Chapter 10　lexical analyzer (lex)

Speaker: Lung-Sheng Chien

Reference book: John R. Levine, lex & yacc 中譯本, 林偉豪譯

Reference ppt: Lecture 2: Lexical Analysis, CS 440/540, George Mason university

Reference URL: http://dinosaur.compilertools.net/

Online manual: http://dinosaur.compilertools.net/flex/index.html

OutLine

- What is lex
- Regular expression
- Finite state machine
- Content of flex
- Application

Recall Exercise 7 in the midterm

Exercise 7 (lexical analyzer): given a document (text file), to find its lexical word is very important. Recall that compiler read a source file and recognize C-keyword, identifier, integer constant, floating constant and string constant. In page 97 of textbook, the author writes a piece of code to obtain an integer from standard input (you can also see Figure 10)

(1) write a driver to test function ***getInt*** in Figure 10, find all possible form of integer that it can recognize.

(2) Do exercise 5-1 in page 97 of textbook.

(3) Modify the code such that ***getInt*** reads an integer from a character string.

(4) Modify the code such that ***getInt*** reads an integer from a file.

(5) Read description **A2.3** of identifier in page 192 of recognize identifier from either character array or f

Question: can we write more compact code to obtain integers?

```
/* getInt: get next integer from input to *pn, page 97 */
int getInt( int *pn )
{
    int c, sign ;

    while( isspace( c = getch()) ) { ;  } // skip white space

    if ( !isdigit(c) && EOF != c && '-' != c ){
        ungetch(c) ;  // it is not a number
        return 0 ;
    }

    sign = ( '-' == c )? -1 : 1 ;
    if ( '+' == c  || '-' == c ){  c = getch() ;  }

    for( *pn = 0 ; isdigit(c) ; c = getch() ){
        *pn = 10 * *pn + ( c - '0' ) ;
    }
    *pn *= sign ;
    if ( EOF != c ){ ungetch(c) ; }
    return c ;
}
```

Exercise 7: remove comments in a file

in C-language, comment is delimited by a pair of **/*** and ***/** whereas in C++, comment starts from **//**. write a program to remove all comments of a given file. You can show result in screen or to another file.

Pseudo-code

```
for each line in a file
    if  line contains "//" not in a string, then
        remove remaining characters after "//".
    if  line contains "/*" not in a string, then
        find conjugate pair "*/" and remove all characters in between
endfor
```

Question: can we have other tool to identify C-comment ?

What is lex

From http://dinosaur.compilertools.net/lex/

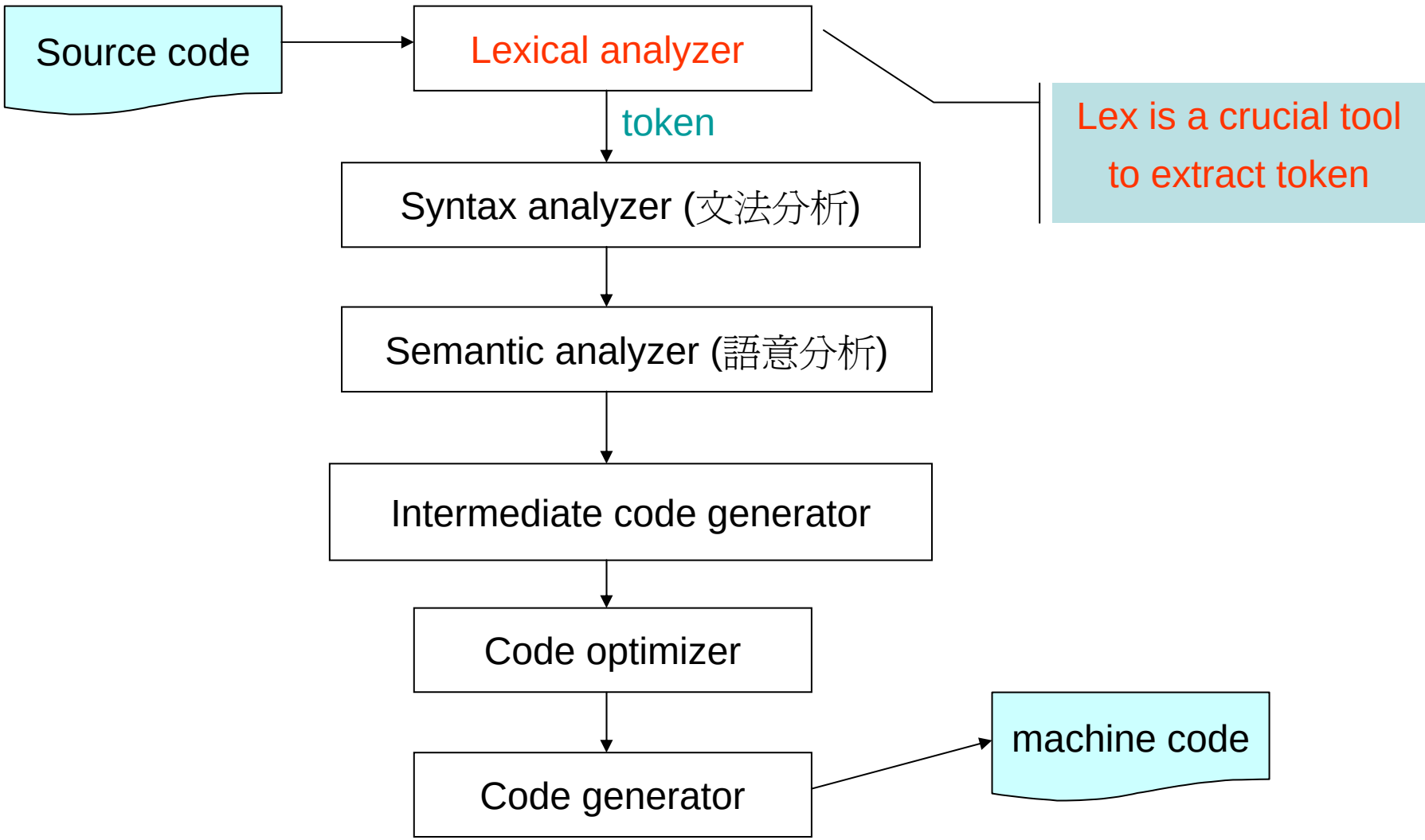
- *Lex* is a *program generator* designed for lexical (語彙的) processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions (正規表示法).
- The regular expressions are specified by the user in the source specifications given to *Lex*.
- *Lex* generates a deterministic finite automaton (DFA, 有限自動機) from the regular expressions in the source.
- The *Lex* written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions.

definition

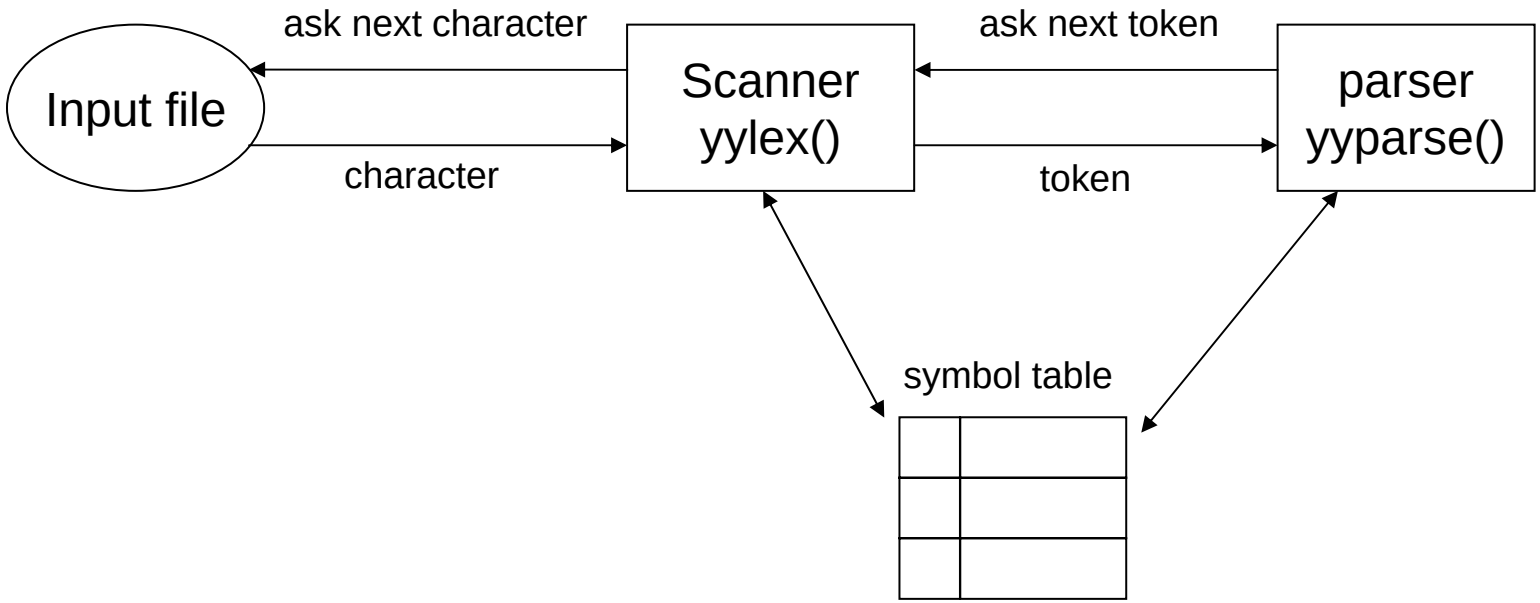
- Token: set of strings defining an atomic element with a defined meaning
- Pattern: a rule describing a set of string
- Lexeme: a sequence of characters that match some pattern

