Lex
a tool for Lexical Analysis

CS 315 – Programming Languages

Pinar Duygulu

Bilkent University

Lexical and syntactic analysis

Lexical analyzer: scans the input stream and converts sequences of characters into tokens.
Token: a classification of groups of characters.

Lexeme	Token
Sum	ID
for	FOR
:=	ASSIGN_OP
=	EQUAL_OP
57	INTEGER_CONST
*	MULT_OP
,	COMMA
(	LEFT_PAREN

Lex is a tool for writing lexical analyzers.

Syntactic Analysis (Parsing):
Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

Yacc is a tool for constructing parsers.

Introduction

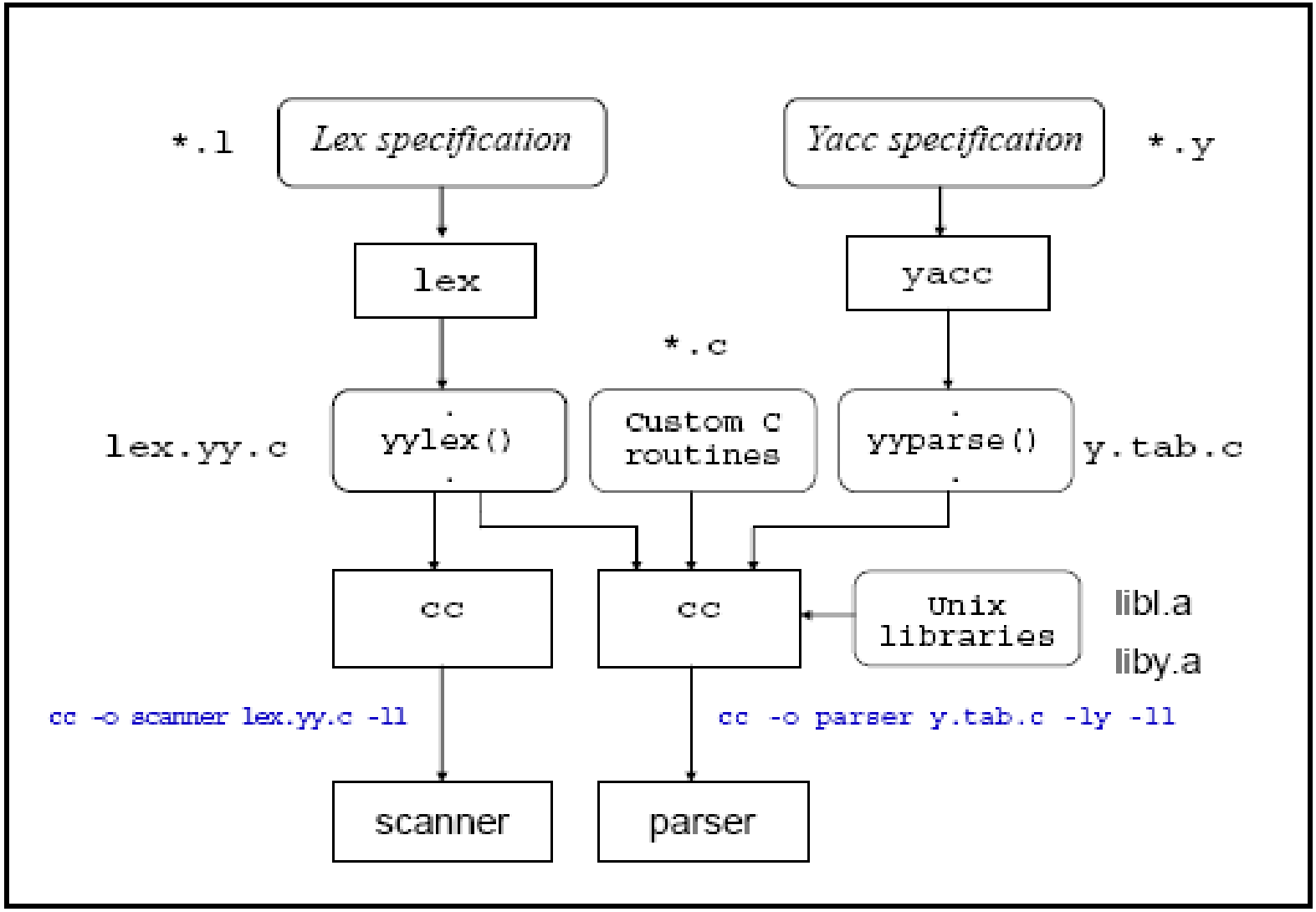
Lex

- reads in a collection of regular expressions, and uses it to write a C or C++ program that will perform lexical analysis. This program is almost always *faster* than one you can write by hand.

Yacc

- reads in the output from lex and parses it using its own set of regular expression rules. This is almost always *slower* than a hand written parser, but *much faster* to implement. Yacc stands for "Yet Another Compiler Compiler".

Using lex and yacc tools



Running lex

On knuth.ug.bcc.bilkent.edu.tr type

```
lex mylex.l
```

it will create `lex.yy.c`

then type

```
gcc -o mylex lex.yy.c -lfl
```

The freeware versions of lex is called “flex”

Lex

In lex, you provide a set of regular expressions and the action that should be taken with each.

Contents of a lex program :

- Definitions
- %%
- Regular expressions and associated actions (rules)
- %%
- User routines

Contents of a lex specification file

Definitions
%%
Regular expressions and associated actions (rules)
%%
User routines

The Simplest lex Program (ex0.l)

```
%%
%%
```

alternatively

```
%%
.|\n   ECHO;
%%
```

What does it do?

Important note: Do not leave extra spaces and/or empty lines at the end of the lex specification file.

The Simplest lex Program

($ is the unix prompt)

```
$emacs ex0.l
```

(alternatively vi ex1.1 or use your favorite editor)

```
$ls
ex0.l
$cat ex0.l
%%
