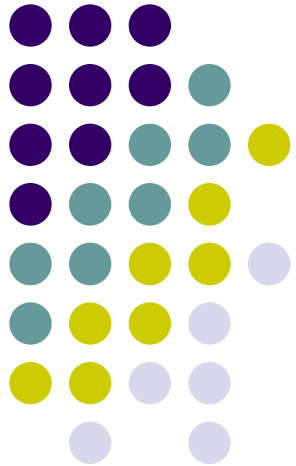
A brief [f]lex tutorial

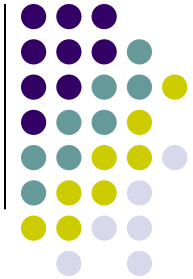
Saumya Debray

The University of Arizona

Tucson, AZ 85721

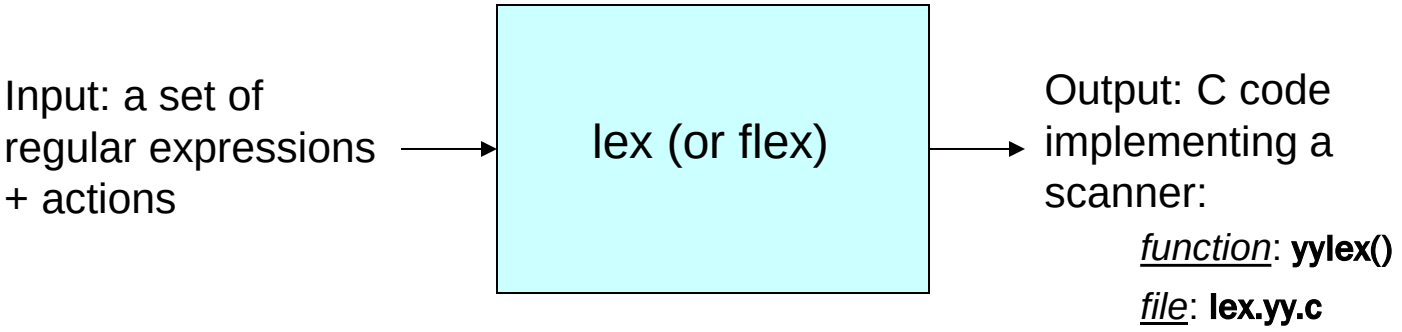


flex (and lex): Overview

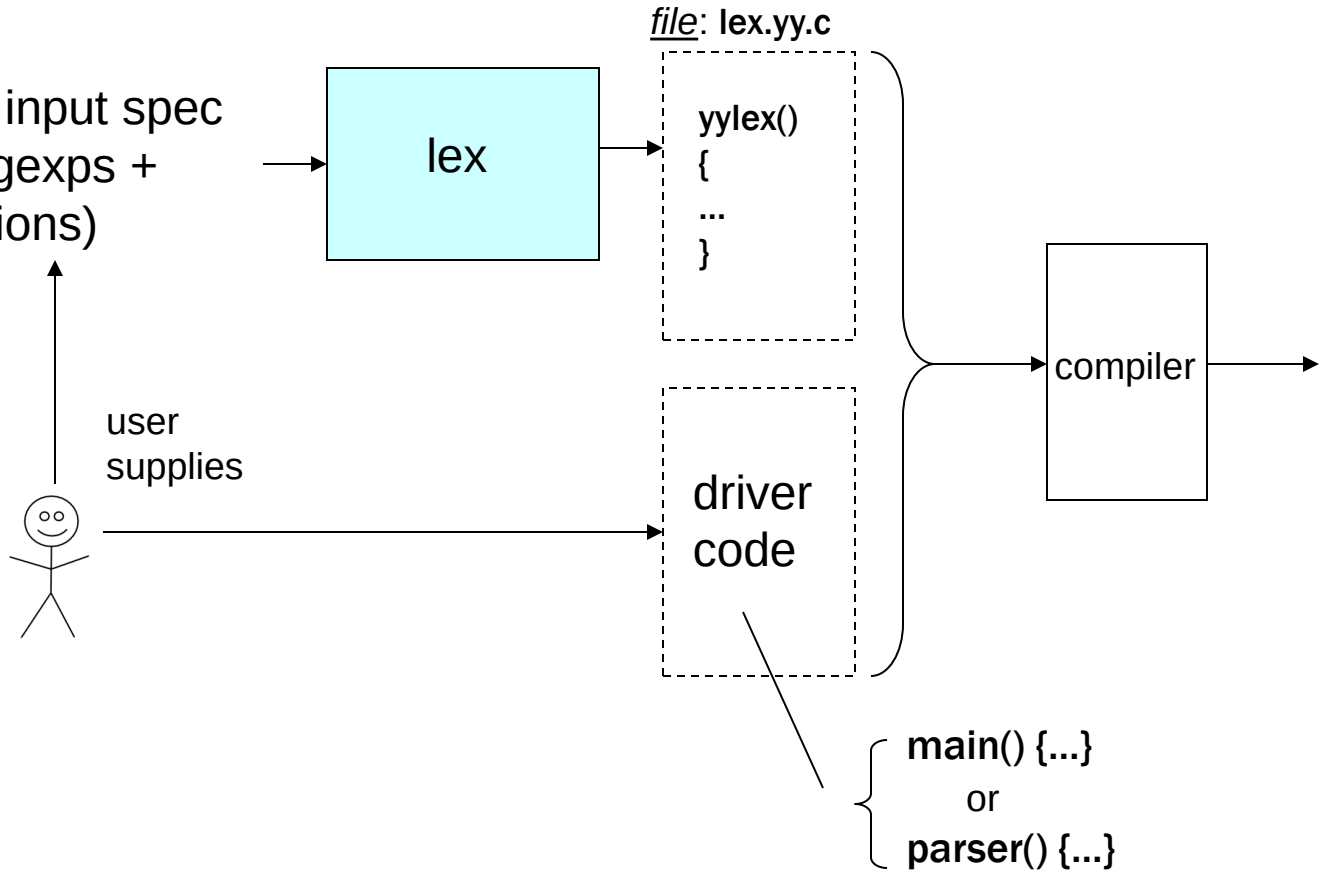


Scanner generators:

- Helps write programs whose control flow is directed by instances of regular expressions in the input stream.

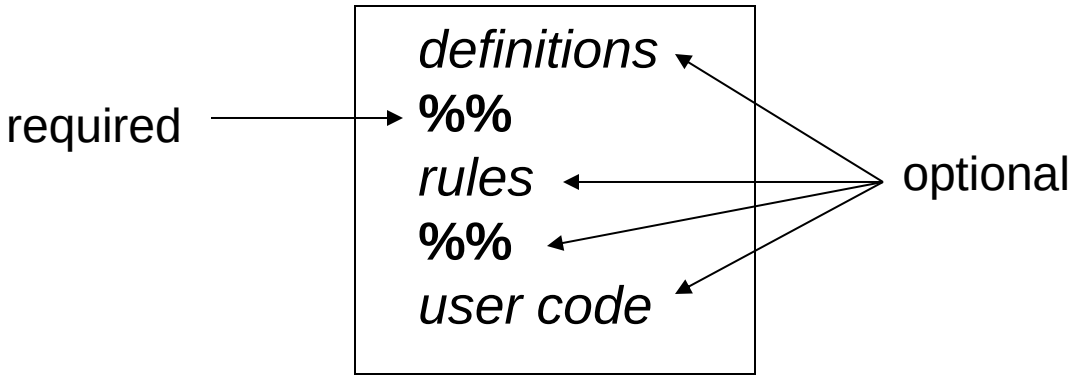


Using flex


file: lex.yy.c
lex input spec
(regexps +
actions)
lex
yylex()
{
...
}
compiler
user
supplies
driver
code
main() {...}
or
parser() {...}

flex: input format

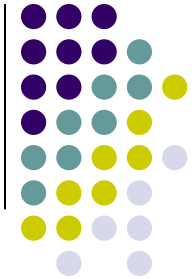
An input file has the following structure:



Shortest possible legal flex input:

```
%%
```

Definitions



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

    name *definition*

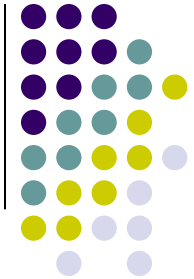
    e.g.:

    DIGIT [0-9]

    CommentStart "/*"

    ID [a-zA-Z][a-zA-Z0-9]*
  - *start conditions*
  - stuff to be copied verbatim into the flex output (e.g., declarations, **#include**s):
    - enclosed in %{ … }%, or
    - indented

Rules

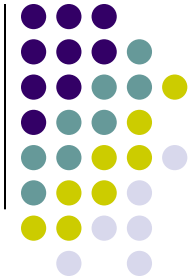


- The *rules* portion of the input contains a sequence of rules.
- Each rule has the form

  pattern *action*

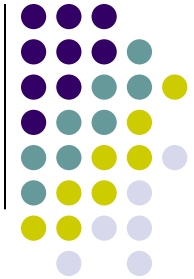
  where:
  - *pattern* describes a pattern to be matched on the input
  - *pattern* must be un-indented
  - *action* must begin on the same line.

Patterns



- Essentially, extended regular expressions.
  - Syntax: similar to grep (see man page)
  - <<EOF>> to match “end of file”
  - Character classes:
    - [:alpha:], [:digit:], [:alnum:], [:space:], etc. (see man page)
  - {*name*} where *name* was defined earlier.
- “start conditions” can be used to specify that a pattern match only in specific situations.