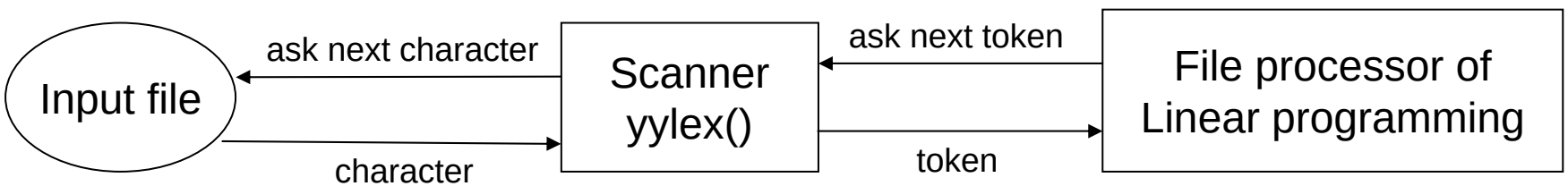
Token	Pattern	Lexeme(詞彙)
integer	(0-9)+	234
identifier	[a-zA-Z]?[a-zA-Z0-9]*	x1
string	Characters between “ “	“hello world”

Phases of a Compiler

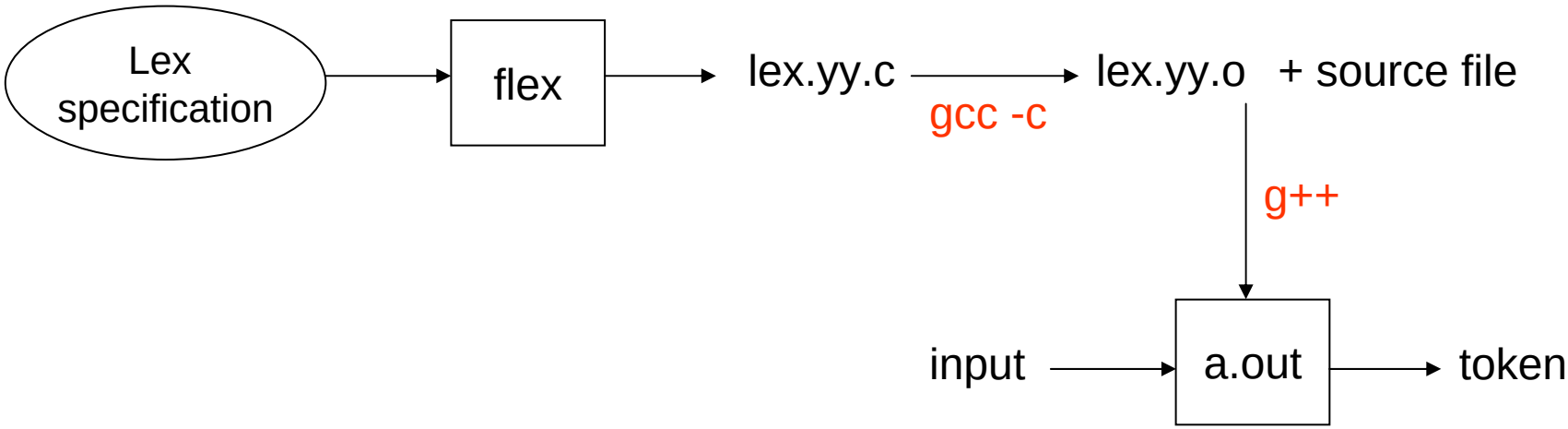


Role of scanner: find token





flex : lexical analyzer generator



- C-code *lex.yy.c* is kernel to extract token, one just need to call function *yylex()*. To use *lex.yy.c* in different platforms, we need to solve several technical problems
  - don’t use library
  - don’t include specific header file
  - mix C with C++ code

flex in RedHat 9

```
[imsl@linux imsl]$ man flex
FLEX(1)                                                    FLEX(1)

NAME
       flex - fast lexical analyzer generator

SYNOPSIS
       flex [-bcdfhilnpstvwBFILTV78+? -C[aefFmr] -ooutput -Pprefix -Sskeleton]
       [--help --version] [filename ...]

DESCRIPTION
       flex is a tool for generating scanners: programs which recognized lexi-
       cal  patterns  in text.  flex reads the given input files, or its stan-
       dard input if no file names are given, for a description of  a  scanner
       to  generate.   The  description  is  in  the  form of pairs of regular
       expressions and C code, called rules.  flex  generates  as  output  a  C
       source  file,  lex.yy.c,  which defines a routine yylex().  This file is
       compiled and linked with the -lfl library  to  produce  an  executable.
       When  the  executable  is run, it analyzes its input for occurrences of
       the regular expressions.  Whenever it finds one, it executes the corre-
       sponding C code.
```

Link with library *libfl.a*

Here is another simple example:

```
            int num_lines = 0, num_chars = 0;

    %%
    \n      ++num_lines; ++num_chars;
    .       ++num_chars;

    %%
    main()
            {
            yylex();
            printf( "# of lines = %d, # of chars = %d\n",
                    num_lines, num_chars );
            }
```

The flex input file consists of three sections, separated by a line with just %% in it:

```
    definitions
    %%
    rules
    %%
    user code
```

Example in the manual of Flex

Count number of lines and number of characters

count_line.txt

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%

\n   { ++num_lines ; ++ num_chars ; }
.    { ++num_chars ; }

%%

int main(int argc, char* argv[])
{
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

```
[imsl@linux count_line]$ ./a.out
This is a book
byebye
# of lines = 2, # of chars = 22
[imsl@linux count_line]$
```

按 enter

按 Ctrl+D

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	h	i	s		i	s		a		b	o	o	k	\n

b	y	e	b	y	e	\n

```
[imsl@linux count_line]$
[imsl@linux count_line]$ flex count_line.txt
[imsl@linux count_line]$ ls
count_line.txt  lex.yy.c
[imsl@linux count_line]$ gcc lex.yy.c -lfl
[imsl@linux count_line]$ ls
a.out  count_line.txt  lex.yy.c
[imsl@linux count_line]$
```

Generate source C-code ***lex.yy.c***

Library libfl.a

Grammar of input file of Flex [1]

Lex copy data enclosed by %{ and %} into C source file

```
 1 %{
 2
 3 #include <stdio.h>
 4 int num_lines = 0, num_chars = 0;
 5
 6 %}
 7
 8 %%
 9
10 \n   { ++num_lines ; ++ num_chars ; }
11 .    { ++num_chars ; }
12
13 %%
14
15 int main(int argc, char* argv[])
16 {
17   yylex() ;
18   printf("# of lines = %d, # of chars = %d\n",
19        num_lines, num_chars );
20   return 0 ;
21 }
```

User code

pattern action

\n { ++num_lines ; ++ num_chars ; }

. { ++ num_chars ; }

wild card character, represent any character expect line feed **\n**

grammar of input file

definition section

%%

rule section → pattern action

%%

user code

When ***pattern*** is matched, then execute ***action***

Grammar of input file of Flex [2]

lex.yy.c

```
char *yytext;
#line 1 "count_line.txt"
#define INITIAL 0
#line 2 "count_line.txt"

#include <stdio.h>
int num_lines = 0, num_chars = 0;

#line 368 "lex.yy.c"
```

```
#if YY_MAIN
int main()
        {
        yylex();
        return 0;
        }
#endif
#line 12 "count_line.txt"
```

default main

```
int main(int argc, char* argv[])
{
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

```
1  %{
2
3  #include <stdio.h>
4  int num_lines = 0, num_chars = 0;
5
6  %}
7
8  %%
9
10 \n   { ++num_lines ; ++ num_chars ; }
11 .    { ++num_chars ; }
12
13 %%
14
15 int main(int argc, char* argv[])
16 {
17   yylex() ;
18   printf("# of lines = %d, # of chars = %d\n",
19       num_lines, num_chars );
20   return 0 ;
21 }
```

Q1: can we compile lex.yy.c without –lfl ? [1]

We want to use *lex.yy.c* on different platforms (Linux and windows), to avoid specific library is lesson one.

