Introduction to Compiler Design

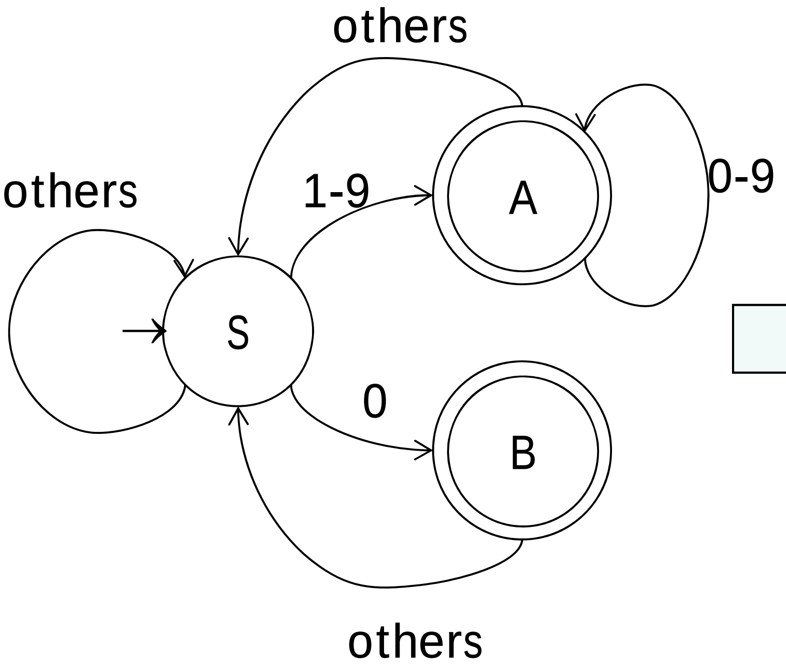
Lex – A Lexical Analyzer Generator

Professor Yi-Ping You
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Why bother using lex and yacc?

- Given a string, what would you do if you want to write a program to know how many “integer” appear?

Well, maybe think with FA first…

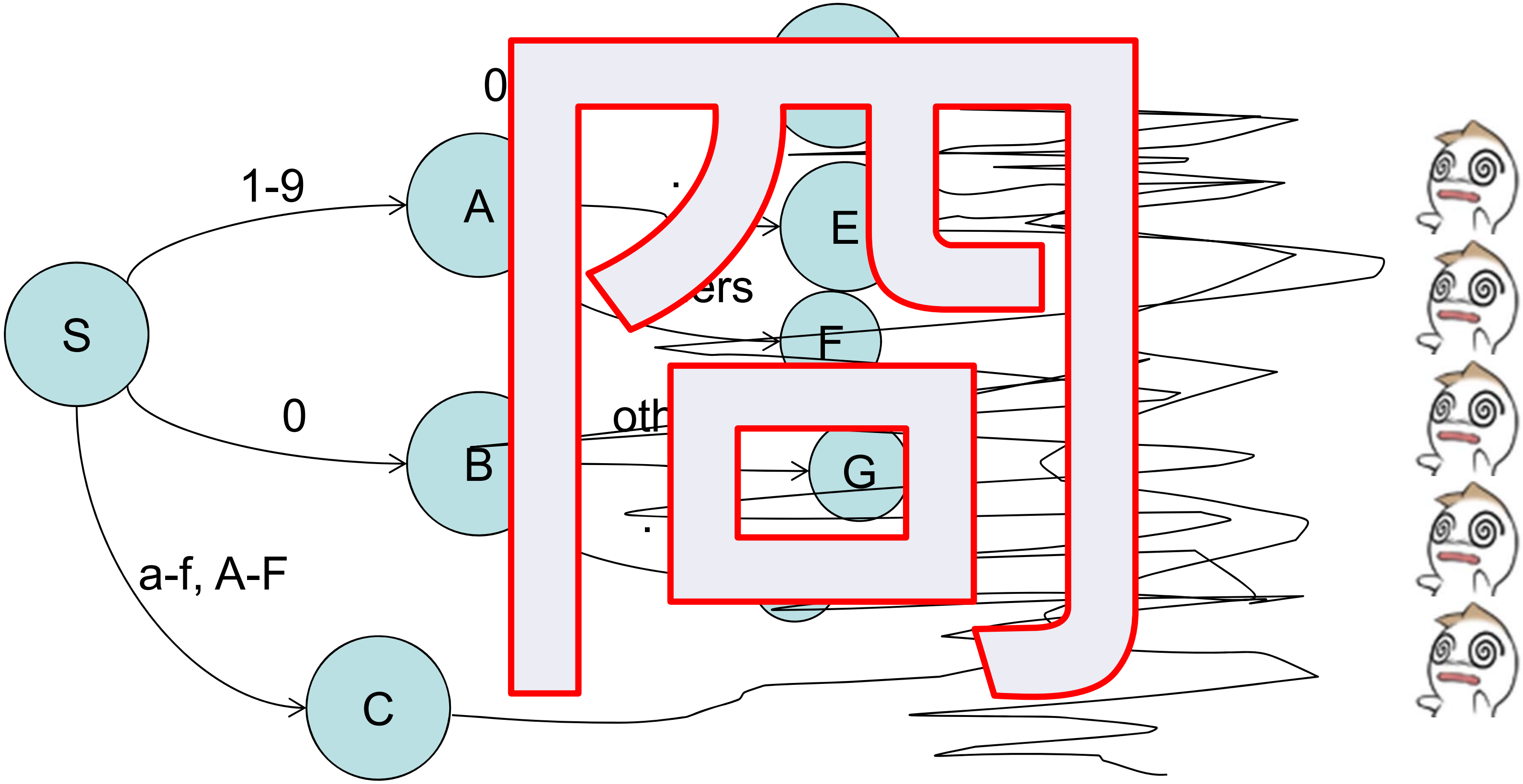


```
while (c = string[i]) {
  if ((c >= “1”) &&
      (c <= “9”)) {
    //check some states
    //change state
    //do something
  } else if (c == “0”)
  {
    :
    :
    :
  }
}
```

Deal with more complex cases

- To recognize “integer in decimal system”
- To recognize “integer in hex system”
- To recognize “real number in decimal system”
- ...

