chapter 7 (Syllabus) Rahaut (1)

LEX and YACC

Introduction:

Lex and yacc programs helps to transform structured input.

For e.g. in a C program we have different units like variables, constants, strings, operators, punctuation etc. This division into units (or tokens) is known as Lexical Analysis. Lex helps us by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical Analyzer or scanner that can identify those tokens. The set of descriptions we give to Lex is called a Lex specification.

The token descriptions that lex uses are known as Regular Expressions. Lex turns these regular expressions into a form that the lexer can use to scan the input text extremely fast, independent of the number of expressions that it is trying to match.

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. This task is known as Parsing and the list of rules that define the relationships that the program understands is a grammar. Yacc (Yet another Compiler-Compiler) takes a concise description of a grammar and produces a C routine that can parse that grammar, called a Parser (Syntax Analyzer).

The yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a syntax error whenever its input doesn't match any of the rules.

(3) Symbol Tables:

A C compiler stores the variable, structure names, labels (in other words all tokens) in the program in its symbol table. Each name is stored along with information describing the nature. In a C compiler the information is the type of the symbol, declaration scope, variable type etc.

The symbol table can be defined using any data structure. If we use linked list the each node definition may look as follows:

```
struct node
{
    char *token_name;
    int token_type;
    struct node *next;
};
```

— whenever we get a token it is entered into the symbol table along with the token_type.

Grammars:

As the input is divided into tokens, a program often needs to establish the relationship among the tokens. A description of such a set of relationship among the tokens and a set of actions is known as a grammar.

Example:

1. One grammar rule might be,

date : month_name Day ',' year ;

i.e. the rule to read & scan a valid date is monthname followed by space followed by Day followed by , followed by year.

The input October 10, 2004 will be matched by the above rule.

2. Another grammar rule might be,

date : monthnumber '/' Day '/' year ;

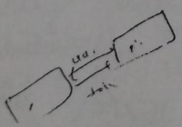
The input 10/10/2004 will be matched by the above rule.

Lex - Yacc communication (Parser - lexer communication)

When we use a lex scanner and a yacc parser together, the parser is the higher level routine. It calls the lexer yylex() whenever it needs a token from the i/p. The lexer then scans through the input recognizing tokens. As soon as it finds a token of interest to the parser, it returns to the parser, returning the token's code as the value of yylex().

Not all tokens are of interest to the parser - in most programming languages, the parser doesn't want to hear about comments and white-spaces, for example. For these ignored tokens, the lexer doesn't return so that it can continue on to the next token without bothering the parser.

The lexer and the parser have to agree what the token codes are. Token code zero is returned for the logical end of the input.

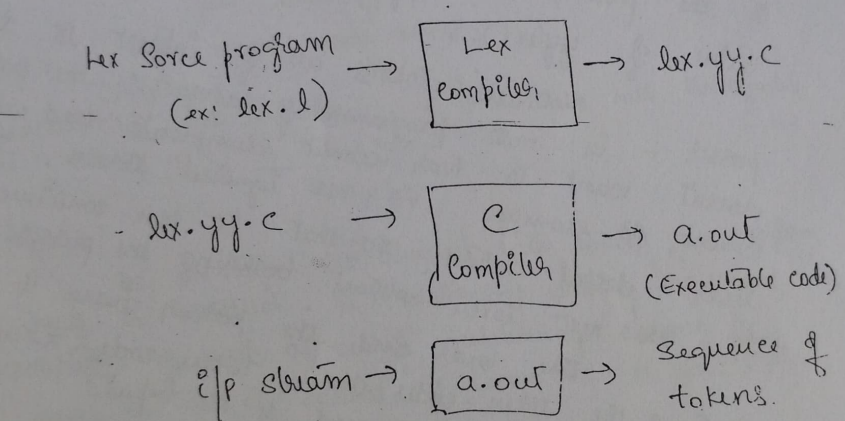


(4)

USING LEX

Lex is a tool for building lexical analyzer & lexers. A lexer takes an arbitrary input string and tokenizes it. i.e. divides it up into lexical tokens. This tokenized output can then be processed further, usually by yacc & it can be the "end product."