Example

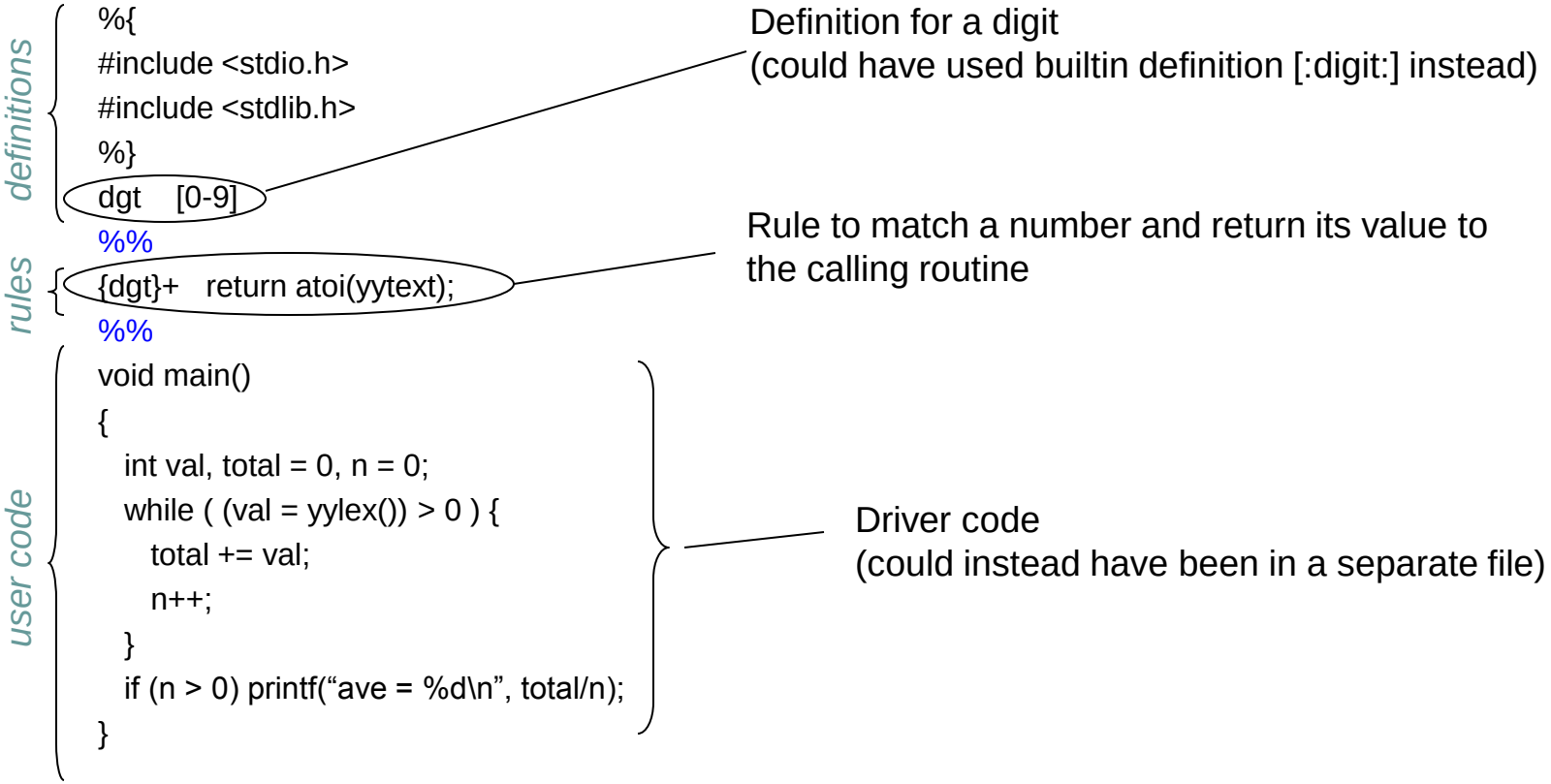


A flex program to read a file of (positive) integers and compute the average:

```
%{
#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);
}
```