Okay, draw FA first



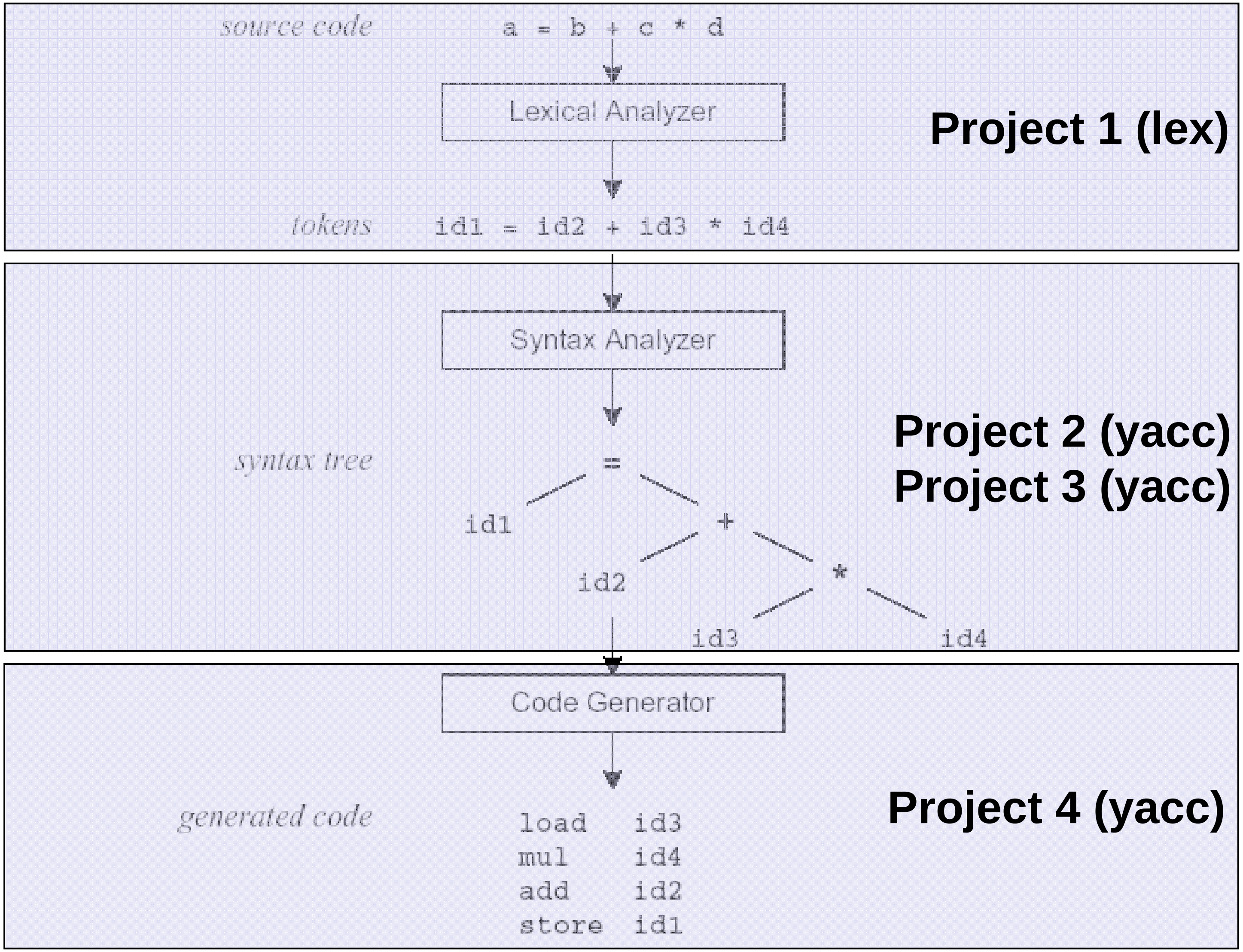
- Even you survive the FA design, the implementation will still be a disaster!!

Let lex and yacc save you!

- Do we must deal them with bare hand from the ground?

- Both tools are developed by AT&T for text analyzing since 1970:
  - Lex
    - Lex generates C code for a lexical analyzer, or scanner
    - Lex uses patterns that match strings in the input and converts the strings to tokens
  - Yacc
    - Yacc generates C code for syntax analyzer, or parser
    - Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree
- Lex divides data into the smallest meaningful elements, and yacc deals the relation between those elements

Compilation Flow



source code a = b + c * d

Lexical Analyzer

tokens id1 = id2 + id3 * id4

Project 1 (lex)

Syntax Analyzer

syntax tree

Project 2 (yacc)
Project 3 (yacc)

Code Generator

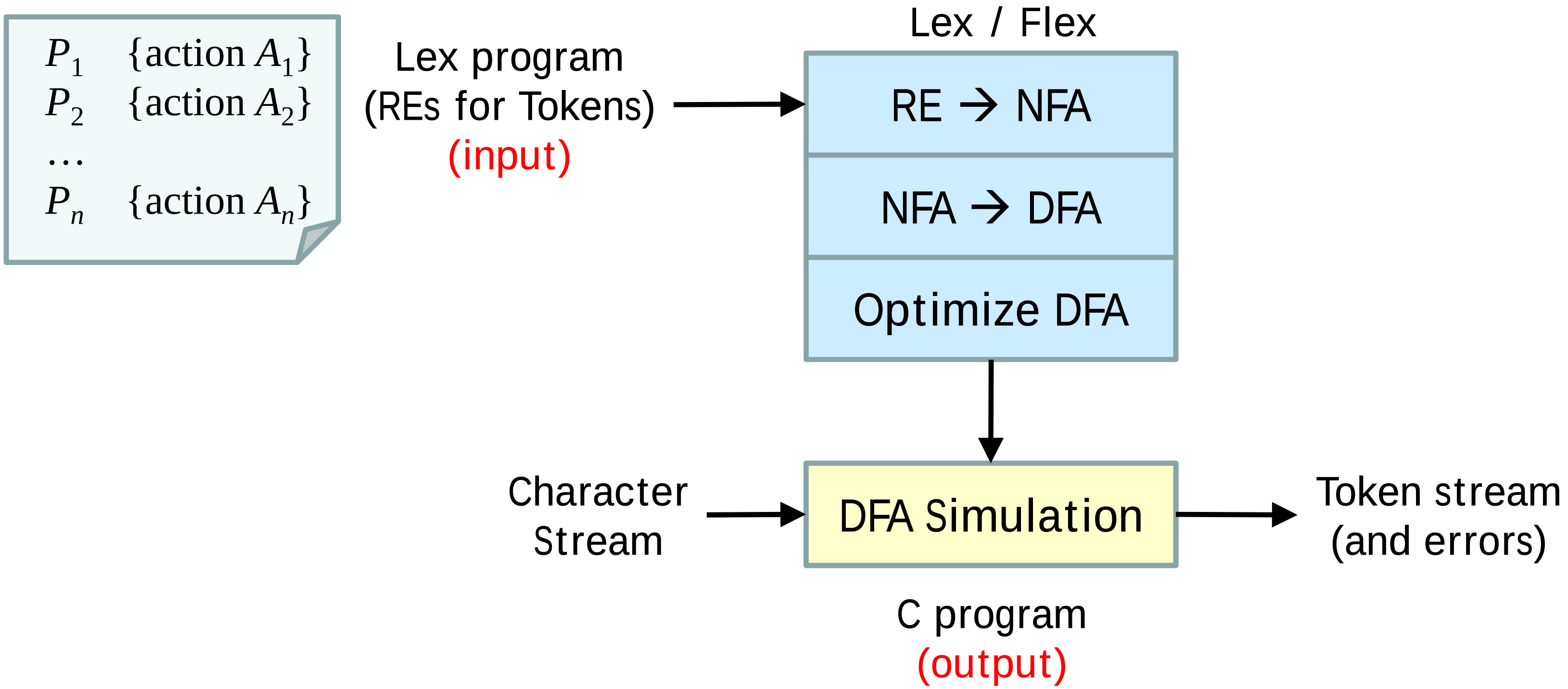
generated code

```
load   id3
mul    id4
add    id2
store  id1
```

