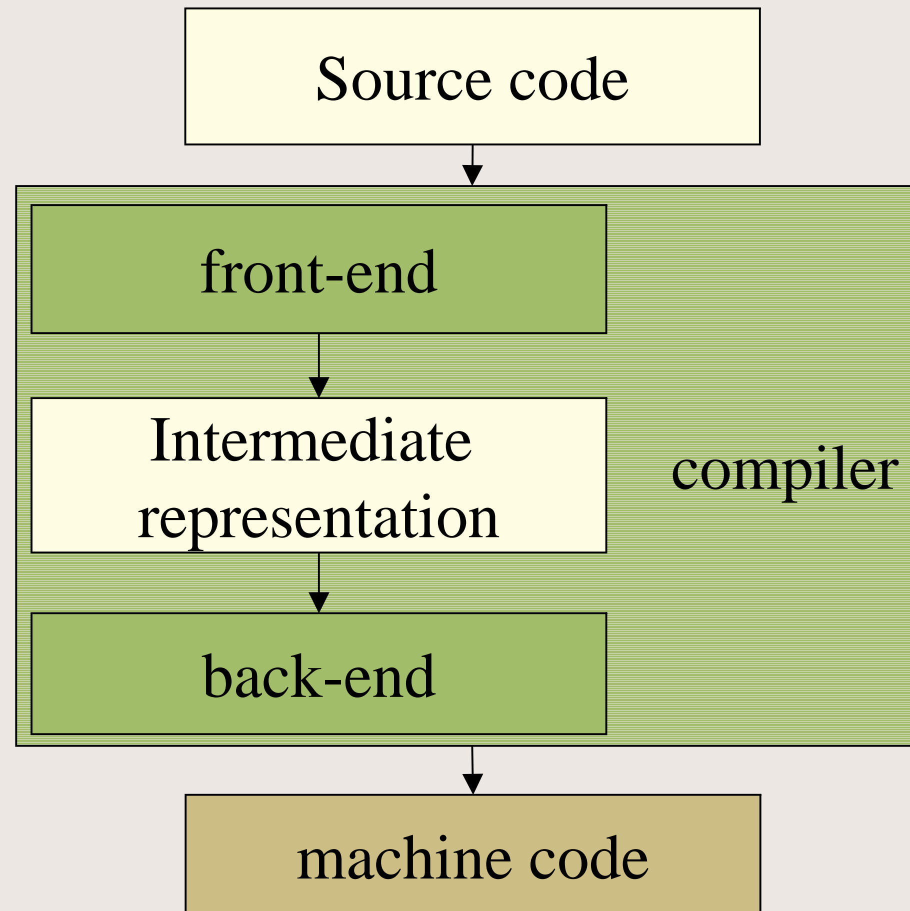
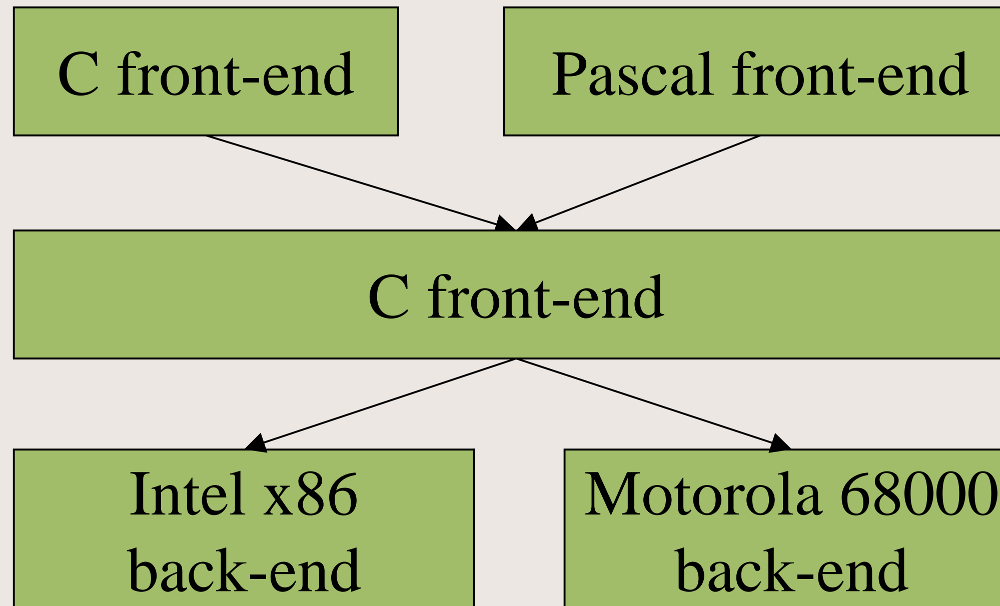
The structure of a compiler