```
[imsl@linux count_line]$ gcc lex.yy.c
/tmp/ccgm0gZ8.o(.text+0x30d): In function `yylex':
: undefined reference to `yywrap'
/tmp/ccgm0gZ8.o(.text+0xa4f): In function `input':
: undefined reference to `yywrap'
collect2: ld returned 1 exit status
[imsl@linux count_line]$
```

Library **libfl.a** contains function *yywrap()*

-lfl means “include library **libfl.a**”, this library locates in /usr/lib

```
[imsl@linux lib]$ pwd
/usr/lib
[imsl@linux lib]$ ls libf*
libfam.a     libfam.so.0.0.0     libfontconfig.so.1.0  libform.so.5.3  libfreetype.so.6
libfam.la    libfl.a             libform.a             libfreetype.a   libfreetype.so.6.3.2
libfam.so    libfontconfig.so    libform.so            libfreetype.la
libfam.so.0  libfontconfig.so.1  libform.so.5          libfreetype.so
[imsl@linux lib]$ ar -t libfl.a
libmain.o
libyywrap.o
[imsl@linux lib]$
```

contains function *yywrap()*

Q1: can we compile lex.yy.c without –lfl ? [2]

count_line.txt

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%
\n      { ++num_lines ; ++ num_chars ; }
.       { ++num_chars ; }

%%

int main(int argc, char* argv[])
{
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}

/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
 *     program is end ( yylex() return 0 )
 *  if return value is 0, then tells yylex() that
 *     new file is ready, it can go on to process new token.
 *
 *  Hence if we have multiple files to be parsed, then we can use yywrap() to
 *  open file one by one
 */

int yywrap()
{
   return 1 ;  /* eof */
}
```

Implement function *yywrap* explicitly

Q2: how to process a file?

count_line.txt

```
%%
\n      {
           ++num_lines ;
           ++ num_chars ;
        }
.       {
           ++num_chars ;
        }

%%

int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over program name*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}

/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
```

lex.yy.c

```
/* Translate the current start state into a value that can b
 * to BEGIN to return to the state.  The YYSTATE alias is fo
 * compatibility.
 */
#define YY_START ((yy_start - 1) / 2)
#define YYSTATE YY_START

/* Action number for EOF rule of a given start state. */
#define YY_STATE_EOF(state) (YY_END_OF_BUFFER + state + 1)

/* Special action meaning "start processing a new file". */
#define YY_NEW_FILE yyrestart( yyin )

#define YY_END_OF_BUFFER_CHAR 0

/* Size of default input buffer. */
#define YY_BUF_SIZE 16384

typedef struct yy_buffer_state *YY_BUFFER_STATE;

extern int yyleng;
extern FILE *yyin, *yyout;

#define EOB_ACT_CONTINUE_SCAN 0
#define EOB_ACT_END_OF_FILE 1
#define EOB_ACT_LAST_MATCH 2
```

yyin is a file pointer in *lex*, function ***yylex()*** read characters from *yyin*

Q3: can we move function *main* to another file?

count_line.txt

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%
\n      {
           ++num_lines ;
           ++ num_chars ;
        }
.       {
           ++num_chars ;
        }

%%

/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
 *     program is end ( yylex() return 0 )
 *  if return value is 0, then tells yylex() that
 *     new file is ready, it can go on to process new token.
 *
 *  Hence if we have multiple files to be parsed, then
 *  we can use yywrap() to open file one by one
 */

int yywrap()
{
   return 1 ;  /* eof */
}
```

code block

main.cpp

```
#include <stdio.h>

extern FILE* yyin ; // yyin is declared in lex.yy.c
extern int num_lines ; // num_lines and num_chars are
extern int  num_chars ; // also declared in lex.yy.c

/* we compile lex.yy.c with gcc (C-compiler), then
   extern "C" tells compiler to treat yylex as
   C-function, NOT C++-function
 */
extern "C" {
  int yylex( void ) ;
}

int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over program name*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

```
[imsl@linux count_line3]$ flex count_line.txt
[imsl@linux count_line3]$ gcc -c lex.yy.c
[imsl@linux count_line3]$ g++ main.cpp lex.yy.o
[imsl@linux count_line3]$
```

Exercise: mix C-code with C++ code

- In this work, *lex.yy.c* is C-code and *main.cpp* is C++-code, what happens if we issue command “g++ main.cpp lex.yy.c”? That’s why we use two steps,
  step 1: gcc –c lex.yy.c
  step 2: g++ main.cpp lex.yy.o
- If we replace

```
extern "C" {
   int yylex( void ) ;
}
```

  with

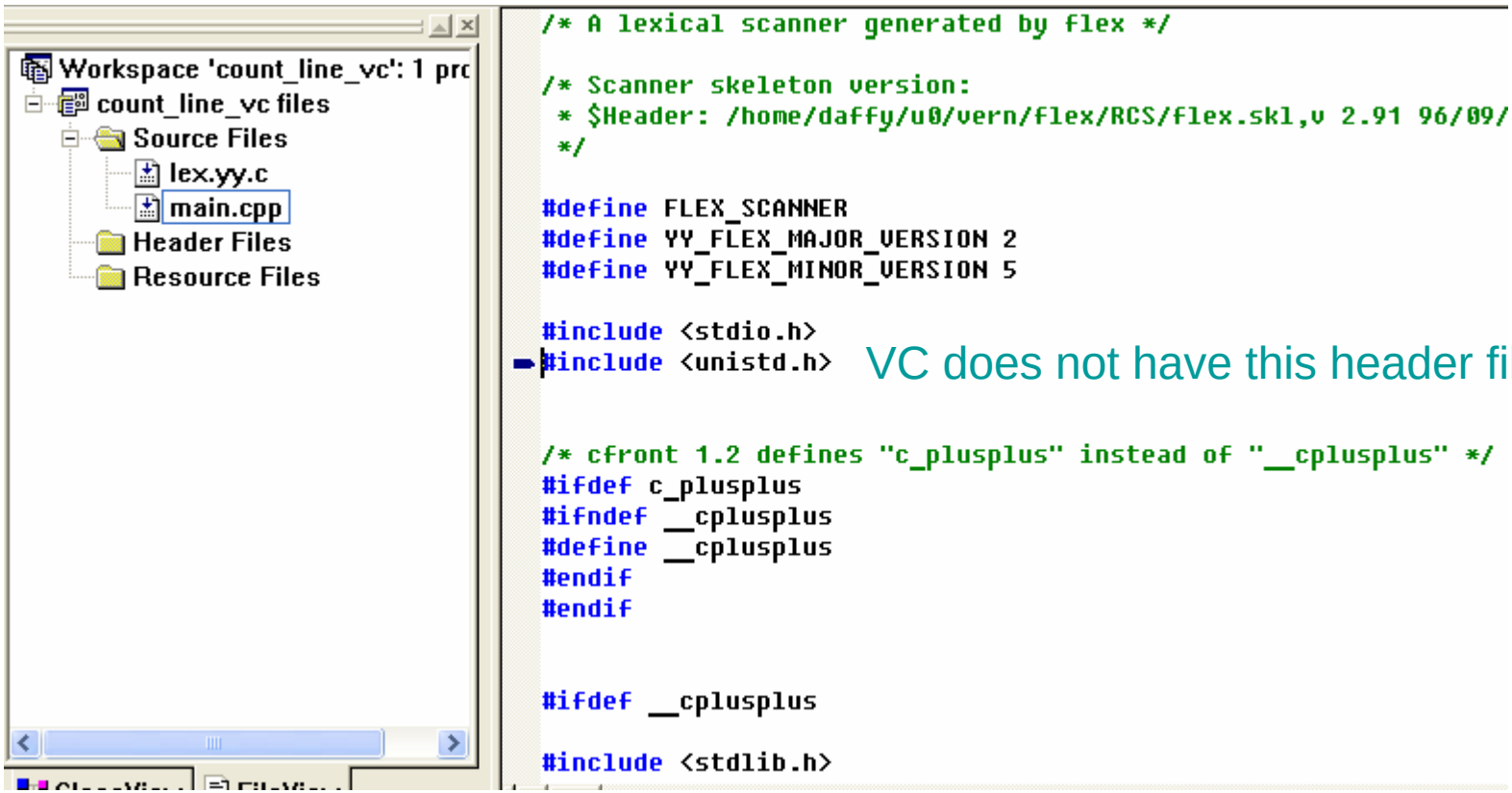
```
int yylex( void ) ;
```

  Does “g++ main.cpp lex.yy.c” work?