Project 4 (yacc)

How Lex help

- It constructs FAs internally from regular expressions which are provided by users, and generates an efficient C codes to recognize them
  - Lots of techniques you learned are applied



A Lex Example

- To recognize “integers” and “real numbers”

```
%%
0|[1-9][0-9]* {
   printf("an integer\n");
}
(0|[1-9][0-9]*)\.[0-9]* {
   printf("a real number\n");
}
. {
   printf("others\n");
}
%%
```

```
Result:
> 012 34.56 789 0.1
an integer
an integer
others
a real number
others
an integer
others
a real number
```

A Yacc Example

```
%%
expression : cterm
 | expression ‘+’ cterm {printf(“+ expression\n”);}
 | expression ‘-’ cterm {printf(“- expression\n”);}
 ;

cterm : cfactor
 | cterm ‘*’ cfactor {printf(“* expression\n”);}
 | cterm ‘/’ cfactor {printf(“/ expression\n”);}
 ;

cfactor : INTEGER {printf(“integer from lex\n”);}
 | REAL {printf(“real number from lex\n”);}
 ;
%%
```

What is Lex?

- Lex is an utility to help you rapidly generate your lexical analyzer

- 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

- Regular expressions define **tokens**

  `[a-zA-Z]+` => a word

Lex Source Program

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

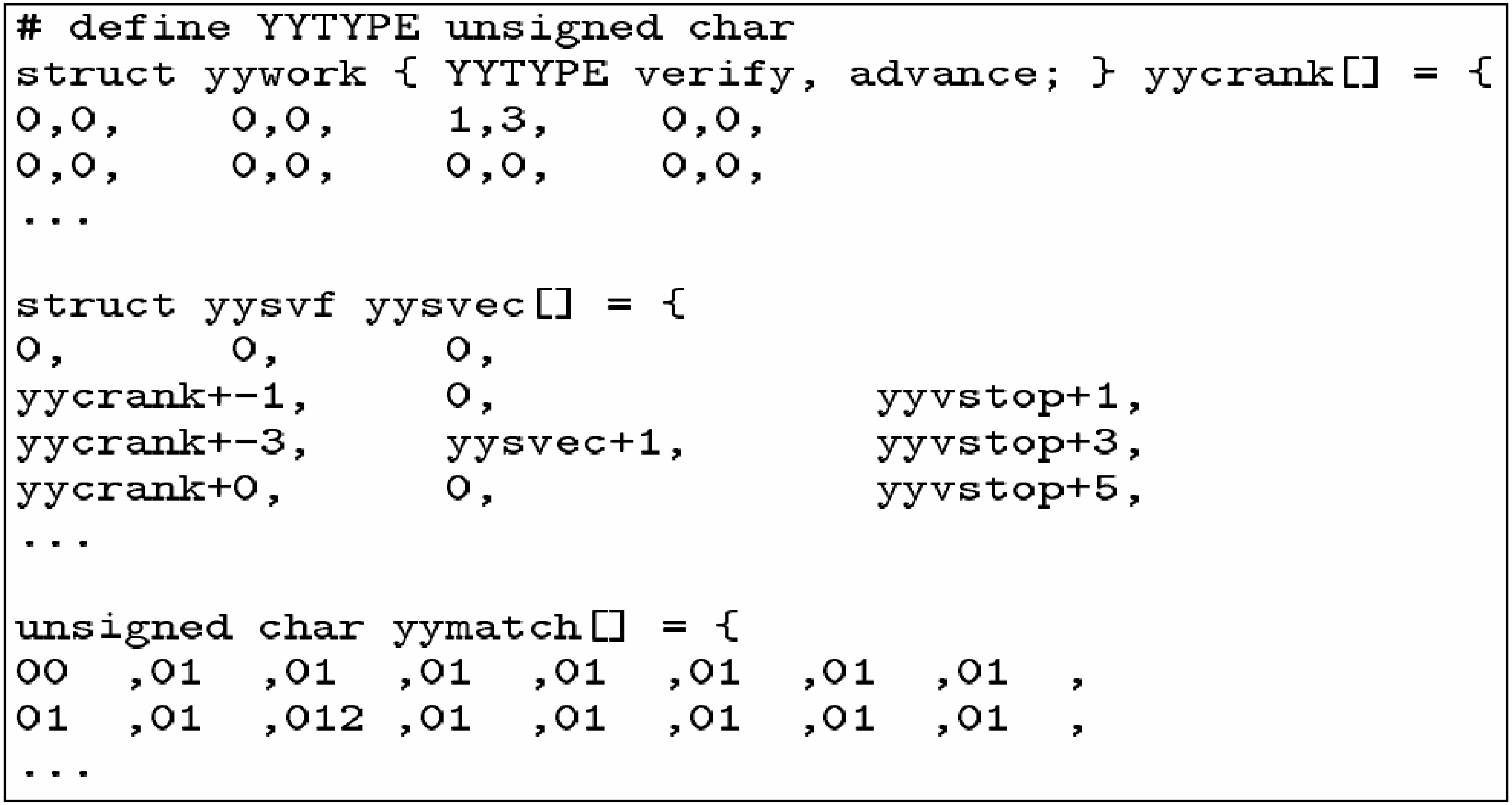
P_1 {action A_1}
P_2 {action A_2}
…
P_n {action A_n}