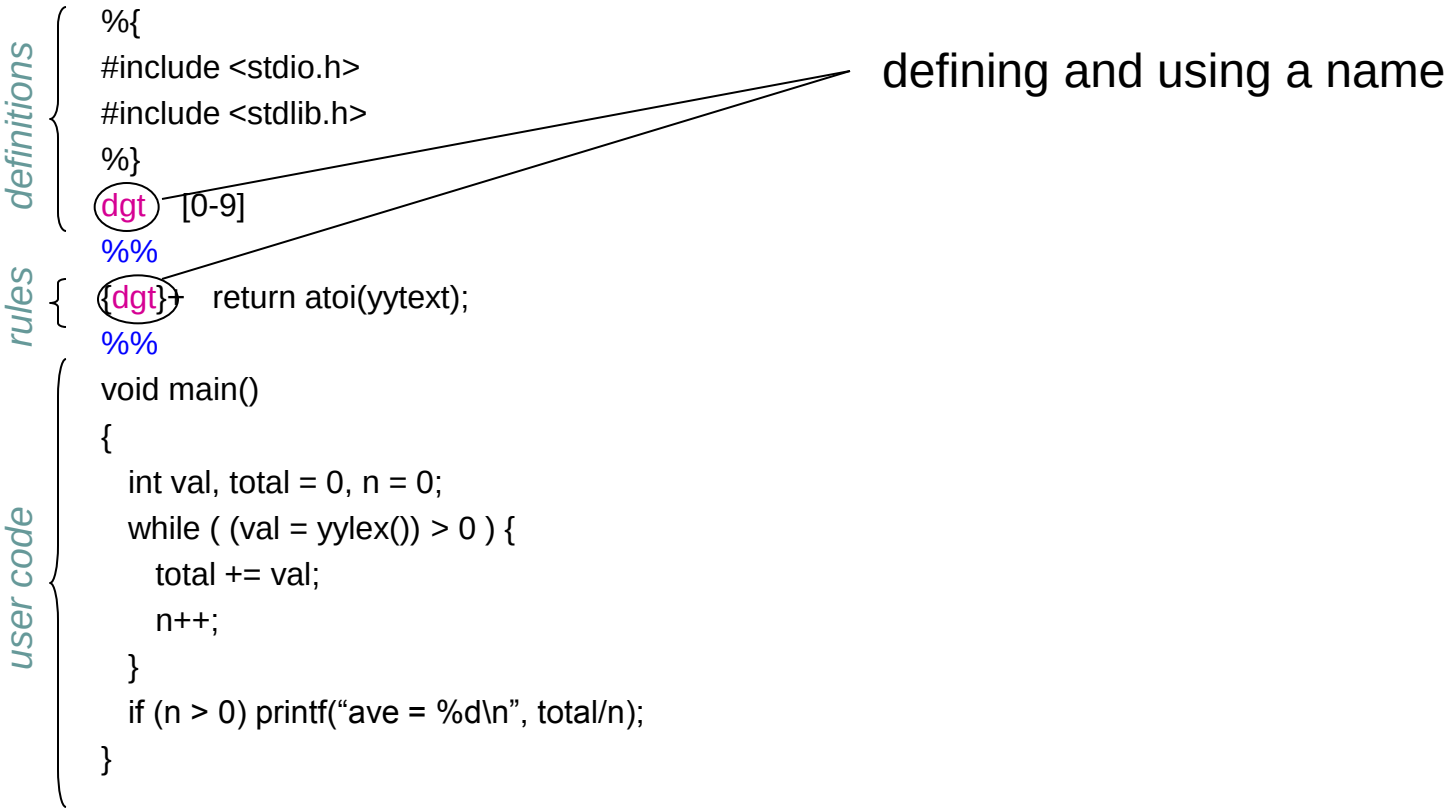
Example

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:

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

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;

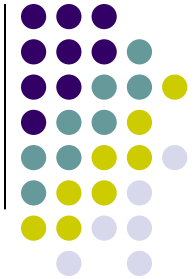
a buffer that holds the input characters that actually match the pattern

Invoking the scanner: **yylex()**

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