Q4: can we compile lex.yy.c in VC6.0? [1]

Download *lex.yy.c* and *main.cpp* in Q3 into local machine

Error occurs when compiling *lex.yy.c*




```
--------------------Configuration: count_line_vc - Win32 Debug--------------------
Compiling...
lex.yy.c
f:\course\2008summer\c_lang\example\chap10\count_line_vc\lex.yy.c(12) : fatal error C1083: Cannot open include file: 'unistd.h': No such file
Error executing cl.exe.

lex.yy.obj - 1 error(s), 0 warning(s)
```

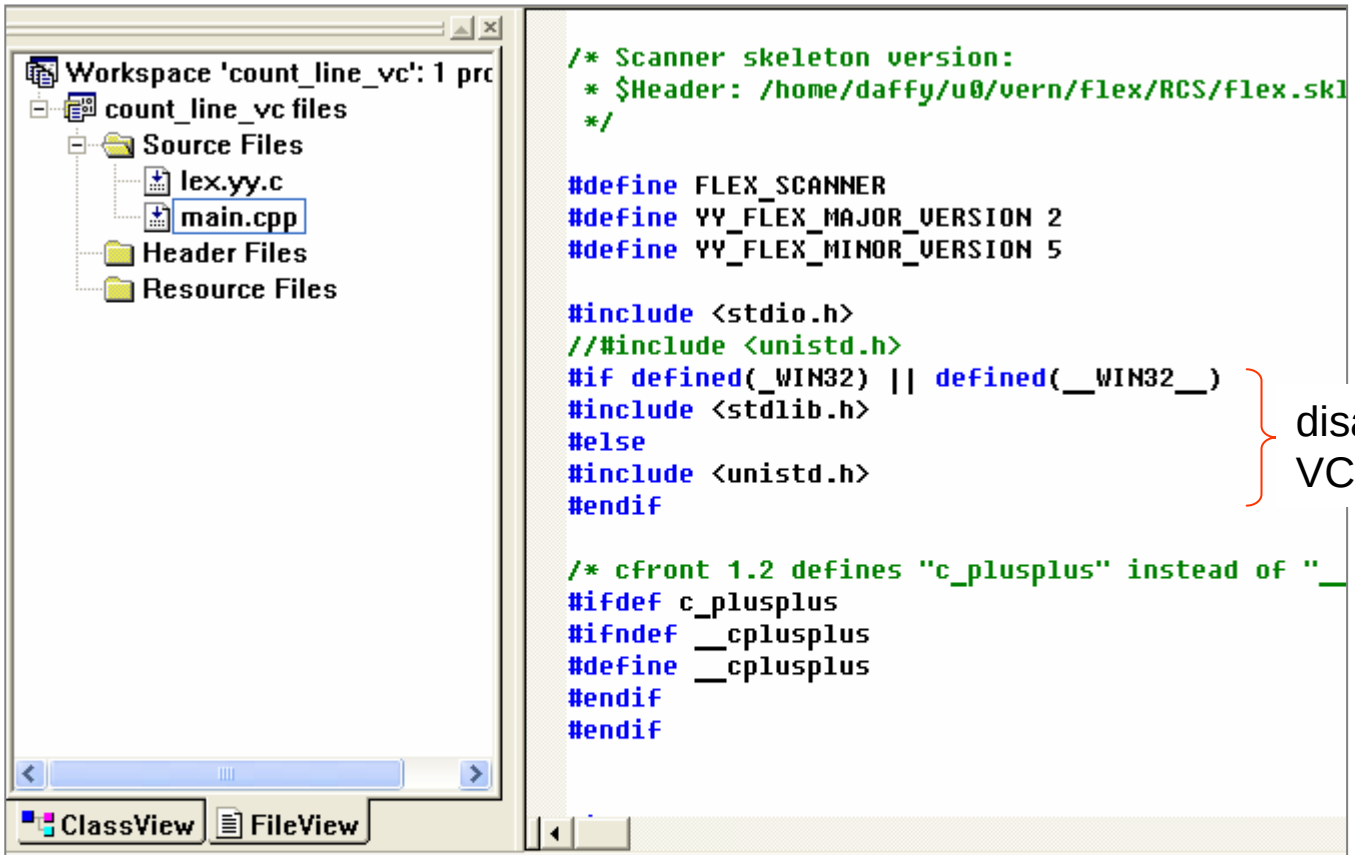
Q4: can we compile lex.yy.c in VC6.0? [2]

```
[imsl@linux include]$ pwd
/usr/include
[imsl@linux include]$ ls unist*
unistd.h
[imsl@linux include]$
```

/usr/include/unistd.h

```
/*
 *      POSIX Standard: 2.10 Symbolic Constants         <unistd.h>
 */

#ifndef _UNISTD_H
#define _UNISTD_H       1

#include <features.h>

__BEGIN_DECLS

/* These may be used to determine what facilities are present at compile time.
   Their values can be obtained at run time from `sysconf'.  */

/* POSIX Standard approved as ISO/IEC 9945-1 as of August, 1988 and
   extended by POSIX-1b (aka POSIX-4) and POSIX-1c (aka POSIX threads).  */
#define _POSIX_VERSION  199506L

/* These are not #ifdef __USE_POSIX2 because they are
   in the theoretically application-owned namespace.  */

/* POSIX Standard approved as ISO/IEC 9945-2 as of December, 1993.  */
#define _POSIX2_C_VERSION       199209L

/* The utilities on GNU systems also correspond to this version.  */
#define _POSIX2_VERSION 199209L

/* If defined, the implementation supports the
   C Language Bindings Option.  */
#define _POSIX2_C_BIND  1

/* If defined, the implementation supports the
   C Language Development Utilities Option.  */
#define _POSIX2_C_DEV   1
```

Q4: can we compile lex.yy.c in VC6.0? [3]



disable "*unistd.h*" in VC6.0

/usr/include/unistd.h

```
/* Return 1 if FD is a valid descriptor associated
   with a terminal, zero if not.  */
extern int isatty (int __fd) __THROW;
```

Error occurs since prototype of function ***isatty*** is declared in *unistd.h*

Q4: can we compile lex.yy.c in VC6.0? [4]

lex.yy.c

```
#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#define YY_FLEX_MINOR_VERSION 5

#include <stdio.h>
//#include <unistd.h>
#if defined(_WIN32) || defined(__WIN32__)
#include <stdlib.h>

int isatty (int __fd) { return 0 ;}

#else
#include <unistd.h>
#endif

/* cfront 1.2 defines "c_plusplus" instead of
#ifdef c_plusplus
#ifndef __cplusplus
#define __cplusplus
#endif
#endif


#ifdef __cplusplus
```