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

Lex Program 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

Snapshot of lex.yy.c

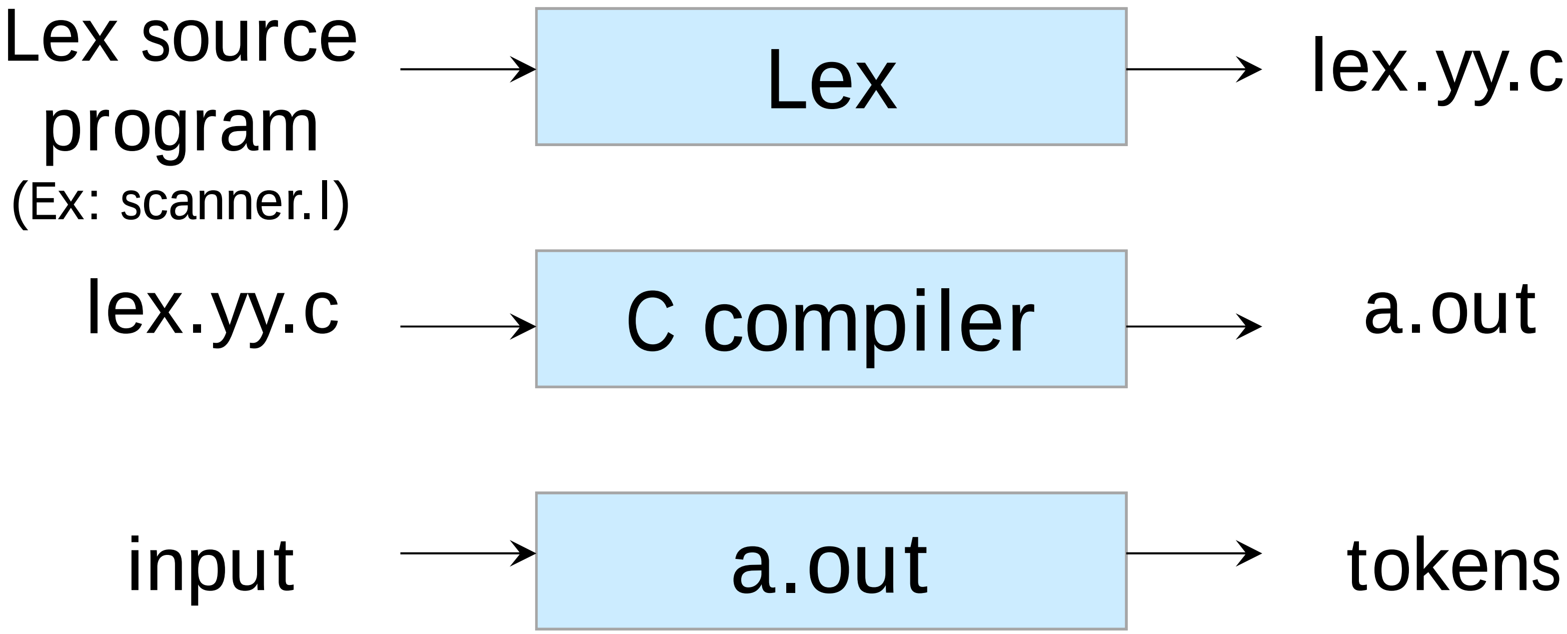


```
# define YYTYPE unsigned char
struct yywork { YYTYPE verify, advance; } yycrank[] = {
0,0,    0,0,    1,3,    0,0,
0,0,    0,0,    0,0,    0,0,
...

struct yysvf yysvec[] = {
0,      0,      0,
yycrank+-1,     0,              yyvstop+1,
yycrank+-3,     yysvec+1,       yyvstop+3,
yycrank+0,      0,              yyvstop+5,
...

unsigned char yymatch[] = {
00  ,01  ,01  ,01  ,01  ,01  ,01  ,01  ,
01  ,01  ,012 ,01  ,01  ,01  ,01  ,01  ,
...
```

An Overview of Lex


Lex source program

(Ex: scanner.l)

Lex

lex.yy.c

lex.yy.c

C compiler

a.out

input

a.out

tokens

Lex Source Program

- Lex program 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

```
%%
```

General Format of Lex Program

```
%{
      C declarations and includes
%}

<name>        <regexp>
<name>        <regexp>
…

%%

<regexp>      <action>
<regexp>      <action>
…

%%

User subroutines (C code)
```

Definitions

Rules

Routines

Lex Regular Expressions

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

Operators

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

- “xyz” = xyz
- If they are to be used as text characters, an escape should be used

  \$ = “$”

  \\ = “\”

