to in a lex file.

Note: yylval : if the token returned by yylex function is associated with a significant value, yylex should place the value in this global variable. When we run yacc with -d option, the full yylval defn is parsed into <y.tab.h> file for access by lex.

The following functions (contained in user function section) are invoked within the yyparse function generated by yacc.

(*) yylex(): The lexical analyzer called by yyparse to recognize each token of i/p. This function is created by lex but is internally called by yyparse()

The end of i/p is marked by a special token called end marker that has a token no. that is zero or negative. If the tokens upto, but not excluding, the end marker form a structure that matches the start symbol, the parser accepts the input.

Date | Page No.

If the endmarker is seen in any other context, it is considered an error.

④ yyerror(): the function that parser calls upon encountering an i/p error. The default action simply prints string to the standard error. The user can redefine the function.

EXIT status
= 0 : successful termination
≠ 0 : error

Running Calculator Program (+ Files created)

① `yacc -d calc.y`

Process the yacc grammar file using '-d' option. This option tells yacc to create a file that defines the tokens it uses, in addition to the C-language source code.

Files created after ① → y.tab.c (C language src file that yacc created for Parser)
→ <y.tab.h> (header file containing #define stmts for tokens used by Parser).

② `lex calc.l`

Files created after ② → lex.yy.c (C language src file that lex created for Lexical Analyzer)

③ Compile & link the 2 C language src file (lexer + Parser)

`cc y.tab.c lex.yy.c -ll`

.o files are temporary

Files created after ③ → y.tab.o (object file for y.tab.c)
→ lex.yy.o (object file for lex.yy.c)
→ a.out (executable pgm file)

④ Run the pgm using `./a.out`