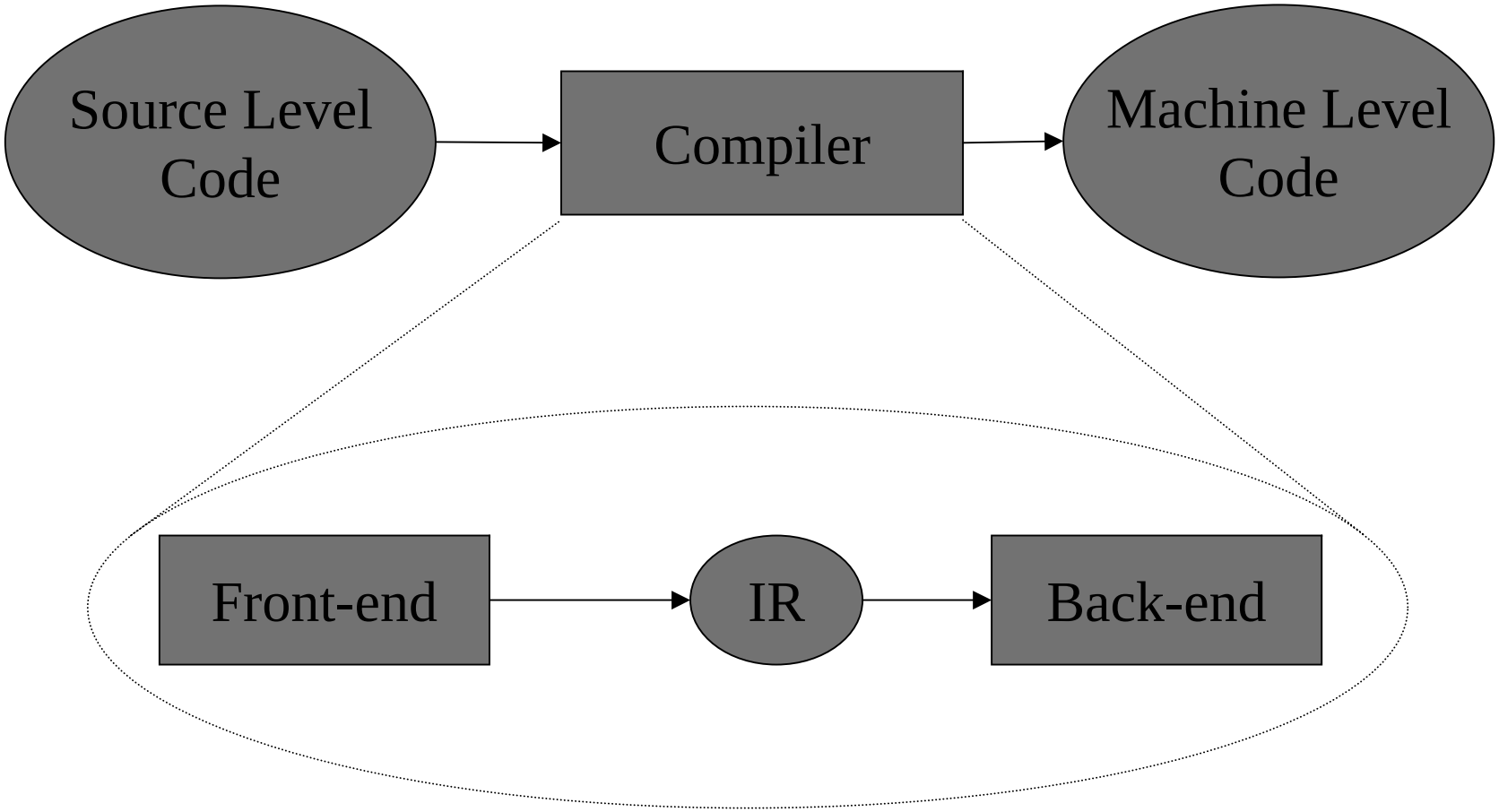
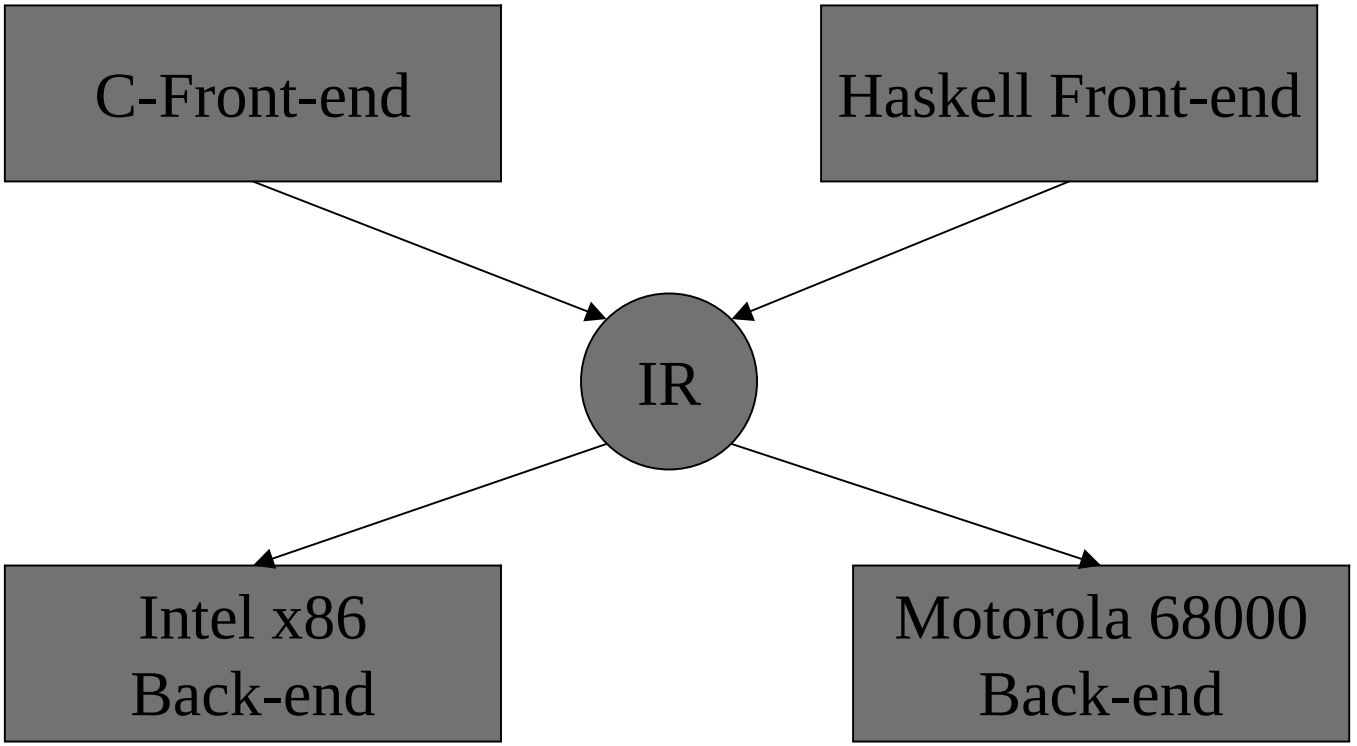
Constructing a compiler using Flex and Bison