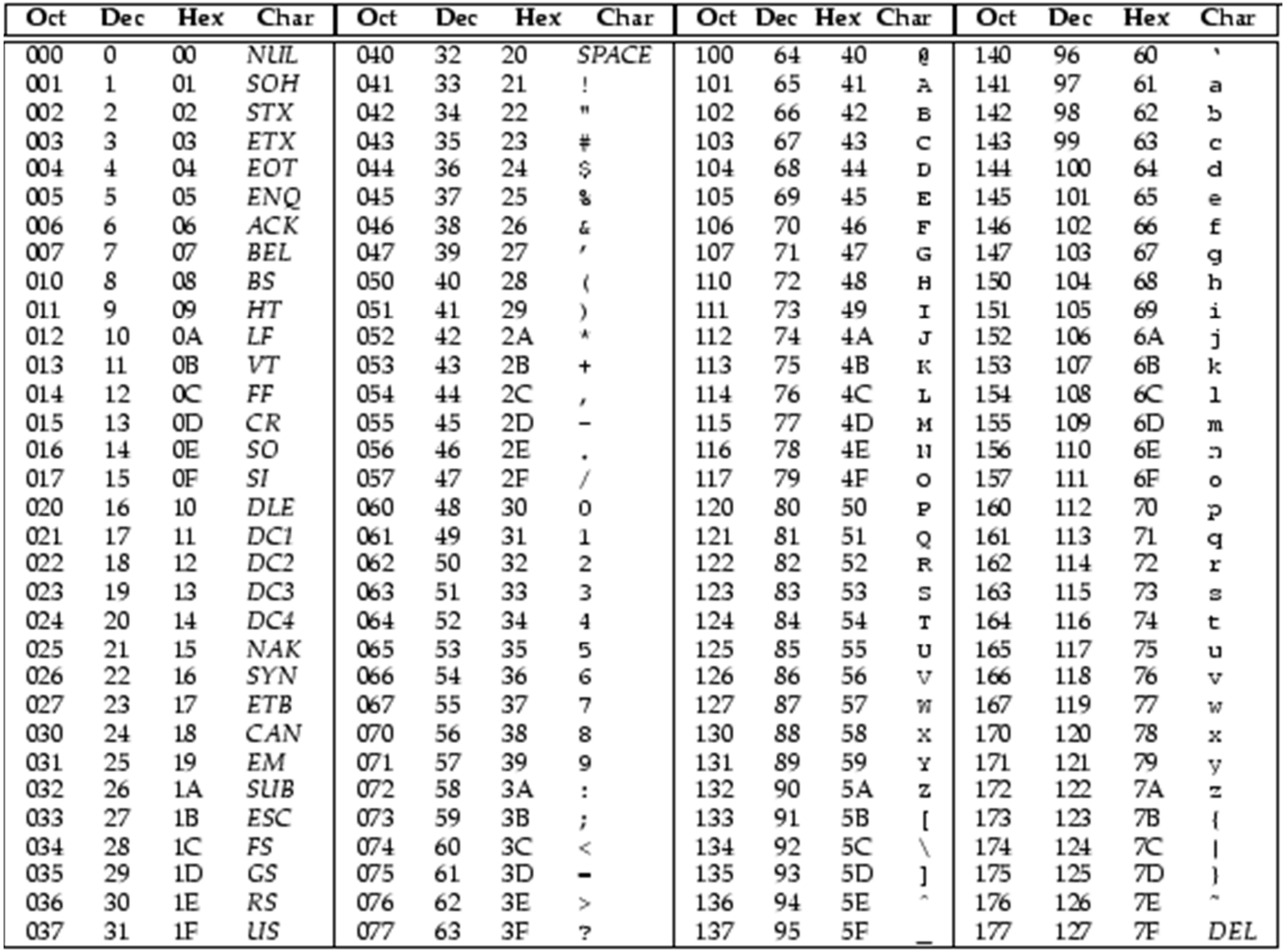
  xyz“++” = “xyz++” = xyz\+\+
- Every character but *blank*, *tab* (\t), *newline* (\n) and the list above is always a text character

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

ASCII Table



Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	*NUL*	040	32	20	*SPACE*	100	64	40	@	140	96	60	`
001	1	01	*SOH*	041	33	21	!	101	65	41	A	141	97	61	a
002	2	02	*STX*	042	34	22	"	102	66	42	B	142	98	62	b
003	3	03	*ETX*	043	35	23	#	103	67	43	C	143	99	63	c
004	4	04	*EOT*	044	36	24	$	104	68	44	D	144	100	64	d
005	5	05	*ENQ*	045	37	25	%	105	69	45	E	145	101	65	e
006	6	06	*ACK*	046	38	26	&	106	70	46	F	146	102	66	f
007	7	07	*BEL*	047	39	27	'	107	71	47	G	147	103	67	g
010	8	08	*BS*	050	40	28	(	110	72	48	H	150	104	68	h
011	9	09	*HT*	051	41	29	)	111	73	49	I	151	105	69	i
012	10	0A	*LF*	052	42	2A	*	112	74	4A	J	152	106	6A	j
013	11	0B	*VT*	053	43	2B	+	113	75	4B	K	153	107	6B	k
014	12	0C	*FF*	054	44	2C	,	114	76	4C	L	154	108	6C	l
015	13	0D	*CR*	055	45	2D	-	115	77	4D	M	155	109	6D	m
016	14	0E	*SO*	056	46	2E	.	116	78	4E	N	156	110	6E	n
017	15	0F	*SI*	057	47	2F	/	117	79	4F	O	157	111	6F	o
020	16	10	*DLE*	060	48	30	0	120	80	50	P	160	112	70	p
021	17	11	*DC1*	061	49	31	1	121	81	51	Q	161	113	71	q
022	18	12	*DC2*	062	50	32	2	122	82	52	R	162	114	72	r
023	19	13	*DC3*	063	51	33	3	123	83	53	S	163	115	73	s
024	20	14	*DC4*	064	52	34	4	124	84	54	T	164	116	74	t
025	21	15	*NAK*	065	53	35	5	125	85	55	U	165	117	75	u
026	22	16	*SYN*	066	54	36	6	126	86	56	V	166	118	76	v
027	23	17	*ETB*	067	55	37	7	127	87	57	W	167	119	77	w
030	24	18	*CAN*	070	56	38	8	130	88	58	X	170	120	78	x
031	25	19	*EM*	071	57	39	9	131	89	59	Y	171	121	79	y
032	26	1A	*SUB*	072	58	3A	:	132	90	5A	Z	172	122	7A	z
033	27	1B	*ESC*	073	59	3B	;	133	91	5B	[	173	123	7B	{
034	28	1C	*FS*	074	60	3C	<	134	92	5C	\	174	124	7C	|
035	29	1D	*GS*	075	61	3D	=	135	93	5D	]	175	125	7D	}
036	30	1E	*RS*	076	62	3E	>	136	94	5E	^	176	126	7E	~
037	31	1F	*US*	077	63	3F	?	137	95	5F	_	177	127	7F	*DEL*