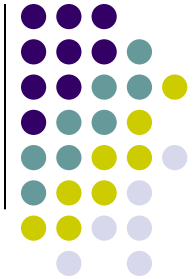
Returns 0 on end-of-file.

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)



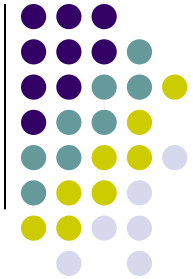
Pattern to match C-style comments: /* … */

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

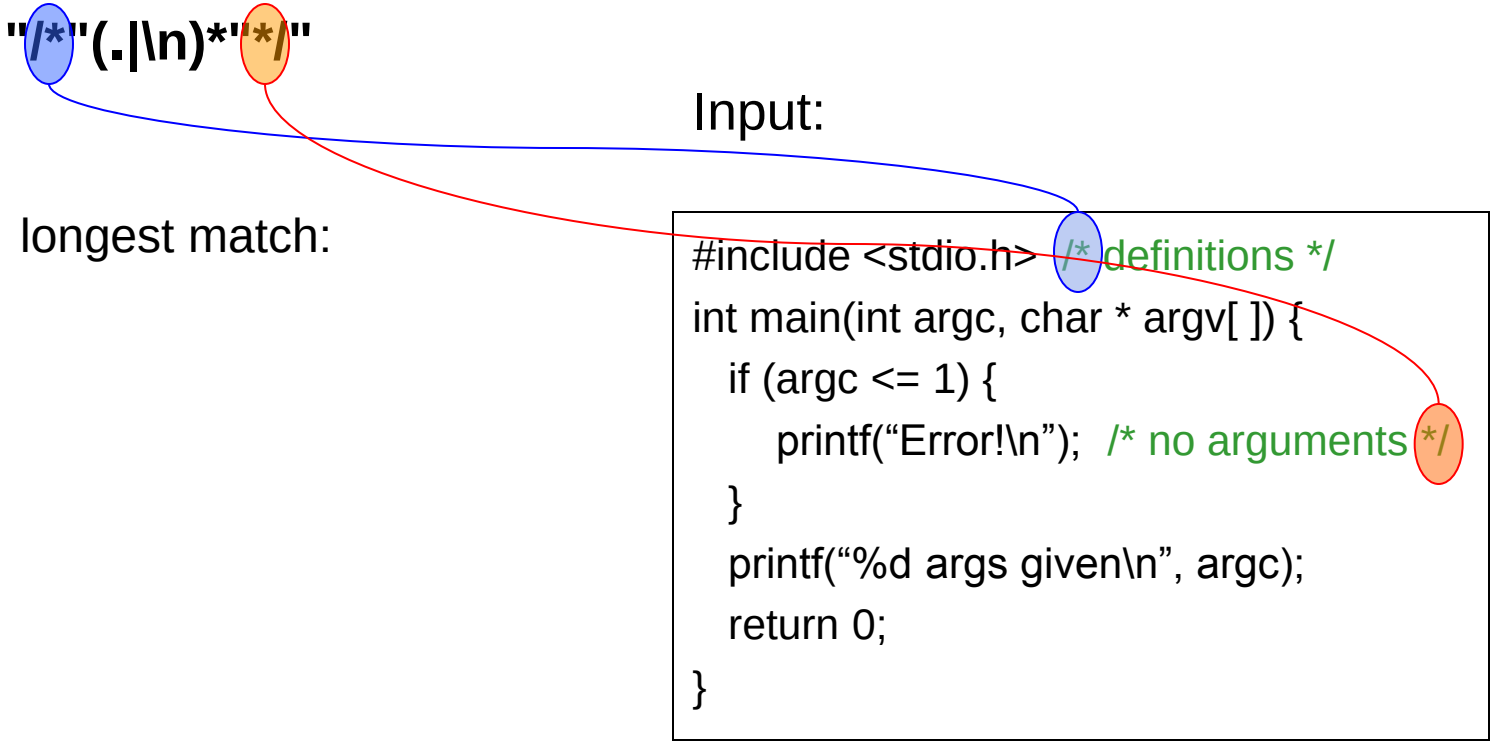
Input:

```
#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;
}
```