.|\n    ECHO;
$lex ex0.l
$ls
ex0.l lex.yy.c
$gcc -o ex0 lex.yy.c -lfl
$ls
ex0 ex0.l lex.yy.c
```

The Simplest lex Program (ex0.l)

```
$vi test0
$cat test0
ali
veli

$ cat test0 | ex0    (or $ex0 < test0)
ali
veli
```

Another lex program (ex1.l)

```
%%
zippy printf("I RECOGNIZED ZIPPY");

$cat test1
zippy
ali zip
veli and zippy here
zipzippy
ZIP

$cat test1 | ex1
I RECOGNIZED ZIPPY
ali zip
veli and I RECOGNIZED ZIPPY here
zipI RECOGNIZED ZIPPY
ZIP
```

Another lex program (ex2.l)

```
%%
zip printf("ZIP");
zippy printf("ZIPPY");

$cat test1 | ex2
ZIPPY
ali ZIP
veli and ZIPPY here
ZIPZIPPY
```

- ***Lex matches the input string the longest regular expression possible***

Another lex program (ex3.l)

```
%%
monday|tuesday|wednesday|thursday|friday|
  saturday|sunday printf("<%s is a day.>",y
ytext);

$cat test3
today is wednesday september 27

today is <wednesday is a day> september 27
```

- ***Lex declares an external variable called `yytext` which contains the matched string***

Designing Patterns

Designing the proper patterns in lex can be very tricky, but you are provided with a broad range of options for your regular expressions.

. A dot will match any single character *except* a newline.

*,+ Star and plus used to match zero/one or more of the preceding expressions.

? Matches zero or one copy of the preceding expression.

Designing Patterns

| | A logical ‘or’ statement - matches either the pattern before it, or the pattern after.

^ Matches the very beginning of a line.

$ Matches the end of a line.

/ Matches the preceding regular expression, but only if followed by the subsequent expression.

Designing Patterns

[ ] Brackets are used to denote a character class, which matches any single character within the brackets. If the *first* character is a '^', this negates the brackets causing them to match any character except those listed. The '-' can be used in a set of brackets to denote a range. C escape sequences must use a '\'.

