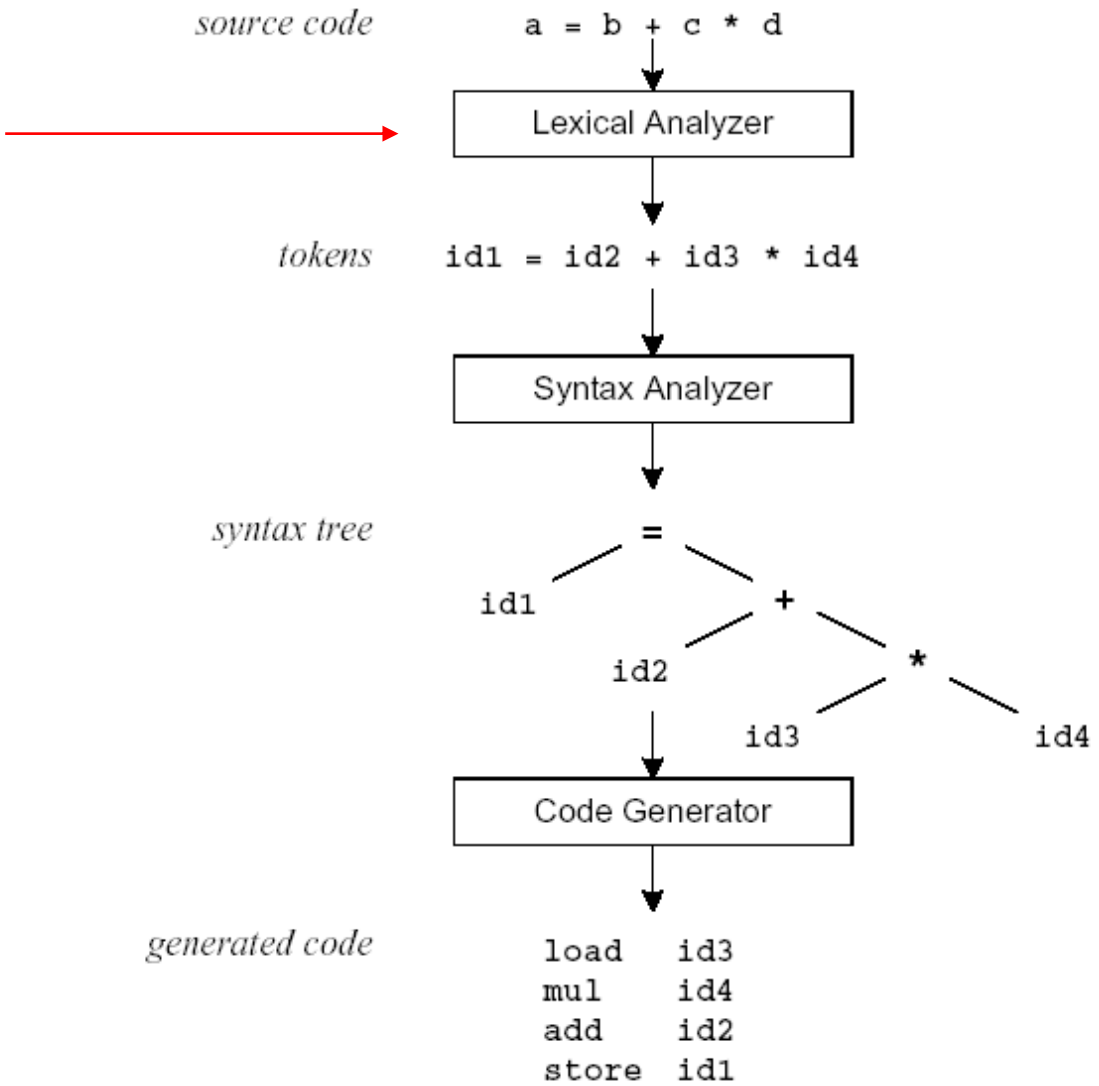
Lex tutorial

Compilation Sequence



What is Lex?

- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

```
a = b + c * d;
```

ID ASSIGN ID PLUS ID MULT ID SEMI

- Lex is an utility to help you rapidly generate your scanners

Why a Tool?