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

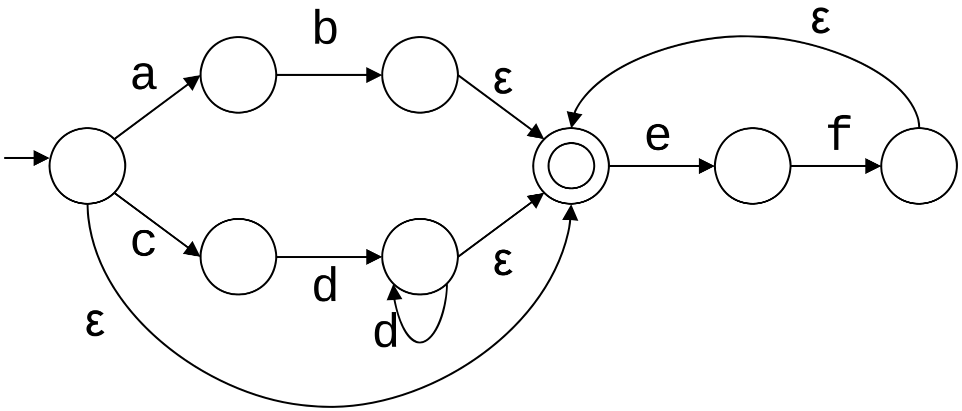
  [a-zA-Z][a-zA-Z0-9]* => all alphanumeric strings with a leading alphabetic character

Alternation | and Grouping ( )

- (ab|cd) = ab|cd => ab or cd
- (ab|cd+)?(ef)*

=> abefef, efefef, cdef, cddd, ...

but not abc, abcd, or abcdef



Context Sensitivity ^ $ /

- ^ab matches the string ab, and only if ab is at the beginning of a line

  (if ^ is the first character of an expression)
- ab$ matches the string ab, and only if ab is at the end of a line

  (if $ is the last character of an expression)
- ab/cd matches the string ab, but only if followed by cd

- ab$ = ab/\n are the same rule
- Recall: [^ab]

Repetitions and Definitions {}

- The operators {} specify either
  - repetitions (if they enclose numbers)
  - definition expansion (if they enclose a name)
- E.g.

  `a{5}` => 5 occurrences of a

  `a{1,5}` => 1 to 5 occurrences of a

  `{digit}` => a predefined string named *digit*

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

Precedence of Operators

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

Recall: Lex Program

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

```
…
%%
<regexp>      <action>
<regexp>      <action>
…
%%
```

```
a = b + c;

↓↓

a operator: ASSIGNMENT b + c;
```

- E.g.,

```
%%
“=“    printf(“operator: ASSIGNMENT”);
```

Transition Rules

- regexp <one or more blanks> action (C code);
- regexp <one or more blanks> { actions (C code) }
- Unmatched patterns will perform a default action, which copies the input to the output
- 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, REJECT, and BEGIN
- | indicates that the action for this rule is from the action for the next rule

```
[ \t\n]   ;
" "       |
"\t"      |
"\n"      ;
```

- ECHO copies the matched string to the output
- An unmatched token uses the default action, ECHO

Ambiguous Source Rules

- Lex is partitioning the input stream, not searching for all possible matches
  - i.e. each character is accounted for once and only once
- When more than one expression can match the current input,
  - The longest match is preferred
  - The rule given first is preferred
- E.g.

```
she      s++;
he       h++;
.        |
\n       ;
```

s=1
h=1

```
~~she      {s++; REJECT;}~~
he       {h++;}
.        |
\n       ;
```

s=1
h=2

Input stream: … Is God a she, he or an it?

Lex Predefined Variables

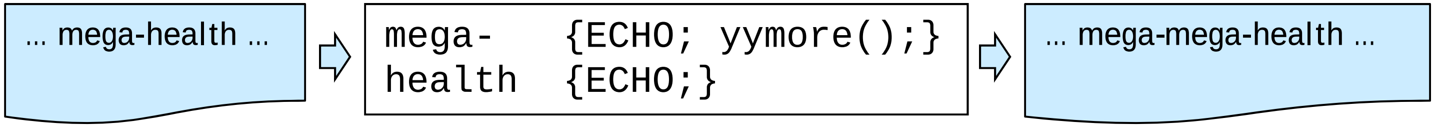
- yytext -- a string containing the lexeme
- yyleng -- the length of the lexeme
- yyin -- the input stream pointer
  - the default input of default main() is **stdin**
- yyout -- the output stream pointer
  - the default output of default main() is **stdout**.
- `linux1: %./a.out < inputfile > outfile`

- 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()
  - Append next token to yytext (instead of overwrite)
  - i.e. keep current token in yytext

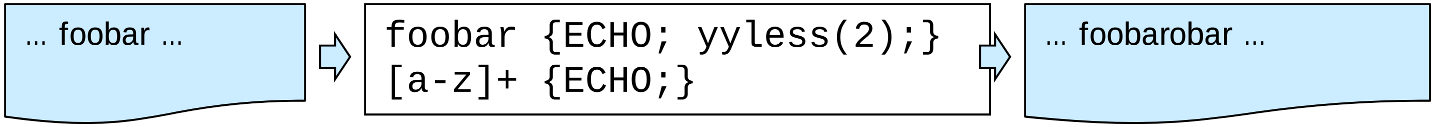


... mega-health ...

```
mega-   {ECHO; yymore();}
health  {ECHO;}
```

... mega-mega-health ...

- yyless(n)
  - return all but the first n characters in yytext to input stream



... foobar ...

```
foobar {ECHO; yyless(2);}
[a-z]+ {ECHO;}
```

... foobarobar ...

- yywrap()
  - is called whenever lex reaches an end-of-file
  - The default yywrap() 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
`BEGIN`	condition switch start condition

Recall

- The format of Lex program is

```
{definitions}
%%
{transition rules}
%%
{user subroutines}
```

Definitions Section

```
%%
[a-zA-Z_]([a-zA-Z_]|[0-9])*  ECHO;
.                            |
\n                           ;
```

Equivalent!!

```
letter      [a-zA-Z_]
digit       [0-9]
%%
{letter}({letter}|{digit})*  ECHO;
.                            |
\n                           ;
```

Definitions Section (Cont’d)

- Users may need additional options to define variables for use in his program and for use by Lex

```
%{
int s=0;
int h=0;
%}
%%
she   s++;
he    h++;
.     |
\n    ;
```

Equivalent! ⟺

```
      int s=0;
      int h=0;
%%
she   s++;
he    h++;
.     |
\n    ;
```

A Simple Example for Lex Definitions

```
%{
  int counter = 0;
%}
digit        [0-9]
space        [ \t]+
letter       [a-zA-Z]
…

%%
-{digit}+    printf(“a negative integer\n”);
\+?{digit}+  printf(“a positive integer\n”);
{letter}+    {printf(“a word\n”); counter++;}
…

%%
…
```

What can be in the definition section?