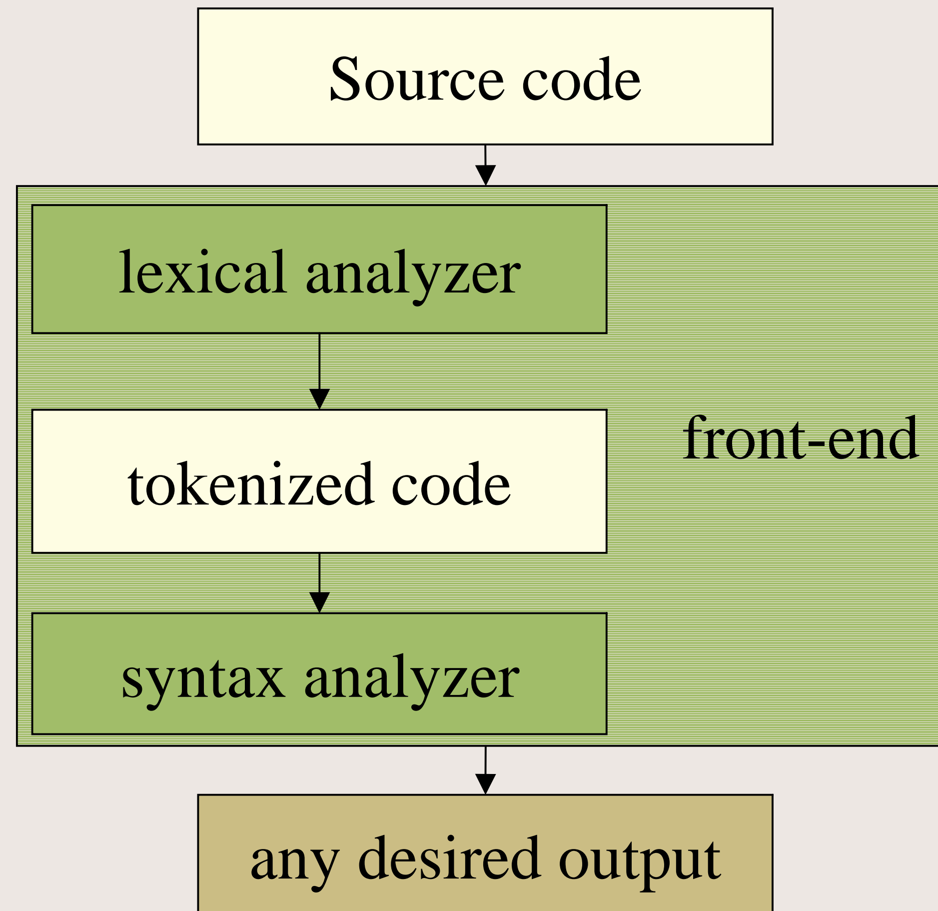


Front-end & Back-end



- The front-end abstracts from the hardware.
- The back-end abstracts from the high level language.

Front-end Structure



Relating to words or the vocabulary of a language as distinguished from its grammar and construction.

Webster’s Dictionary

Lexical Analysis

- It recognizes patterns in a stream of characters.
- A pattern represents a category of lexical elements, named “tokens”.
- Each token can have one or more attributes describing, for example, its position in the original text.

Example

WORD	a word, made by one or more alphabetical characters (in upper or lower case);
SPACE	a sequence of one or more blank spaces;
Attributes	characters constituting the token; position and length of the token expressed in number of characters;

Input stream

```
  1 2 3 4 5 6 7
1 A   s i m p l
2 e   e x a m p
3 l e
```

The result of the lexical analysis over the above text follows:

```
WORD: 'A', (1,1), 1
SPACE: ' ', (1,2), 1
WORD: 'simple', (1,3), 6
SPACE: ' ', (2,2), 1
WORD: 'example', (2,3), 7
```

The scanner

- A program that performs lexical analysis.
- Construction by hand is a tedious work.
- There are programs that generate scanners automatically.