- Starting from scratch is difficult
- Use by defining patterns

Standard tools

- LEX
- FLEX
- JLEX

Lex Source Program

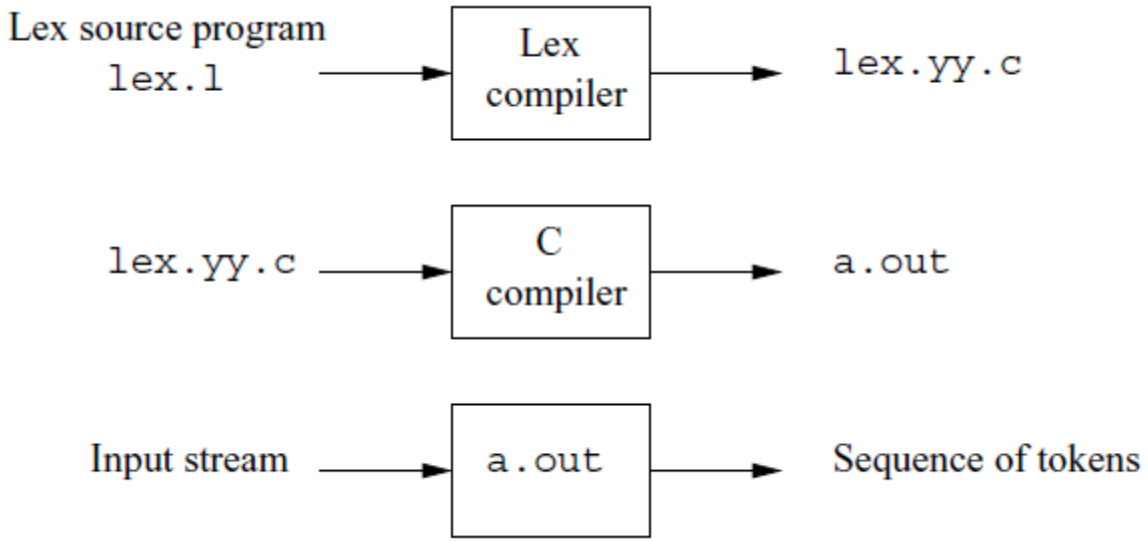
- Lex source is a table of
  - regular expressions and
  - corresponding program fragments

```
digit  [0-9]
letter [a-zA-Z]
%%
{letter}({letter}|{digit})*       printf("id: %s\n", yytext);
\n                                printf("new line\n");
%%
main() {
       yylex();
}
```

Lex Source to C Program

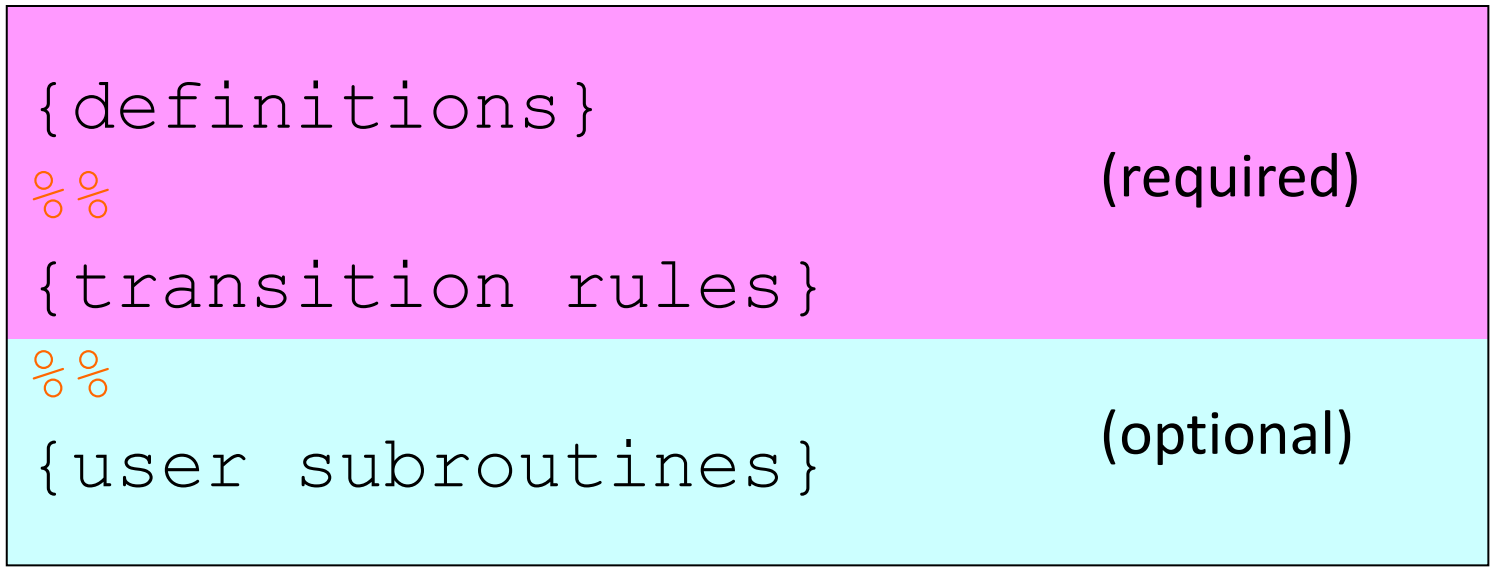
- The table is translated to a C program (lex.yy.c) which
  - reads an input stream
  - partitioning the input into strings which match the given expressions and
  - copying it to an output stream if necessary

