FLex

Overview of 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.

Cont.

- **Purpose:** to construct the scanner
- **Input:** a table of regular expressions and corresponding program fragments
  - Used to construct a deterministic finite automaton
- **Output:** a scanner, written in C, which
  - Reads an input stream (source language program)
  - Partitions input stream into strings which match regular expressions
  - Produces an output stream (list of tokens)

Skeleton of a Lex Specification (.l file)

x.l → lex.yy.c is generated after running

```
> lex x.l
```

%{

< C global variables, prototypes, comments >

%}

[DEFINITION SECTION] → This part will be embedded into lex.yy.c

%%

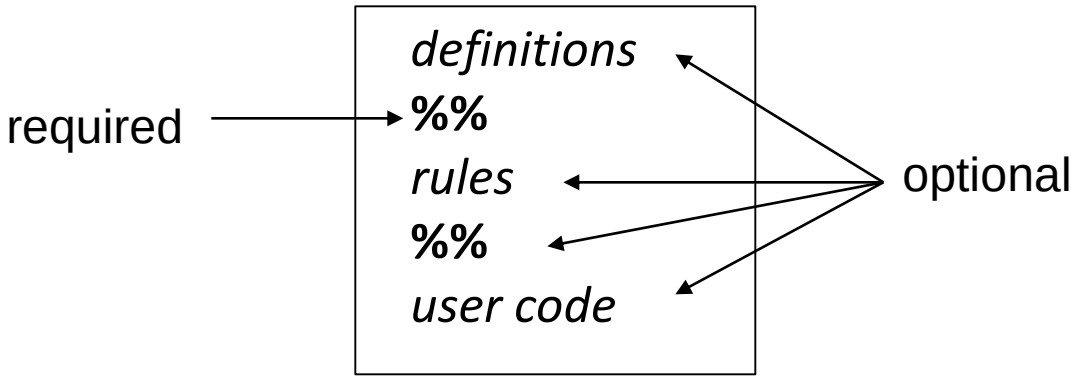
[RULES SECTION] → Define how to scan and what action to take for each token

%%

C auxiliary subroutines/code → Any user code.

Lex Source

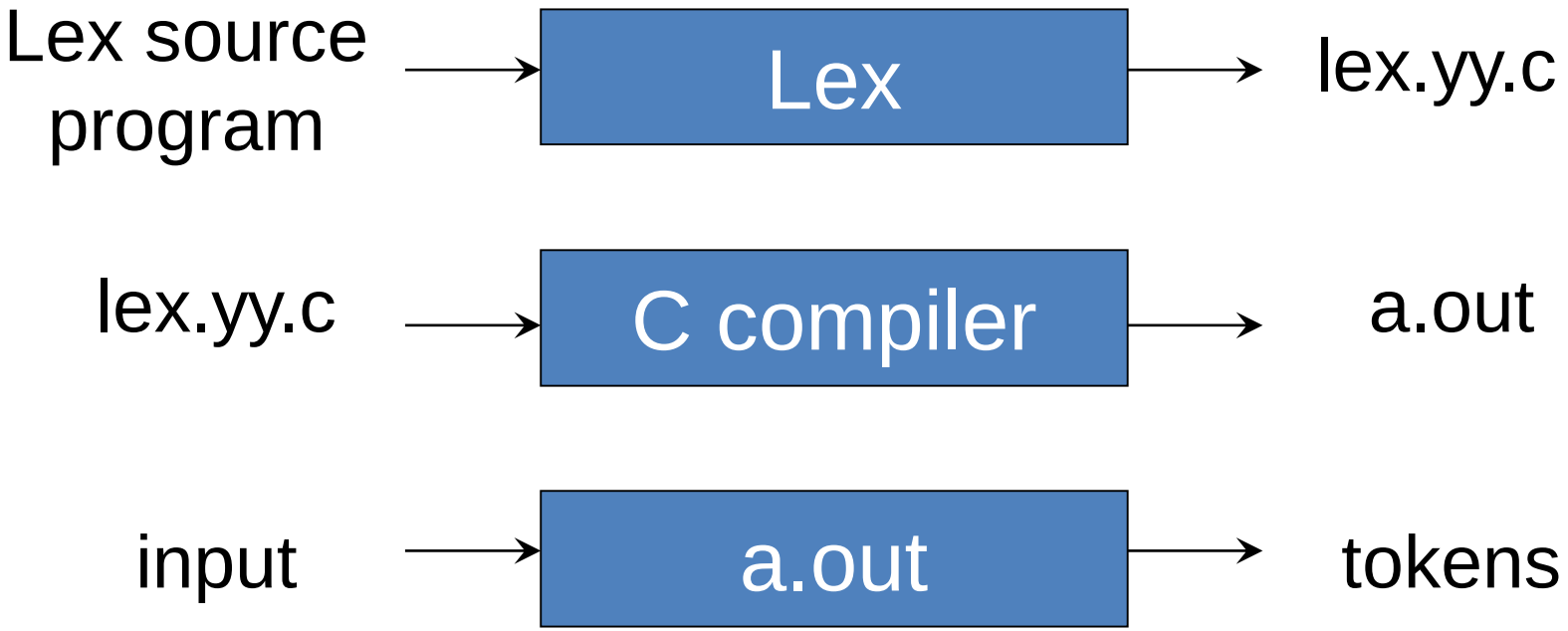
- Lex source is separated into three sections by %% delimiters