" " Match everything within the quotes literally - don't use any special meanings for characters.

( ) Group everything in the parentheses as a single unit for the rest of the expression.

Patterns – Regular expressions

- `a` matches `a`
- `abc` matches `abc`
- `[abc]` matches `a`, `b` or `c`
- `[a-f]` matches `a`, `b`, `c`, `d`, `e`, or `f`
- `[0-9]` matches any digit
- `X+` mathces one or more of X
- `X*` mathces zero or more of X
- `[0-9]+` matches any integer
- `(...)` grouping an expression into a single unit
- `|` alternation (or)
- `(a|b|c)*` is equivalent to `[a-c]*`

Regular expressions in Lex (cont)

- `X?` X is optional (0 or 1 occurrence)
- `if(def)?` matches `if` or `ifdef` (equivalent to `if|ifdef`)
- `[A-Za-z]` matches any alphabetical character
- `.` matches any character except newline character
- `\.` matches the . character
- `\n` matches the newline character
- `\t` matches the tab character
- `\\` matches the `\` character
- `[ \t]` matches either a space or tab character
- `[^a-d]` matches any character other than a,b,c and d

Examples

- Real numbers, e.g., 0, 27, 2.10, .17
  - [0-9]*(\.)?[0-9]+
- To include an optional preceding sign:
  - [+-]?[0-9]*(\.)?[0-9]+
- Integer or floating point number
  - [0-9]+(\.[0-9]+)?
- Integer, floating point, or scientific notation.
  - [+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?

A slightly more complex program (ex4.l)

```
%%
[\t ]+ ;
monday|tuesday|wednesday|thursday|friday|
saturday|sunday printf("%s is a day.",y
ytext);
[a-zA-Z]+ printf("<%s is not a day.>", yytext);
```

A tiny bit more… (ex5.l)

```
%%
[\t ]+ ;
Monday|Tuesday|Wednesday|Thursday|Friday
  printf("%s is a week day.",yytext);
Saturday|Sunday
  printf("%s is a weekend.",yytext);
[a-zA-Z]+
  printf("%s is not day.",yytext);
```

Structure of a lex program

Notice that all of the lex programs seem to have three sections, separated by a pair of percent signs "%%".

Section one is the **definition** section. Here we introduce any code that we want at the top of the C program.

Section two is the **rules** section. Here we link *patterns* with the *action* that they should trigger.

Section three is the **user sub-routines** section. Lex will copy these sub-routines after the code it generates.

Definitions

```
%%
[+-]?[0-9]*(\.)?[0-9]+ printf("FLOAT");
```

The same lex specification can be written as:

```
digit [0-9]
%%
[+-]?{digit}*(\.)?{digit}+ printf("FLOAT");
```

input: `ab7.3c--5.4.3+d++5`

output: `abFLOATc-FLOATFLOAT+d+FLOAT`

Definitions

```
digit [0-9]
sign [+-]
%%
float val;
{sign}?{digit}*(\.)?{digit}+
  {sscanf(yytext, "%f", &val);
printf(">%f<", val);}
```

Input Output

```
ali-7.8veli ali>-7.800000<veli
ali--07.8veli ali->-7.800000<veli
+3.7.5 >3.700000<>0.500000<
```

Definitions

```
/* echo-uppercase-words.l */
%%
[A-Z]+[ \t\n\.\,] printf("%s",yytext);
. ; /* no action specified */
```

- The scanner for the specification above echo all strings of capital letters, followed by a space tab (`\t`)or newline (`\n`) dot (`\.`) or comma (`\,`) to stdout, and all other characters will be ignored.

Input	Output
`Ali VELI A7, X. 12`	`VELI X.`
`HAMI BEY a`	`HAMI BEY`

Definitions

Definitions can be used in definitions

```
/* def-in-def.l */
alphabetic [A-Za-z]
digit [0-9]
alphanumeric ({alphabetic}|{digit})
%%
{alphabetic}{alphanumeric}* printf("Variable");
\, printf("Comma");
\{ printf("Left brace");
\:\= printf("Assignment");
```

User rules

The user sub-routines section is for any additional C or C++ code that you want to include. The only required line is:

```
main() { yylex(); }
```

This is the main function for the resulting program. Lex builds the yylex() function that is called, and will do all of the work for you.