flex: a scanner generator

- **flex** is a scanner generator, a complete rewriting of the AT&T Unix tool **lex**.
- You can find **flex** at the GNU site at the following address:

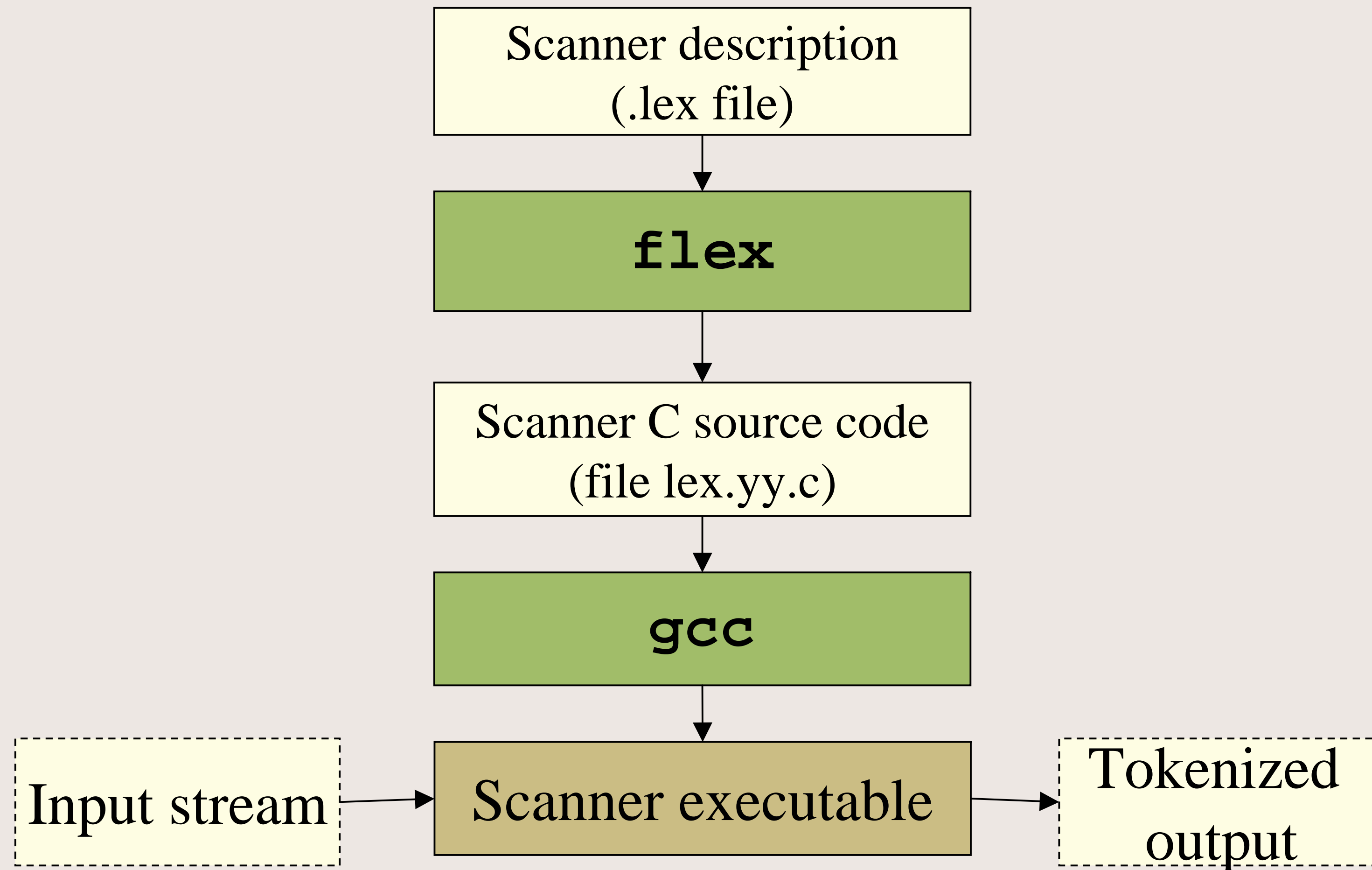
  www.gnu.org/software/flex/flex.html
- **flex** is free, and distributed under the terms of GNU General Public License (GPL).
- A useful book to understand **flex** is:

  lex & yacc, 2nd Edition

  by John Levine, Tony Mason & Doug Brown

  O'Reilly

How **`flex`** Works



```
%option noyywrap
UPPER    [A-Z]
LOWER    [a-z]
BLANK    [ ]
TAB      [\t]
NEWLINE  [\n]
%%
{UPPER}    { printf("%c",tolower(*yytext));
            /* replace uppercase letters
               with lowercase ones */}
{NEWLINE}  { printf(".\n");
            /* replace newlines with a
               dot*/}
{BLANK}+   { printf(" ");
            /* replace any number of spaces
               with a single space */}
%%
int main() { return yylex();}
```

```
%{ // A more complicated example
#define max_col 7
%}
%option noyywrap
LETTER [a-zA-Z]
SPACE [ ]
%%
%{
  int col=0;
  int row=1;
%}
{LETTER}+ {printf("WORD: '%s', (%d,%d),%d\n",
                    yytext, row, col+1, yyleng);
           col=((col+yyleng)<=max_col) ? col+yyleng :
                    row++,(col+yyleng)%max_col;
          }
{SPACE}+  {printf("SPACE: '%s', (%d,%d), %d\n",
                    yytext, row, col+1, yyleng);
           col=((col+yyleng)<=max_col) ? col+yyleng :
                    row++,(col+yyleng)%max_col;
          }
%%
int main() { return yylex();}
```

The format of a **`flex`** input file

definitions

%%

rules

%%

user code

The format of a **`flex`** input file (2)

DEFINITIONS

- Name definitions example:

  `LETTER [a-zA-Z]`

- The notation […] represents a class of character.
- In the rules section, each occurrence of **`{LETTER}`** is substituted by **`([a-zA-Z])`**.

The format of a **flex** input file (3)

RULES

- Rule example:

  {LETTER}+ {a block of C code}
- Pattern condition: a regular expression;
- Action: a fragment of C code to be executed each time a token in the input stream matches the associated pattern.

The format of a **`flex`** input file (4)

USER CODE

- User C code is copied to the generated scanner source “as is”.
- This section can contain routines called by actions, or code which calls the scanner.

The format of a **`flex`** input file (3)

ADDITIONAL CODE

```
%{
   …
%}
```

- It can be put in the definitions and in the rules sections.
- This code is copied into the generated scanner source code as is.

Regular Expression Rules

R	the regular expression R
RS	the concatenation of R and S
R | S	either R or S
R*	zero or more R's
R+	one or more R's
R?	zero or one R's
R{m,n}	a number of R's ranging from m to n
R{n,}	n or more times R's
R{n}	exactly n R's
(R)	(parentheses to override precedence)
R/S	R, but only if followed by S
^R	R, but only at beginning of a line
R$	R, but only at end of a line
<s1>R	R, but only in start condition s1
<s1,…,sn>R	R, in any of start conditions s1,…,sn
<*>R	R, in any start condition, even an exclusive one

Regular Expression Rules

x	matches the character ‘x’
.	matches any character except newline
[xyz]	any character in the given set (x | y | z)
[a-z]	any character in the given range (a | b |…| z)
[^A-Z]	any character but those in the range
{x}	expansion of x’s definition
“…”	a literal string
\x	ANSI-C interpretation of \x, if any, otherwise the literal x (to escape operators)
\0	the NUL character
<<EOF>>	the end-of-file

How the generated scanner works

- It reads the input stream, looking for strings that match any of its patterns.
- **Longest matching rule**:
  if more than one matching string is found, the rule that generates the longest one is selected.
- **First Rule**:
  if more than one string with the same length are found, the rule listed first in the rules section is selected;

How the generated scanner works (2)

- If no rules were found, the scanner performs the standard action: the next character in input is considered matched and it is copied to the output stream; then the scanner goes on.
- Once the right match is determined, the corresponding text is made available thru the global **`char *`** variable **`yytext`**, and its length is available in **`yyleng`**.

Rule actions

- Each rule can have its own action, specified as a block of C code.
- The default action is to discard matched text.
- A ‘|’ symbol in place of a block of C code instructs **`flex`** to use the same rule as the following pattern.

Special directives in actions

- **`ECHO`** copies yytext to the output.
- **`BEGIN`** sx:
  places the scanner in the corresponding start condition;
- **`REJECT`** chooses the next best matching rule;
- **`yymore()`** the next matched text is appended to yytext.
- **`yyless(n)`**
  sends back to the input stream all but the first n characters of the matched string.