- Definitions

  name space translation

- Included code
  - space code
  - %{

    code

    %}
- Start conditions
  - %Start name1 name2 ...

The Use of Start Conditions

- Lex allows the user to explicitly declare multiple states (in definition section)
  - `%Start name1 name2 …`
- The word Start may be abbreviated to s or S
  - i.e. `%Start name`

    `= %S name`

    `= %s name`

The Use of Start Conditions (Cont’d)

- Consider the following problem:
  - copy the input to the output,
  - changing the word ***magic*** to ***first*** on every line which began with the letter ***a***,
  - changing the word ***magic*** to ***second*** on every line which began with the letter ***b***,
  - changing the word ***magic*** to ***third*** on every line which began with the letter ***c***

The Use of Start Conditions (Cont’d)

- The default state is `<INITIAL>` or `0`

```
    int flag;
%%
^a          {flag=‘a’; ECHO;}
^b          {flag=‘b’; ECHO;}
^c          {flag=‘c’; ECHO;}
\n          {flag=0; ECHO;}
magic       { switch(flag) {
                case ‘a’: printf(“first”); break;
                case ‘b’: printf(“second”);
                            break;
                case ‘c’: printf(“third”); break;
                default: ECHO;
              }
            }
```

```
%Start AA BB CC
%%
^a          {ECHO; BEGIN AA;}
^b          {ECHO; BEGIN BB;}
^c          {ECHO; BEGIN CC;}
\n          {ECHO; BEGIN 0;}
<AA>magic       printf(“first”);
<BB>magic       printf(“second”);
<CC>magic       printf(“third”);
magic           ECHO;
```

Equivalent!!

Any rule not beginning with the <> prefix operators is always active

The Use of Start Conditions (Cont’d)

- The default state is `<INITIAL>` or `0`

```
%Start COMMENT
%%
<COMMENT>.          ;
<COMMENT>”*/”       BEGIN INITIAL;
<INITIAL>.          ECHO;
<INITIAL>”/*”       BEGIN COMMENT;
```

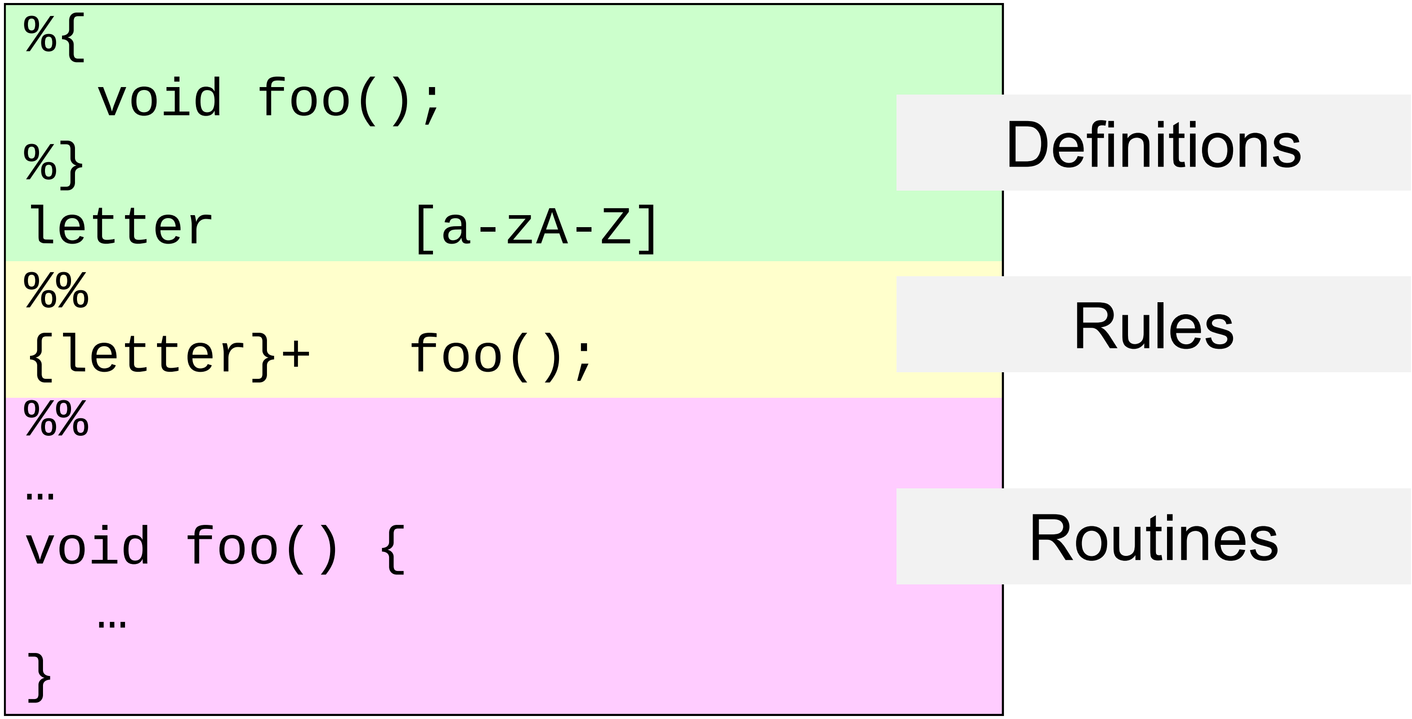
```
/* comments */
a = b + c;   /* another comment */

↓↓

a = b + c;
```

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() {
   …
}
```

Definitions

Rules

Routines

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

  `gcc lex.yy.c –ll` or

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

- 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
- GNU -- flex
  http://flex.sourceforge.net/manual/
- a Win32 version of flex :
  http://gnuwin32.sourceforge.net/packages/flex.htm

  or Cygwin :
  http://sources.redhat.com/cygwin/

- Lex on different machines may generate different results

Default Rules and Actions

- The first and second section must exist, but may be empty, the third section and the second %% are optional
- If the third section dose not contain a main(), -ll (or –lfl) will link a default main() which calls yylex() then exits
- Unmatched patterns will perform a default action, which copies the input to the output

The Shortest Possible Lex File

```
%%
```

⇕ Equivalent!!