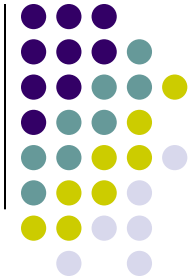
Matching the Input (cont'd)



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



Matching the Input (cont'd)



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

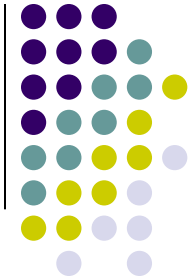
"/*"(.|\n)*"*/"

Input:

longest match:
Matched text
shown in blue

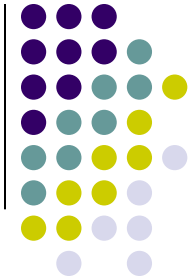
```
#include <stdio.h> /* definitions */
int main(int argc, char * argv[ ]) {
   if (argc <= 1) {
      printf("Error!\n");  /* no arguments */
   }
   printf("%d args given\n", argc);
   return 0;
}
```

Start Conditions



- Used to activate rules conditionally.
  - Any rule prefixed with <*S*> will be activated only when the scanner is in start condition *S*.
- Declaring a start condition *S*:
  - in the definition section: %x *S*
    - "%x" specifies "exclusive start conditions"
    - flex also supports "inclusive start conditions" ("%s"), see man pages.
- Putting the scanner into start condition S:
  - action: BEGIN(*S*)

Start Conditions (cont'd)

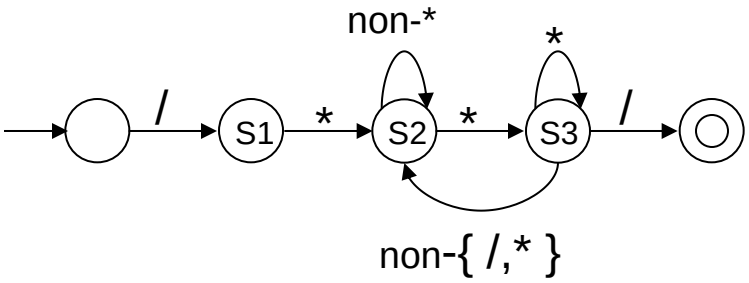


- Example:
  - <STRING>[^"]* { …match string body… }
    - [^"] matches any character other than "
    - The rule is activated only if the scanner is in the start condition STRING.
- INITIAL refers to the original state where no start conditions are active.
- <*> matches all start conditions.