Shortest possible legal flex input:

```
%%
```

In Context of C



The Shortest Lex program

```
%%
```

- This program contains no definitions, no rules, and no user subroutines!
- It copies the input to the output without change.

Lex Program to Delete White Space at End of Lines

```
%%
[ \t]+$      ;
```

`\t` means “tab”

`[ \t]` means “either ‘space’ or ‘tab’”

`[ \t]+` means “a string of one or more ‘spaces’ or ‘tabs’”

`$` means “end of line”

`[ \t]+$` means “a string of one or more ‘spaces’ or ‘tabs’ followed by ‘end of line’”

There is no code fragment, so the text which matches the pattern is erased and not replaced with anything.

Lex Program to Compress White Space

```
%%
[ \t]+$    ;
[ \t]+     printf(" ");
```

Ex. Identifier in Pascal

```
Digit                [0-9]
Letter               [a-zA-Z]
%%
{Letter}({Digit} | {Letter})*   printf("\n The found identifier is =
    %s", yytext);
```

Definition Section

- A series of:
  - *name definitions*, each of the form

    name *definition*

    e.g.:

    ```
    DIGIT          [0-9]
    CommentStart   "/*"
    ID             [a-zA-Z][a-zA-Z0-9]
    ```

    These definitions can be used in rules section as
    {DIGIT}+ {….

  - stuff to be copied verbatim into the flex output (e.g., declarations, **#include**s):
    - enclosed in %{ … }%, or
    - indented

Rules Section

- The *rules* portion of the input contains a sequence of rules.
- Each rule has the form

  patterns *actions*

  where:
  - *Patterns are* regular expression which describes a pattern to be matched on the input
  - *pattern* must be un-indented
  - *actions* are either a single C command or a sequence enclosed in braces. It must begin on the same line of patterns.

Count no.of chars and lines

```
%{
int charcount=0,linecount=0;
%}
%%
. charcount++;
\n {linecount++; charcount++;}
%%
int main()
{
yylex();
printf("There were %d characters in %d lines\n",
charcount,linecount);
return 0;
}
```

Count no.of chars, words and lines

```
%{
int charcount=0,linecount=0,wordcount=0;
%}
letter [^ \t\n]
%%
{letter}+ {wordcount++; charcount+=yyleng;}
. charcount++;
\n {linecount++; charcount++;}
```

Patterns

- Essentially, extended regular expressions.
  - Syntax: similar to grep (see man page)

Metacharacters

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: Examples

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`

Operators

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

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

  `\$` = "$"

  `\\` = "\"

- Every character except *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*))`

Regular Expression

x	match the character 'x'
.	any character (byte) except newline
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ]	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z]	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n]	any character EXCEPT an uppercase letter or a newline

Regular Expression

r* zero or more r's, where r is any regular expression

r+ one or more r's

r? zero or one r's (that is, "an optional r")

r{2,5} anywhere from two to five r's

r{2,} two or more r's

r{4} exactly 4 r's

{name} the expansion of the "name" definition

"[xyz]\"foo“ the literal string: [xyz]"foo

\X if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v',
then the ANSI-C interpretation of \x.
Otherwise, a literal 'X' (used to escape
operators such as '*')