Other functions here can be called from the rules section

Rule order

- If more than one regular expression match the same string the one that is defined earlier is used.

```
/* rule-order.l */
%%
for printf("FOR");
[a-z]+ printf("IDENTIFIER");
```

- for input
- `for count := 1 to 10`
- the output would be
- `FOR IDENTIFIER := 1 IDENTIFIER 10`

Rule order

However, if we swap the two lines in the specification file:

```
%%
[a-z]+ printf("IDENTIFIER");
for printf("FOR");
```

for the same input

the output would be

```
IDENTIFIER IDENTIFIER := 1 IDENTIFIER 10
```

Example Number Identifications (ex6.l)

```
%%
[\t ]+  /* Ignore Whitespace */;

[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
printf(" %s:number", yytext)

[a-zA-Z]+
printf(" %s:NOT number", yytext)
%%
main() { yylex(); }
```

Tracking line numbers (ex7.l)

```
%{
    int line_num = 1;
%}
%%
[\t ]+  /* Ignore Whitespace */;

\n    { line_num++; }

[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
printf("line_num:%d\n" yytext);
%%
main() { yylex(); }
```

More patterns…

What about literal strings?

Does this work? `\".*\"`

What about: `\"[^\"]*\"`

We need to use: `\"[^\"\n]*\"`

Counting Words (ex8.l)

```
%{
int char_count = 0;
int word_count=0;
int line_count=0;
%}
word    [^ \t\n]+
eol     \n
%%
{word} {word_count++; char_count+=yyleng;}
{eol} {char_count++; line_count++;}
. char_count++;
%%
main() {
yylex();
printf("line_count = %d , word_count = %d,
char_count = %d\n", line_count, word_
count, char_count);
}
```

Counting words (cont)

```
$ cat test8
how many words
and how many lines
are there
in this file

ex8 < test8
line_count = 5 , word_count = 12,char_count = 61
```

(ex9.l)

```
%%
 int k;
-?[0-9]+  {
    k = atoi(yytext);
    printf("%d", k%7 == 0 ?  k+3 : k+1) ;
}
-?[0-9\.]+  ECHO;
[A-Za-z][A-Za-z0-9]+ printf("<%s>",yytext);

%%
```

ex10.l

```
int lengs[100];
%%
[a-z]+ {lengs[yyleng]++ ;
if(yyleng==1) printf("<%s> ", yytext); }
. |
\n ;
%%
yywrap()
{
 int i;
printf("Lenght   No. words\n");
for(i=0; i<100; i++) if(lengs[i] >0)
  printf("%5d%10d\n", i, lengs[i]);
return(1) ;
}
```

yywrap is called whenever lex reaches an end-of-file

ex11.l (sets)

```
space       [ \t]*
alphabetic  [A-Za-z]
digit       [0-9]
alphanumeric ({alphabetic}|{digit})
element {space}{alphanumeric}+{space}
%%
\<\-                                  printf("ASSIGNMENT ");
\=                                    printf("EQUALS ");
\<\>                                  printf("NOT-EQUALS");
\<                                    printf("LESS-THAN ");
\+                                    printf("UNION ");
\*                                    printf("INTERSECTION ");
[Ww][Hh][Ii][Ll][Ee]                  printf("WHILE ");
[Dd][Oo]                              printf("DO ");
{alphabetic}{alphanumeric}*           printf("IDENTIFIER ");
\{(({element}\,)*{element})?{space}\} printf("SET ");
{space} ; /* skip spaces */
```

ex12.l (comments)

```
%{
int comms=0 ;
%}
%%
\/\*(.)+\*\/   {comms++;
  printf("COMMENT\n");}
. | \n printf("%s", yytext);
%%
yywrap()
{
printf("NO. of Comments = %d\n", comms);
return(1) ;
}
```