main.cpp

```
#include <stdio.h>

/* we compile lex.yy.c with gcc (C-compiler), then
   extern "C" tells compiler to treat yylex as
   C-function, NOT C++-function
 */
extern "C" {
    extern FILE* yyin ; // yyin is declared in lex.yy.c
    extern int num_lines ; // num_lines and num_chars are
    extern int  num_chars ; // also declared in lex.yy.c
    int yylex( void ) ;
}

int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over program name*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

OutLine

- What is lex
- Regular expression
- Finite state machine
- Content of flex
- Application

Regular expression

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

- A regular expression, often called a **pattern**, is an expression that describes a set of strings.
- The origins of regular expressions lie in automata theory and formal language theory, both of which are part of theoretical computer science . In the 1950s, mathematician Stephen Cole Kleene described these models using his mathematical notation called *regular sets*.
- *Most formalisms provide the following operations to construct regular expressions*
  - alternation: A vertical bar separates alternatives. For example, gray|grey can match "gray" or "grey".
  - grouping: use parentheses to define the scope and precedence of the operators. For example, gray|grey and gr(a|e)y are equivalent.
  - quantification (量化): a quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur.

Syntax of regular expression [1]

metasequence	description
.	matches any single character except newline
[ ]	matches a single character that is contained within the brackets. [abc] = { a, b, c } [0-9] = {0,1,2,3,4,5,6,7,8,9}
[^]	matches a single character that is not contained within the brackets. [^abc] = { x is a character : x is not a or b or c }
^	matches the starting position within the string
$	matches the ending position of the string or the position just before a string-ending newline
{m,n}	matches the preceding element at least *m* and not more than *n* times. a{3,5} matches only “aaa”, “aaaa” and “aaaaa”, NOT “aa”
< >	在方括號中如果放的是名稱, 且放在樣式開頭的話, 代表這個樣式只用在某個開始狀態

Syntax of regular expression [2]

metasequence	description
*	matches the preceding element zero or more times ab*c matches “ac”, “abc”, “abbc”
+	matches the preceding element one or more times [0-9]+ matches “1”, “14”, “983”
?	matches the preceding element zero or one time [0-9]? matches “ ”, “9”
|	the choice (aka alternation or set union) operator matches either the expression before or the expression after the operator. abc|def matches “abc” or “def”
( )	group to be a new expression (01) denotes string “01”
\	escape character * means wild card, * means ASCII code of *
“…”	代表引號中的全部字元, 所有引號中的後設字元都失去它們特別的意義, 除 \ 之外 “/*” 代表兩個字元 / 和 *

Example: based-10 integer

one digit of regular expression *[0-9]*

positive integer is composed of many digits *[0-9]+*

*[0-9]** is not adequate, since *[0-9]** can accept empty string

we need a *sign* to represent all integers *-?[0-9]+*

Accepted string: “-5”, “1234”, “0000”, “-000”, “9276000”

Question: How to represent based-16 integer under regular expression?

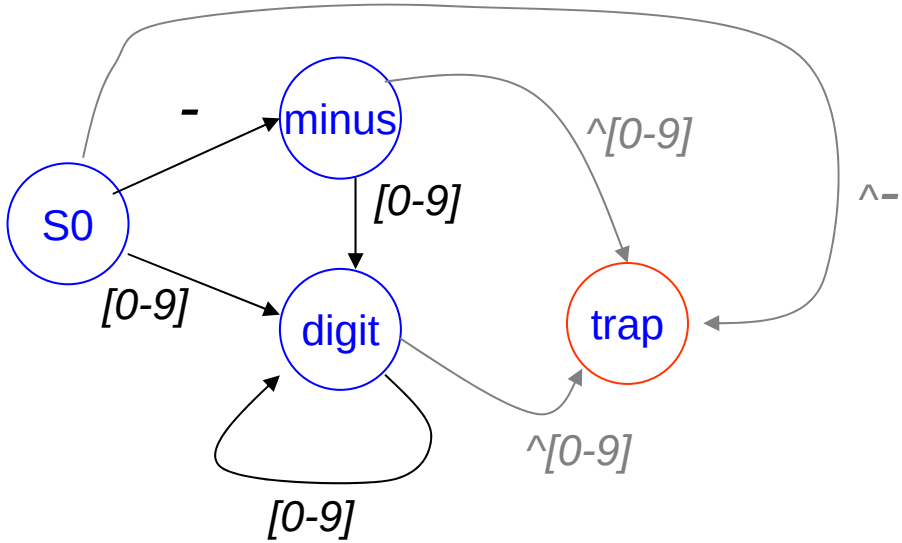
OutLine

- What is lex
- Regular expression
- Finite state machine
- Content of flex
- Application

Finite state machine (FSM)

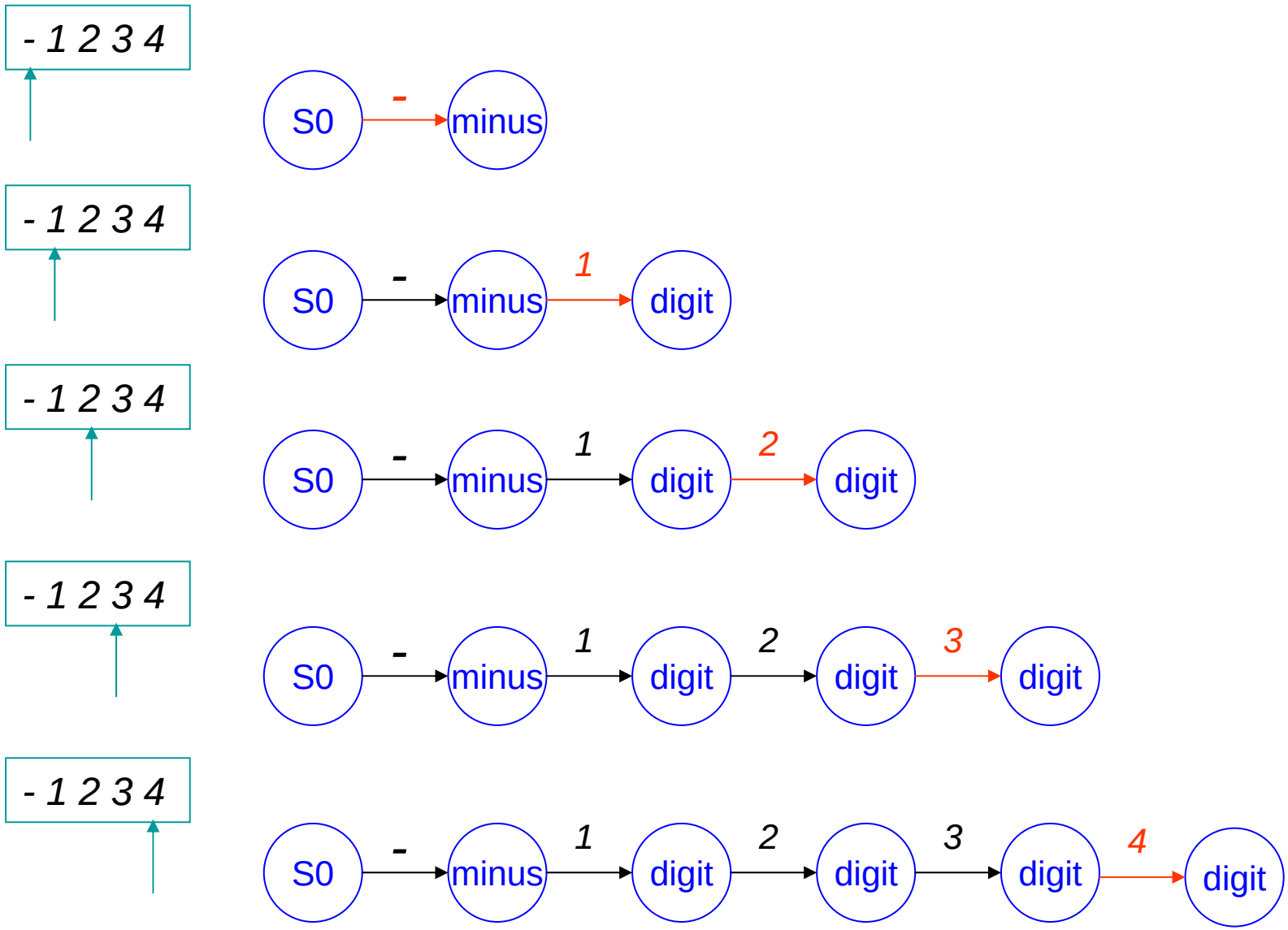
integer -?[0-9]+

state transition diagram



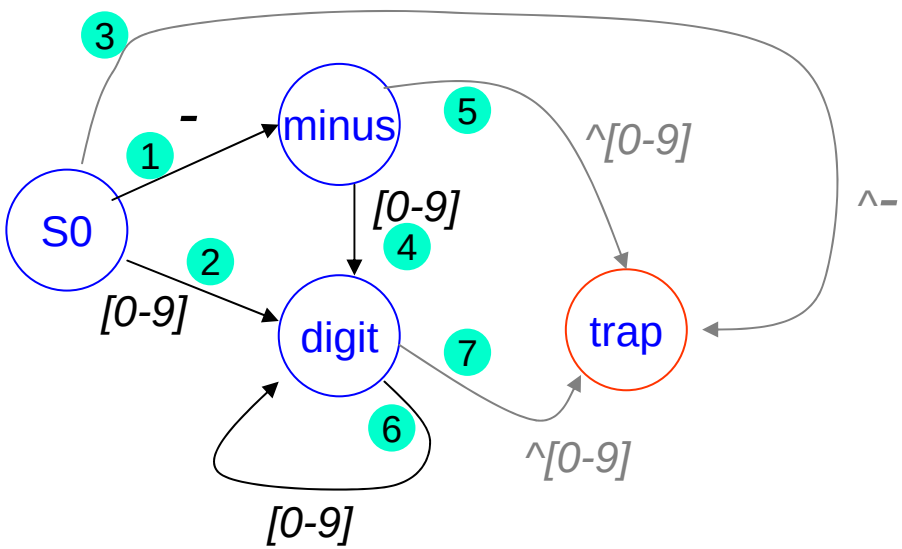
Current state	Input token (transition function)	Next state	description
S0	-	minus	S0 is initial state
	[0-9]	digit	
minus	*[0-9]*	digit	minus state recognize string “-”
digit	*[0-9]*	digit	digit state recognize string “-[0-9]+” or “[0-9]+”
trap			terminate

State sequence



Transform FSM to C-code

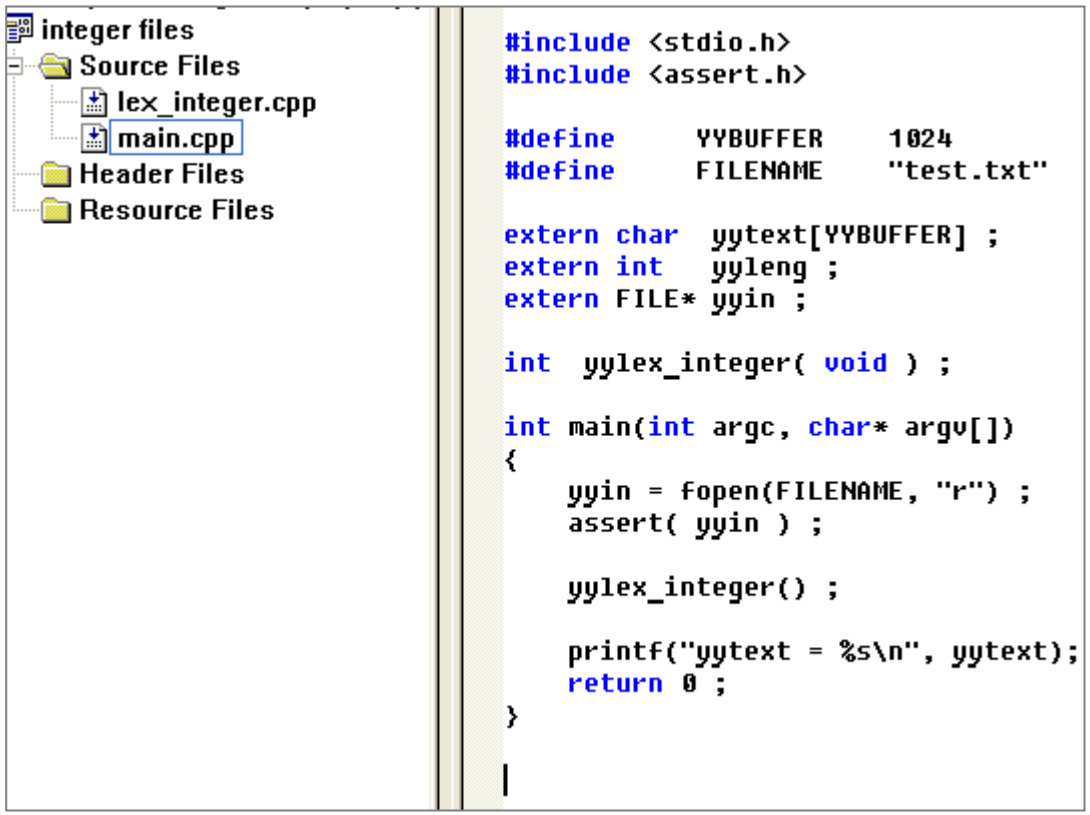
```
#include <stdio.h>
#define  YYBUFFER   1024
char  yytext[YYBUFFER] ; // store token
int   yyleng ; // length of token
FILE* yyin ; // input file pointer