Lex turns the user's expressions and actions into the Host-general purpose language (usually C language). Creating a lexical analyzer with lex is shown in following steps.



First, a specification of a lexical analyzer is prepared by creating a program lex.l in the lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c. The program lex.yy.c consists of a tabular representation of a transition diagram constructed from the regular expressions of lex.l, together with a standard routine that uses the table to recognize lexemes. The actions associated with regular expressions in

lex.l are pieces of C code and are carried over directly to lex.yy.c

Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an i/p stream into a sequence of tokens.

LEX Specifications:

A lex program consists of 3 parts:

```
%{
Declarations (Definition section)
%}
%%
Translation rules
%%
Auxiliary procedures
```

The definition section include declarations of variables, constants and regular definitions.

We surround the C code with the special delimiters "%{" and "%}". Lex copies the material between "%{" and "%}" directly to the generated C file, so we may write any valid C code here.

The next section is the rules section. Each rule is made up of two parts: a pattern and an action, separated by whitespace. The lex will execute the action when it recognizes the pattern.

The final section is the auxiliary procedures section, which can consist of any legal C code. Lex copies it to the C file after the end of the lex generated code.

NOTE: To run a lex pgm;

1. lex a.l ↵
2. cc lex.yy.c -ll ⟼ link lex library

(6) Lex Regular Expressions:

A regular expression is a pattern description using a "meta" language, a language that use to describe particular patterns of interest. The characters that form regular expressions are:

. - Matches any single character except the new line character \n.

* - Matches zero or more copies of the preceding expression.

+ - Matches one or more occurrences of the preceding regular expression.

Ex: [0-9]+ matches "1", "123", or "123456" but not an empty string.

[ ] - A character class which matches any character within the brackets. If the first character is a circumflex ("^") it changes the meaning to match any character except the ones within the brackets)

A dash "-" inside the square brackets indicates a character range.

ex: [0-9] means [0123456789].

If we want to include the character "-" in a character class, it should be first or last.

ex: [-+0-9] matches all the digits and "-" and "+".

$ - matches the end of a line as the last character of a regular expression.

{ } - Indicates how many times the previous pattern is allowed to match when containing one or two numbers.

For ex: A{1,3} matches one to three occurences of the letter A.

? - Optional operator matches zero or one occurence of the preceding regular expression

For ex: ① ab?c matches either ac or abc

② [+]?[0-9]+ matches a +ve no including an optional leading +. +50 and 50 matches to the above regular expression.

| - matches either the preceding regular expression or the following regular expression

Ex: "cow"|"sheep"|"cat" matches any of the 3 words.

/ - matches the preceding regular expression but only if followed by the following regular expression.

For ex: 0/1 matches "0" in the string "01" but would not match anything in the strings "0" or "02". The material matched by the pattern following the slash is not "consumed" and remains to be tested by subsequent tokens. Only one slash is permitted

(8) Lex Regular Expressions:

(6) ( ) - Groups a series of regular exp. Together into a new regular expressi
For ex: (01) represents the chara sequence 01. Parantheses are usef when building up complex patterns with *, + and |.

Examples of lex regular expressions:

[-][0-9]+ - regular expression for -ve integs.
[+]?[0-9]+ - " " for +ve integs.
[+]?[0-9]*\.[0-9]+ - " " for +ve fraction.

Notice that "\" before "." to make it separate from a wild card characters '.'.

[a-zA-Z][a-zA-Z0-9]* - is a typical regular expression for identifying identifiers in programming languages.

8

Lex Actions:

When an expression written as above is matched, Lex executes the corresponding action. Note that there is a default action, which consists of copying the input to the output. If the file pointer yyin is not initialized then it reads from keyboard else from specified file. Illy if the file pointer yyout is not initialized then it writes to monitor else to specified file.

Usually lex leaves the text that matched some regular expression, in an external character array named yytext.

Ex: [a-z]+ { printf("%s", yytext); }

will print the string in yytext.
This action is so common that it can be written as ECHO.

i.e. [a-z]+ { ECHO; } is the same as above.

Lex provides a counter variable yyleng which is the no. of characters matched.