An Overview of Lex



Lex Source

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is



- The absolute minimum Lex program is thus

```
%%
```

Regular Expressions

Lex Regular Expressions (Extended Regular Expressions)

- A regular expression matches a set of strings
- Regular expression
  - Operators
  - Character classes
  - Arbitrary character
  - Optional expressions
  - Alternation and grouping
  - Context sensitivity
  - Repetitions and definitions

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

Character Classes [ ]

- `[abc]` matches a single character, which may be `a`, `b`, or `c`
- Every operator meaning is ignored except `\` `-` and `^`
- e.g.

  `[ab]` => `a` or `b`

  `[a-z]` => `a` or `b` or `c` or … or `z`

  `[-+0-9]` => all the digits and the two signs

  `[^a-zA-Z]` => any character which is not a letter

Arbitrary Character

- To match almost character, the operator character `.` is the class of all characters except newline

- `[\40-\176]` matches all printable characters in the ASCII character set, from octal 40 (blank) to octal 176 (tilde~)

Optional & Repeated Expressions

- `a?` => zero or one instance of `a`
- `a*` => zero or more instances of `a`
- `a+` => one or more instances of `a`

- E.g.

  `ab?c` => `ac` or `abc`

  `[a-z]+` => all strings of lower case letters

  `[a-zA-Z][a-zA-Z0-9]*` => all alphanumeric strings with a leading alphabetic character

Precedence of Operators

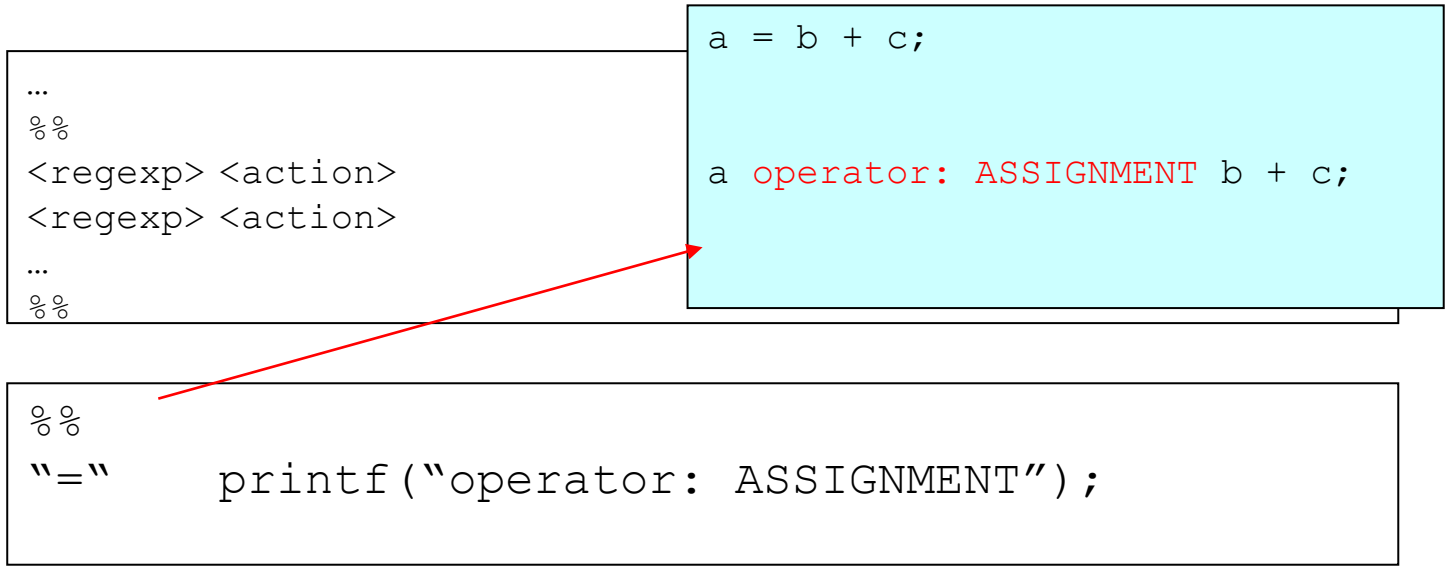
- Level of precedence
  - Kleene closure (*) , ?, +
  - concatenation
  - alternation (|)
- All operators are left associative.
- **Ex:** `a*b|cd*` = `((a*)b)|(c(d*))`