// return length of token or EOF
int  yylex_integer( void )
{
    enum stateVar { S0_state, minus_state, digit_state, trap_state } ;
    stateVar  state = S0_state ; // initial state
    int c ;
    yyleng = -1 ; // no input so far

    while(1){
        if ( trap_state != state ){
            c = fgetc( yyin ) ; // read next character
            yyleng ++ ;
            yytext[ yyleng ] = c ;
            if ( EOF == c ) {
                yytext[ yyleng ] = '\0' ;
                return EOF ;
            }
        }// !trap_state
```



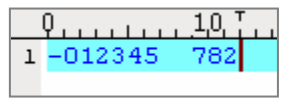

```
        switch( state ){
        case S0_state :
            if ( c == '-' ){
                state = minus_state ;
            }else if ( ('0' <= c) && ('9' >= c) ){
                state = digit_state ;
            }else{
                state = trap_state ;
            }
            break ;
        case minus_state :
            if ( ('0' <= c) && ('9' >= c) ){
                state = digit_state ;
            }else{
                state = trap_state ;
            }
            break ;
        case digit_state:
            if ( ('0' <= c) && ('9' >= c) ){
                state = digit_state ;
            }else{
                state = trap_state ;
            }
            break ;
        case trap_state:
            ungetc(c, yyin) ;
            yytext[ yyleng ] = '\0' ;
            return yyleng ;
        }
    }// forever
}
```

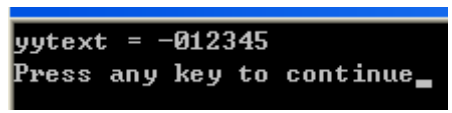
Driver to yylex_integer

main.cpp



test.txt





Exercise: extract real number

real number -?[0-9]*\.[0-9]+(([Ee][-+]?[0-9]+)?)

- why do we need a escape character for dot, “\.” ?
- Can this regular expression identify all real numbers?
- depict state transition diagram of finite state machine for this regular expression.
- Implement this state transition diagram and write a driver to test it
- Use *flex* to identify (1) integer (2) real number, note that you need to neglect space character [\t\n]

OutLine

- What is lex
- Regular expression
- Finite state machine
- Content of flex
- Application

How flex works

- *flex* works by processing the file one character at a time, trying to match a string starting from that character
  1. *flex* always attempts to match the *longest* possible string
  2. if two rules are matched (and match strings are same length), the first rule in the specification is used.

- Once it matches a string, it starts from the character after the string.

- Once a rule is matched, *flex* execute corresponding action, if no "return" is executed, then *flex* automatically matches next token.

- *flex* always creates a file named "*lex.yy.c*" with a function *yylex().*

- The *flex* library supplies a default "*main"*:

  main(int argc, char* argv[]) { return yylex() ; }

  However we prefer to write our "*main*".

Lex states

- Regular expressions are compiled to finite state machine
- *flex* allows the user to explicitly declare multiple states

```
%x  CMNT        //exclusive starting condition
%s  STRING      //inclusive starting condition
```

- Default initial state is INITIAL (0)
- Actions for matched strings may be different for different state

yylex()

- 當 token 配對到樣式後, 會執行一段 C 語言程式碼, 然後藉由 return 會讓 *yylex()* 傳回一個傳回值給呼叫程式. 等到下次再呼叫 *yylex()* 時, 字彙分析器就從上次停下來的地方繼續做下去
- *yylex()* return 0 when encounters EOF.

count_line.txt

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%
\n      {
            ++num_lines ;
            ++ num_chars ;
            return '\n' ;
        }
.       {
            ++num_chars ;
            return '\.' ;
        }

%%
```

main.cpp

```
int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over program name*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  while( yylex() ){
     printf("%s",  yytext );
  }
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

```
[imsl@linux count_line3]$
[imsl@linux count_line3]$ ./a.out test.txt
This is a book
byebye

# of lines = 3, # of chars = 23
[imsl@linux count_line3]$
```

return to caller when matching a token

call *yylex()* till End-Of-File

yytext

- 當字彙分析器辨識出一個 token 之後, token 的文字會存在 yytext 字串中, 且以空字元 (null, \0) 結尾. 且 token 的長度記錄在 yyleng, 即 yyleng = strlen(yytext)

- yytext 是字元陣列, 宣告爲
  extern char yytext[] ; 或
  extern char *yytext ;

- yytext 的內容在每辨識出一個新的 token 之後, 就會被更新. 假如之後想用到 yytext 的內容, 請自行複製

- 因爲 yytext 是陣列型態, 比 yytext 還長的 token 將導致 overflow. 在 *flex* 中, 預設的 I/O 暫存區是 16KB, 所以可以處理 8KB 的 token. 即便 token 是一段注解是不會產生 overflow 的問題

lex.yy.c

```
typedef unsigned char YY_CHAR;
FILE *yyin = (FILE *) 0, *yyout = (FILE *) 0;
typedef int yy_state_type;
extern char *yytext;
#define yytext_ptr yytext
```

yywrap()

- 當字彙分析器讀到檔案結尾時, 它會呼叫 yywrap() 函式來看看接下來要做什麼. 假如 yywrap() 函式傳回 0, 則字彙分析器繼續作分析 ;假如 yywrap() 函式傳回 1,則字彙分析器傳回一個 token **0** 來代表遇到檔案結尾

- 在 lex 函式庫中的標準 yywrap() 函式永遠會傳回 1, 但是你可以用自己寫的來代替它.假如 yywrap() 函式傳回 0, 表示還有其它的輸入資料, 這個時候需要先重新設定 yyin 指向新的檔案 (用 fopen 來設定)

- 在我們的 lex 輸入檔中, 我們定義 yywrap() 永遠回傳 1, 表示只有一個檔案需要處理

count_line.txt

```
/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
 *     program is end ( yylex() return 0 )
 *  if return value is 0, then tells yylex() that
 *     new file is ready, it can go on to process new token.
 *
 *  Hence if we have multiple files to be parsed, then
 *  we can use yywrap() to open file one by one
 */

int yywrap()
{
   return 1 ;  /* eof */
}
```

yyinput(), yyunput()

- *flex* 提供 yyinput() 以及 yyunput() 來包裝 input(), unput().

- unput(c) 函式會將字元 c 放回輸入資料中. 和一般 stdio 中 unputc() 函式不同的是: 你可以連續呼叫 unput() 來將一堆字元放回去.

lex.yy.c

```
#ifdef YY_USE_PROTOS
#define YY_PROTO(proto) proto
#else
#define YY_PROTO(proto) ()
#endif

#ifndef YY_NO_INPUT
#ifdef __cplusplus
static int yyinput YY_PROTO(( void ));
#else
static int input YY_PROTO(( void ));
#endif
#endif

#ifndef YY_NO_UNPUT
#ifdef YY_USE_PROTOS
static void yyunput( int c, register char *yy_bp )
#else
static void yyunput( c, yy_bp )
```

yyless(), yymore()

- 在動作程式碼中呼叫 yyless(n), 會將該規則配對到的 token 保留前 *n* 個字元, 其它的則 “放” 回去. 在判斷 token 的邊界時, 而且又不容易表示成常規表示法時很有用. yyless 和 yymore 可搭配使用, 利用 yymore 來告訴 lex 將下一個 token 附加到目前的 token 上

extract string literal