Ex: [a-zA-Z]+ { words++; chars = chars + yyleng; }

which counts both the no. of words and the no of characters in the words recognized.

Example program:

The following lex program counts the no. of characters, words and newlines in a stream of i/p characters string.

```
%{
    int c=0, w=0, l=0;
%}
WORD    [^ \t\n]+
EOL     [\n]
%%
{WORD}      { w++; c = c+yyleng; }
{EOL}       { c++; l++; }
.           { c++; }
```

(10)

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

main()
{
   printf("Enter the string upto ^d\n");
   yylex();
   printf("No. of chars = %d\n No. of words = %d\n No. of
   new lines = %d\n", c, w, l);
}
```

The section bracketed by "%{" and "%}" is C code which is copied directly to C code. The next 2 lines are definitions. Lex provides a simple substitution mechanism to make it easier to define long & complex patterns.

The next section is rules and action part. if ~~more~~ more than one action is there, then the action part should be enclosed between { & }.

The main program first calls yylex() which is the lexer's entry point. When yylex() reaches the end of its input file, it calls yywrap(), which returns a value of 0 or 1. If the value is 1, the program is done and there is no more input. If the value is 0, on the other hand, the lexer assumes that yywrap() has opened another file for it to read, and continues to read from yyin. The default yywrap() always

returns 1. By providing our own version of yywrap(), we can have our program read all of the files named on the command line, one at a time.

The lex provides the routines input() and unput()

The input() routine handles calls from the lexer to obtain characters. When the current argument is exhausted it moves to the next argument, if there is one, and continues scanning. If there are no more arguments, we treat it as the lexer's end-of-file condition and return a zero byte.

The unput() routine handles calls from the lexer to "push back" characters into the input stream. It does this by reversing the pointers direction, moving backwards in the string.

[Note: For example refer variable count, tab program]

Start states:

Start-state is a method of capturing context sensitive information within the lexer. Tagging rules with start states tells the lexer only to recognize the rules when the start state is in effect. The start state is defined in the definition section using %S followed by state name. In the rules section we can use these state name which is enclosed between < and >. These rules are only

recognized when the lexer is in state stated in the definition section.

We enter a new state with a BEGIN statement. To change back to the default state, we use "BEGIN 0". (The default state zero, is also known as INITIAL.)

Ex:

To count the no. of comment lines in a C pgm.