Regular Expression

\0	a NULL character (ASCII code 0)
\123	the character with octal value 123
\x2a	the character with hexadecimal value 2a
(r)	match an r; parentheses are used to override precedence
rs	the regular expression r followed by the regular expression s; called "concatenation"
r|s	either an r or an s
^r	an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
r$	an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Two Notes on Using Lex

1. Lex matches token with longest match

Input: *abc*
Rule: `[a-z]+`

→ Token: *abc* (not "*a*" or "a*b*")

2. Lex uses the first applicable rule

Input: *post*
Rule1: `"post"  {printf ("Hello,"); }`
Rule2: `[a-zA-Z]+  {printf ("World!"); }`

→ It will print Hello, (not "World!")

Features

- Some limitations, Lex cannot be used to recognize nested structures such as parentheses, since it only has states and transitions between states.

- Echo is an action and predefined macro in lex that writes code matched by the pattern.

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

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

Features (cont)

- Text enclosed by %{ and %} is assumed to be C code and is copied verbatim
- Line which begins with white space is assumed to be a comment and is ignored
- Other lines are assumed to be definitions
- All input characters which are not matched by a lex rule are copied to the output stream (the file lex.yy.c which contains function yylex)
- Definitions from the definitions section are physically substituted into the rules

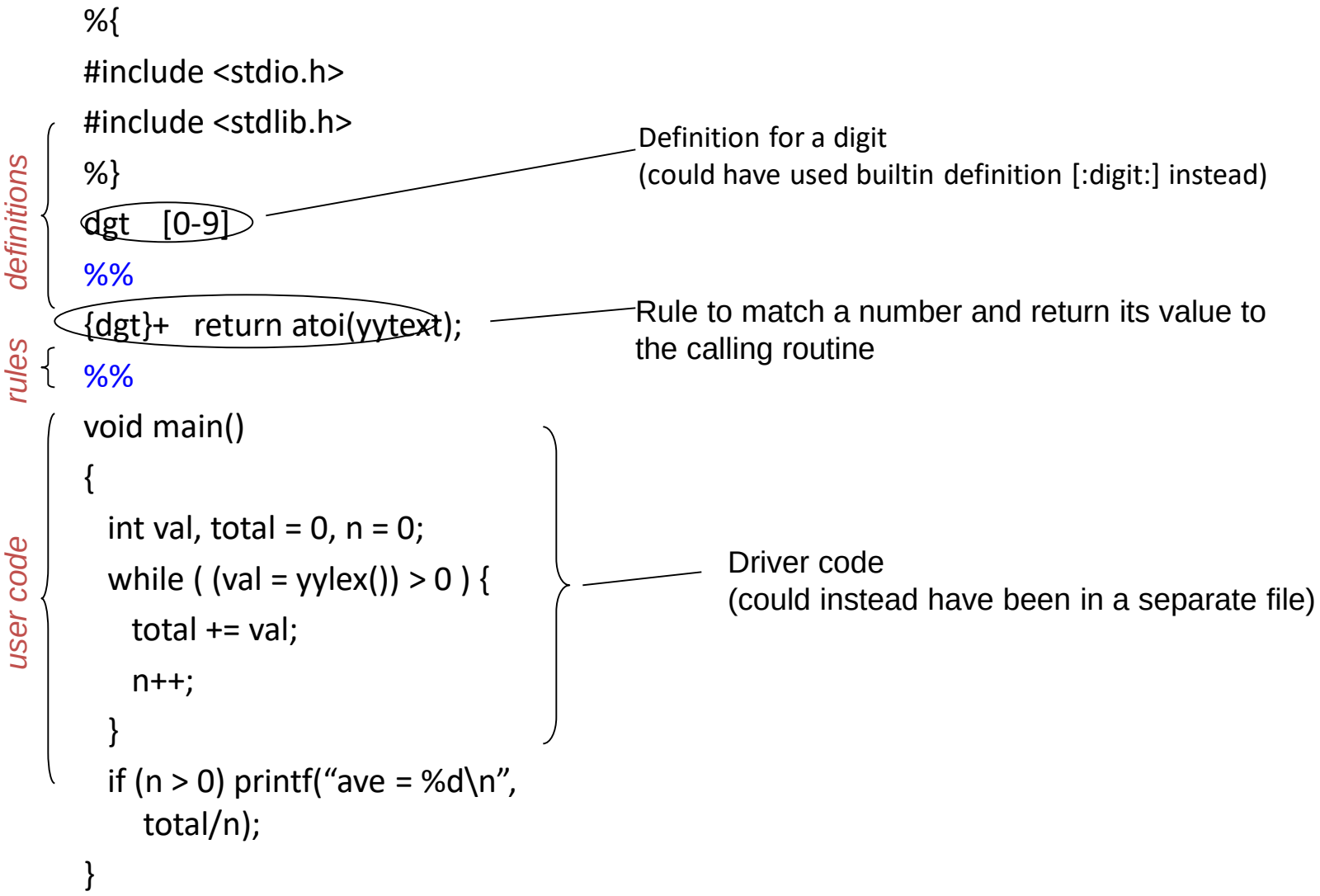
Example