```
\"[^"]*\"    {
// How to deal with "abc\"mac"
// step 1: use rule to identify "abc\"
// step 2: check if character before last character is '\' or NOT, if so, use yyless()
//          to keep  "abc\  and call yymore() to process next string "mac" and
//          this string would be added into yytext, hence finally, yytext = "abc\"mac"
        if ( '\\' == yytext[yyleng-2] ){
          yyless( yyleng - 1 ) ;
          yymore() ; // add next string
        }else{
          return  LITERAL_ ;
        }
      }
```

傳回最後一個引號

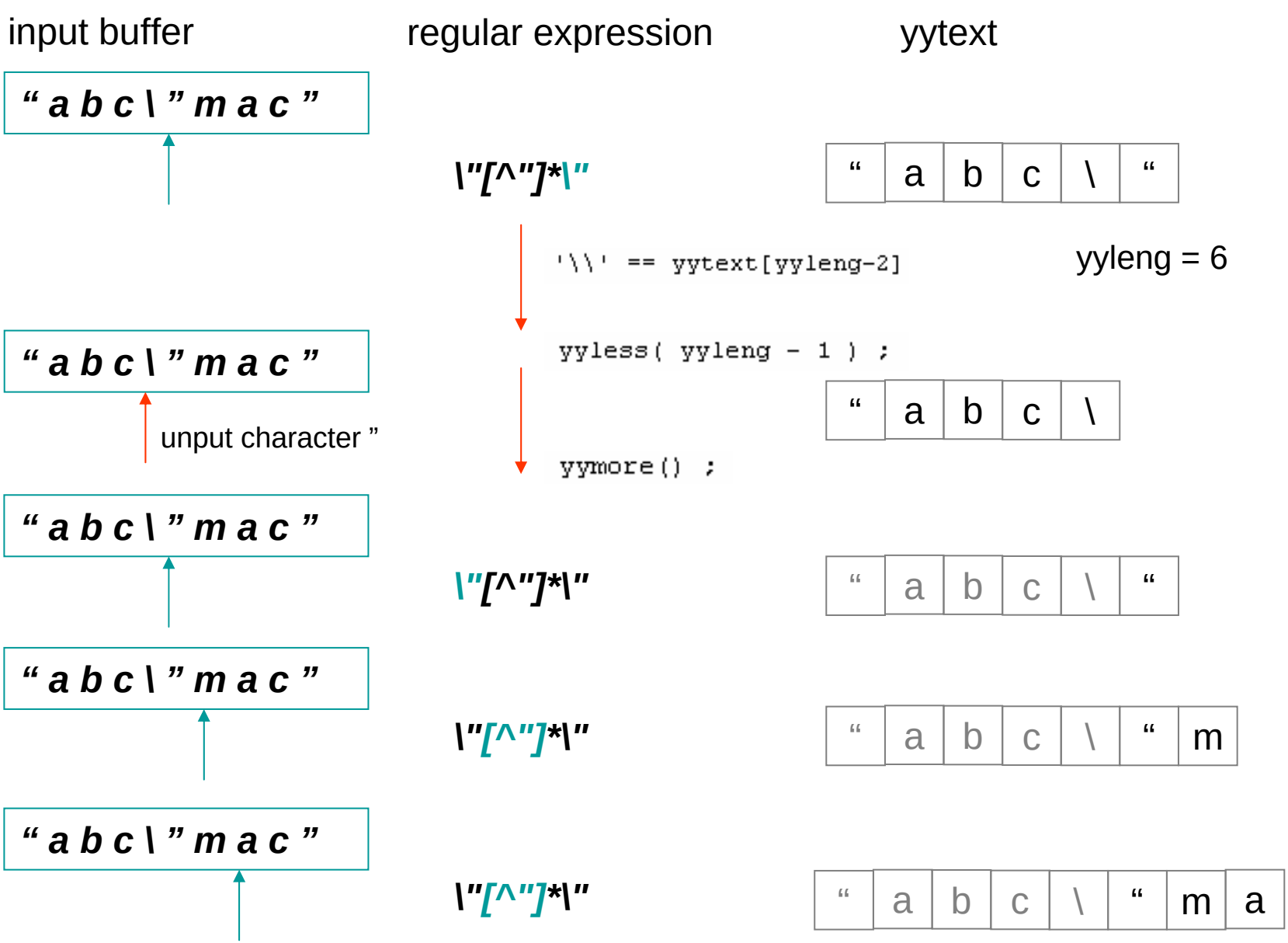
加入下一個字串

“abc\”mac” —— \"[^"]*\" ——> “abc\”

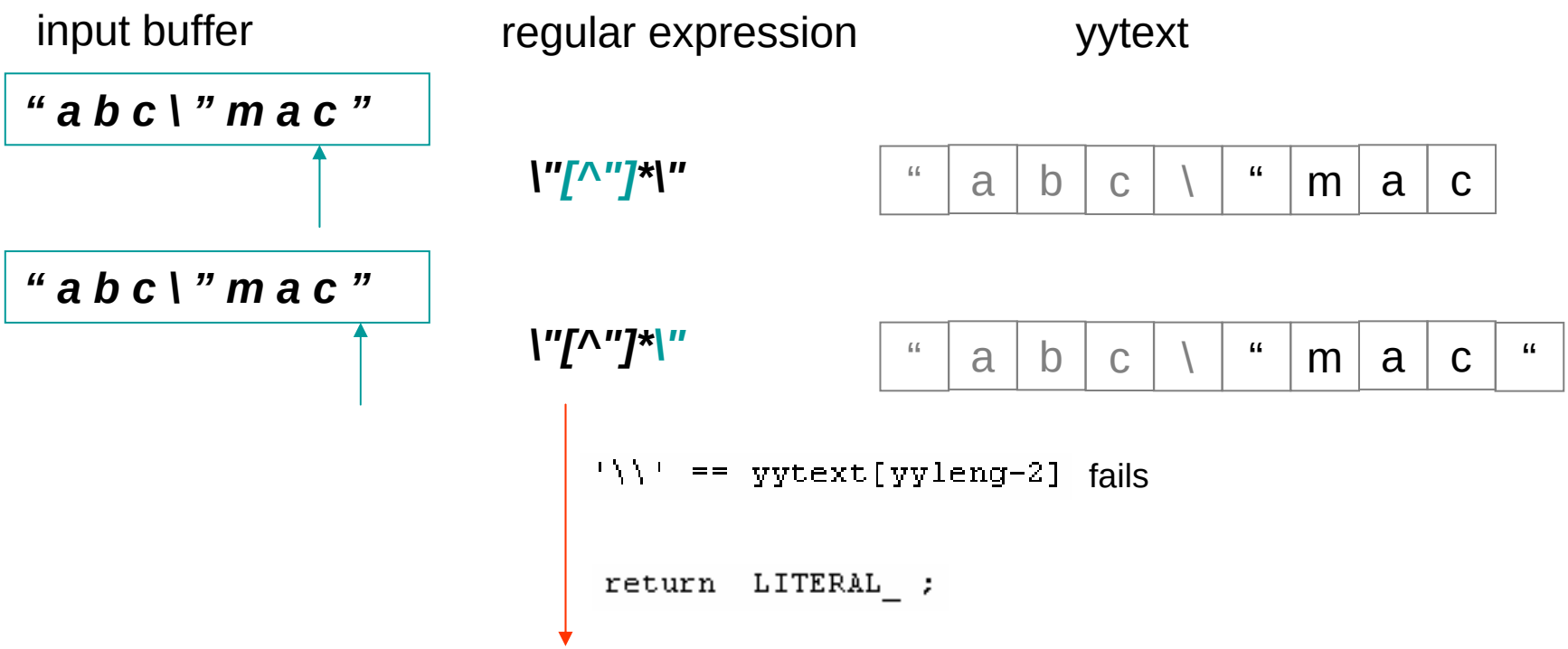
?

Analyzing process [1]

Analyzing process [2]



Analyzing process [3]



yytext | “ | a | b | c | \ | “ | m | a | c | “ | \0 |

yyleng = 10

Starting condition (開始狀態)

- *flex* provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc".
- Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either `%s' (*inclusive* start conditions) or `%x' (*exclusive* start conditions)
- Initial starting condition of *flex* is 0 (INITIAL)
- A start condition is activated using the BEGIN action. Until the next BEGIN action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive.
- If the start condition is *inclusive*, then rules with no start conditions at all will also be active.
- If it is *exclusive*, then only rules qualified with the start condition will be active.

Inclusive v.s. exclusive

The following three *lex* input are equivalent

```
%s example
%%
<example>foo  do_something();
bar      something_else();
```

```
%s example
%%
<example>foo  do_something();
<INITIAL,example>bar something_else();
```

```
%x example
%%
<example>foo    do_something();
<INITIAL,example>bar   something_else();
```

pattern ***foo*** is activated in starting condition, example

pattern ***bar*** does not specify starting conditions, then all starting conditions declared as inclusive (s) will execute pattern ***bar***

How to recognize comment in C, /* … */

main.cpp

```
#include <stdio.h>

extern "C" {
  extern FILE* yyin ;
  extern char *yytext ;
  int yylex( void ) ;
}

int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over command*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  while( yylex() ){
     printf("%s",  yytext );
  }
  return 0 ;
}
```

comment.txt

```
%{
#include <stdio.h>
%}

%x CMNT
%%
"/*"         { BEGIN CMNT ; }
<CMNT>.
<CMNT>\n
<CMNT>"*/"   { BEGIN INITIAL ; }

.        {
            return '\.' ;
         }

%%

int yywrap()
{
   return 1 ;  /* eof */
}
```

CMNT is an exclusive starting condition

If read **/***, change to *CMNT*

If read ***/**, back to *INTIAL*

test.txt

```
/*** // comment 1*/
gogo    /* comment 2*/
This is a book
/** comment 3
    continue ***/
byebye
```

```
[imsl@linux commentl]$
[imsl@linux commentl]$ ./a.out test.txt

gogo
This is a book

byebye

[imsl@linux commentl]$
```

Can you explain output?

Exercise

- C++ support another kind of comment, starting by //, write a regular expression to recognize this kind of comment and build it into *flex* input file. Write a C program with C-comment and C++-comment to test scanner generated by *flex*.

- Depict state transition diagram for C-comment and C++ comment, write code to implement this state transition diagram and measure program size. Do you think *flex* helps you identify C-comment very well?

- Can you have other method to identify C-comment by using *flex*?
  Hint: use *flex* to identify /*, then write code to find */ by yyinput() or input()

comment.txt

```
/*** // comment 1*/
gogo    /* comment 2*/
    // c++ comment
This is a book // C++ comment
/** comment 3
    continue ***/
byebye
```