Special directives in actions (2)

- `unput(c)` sends character c back to the input stream;

  WARNING: calls to unput(c) trash the contents of yytext; therefore contents of yytext must be copied before calling unput(c), if required.
- `input():` consumes the next character in input.

The **`yylex()`** scanner function

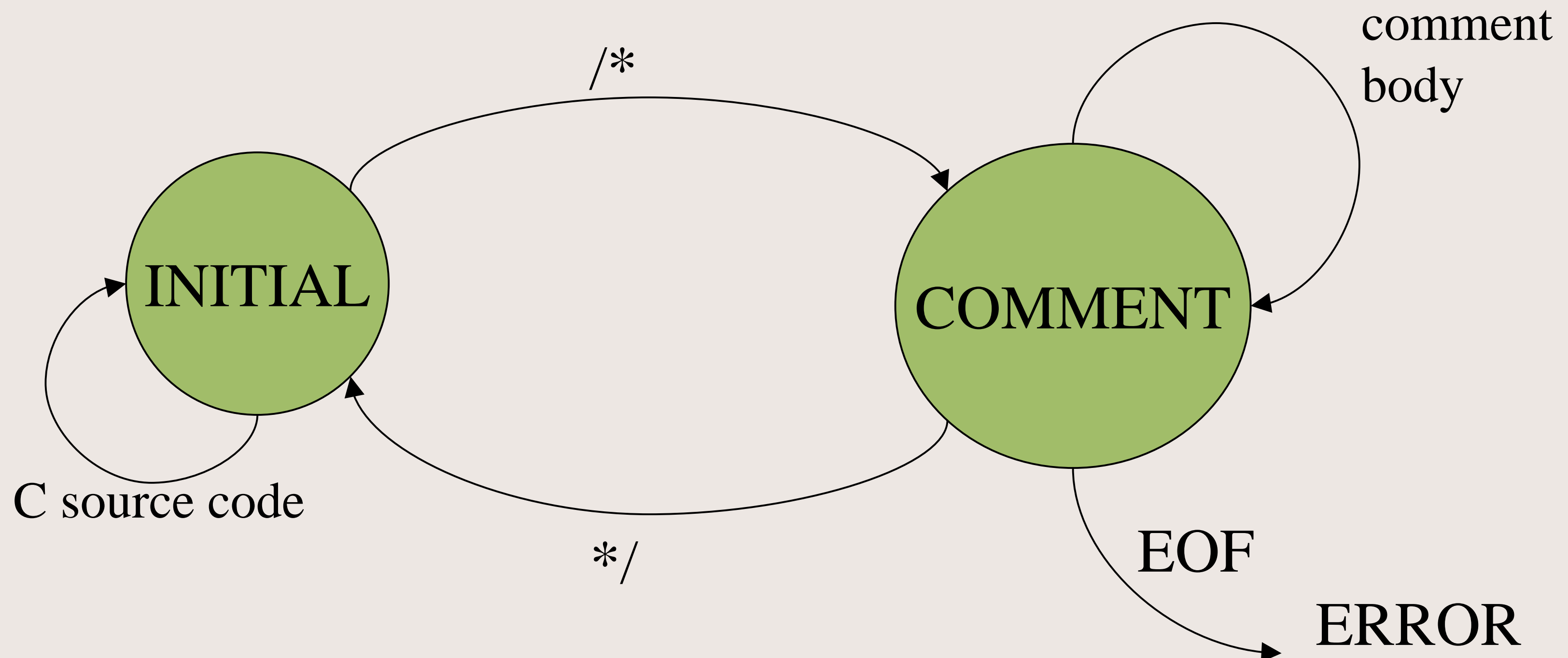
- Default signature: int yylex()
- You can modify it by, e.g., as follows:

```
#define YY_DECL float lexscan(float a)
```

- The yylex input is the global input stream **`yyin`**, which by default is assigned to **`stdin`**.
- The yylex output is the global output stream, **`yyout`** which by default is assigned to **`stdout`**.

Multiple Scanners

- Sometimes it is useful to have more than one scanner together.
- A classic example: comments in a C source code.



Start conditions

In **`flex`** it is possible to model this behavior as follows:

```
%x COMMENT
%option noyywrap
%%
<INITIAL>[^/]* ECHO;
<INITIAL>"/"+[^*/]* ECHO;
<INITIAL>"/*" BEGIN(COMMENT);
<COMMENT>[^*]*
<COMMENT>"*"+[^*/]*
<COMMENT>"*"+"/" BEGIN(INITIAL);
%%
int main(){
 return yylex();
}
```

Start conditions (2)

- A pattern preceded by an <s> start condition is active only when the scanner is in such a state.
- A start condition can be declared with %x (exclusive mode) or %s (inclusive mode).
- The special start condition <*> matches every start condition.
- The initial start condition is INITIAL.
- Start conditions are stored as integer values.
- The current start condition is stored in YY_START variable.

