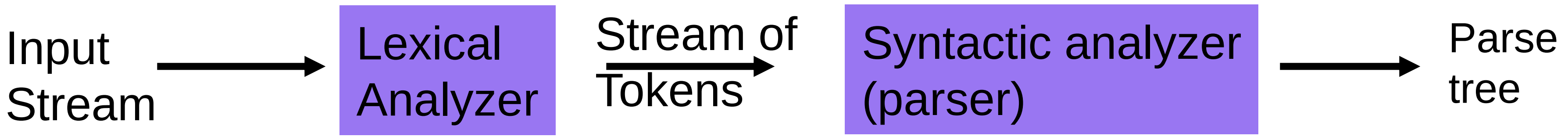
# `Lex` – A tool for lexical analysis

# Lexical and syntactic analysis



- **Lexical analyzer:** scans the input stream and converts sequences of characters into tokens.
  (char list) → (token list)
- **`Lex`** is a tool for writing lexical analyzers.

- **Syntactic Analysis:** reads tokens and assembles them into language constructs using the grammar rules of the language.
- **`Yacc`** is a tool for constructing parsers.

# Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
     z = 0;
else
     z = 1;
```

- The input is just a sequence of characters:

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```

- **Goal:** Partition input strings into substrings
  - And classify them according to their role

## Tokens

- Output of lexical analysis is a list of tokens
- A token is a syntactic category
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - Identifier, Integer, Keyword, Whitespace, ...
- Parser relies on token distinctions:
  - e.g., identifiers are treated differently than keywords

## Example

- Recall:

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```

- Token-lexeme pairs returned by the lexer:
  - <Keyword, "`if`">
  - <Whitespace, " ">
  - <OpenPar, "`(`">
  - <Identifier, "`i`">
  - <Whitespace, " ">
  - <Relation, "`==`">
  - <Whitespace, " ">
  - ...

Slide credit: Wes Weimer

# Implementation of A Lexical Analyzer

- The lexer usually discards **uninteresting** tokens that don't contribute to parsing.
- Examples: Whitespaces, Comments
  - Exception: which language cares about whitespaces?
- The goal is to partition the string. That is implemented by reading left-to-right, recognizing one token at a time.
- Lexical structure described can be specified using ***regular expressions*.**

Slide credit: Wes Weimer

# Regular Expressions

In computing, a **regular expression**, also referred to as "regex" or "regexp", provides a concise and flexible means for **matching strings of text**, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a **regular expression processor**.

http://en.wikipedia.org/wiki/Regular_expression

# Regular Expressions

- Regular expressions are used in many programming languages and software tools to specify patterns and match strings.
- Regular expressions are well suited for matching lexemes in programming languages.
- Regular expressions use a finite alphabet of symbols and defined by the operators
  - (i) union
  - (ii) concatenation
  - (iii) Kleene closure.

Slide credit: Wes Weimer

# Lex − A Lexical Analyzer Generator

**M. E. Lesk and E. Schmidt**

Bell Laboratories
Murray Hill, New Jersey 07974

*ABSTRACT*

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

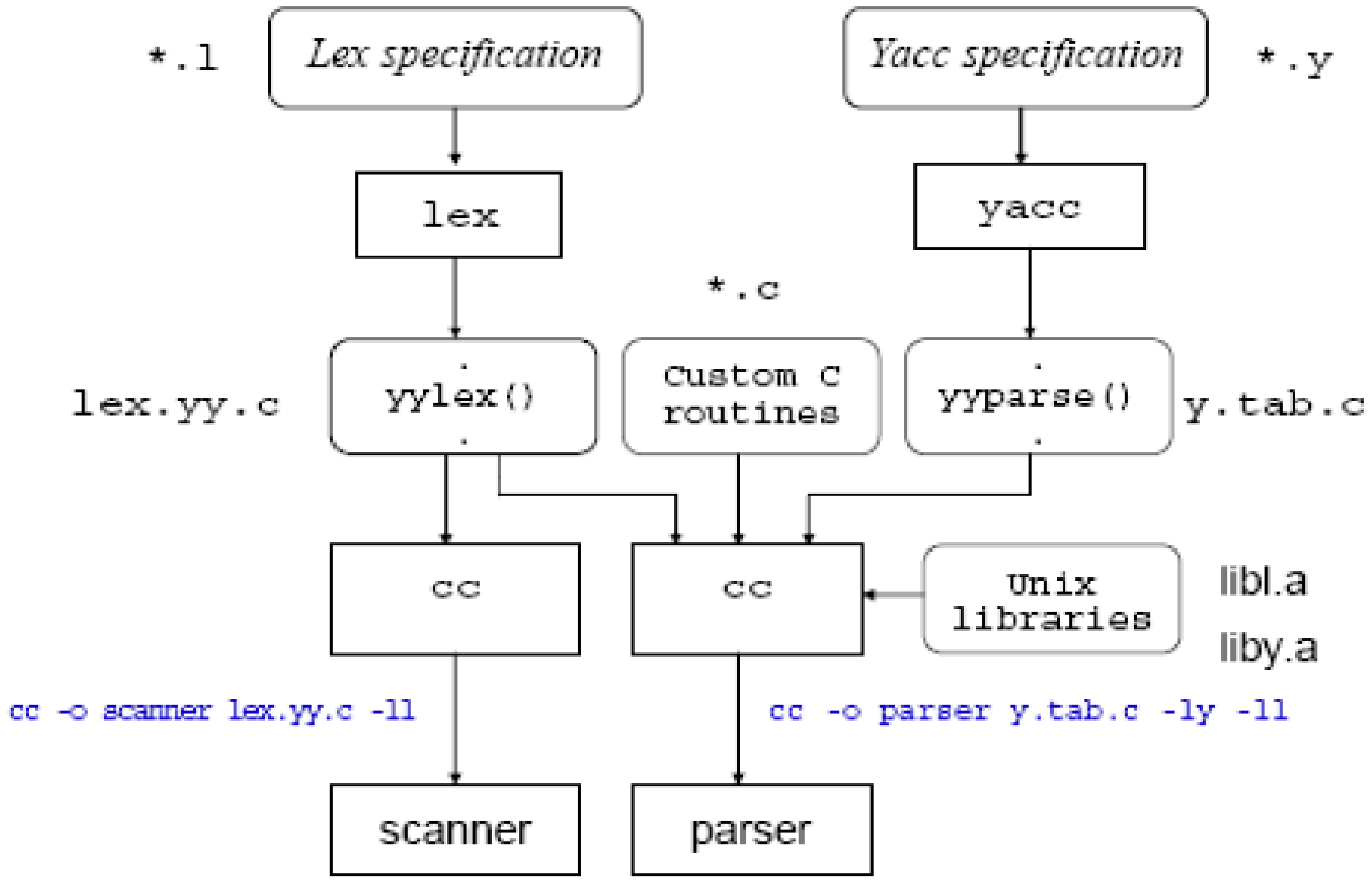
Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

July 21, 1975

## 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 handwritten parser, but much faster to implement.
    Yacc stands for “Yet Another Compiler Compiler”.

# Using `lex` and `yacc` tools



## Running `lex`

On Unix system

