Lecture 4

Lexical Analysis

Course Leader:

K S Suvidha

Objectives

At the end of this lecture, the student will be able to answer,

- What is lex?
- Source Program
- Operators
- Regular Expression

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

Lex – Lexical Analyzer

- Lexical analyzers **tokenize** input streams
- Tokens are the **terminals** of a language
  - English
    - words, punctuation marks, …
  - Programming language
    - Identifiers, operators, keywords, …
- Regular expressions define **terminals/tokens**

Lex Source Program

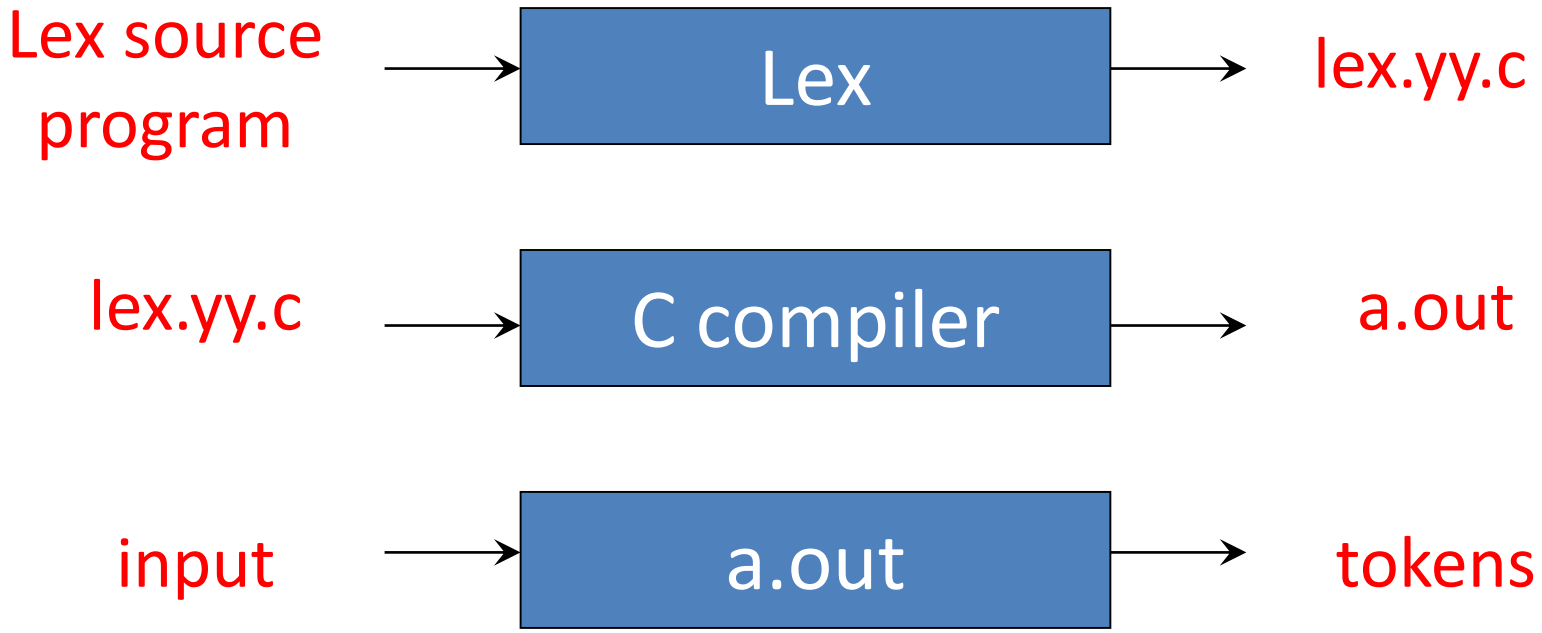
- Lex source is a table of
  - regular expressions and
  - corresponding program fragments

```
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