Example:

```
%x COMMENT
%option noyywrap
SLASH [/]
STAR [*]
%%
%{
  int nesting_level=0;
  int comment_caller[10];
%}
<INITIAL>[^/]* ECHO;
<INITIAL>{SLASH}+[^*/]* ECHO;
<INITIAL>{SLASH}{STAR} { comment_caller[nesting_level++]=YY_START;
                        BEGIN(COMMENT);
                       }
<COMMENT>[^/*]*
<COMMENT>{SLASH}+[^*/]*
<COMMENT>{SLASH}{STAR} { comment_caller[nesting_level++]=YY_START;
                        BEGIN(COMMENT);
                       }
<COMMENT>{STAR}+[^*/]*
<COMMENT>{STAR}+{SLASH} BEGIN(comment_caller[--nesting_level]);
%%
int main() {
  return yylex();
}
```

Good regular expressions

CONCISENESS

```
%x COMMENT
%option noyywrap
%%
<INITIAL>([^/]*("/"+[^*/])*)* ECHO;
<INITIAL>"/*" BEGIN(COMMENT);
<COMMENT>([^*]*("*"+[^*/])*)*
<COMMENT>"*"+"/" BEGIN(INITIAL);
%%
void main(){
   return yylex();
}
```

Good regular expressions (2)

READABILITY

```
NOT_SLASH [^/]
NOT_STAR [^*]
NOT_SLASH_STAR [^*/]
SLASH [/]
STAR [*]
%%
<INITIAL>{
   ({NOT_SLASH}*({SLASH}+{NOT_SLASH_STAR})*)* ECHO;
   {SLASH}{STAR} BEGIN(COMMENT);
}
<COMMENT>{
   ({NOT_STAR}*({STAR}+{NOT_SLASH_STAR})*)*
   {STAR}+{SLASH} BEGIN(INITIAL);
}
```

Common pitfalls

```
<COMMENT>[^*]*                    (rule 1)
<COMMENT>"*"+[^/]*                (rule 2)
```

/* * what you want */
rule 2 matches "* what you want *"
rule 1 matches "/"
results: the end of comment is lost.

```
<COMMENT>([^*]*("*"+[^*/]*)*)*        (rule 1)
```

/* * what you want */
rule 1 matches "* what you want *"
rule 1 matches "/"
results: the end of comment is lost.

Common pitfalls (2)

```
<INITIAL>"/""*"+ BEGIN(COMMENT);        (rule 1)
<COMMENT>[^*]*                          (rule 2)
```

/***************/

rule 1 matches “/***************”

rule 2 matches “/”

results: the end of comment is lost.

Stacked start conditions

```
%x COMMENT
%option noyywrap
%option stack
SLASH [/]
STAR [*]
%%
<INITIAL>{
  [^/]* ECHO;
  {SLASH}+[^*/]* ECHO;
  {SLASH}{STAR} { yy_push_state(YY_START); BEGIN(COMMENT);}
}
<COMMENT>{
  [^*/]* ;
  {SLASH}+[^*/]* ;
  {SLASH}{STAR} yy_push_state(YY_START);
  {STAR}+[^*/]* ;
  {STAR}+{SLASH} yy_pop_state();
}
%%
int main(int argc, char* argv[]){
  argv++;  argc--;
  yyin=fopen(argv[0],"r");
  yylex();
}
```

Multiple input buffers

- It is sometimes useful to switch among multiple input buffers.
- A classical example: header files included in a C source code.

```
YY_BUFFER_STATE yy_create_buffer(FILE * file, int size)

void yy_switch_to_buffer(YY_BUFFER_STATE buffer)

void yy_delete_buffer(YY_BUFFER_STATE buffer)

YY_CURRENT_BUFFER
```

Other useful options

- **`-d`** enables the debugging mode;
- **`-s`** suppresses the default rule and raises an error whenever text cannot be matched by any rule;
- **`-v`** increases the output verbosity;
- **`%option yylineno`**

  an integer variable named **`yylineno`** stores the number of line currently being read.