Using Start Conditions

- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

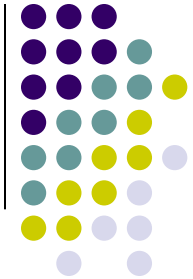
FSA for C comments:



flex input:

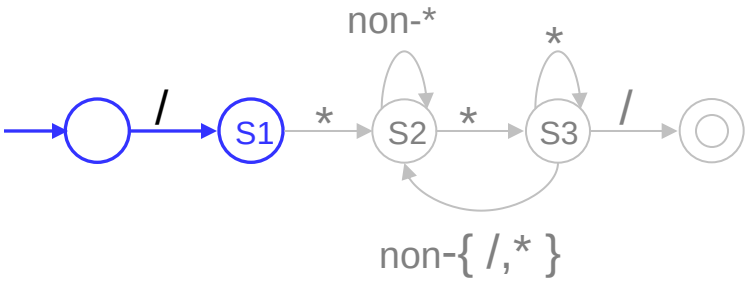
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*"         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

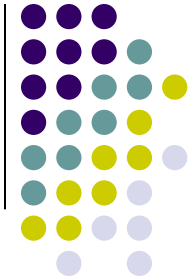
FSA for C comments:



flex input:

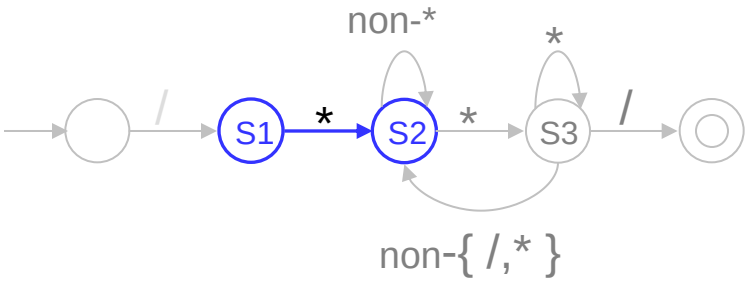
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"          BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

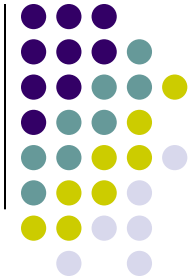
FSA for C comments:



flex input:

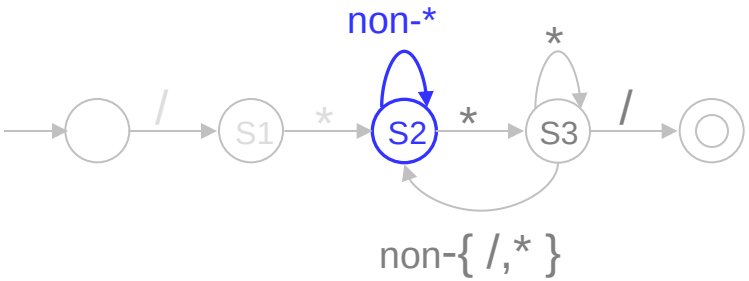
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

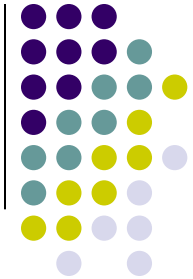
FSA for C comments:



flex input:

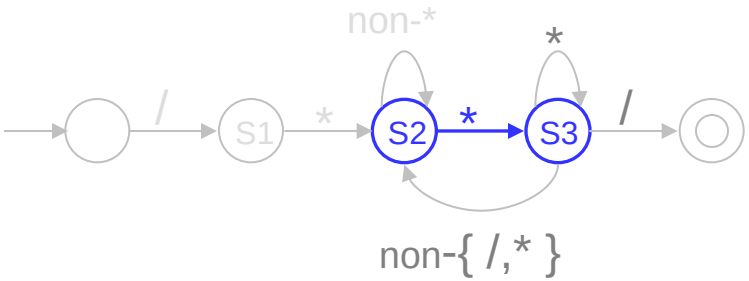
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"          BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

