Lex & Yacc

Prepared by:
Taha Malampattiwala 140050107052

Outline

- Lex :
  - Theory
  - Execution
  - Example
- Yacc :
  - Theory
  - Description
  - Example
- Lex & Yacc linking

Lex

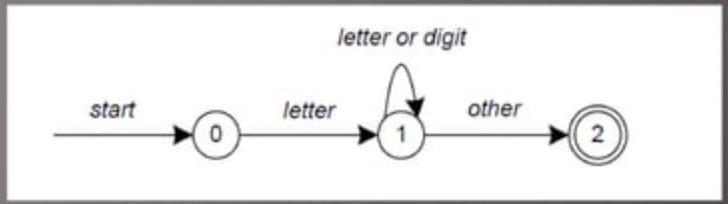
- lex is a program (generator) that generates lexical analyzers, (widely used on Unix).
- It is mostly used with Yacc parser generator.
- Written by Eric Schmidt and Mike Lesk.
- It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions); then produces C code for a lexical analyzer that scans for identifiers.

Lex

- A simple pattern: **letter(letter|digit)***

- Regular expressions are translated by lex to a computer program that mimics an FSA.

- This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.



Lex


```
start:   goto state0

state0:  read c
         if c = letter goto state1
         goto state0

state1:  read c
         if c = letter goto state1
         if c = digit goto state1
         goto state2

state2:  accept string
```

- Some limitations, Lex cannot be used to recognize nested structures such as parentheses, since it only has states and transitions between states.
- So, Lex is good at pattern matching, while Yacc is for more challenging tasks.

Lex

- The input structure to Lex

……..Definitions section……

%%

……Rules section……..

%%

…….C code section (subroutines)……..

Operators

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- If they are to be used as text characters, an escape should be used

  \$ = "$"

  \\ = "\"

- Every character but *blank*, *tab* (\t), *newline* (\n) and the list above is always a text character

Precedence of Operators

- Level of precedence
  - Kleene closure (*), ?, +
  - concatenation
  - alternation (|)
- All operators are left associative.
- Ex: `a*b|cd*` = `((a*)b)|(c(d*))`

Lex

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

- Pattern Matching Primitives

Lex

Expression	Matches
`abc`	`abc`
`abc*`	`ab abc abcc abccc ...`
`abc+`	`abc abcc abccc ...`
`a(bc)+`	`abc abcbc abcbcbc ...`
`a(bc)?`	`a abc`
`[abc]`	one of: `a`, `b`, `c`
`[a-z]`	any letter, a-z
`[a\-z]`	one of: `a`, `-`, `z`
`[-az]`	one of: `-`, `a`, `z`
`[A-Za-z0-9]+`	one or more alphanumeric characters
`[ \t\n]+`	whitespace
`[^ab]`	anything except: `a`, `b`
`[a^b]`	one of: `a`, `^`, `b`
`[a|b]`	one of: `a`, `|`, `b`
`a|b`	one of: `a`, `b`

- Pattern Matching examples.

Lex

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

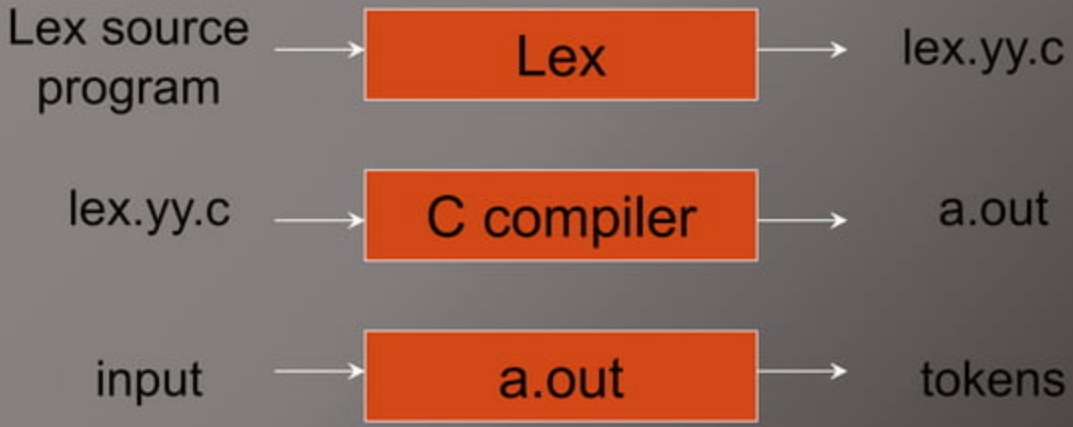
- Lex predefined variables.