A flex program to read a file of (positive) integers and compute the average:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt    [0-9]
%%
{dgt}+   return atoi(yytext);
%%
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) > 0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n",
    total/n);
}
```

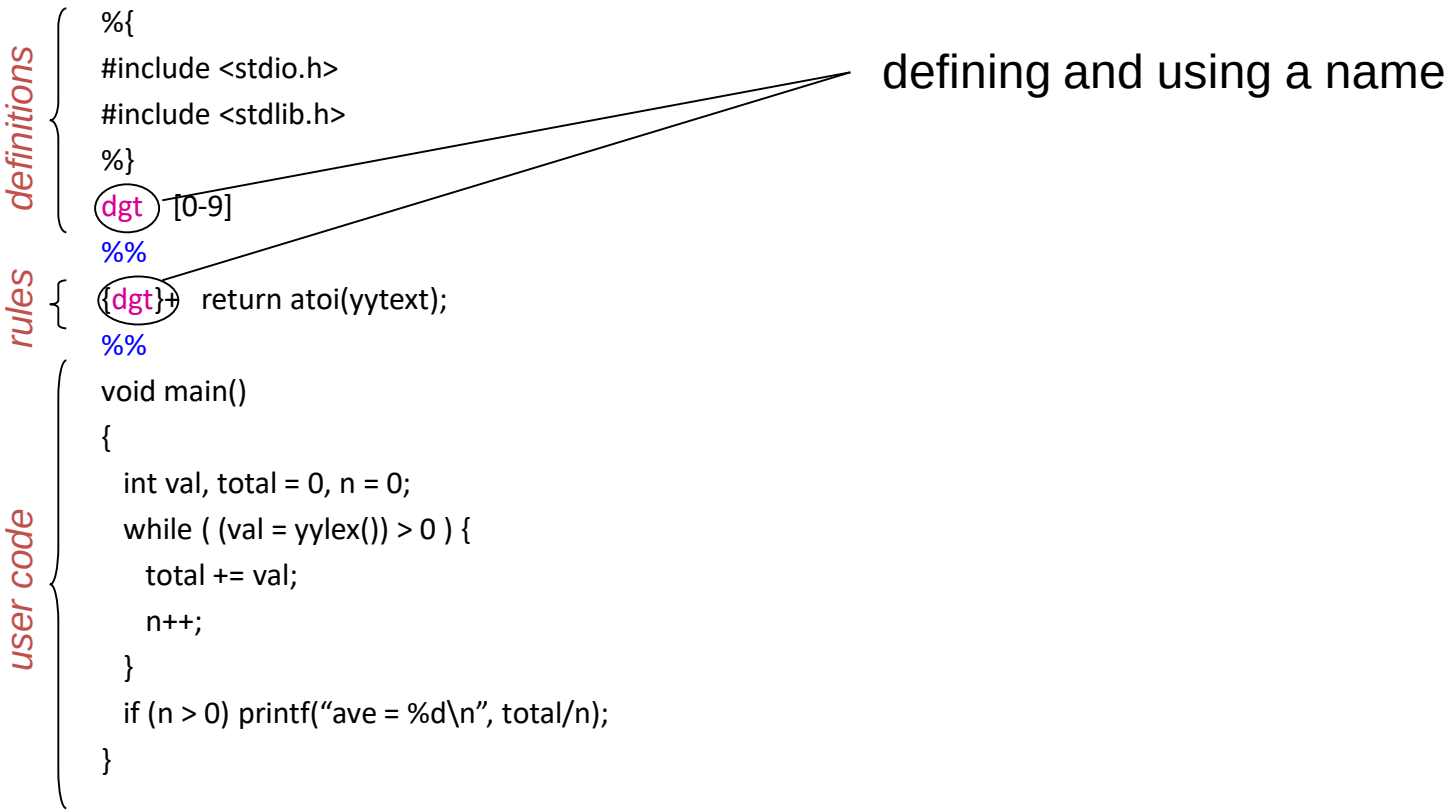
Example

A flex program to read a file of (positive) integers and compute the average:



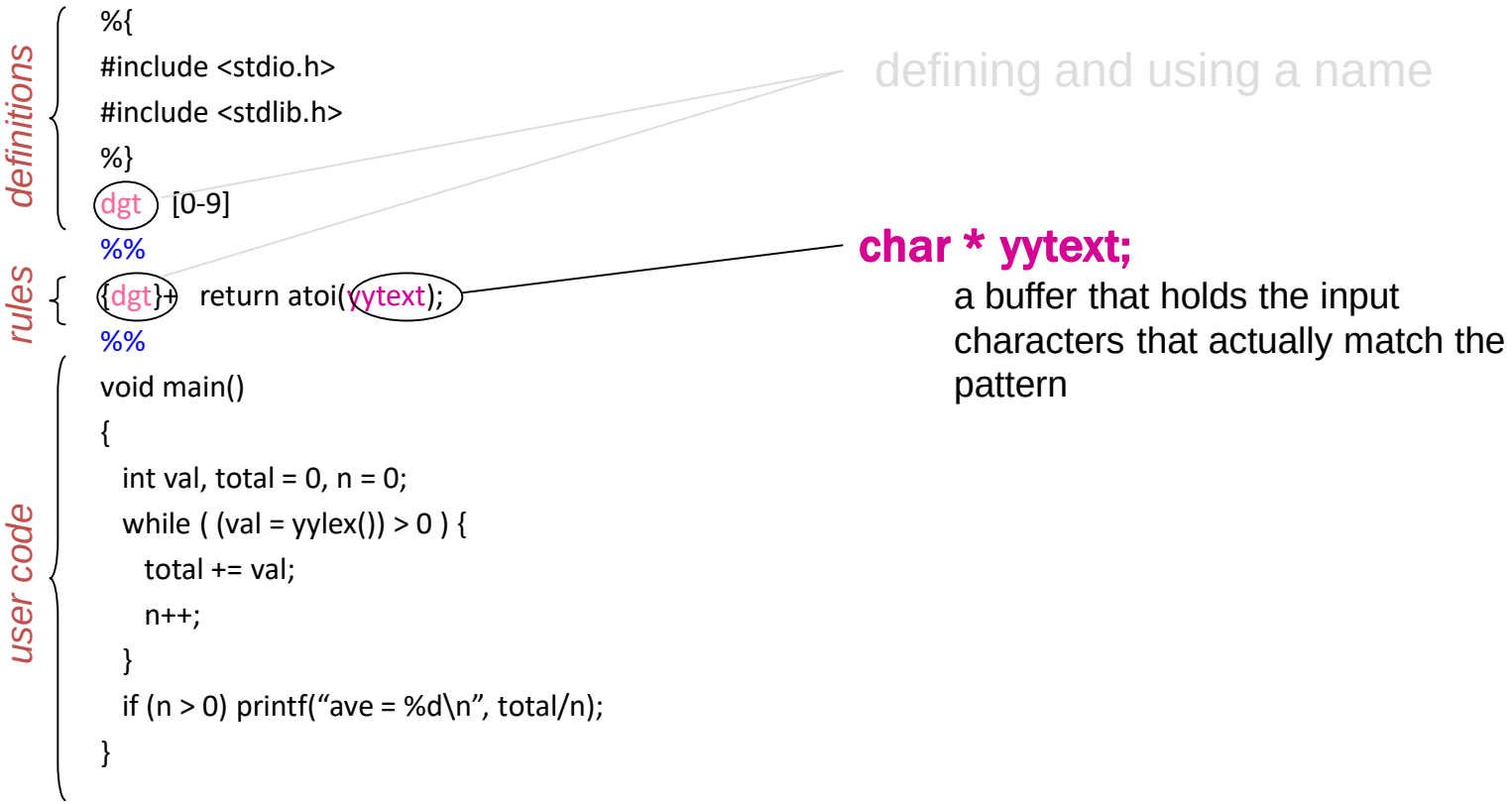
Example

A flex program to read a file of (positive) integers and compute the average:



Example

A flex program to read a file of (positive) integers and compute the average:



Example

A flex program to read a file of (positive) integers and compute the average:

definitions

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
dgt [0-9]
```