```
%{
    int com=0;
%}
%s COMMENT
%%
[ \t]* "/*"     { BEGIN COMMENT; }
<COMMENT>\n     { com++; }
<COMMENT>"*/"   { BEGIN 0; com++; }
<COMMENT>.      { ; }
%%
main()
{
    yylex();
    printf("no. of comments = %d", com);
}
```

Roshan Fernandes

(Yet Another Compiler Compiler)

80 copies

USING YACC

Lex divides the input stream into pieces (tokens) and then Yacc takes these pieces and groups them together logically.

Grammars: Yacc takes a grammar that we specify and writes a parser that recognizes valid "sentences" in that grammar. A grammar is a series of rules that the parser uses to recognize syntactically valid input.

Consider the following grammar;

statement → NAME = expression
expression → NUMBER + NUMBER | NUMBER - NUMBER

The vertical bar "|" means there are 2 possibilities for the same symbol, i.e. an expression can be either an addition or a subtraction.

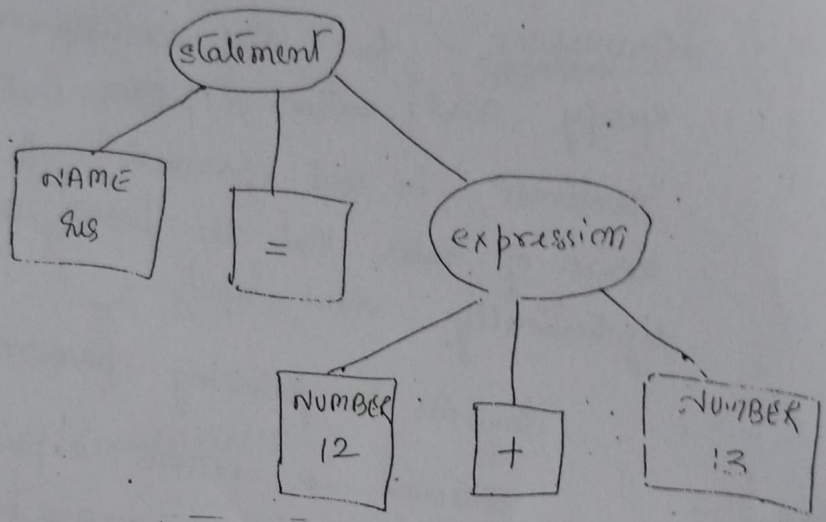
The symbol to the left of the '→' is known as the Left-Hand side of the rule (LHS) and the symbols to the right are Right-Hand side of the rule (RHS).

Symbols that actually appear in the input and are returned by the lexer are terminal symbols or tokens. Symbols that appear on the left hand side of the rules are Non-terminal symbols or non terminals.

Usually Terminal Symbols are given in uppercase letters and Non Terminals are given in lowercase letters

The usual way to represent a parsed sentence is as a tree.

For example, if we parsed the input "res = 12 + 13" with the above grammar, the tree would look like below.



"12 + 13" is an expression, and "res = expression" is a statement.

Note: A Pace parser doesn't actually create this tree as a data structure.

Every grammar includes a start symbol, the one that has to be at the root of the parse tree.

In the above grammar, Statement is the start symbol.

Recursive rules:

Rules can refer directly or indirectly to themselves. This important ability makes it possible to parse arbitrarily long input sequences.

Example:

expression → NUMBER
| expression + NUMBER
| expression − NUMBER

Now we can parse a sequence like "res = 14 + 23 − 11 + 7" by applying the expression rules repeatedly.

Shift/Reduce Parsing:

A Yacc parser uses Shift/Reduce Parsing technique.

As the parser reads tokens, each time it reads a token that doesn't complete a rule, it pushes the token on an internal stack and switches to a new state reflecting the token it just read. This action is called **shift**. When it has found all the symbols that constitute the right-hand side of a rule, it pops the right-hand side symbols off the stack, pushes the left-hand side symbol onto the stack and switches to a new state reflecting the new symbol on the stack. This action is called **Reduction**, since it reduces the number of items on the stack. Whenever Yacc reduces a rule, it executes user code associated with the rule.

Let us look how it parses the input "res = 12 + 13" using the simple rules,

statement → NAME = expression
expression → NUMBER + NUMBER | NUMBER −

The parser starts by shifting tokens on to internal stack one at a time:

res

res =

res = 12

res = 12 +

res = 12 + 13

At this point it can reduce the rule "expression → NUMBER + NUMBER" so it pops 13, +, and 12 from the stack and replaces them with expression.

i.e. res = expression.

Now it reduces the rule "statement ⇒ NAME = expression", so it pops expression, = and res from the stack and replaces them with statement. We have reached the end of the input and the stack has been reduced to the start symbol, so the input was valid according to the grammar.

What Yacc cannot Parse;

Yacc cannot handle ambiguous grammars, ones in which the same input can match more than one parse tree. [But yacc can deal with a limited but useful set of ambiguous grammars.]

Yacc also cannot deal with grammars that need more than one token of lookahead to tell whether it has matched a rule.

Example:

Consider the following grammar:

```
phrase → cart_animal AND CART
       | work_animal , AND PLOW

cart_animal → HORSE | GOAT

work_animal → HORSE | OX
```

Yacc cannot handle it because of requiring two symbols of lookahead: In particular, in the input "HORSE AND CART" it cannot tell whether HORSE is a cart_animal or a work_animal until it sees CART, and Yacc cannot look that far ahead.

If we changed the first rule to this:

```
phrase → cart_animal CART
       | work_animal PLOW
```

Yacc would have no problem, since it can look one token ahead to see whether an input of HORSE is followed by CART in this case the HORSE is a cart_animal; if it is followed by PLOW, then the horse is a work_animal.

A Yacc Parser:

A Yacc grammar has a 3-part structure as a lex specification. The 3 sections are Definition section, rules section and programs | auxiliary procedures section.

sections is separated by %%.

(18) Definition section:

It can include a declaration part enclosed between %{ and %}.

Tokens & the terminal symbols have to be defined using %token.

Ex: %token NAME NUMBER

We can use single quoted characters as tokens without declaring them.

Example: '=', '+', etc.

The Rules Section:

The rules section consists of a list of grammar rules. In rules we use a ":" (colon) between the left- and right-hand sides of a rule. and we put a semicolon at the end of each rule:

```
Ex:   %token NAME NUMBER
      %%
      statement: NAME '=' expression
               | expression
               ;
      expression: NUMBER '+' NUMBER
                | NUMBER '-' NUMBER
                ;
```

The symbol on the left hand side of the first rule in the grammar is normally the start symbol.

We can use %start declaration to specify a starting symbol.

-10-

Symbol Values and Actions:

Every symbol in a yacc parser has a value. The value gives additional information about a particular instance of a symbol. If a symbol represents a number, the value would be the particular number. If it represents a literal text string, the value would probably be a pointer to a copy of the string. If it represents a variable in a program, the value would be a pointer to a symbol table entry describing the variable. Some tokens don't have a useful value, ex: a token representing a close parenthesis, since one close parenthesis is the same as another.

Whenever the parser reduces a rule, it executes user C code associated with the rule, known as the rule's action. The action appears in braces after the end of each rule, before the semicolon or vertical bar.

The action code can refer to the values of the right-hand side symbols as $1, $2, ... and can set the value of the left-hand side by setting $$.

Example:

```
%token NAME NUMBER;
%%
statement : NAME '=' expression
          | expression { printf("%d\n", $1); }
          ;
expression : expression '+' NUMBER { $$ = $1 + $3;
           | expression '-' NUMBER { $$ = $1 - $3; }
```

| NUMBER { $$ = $1; }
;

the expression building rules. the first and second numbers' values are $1 and $3 respectively. the operator's value would be $2.

The Lexer;

The parser is a higher level routine and calls the lexer yylex() whenever it needs a token from the input. As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value. Yacc defines the token names in the parser as C preprocessor names in y.tab.h

Example, Here is a simple lexer to provide tokens for our parser;

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t]       ; /* ignore white space */
\n          { return 0; } /* end of input */
.           { return yytext[0]; }
%%
```

Whenever the lexer returns a token to the parser, if the token has an associated value,

the lexer must store the value in yylval before returning the token.

Compiling and running a simple parser:

On a UNIX system, yacc takes our grammar and creates y.tab.c, the C language parser and y.tab.h, the include file with the token numbers definitions. Lex creates lex.yy.c, the C language lexer. We need only compile them together with the yacc & lex libraries.

The main() program calls the parser, yyparse() and exits.

The different steps are given below:

```
yacc -d pgm1.y      → makes y.tab.c & y.tab.h
lex pgm1.l          → makes lex.yy.c
cc y.tab.c lex.yy.c -ly -ll
                    → compile & link
```

```
./a.out ↵
10+50 ↵
60
./a.out ↵
100+-50 ↵
syntax error.
```

It correctly reports a message "syntax error" when we enter something that doesn't confirm to the grammar.

We can also call a routine yyerror()

Ex: if ($2 == 0)
yyerror("Divide by zero");

Ambiguity and Conflicts:

In any arithmetic expression grammar, operators are grouped into levels of precedence from lowest to highest.

If there are operators of same precedence then associativity has to be specified.

Example: a-b-c can be (a-b)-c or a-(b-c). So the operator can be grouped as left associative and right associative

left associative example → (a-b)-c

right associative example → a=(b=c)

The precedences and associativity are attached to tokens in the declaration section. This is done by a series of lines begining with a yacc keyword %left, %right or %nonassoc, followed by a list of tokens. The keyword %left is used to describe left associative operators, %right is used to describe right associative operators and %nonassoc is used to describe operators that may not associate with themselves.

NOTE: All the tokens on the same line are assumed to have the same precedence level and associativity. The lines are listed in order of increasing precedence or binding strength.