```
%%
        /*  match everything except newline */
.       ECHO;
        /*  match newline */
\n      ECHO;
%%
int main(void) {
   yylex();
   return 0;
}
```

Some Simple Lex Program Examples

- A minimum Lex program:

```
%%
```

It only copies the input to the output unchanged

- A trivial program to deletes three spacing characters:

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

- Another trivial example:

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

It deletes from the input all blanks or tabs at the ends of lines

Example 1

```
digit   [0-9]
letter  [_a-zA-Z]
%{
   int count;
%}

%%
{letter}({letter}|{digit})*  {printf(“ID:%s\n”, yytext); count++;}
…

%%
int main(void) {
   yylex();
   printf(“\n\nnumber of identifiers = %d\n”, count);
   return 0;
}
```

Example 2

```
%{
   int nchar, nword, nline;
%}
word   [^ \t\n]+

%%
{word} { nword++; nchar += yyleng; }
\n     { nline++; nchar++; }
.      { nchar++; }

%%
int main(void) {
   yylex();
   printf(“%d\t%d\t%d\n”, nchar, nword, nline);
   return 0;
}
```

Example 3

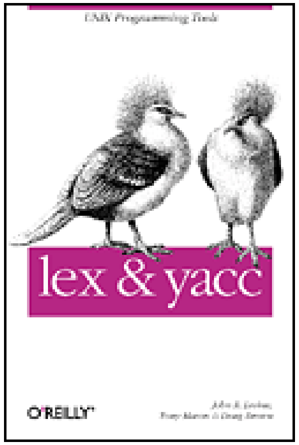
```
%{
int icount = 0;
%}

%%
int    { printf(“double”); icount++;}

%%
int main() {
  yylex();
  printf(“Change %d int to double\n”, icount);
  return 0;
}
```

Reference Books

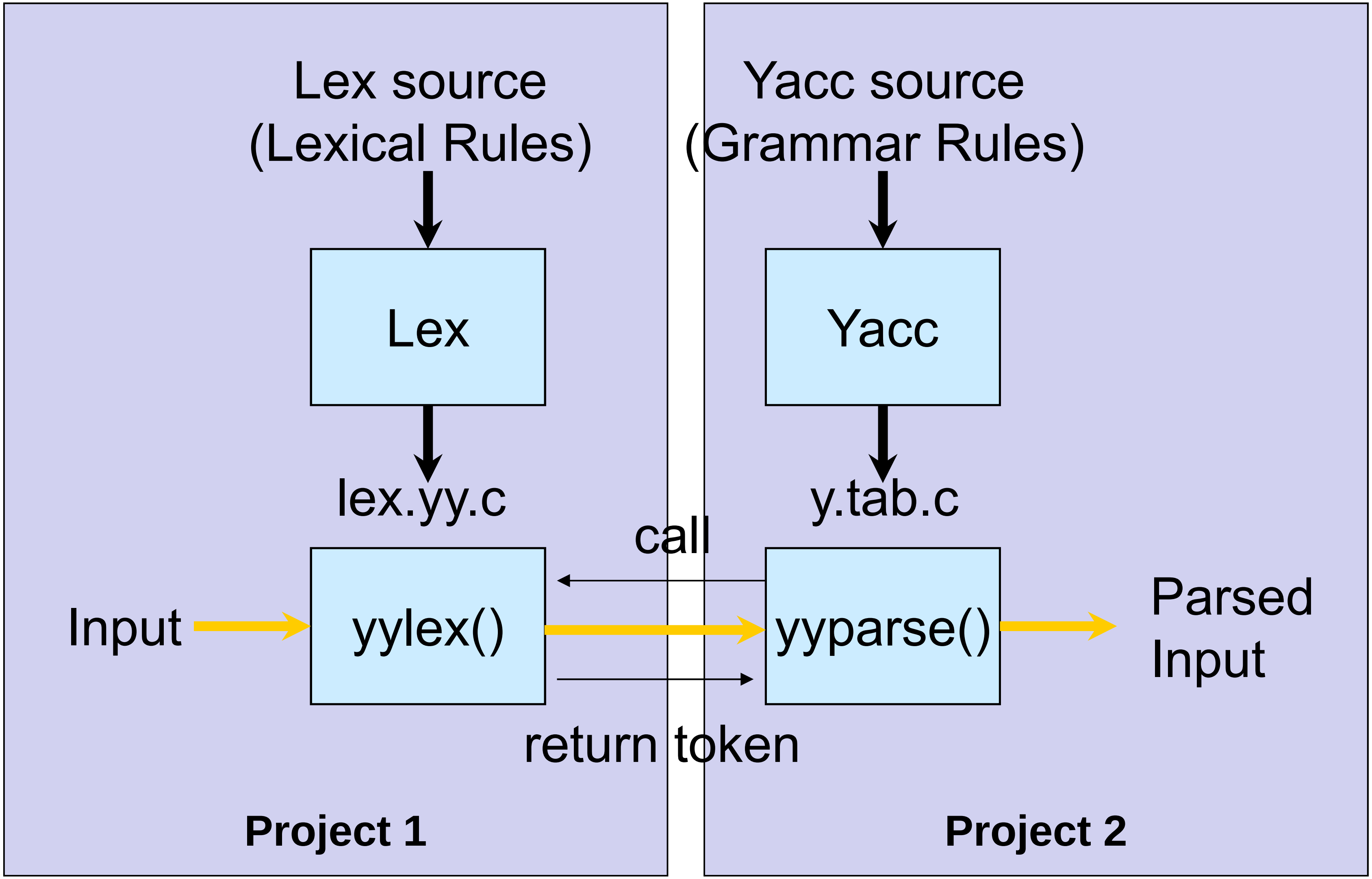
- lex & yacc, 2nd Edition
  - by John R.Levine, Tony Mason & Doug Brown
  - O’Reilly
  - ISBN: 1-56592-000-7



- Mastering Regular Expressions
  - by Jeffrey E.F. Friedl
  - O’Reilly
  - ISBN: 1-56592-257-3

- Some useful documents could be found on our course webpage

Term Project: A *P* Compiler



How to Add Comments in LEX programs!?

```
/* this is comment 1 */
%%
0 |
[1-9][0-9]* {
   printf("a integer\n");
}
  /* this is comment 2 */

(0|[1-9][0-9]*)\.[0-9]* {
   printf("a real number\n");
}

[0-]+ {
   printf("a sign number\n");
}

. {
   printf("others\n");
}
%%
/* this is comment 3 */
```