MSc Anna Beletska, beletska@elet.polimi.it

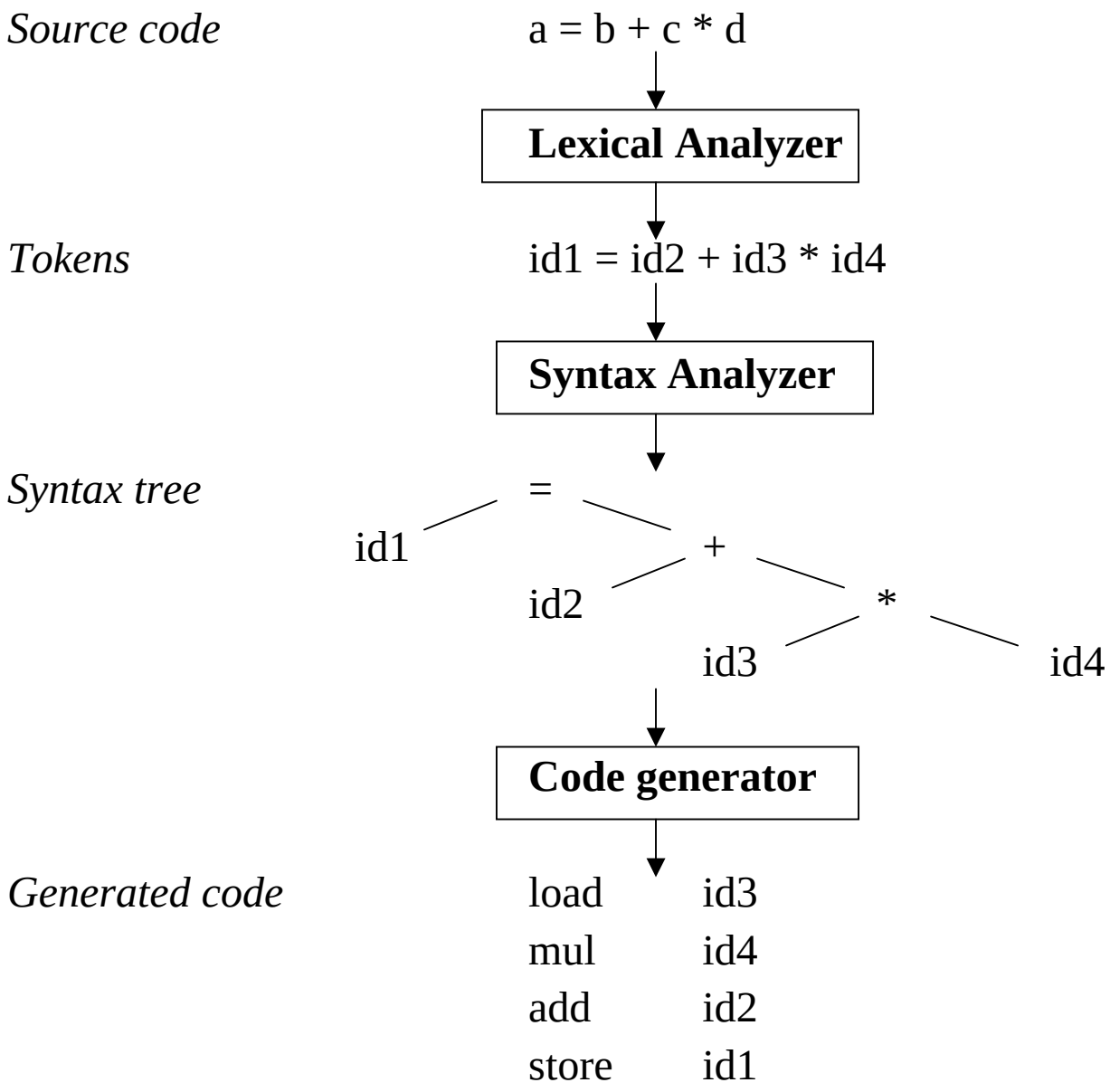
Compiler Structure



Front-end & Back-end



Compilation sequence



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

```
  0 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

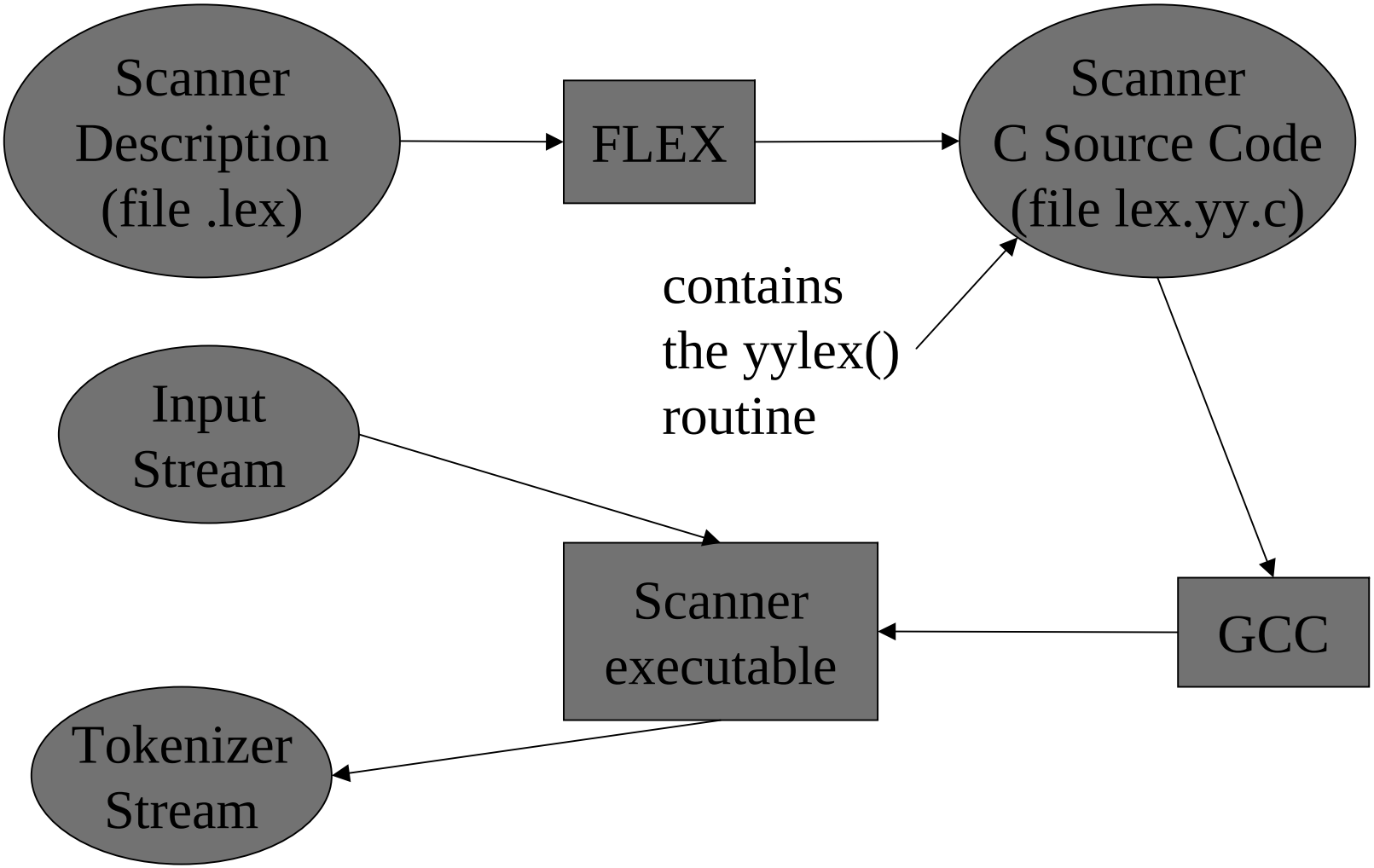
The Scanner

- A program that performs lexical analysis – recognizes lexical patterns in text.
- Construction by hand is a boring and time-consuming work.
- There are programs that generate scanners automatically.

Flex: a tool for generating scanners

- **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 understanding Flex is: lex & yacc, 2nd Edition by John Levine, Tony Mason & Doug Brown O'Reilly

How flex works



The format of a **Flex** input file

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

definitions
%%
rules
%%
user code

Definitions

Definitions contain declarations of simple ***name*** definitions to simplify the scanner specification, and declarations of ***start conditions***.

Name definitions have the form: **name definition**

DIGIT [0-9]
ID [a-z][a-z0-9]*
defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits.

{DIGIT}+"."{DIGIT}* is identical to ([0-9])+"."([0-9])*

Rules

The rules section of the flex input contains a series of rules of the form: **pattern action**

- **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.
- Rule example:
  {LETTER}+ {a block of C code}

User code

- User C code is copied to the generated scanner source "as is“ (lex.yy.c).
- This section can contain routines called by actions, or code which calls the scanner.
- The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too

Additional Code

%{

.....

%}

- Text enclosed in the definitions and in the rules sections.
- Code that is copied into the generated scanner source code as is.