Lex


```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*        count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

- Whitespace must separate the defining term and the associated expression.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with "%{" and "%}" markers.
- substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.

An Overview of Lex



Usage

- To run Lex on a source file, type
  ```
  lex scanner.l
  ```
- It produces a file named lex.yy.c which is a C program for the lexical analyzer.
- To compile lex.yy.c, type
  ```
  cc lex.yy.c -ll
  ```
- To run the lexical analyzer program, type
  ```
  ./a.out < inputfile
  ```

Any Question So Far?

Yacc - **Yet Another Compiler-Compiler**

- Yacc reads the grammar and generate C code for a parser .
- Grammars written in Backus Naur Form (BNF) .
- BNF grammar used to express *context-free languages* .
- e.g. to parse an expression, do reverse operation(reducing the expression)
- This known as *bottom-up or shift-reduce parsing* .
- *Using* stack for storing (LIFO).

Yacc

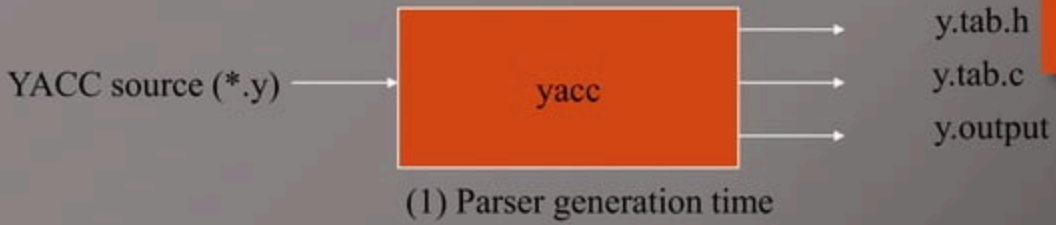
- Input to yacc is divided into three sections.

...definitions...
%%
...rules...
%%
... subroutines...

Yacc

- **The definitions section consists of :**
  - token declarations .
  - C code bracketed by “%{“ and “%}”.
- **the rules section consists of** :
  - BNF grammar.
- **the subroutines section** consists of :
  - user subroutines.

An Overview of YACC



Lex v.s. Yacc

- Lex
  - Lex generates C code for a lexical analyzer, or scanner
  - Lex uses patterns that match strings in the input and converts the strings to tokens
- Yacc
  - Yacc generates C code for syntax analyzer, or parser.
  - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

Yacc & Lex in Together

- **The grammar:**

  program -> program expr | ε

  expr -> expr + expr | expr - expr | id

- **Program and expr are nonterminals.**
- **Id are terminals** (tokens returned by lex).

- **expression may be :**
  - sum of two expressions .
  - product of two expressions .
  - Or an identifiers

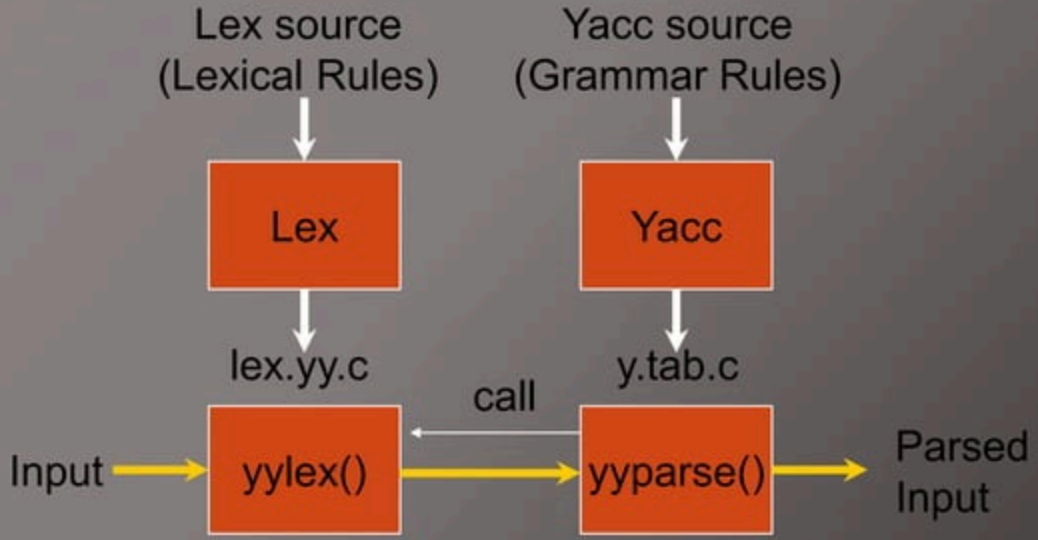
Lex file


```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+       {
                 yylval = atoi(yytext);
                 return INTEGER;
             }

[-+\n]       return *yytext;

[ \t]        ; /* skip whitespace */

.            yyerror("invalid character");

%%

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

Yacc file


```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program:
        program expr '\n'          { printf("%d\n", $2); }
        |
        ;

expr:
        INTEGER                    { $$ = $1; }
        | expr '+' expr            { $$ = $1 + $3; }
        | expr '-' expr            { $$ = $1 - $3; }
        ;

%%

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

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

Linking lex & yacc



Thank You