```
$ lex mylex.l
```

it will create **`lex.yy.c`**

then type

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

The open-source version of **`lex`** is called **“`flex`”**

# Using `lex`

- Contents of a **`lex`** program:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.

- The translation rules are each of the form

  ```
  pattern {action}
  ```

  - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
  - Each action is a fragment of C-code.

- The auxiliary functions section starting with the second `%%` is optional. Everything in this section is copied directly to the file **`lex.yy.c`** and can be used in the actions of the translation rules.

# Simple `lex` program (`ex0.1`)

**ex0.1 :** (edit with emacs, vi, etc.)

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

($ is the unix prompt)

```
$lex ex0.1
$gcc -o ex0 lex.yy.c -lfl
$ls
ex0 ex0.1 lex.yy.c
```

# Simple `lex` program (`ex0.1`)

```
$vi test0
$cat test0
ali
Veli

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

**Simply echos the input file contents**

# Example lex program (ex1.1)

```
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
```

# Example `lex` program (`ex2.1`)

```
%%
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!**

## Example `lex` program (`ex3.1`)

```
%%
monday|tuesday|wednesday|thursday|friday|
saturday|sunday printf("<%s is a day.>",
  yytext);
$cat test3
today is wednesday september 27

$ex3 < test3
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.
- “ ”	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.

## Regular expressions in `lex`

- `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+` matches one or more of `X`
- `X*` matches 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`

- **`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 (`ex4.1`)

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

# ex5.1

```
%%
[\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

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.

- The translation rules are each of the form
  `pattern {action}`

  - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
  - Each action is a fragment of C-code.

- The auxiliary functions section starting with the second `%%` is optional. Everything in this section is copied directly to the file **`lex.yy.c`** and can be used in the actions of the translation rules.

## Declarations

```
%%
[+-]?[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`

## Declarations

```
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<
```

## Declarations

```
/* 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**

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

**Output**

```
VELI X.
HAMI BEY
```

## Declarations

Declarations can be used in Declarations

```
/* 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");
```

# Lex file structure

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

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

# Auxiliary functions

- 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 (`ex6.1`)

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

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

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

# Counting Words (`ex7.1`)

```
%{
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);
}
```

**`lex`** also provides a count **`yyleng`** of the number of characters matched

# Counting words (cont'd.)

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

$ex7 < test8
line_count = 5, word_count =
  12,char_count = 58
```

# ex8.1

```
%%
 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);
%%
```

# ex9.1

```
%{
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

# romans.l

```
WS      [ \t]+

%%
        int total=0;
I       total += 1;
IV      total += 4;
V       total += 5;
IX      total += 9;
X       total += 10;
XL      total += 40;
L       total += 50;
XC      total += 90;
C       total += 100;
CD      total += 400;
D       total += 500;
CM      total += 900;
M       total += 1000;

{WS}    |
\n      return total;
%%
int main (void) {
   int first, second;

   first = yylex ();
   second = yylex ();

   printf ("%d + %d = %d\n", first, second,
    first+second);
   return 0;
   }
```