```
%option noyywrap
UPPER [A-Z]
BLANK [ ]
NEWLINE [\n]

%%
{UPPER}       {printf("%c",tolower(*yytext));
/* substitute upper case letters with lowercase letters */}
{NEWLINE} {printf(".\n");
/* substitute newlines with a dot */}
{BLANK}+     {printf(" ");
/* substitute blank spaces with single space */}

%%
int main(){ yylex();}
```

Patterns: Regulal Expressions

x	matches the character `x‘
.	matches any character except newline
[xyz]	a “*character class*”, the pattern matches either ‘x’ or ‘y’ or ‘z’
[a-z]	a “*character class*” with a *range* in it, the pattern matches any letter from ‘a’ to ‘z’
[^a-z]	a “*negated character class*” , the pattern matches any character but those in the range
{*name*}	expansion of *name*'s definition
“string”	a literal string
\x	ANSI-C interpretation of \x, if any, otherwise the literal x (to escape operators like ‘*’)
\0	the NUL character
<<EOF>>	the end-of-file

Patterns: Regular Expressions

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

Patterns: Regular Expressions

In addition to characters and ranges of characters, character classes can also contain ***character class expressions*:**

[:alnum:] [:alpha:] [:blank:] [:cntrl:] [:digit:] [:graph:] [:lower:] [:print:] [:punct:][:space:] [:upper:] [:xdigit:]

These expressions all designate a set of characters equivalent to the corresponding standard C `isXXX' function. For example, the following four character classes are all equivalent:

[[:alnum:]]	[[:alpha:][:digit:]]
[[:alpha:]0-9]	[a-zA-Z0-9]

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

- 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.

  Thus, the simplest legal flex input is:

  %%

  which generates a scanner that simply copies its input (one character at a time) to its output.

How the generated scanner works

- Once the right match is determined, the corresponding text called ***token*** is made available thru the global character pointer yytext, and its length is available in yyleng.

- The ***action*** corresponding to the matched pattern is then executed.

- The remaining input is scanned for another match.

Rule Actions

- Each rule can have its own action, specified as a block of C code.
- If the action is empty, then when the pattern is matched the input token is simply discarded (the default action).
- An action consisting solely of a vertical bar ('|') means "same as the action for the next rule."

Rule Actions

- If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines.

```
pattern     {  action_line_1
               action_line_2
               ...
               action_line_n  }
```

Special directives in actions

- **ECHO** copies yytext to the output.
- **BEGIN sc** 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.

Example