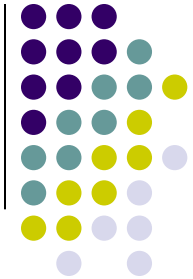
FSA for C comments:



flex input:

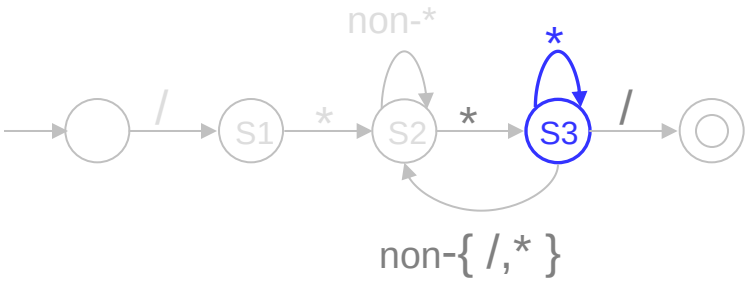
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

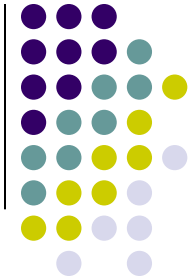
FSA for C comments:



flex input:

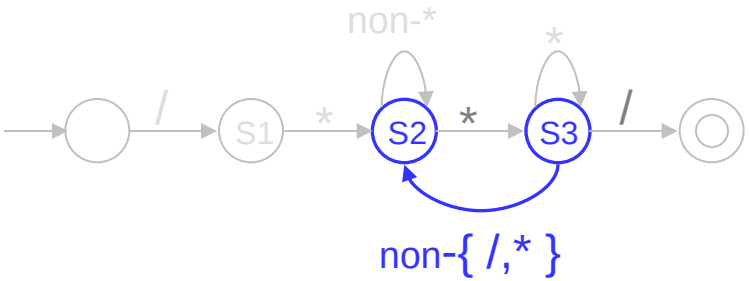
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

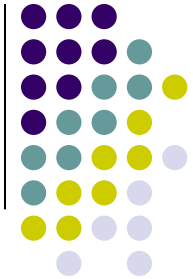
FSA for C comments:



flex input:

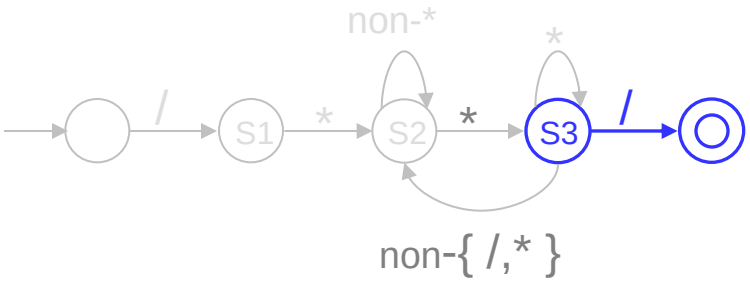
```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Using Start Conditions



- Start conditions let us explicitly simulate finite state machines.
- This lets us get around the “longest match” problem for C-style comments.

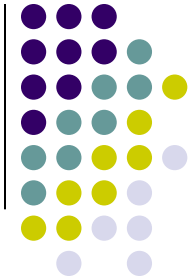
FSA for C comments:



flex input:

```
%x S1, S2, S3
%%
"/"             BEGIN(S1);
<S1>"*"         BEGIN(S2);
<S2>[^*]        ;     /* stay in S2 */
<S2>"*"         BEGIN(S3);
<S3>"*“         ;     /* stay in S3 */
<S3>[^*/]       BEGIN(S2);
<S3>"/"         BEGIN(INITIAL);
```

Putting it all together



- Scanner implemented as a function

  int yylex();

  - return value indicates type of token found (encoded as a +ve integer);
  - the actual string matched is available in yytext.
- Scanner and parser need to agree on token type encodings
  - let yacc generate the token type encodings
    - yacc places these in a file y.tab.h
  - use "#include y.tab.h" in the definitions section of the flex input file.
- When compiling, link in the flex library using "-lfl"