```
{definitions}
%%                          (required)
{transition rules}
%%
                            (optional)
{user subroutines}
```

- The absolute minimum Lex program is thus

```
%%
```

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

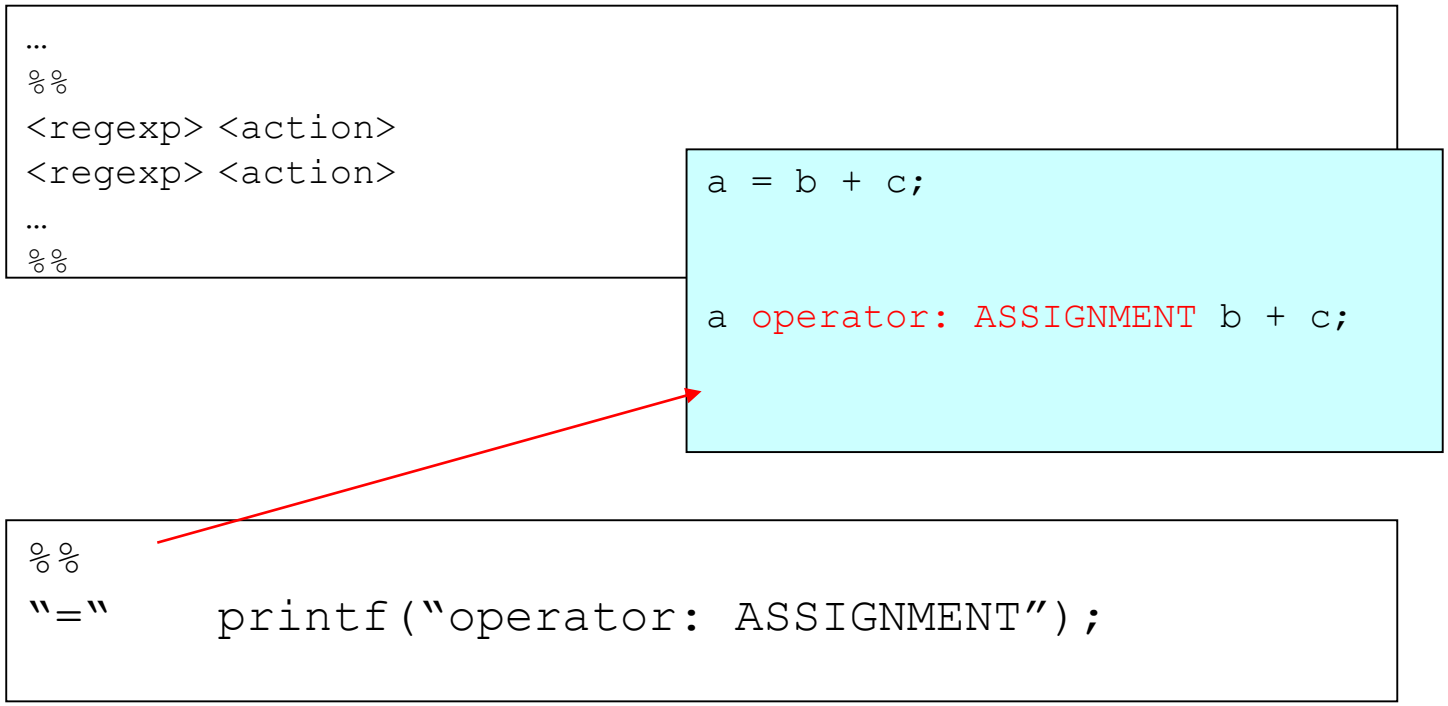
  [a-zA-Z][a-zA-Z0-9]* => all alphanumeric strings with a leading alphabetic character

Pattern Matching Primitives

Meta character	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)



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;
```

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**.
- **cs20: %./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() {
   …
}
```

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);
}
```

Summary

- Lexical Analyzer Generator – Lex
- Working of lex
- You can use your Lex routines in the same ways you use routines in other programming languages