Pattern Matching Primitives

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 / complement
`$`	end of line
`a|b`	`a` or `b`
`(ab)+`	one or more copies of `ab` (grouping)
`[ab]`	`a` or `b`
`a{3}`	3 instances of `a`
`"a+b"`	literal "`a+b`" (C escapes still work)

Recall: Lex Source

- Lex source is a table of
  - regular expressions and
  - corresponding program fragments (actions)



```
/* regular definitions */
delim      [ \t\n]
ws         {delim}+
letter     [A-Za-z]
digit      [0-9]
id         {letter}({letter}|{digit})*
number     {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Transition Rules

- regexp <one or more blanks> action (C code);
- regexp <one or more blanks> { actions (C code) }

- A null statement ; will ignore the input (no actions)

```
[ \t\n] ;
```

  - Causes the three spacing characters to be ignored

```
a = b + c;
d = b * c;

↓ ↓

a=b+c;d=b*c;
```

```
{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
```

Transition Rules (cont'd)

- Four special options for actions:
  |, ECHO;, BEGIN, and REJECT;
- | indicates that the action for this rule is from the action for the next rule
  - `[ \t\n]    ;`
  - `" "        |`
    `"\t"       |`
    `"\n"       ;`
- The unmatched token is using a default action that ECHO from the input to the output

Transition Rules (cont'd)

- REJECT
  - Go do the next alternative

```
…
%%
pink        {npink++; REJECT;}
ink         {nink++; REJECT;}
pin         {npin++; REJECT;}
. |
\n          ;
%%
…
```

Lex Predefined Variables

- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - the default output of default main() is **stdout**.
- **`./a.out < inputfile > outfile`**

- E.g.

```
[a-z]+          printf("%s", yytext);
[a-z]+          ECHO;
[a-zA-Z]+       {words++; chars += yyleng;}
```

Lex Library Routines

- yylex()
  - The default main() contains a call of yylex()
- yymore()
  - return the next token
- yyless(n)
  - retain the first n characters in yytext
- yywarp()
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1

Review of Lex Predefined Variables

Name	Function
`char *yytext`	pointer to matched string
`int yyleng`	length of matched string
`FILE *yyin`	input stream pointer
`FILE *yyout`	output stream pointer
`int yylex(void)`	call to invoke lexer, returns token
`char* yymore(void)`	return the next token
`int yyless(int n)`	retain the first n characters in yytext
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`ECHO`	write matched string
`REJECT`	go to the next alternative rule
`INITAL`	initial start condition
`BEGIN`	condition switch start condition

User Subroutines Section

- You can use your Lex routines in the same ways you use routines in other programming languages.

```
%{
   void foo();
%}
letter       [a-zA-Z]
%%
{letter}+    foo();
%%
…
void foo() {
   …
}
```

```
int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

User Subroutines Section (cont'd)

- The section where main() is placed

```
%{
   int counter = 0;
%}
letter [a-zA-Z]

%%
{letter}+     {printf("a word\n"); counter++;}

%%
main() {
   yylex();
   printf("There are total %d words\n", counter);
}
```

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`

Versions of Lex

- AT&T -- lex
  http://www.combo.org/lex_yacc_page/lex.html

- Lex on different machines is not created equal.

Example

```
int num_lines = 0;
%%
\n      ++num_lines;
.       ;
%%
main()
{   yylex();
  printf( "# of lines = %d\n", num_lines);  }
```

Example

```
%{int s=0,c=0,l=0;%}
%%
[ \t] {s++;}
[a-zA-Z0-9] {c++;}
[\n] {l++;}
EOF {printf("\n\t\t Characters = %d \n\n\t Words = %d Lines =%d",c,s,l);exit(0);}
%%
int main(int argc , char *argv[])
{system("clear");
yyin=fopen(argv[1],"r");       //printf("Enter the String=\n");
yylex();
printf("\n\t\t Characters = %d \n\n\t Words = %d Lines =%d",c,s,l);
fclose(yyin);
 }
```