```
[imsl@linux comment1]$
[imsl@linux comment1]$ ./a.out test.txt

gogo

This is a book

byebye

[imsl@linux comment1]$
```

OutLine

- What is lex
- Regular expression
- Finite state machine
- Content of flex
- Application
  - scan configuration file of linear programming
  - C-program analyzer

Application 1: configuration file of Linear Programming

Objective: read configuration file, extract coefficient of vector ***c***, **b** and matrix ***A***, then output ***c***, ***b***, ***A***

configure.txt

```
1
2  // minimize z = C' *x
3  <objective>
4    1*x1 + 0.5*x2 + x4
5  </objective>
6
7  // subject to Ax <= b
8  // x >= 0 is implicit
9  <constraint>
10   -2*x1 + x2 <= 5.0
11   3*x2 - x5 >= 7
12   6*x2 + 3.14*x1 = 6
13 </constraint>
14
```

$$\min z = c^T x$$

$$\text{subject to } Ax \le b, x \ge 0$$

token

<objective> <constraint>

</objective> </constraint>

x1 x2 x4 x5

integer real number

\+ - * >= <= =

C++-comment

LP.txt

You need to add rule for C++-comment

```
 1 %{
 2 #include <stdio.h>
 3 #include "y.tab.h"
 4 int  yylineno = 1 ;
 5 %}
 6
 7 ws      [ \t]+
 8 digit   [0-9]
 9 alpha   [a-zA-Z]
10 alnum   [a-zA-Z0-9]
11 sign    [+\-]
12 integer_num {sign}?{digit}+
13 exponent  [Ee]{integer_num}
14 float_num {sign}?(({digit}+\.{digit}*)|(\.{digit}+))
15 scientific_num  ({integer_num}|{float_num}){exponent}
16 identifier  ({alpha}|\_|$)({alnum}|\_|$)*
17
18 %x CMNT
19
20 %%
21 "\n"     { yylineno++ ; return '\n' ; }
22 {ws}     ; /* do nothing */
23 {identifier}     { return IDENTIFIER_ ; }
24 {integer_num}  { return INTEGER_ ; }
25 {float_num}  { return  REAL_ ; }
26 {scientific_num} { return  REAL_ ; }
27 "+"    { return '+' ; }
28 "-"    { return  '-' ; }
29 "*"    { return  '*' ; }
30 "/"    { return  '/' ; }
31 ">="     { return  GE_ ; }
32 "<="     { return  LE_  ; }
33 "="    { return  '='  ; }
34
35 "<objective>"      { return OBJECTIVE_ ; }
36 "</objective>"     { return END_OBJECTIVE_ ; }
37 "<constraint>"     { return CONSTRAINT_ ; }
38 "</constraint>"    { return END_CONSTRAINT_ ; }
39
```

definition of code of token

how many lines are processed

substitution rule

```
40 "/*"              { BEGIN CMNT ; }
41 <CMNT>.
42 <CMNT>\n
43 <CMNT>"*/"       { BEGIN INITIAL ; }
44
45 .    {
46       printf("NOT matched: line = %d\t error code = %s\n",
47             yylineno,yytext) ;
48       exit(1) ;
49     }
50 %%
51
52 int yywrap()
53 {
54   return 1 ;  /* eof */
55 }
```

driver: show all tokens [1]

y.tab.h

```
#ifndef Y_TAB_H
#define Y_TAB_H

#define INTEGER_  289
#define REAL_     290
#define LITERAL_  291
#define GE_     292
#define LE_     293
#define IDENTIFIER_   294

#define OBJECTIVE_    295
#define END_OBJECTIVE_  296
#define CONSTRAINT_   297
#define END_CONSTRAINT_   298

#endif
```

main.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"

extern "C" {
  extern FILE* yyin ;
  extern char *yytext ;
  int yylex( void ) ;
}

int main(int argc, char* argv[])
{
  int token ;
  ++argv ;
  --argc ; /* skip over command*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
```

```
  while( token = yylex() ){
    switch( token ){
    case '\n' :
        printf("\n");
        break ;
    case INTEGER_ :
        printf("INTEGER_: %d\n", atoi( yytext) ) ;
        break ;
    case REAL_ :
        printf("REAL_ : %25.15E\n", atof(yytext) ) ;
        break ;
    case IDENTIFIER_ :
        printf("IDENTIFIER_: %s\n", yytext) ;
        break ;
    case OBJECTIVE_ : case END_OBJECTIVE_ :
    case CONSTRAINT_ : case END_CONSTRAINT_ :
        printf("%s\n", yytext) ;
        break ;
    case GE_ :
        printf(">=\n");
        break ;
    case LE_ :
        printf("<= \n");
        break ;
    default:
        printf("%c\n", token );
        break ;
    }// switch(token)
  }// for each token
  return 0 ;
}
```

driver: show all tokens [2]

configure.txt

```
 1
 2 // minimize z = C' *x
 3 <objective>
 4   1*x1 + 0.5*x2 + x4
 5 </objective>
 6
 7 // subject to Ax <= b
 8 // x >= 0 is implicit
 9 <constraint>
10   -2*x1 + x2 <= 5.0
11   3*x2 - x5 >= 7
12   6*x2 + 3.14*x1 = 6
13 </constraint>
14
```

1. Space character is removed automatically

2. It is not necessary to keep space character between two tokens since flex would identify them very well

```
[imsl@linux LP]$ ./a.out configure.txt


<objective>

INTEGER_: 1
*
IDENTIFIER_: x1
+
REAL_ :     5.000000000000000E-01
*
IDENTIFIER_: x2
+
IDENTIFIER_: x4

</objective>
<constraint>

INTEGER_: -2
*
IDENTIFIER_: x1
+
IDENTIFIER_: x2
<=
REAL_ :     5.000000000000000E+00

INTEGER_: 3
*
IDENTIFIER_: x2
-
IDENTIFIER_: x5
>=
INTEGER_: 7

INTEGER_: 6
*
IDENTIFIER_: x2
+
REAL_ :     3.140000000000000E+00
*
IDENTIFIER_: x1
=
INTEGER_: 6

</constraint>
```

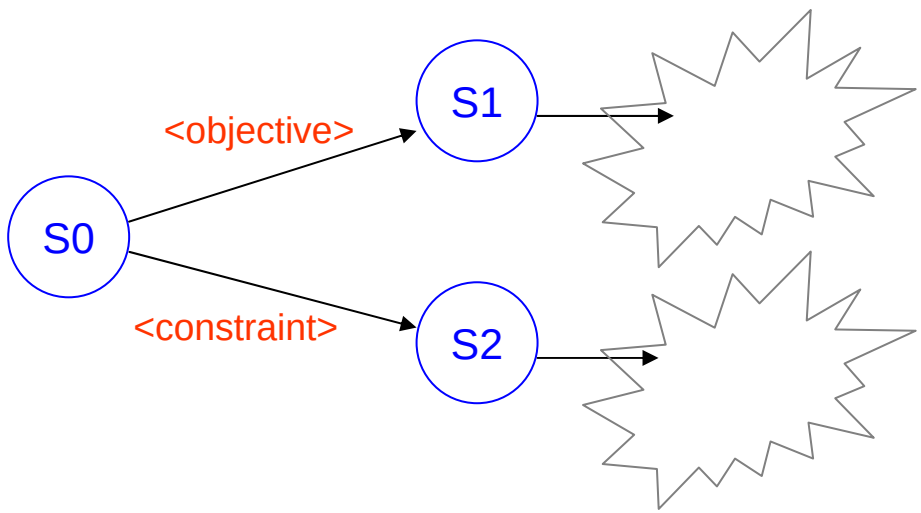
Exercise

- Complete input file for *flex* (add rule to deal with C++-comment) and test the scanner for different cases.

- Depict state transition diagram to collect information from configuration file and construct vector ***c***, ***b*** and matrix ***A***

configure.txt

```
 1
 2 // minimize z = C' *x
 3 <objective>
 4   1*x1 + 0.5*x2 + x4
 5 </objective>
 6
 7 // subject to Ax <= b
 8 // x >= 0 is implicit
 9 <constraint>
10   -2*x1 + x2 <= 5.0
11   3*x2 - x5 >= 7
12   6*x2 + 3.14*x1 = 6
13 </constraint>
14
```



Applicatoin2: C program analyzer

token	Lexeme
identifier	x1
integer	1234
real	3.14, 1.0E-5
Arithmetic operator	+, -, *, /, %
Increment operator	++, --
Arithmetic assignment operator	+=, -=, *=, /=, %=, =
Relational operator	==, !=, >, <, >=, <=
Boolean logical operator	&, |, ^
Logical operator	&&, ||
marker	( ) , [ ] , { } , , , ; , . , “ “ , ‘ ‘
Conditional operator	? :
Escape sequence	\n, \t, \r, \\, \”
comment	//, /* … */

Exercise

- Write a scanner for C-program, we have shown how to write regular expression for identifier, integer, real and comment, you need to add regular expression for
  - arithmetic operator
  - logical operator
  - relational operator
  - marker
  - string and character
  - distinguish keyword (reserved word) from identifier

  note that you need to define integer-value token for above operator in *y.tab.h*