rules

```
%%
{dgt}+ return atoi(yytext);
%%
```

user code

```
void main()
{
  int val, total = 0, n = 0;
  while ( (val = yylex()) >0 ) {
    total += val;
    n++;
  }
  if (n > 0) printf("ave = %d\n", total/n);
}
```

defining and using a name

char * yytext;

a buffer that holds the input characters that actually match the pattern

Invoking the scanner: **yylex()**

Each time yylex() is called, the scanner continues processing the input from where it last left off.

Returns 0 on end-of-file.

Matching the Input

- When more than one pattern can match the input, the scanner behaves as follows:
  - the longest match is chosen;
  - if multiple rules match, the rule listed first in the flex input file is chosen;
  - if no rule matches, the default is to copy the next character to **stdout**.
- The text that matched (the “token”) is copied to a buffer **yytext**.

Matching the Input (cont'd)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

```
#include <stdio.h>  /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf("Error!\n");  /* no arguments */
  }
  printf("%d args given\n", argc);
  return 0;
}
```

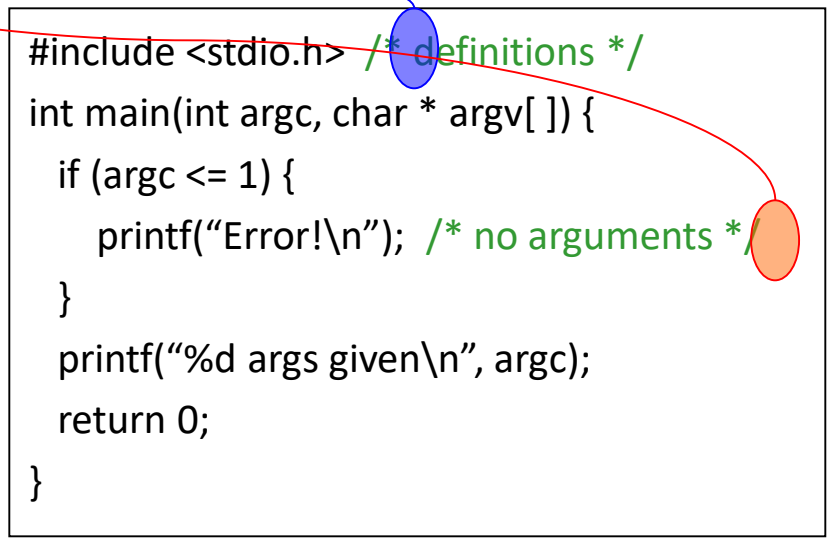
Matching the Input (cont'd)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

longest match:



Matching the Input (cont'd)

Pattern to match C-style comments: /* … */

"/*"(.|\n)*"*/"

Input:

longest match:
Matched text
shown in blue

```
#include <stdio.h>  /* definitions */
int main(int argc, char * argv[ ]) {
  if (argc <= 1) {
     printf("Error!\n");  /* no arguments */
  }
  printf("%d args given\n", argc);
  return 0;
}
```

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**.

- 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(), a function of lex.yy.c file generated after using command lex
- 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

To count no of Identifiers

```
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;
}
```

- White space 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.

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);
}
```

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
  ```