```
%%
a           |
ab          |
abc         |
abcd        ECHO; REJECT;
.|\n        /* eat up any unmatched character */
```

when this scanner scans the token "abcd",
it will write "abcdabcaba" to the output.

Special directives in actions

- **unput(c)** sends character c back to the input stream; it will be the next character scanned.

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 generated scanner

The output of flex is the file `**lex.yy.c**', which contains :

- the scanning routine `**yylex()**',
- a number of tables used by it for matching tokens,
- a number of auxiliary routines and macros.

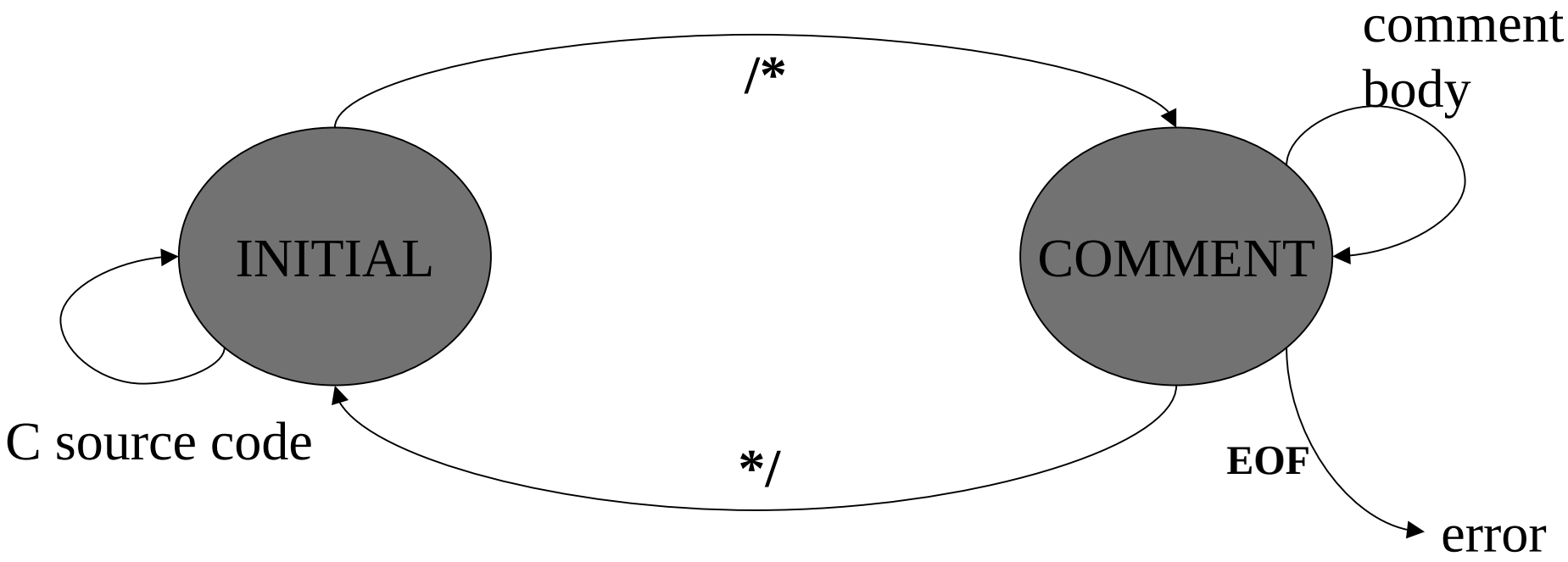
The **yylex()** scanner function

- 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.
- Classic example: comments in a source code.



Start conditions

- 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”. For example:

  <STRING>[^"]* { /* eat up the string body ... */ }

  will be active only when the scanner is in the "STRING" start condition.

- Start conditions are declared in **definitions** section of the input beginning with either `%s' or `%x' followed by a list of names.

Start conditions

- Start conditions are activated using BEGIN action.
- 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.

Start conditions

```
%x COMMENT
%option noyywrap
%%
<INITIAL>[^/]* ECHO;
<INITIAL>"/"+[^*/]+ ECHO;
<INITIAL>"/*" BEGIN(COMMENT);
<COMMENT>[^*]*
<COMMENT>"*"+[^*/]+
<COMMENT>"*"+"/" BEGIN(INITIAL);

%%
int main(int argc, char* argv[]){
argv++;  argc--;
yyin=fopen(argv[0],"r");
yylex();
}
```

```
%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(int argc, char* argv[]){
argv++;
argc--;
yyin=fopen(argv[0],"r");
yylex();
}
```

Good regular expressions

CONCISENESS

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

Good regular expressions

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);
}
```

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.