A brief [f]lex tutorial

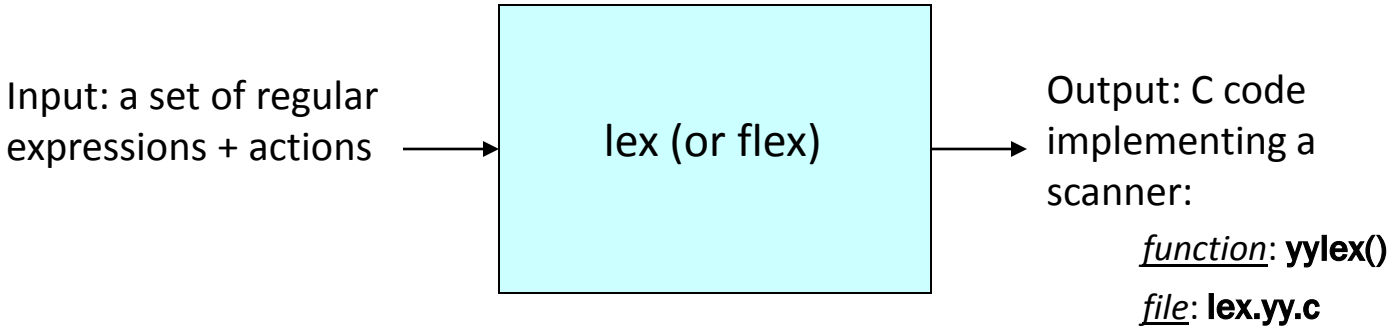
Saumya Debray
The University of Arizona
Tucson, AZ 85721

Modified by
Sandeep Dasgupta

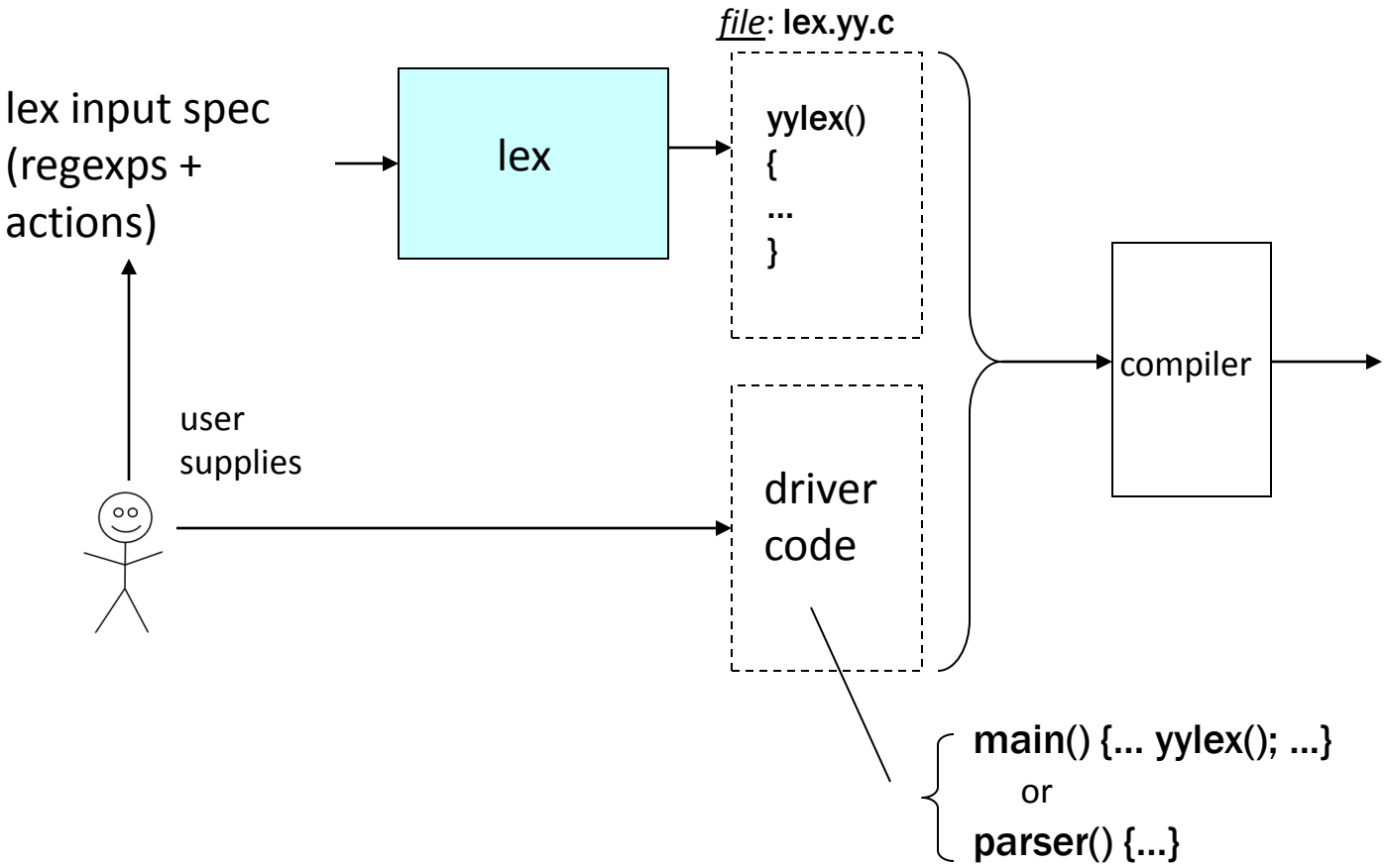
flex (and lex): Overview

Scanner generators:

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

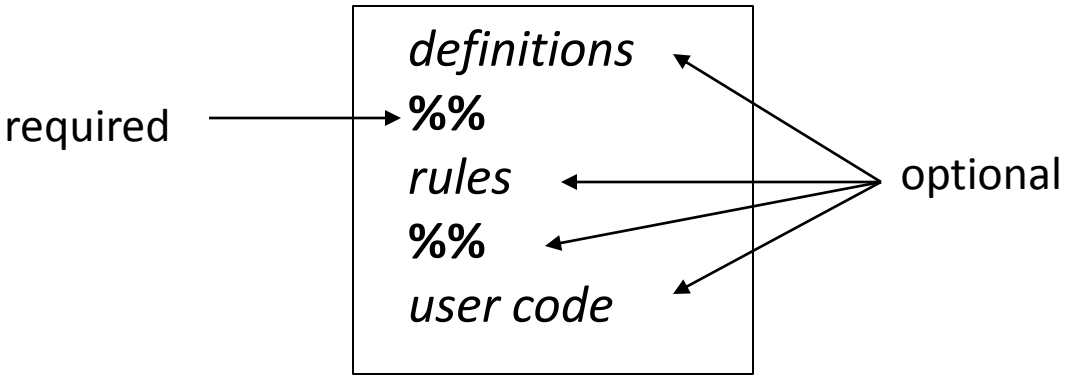


Using flex


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

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.(version dependent), for multi lined action : use {}

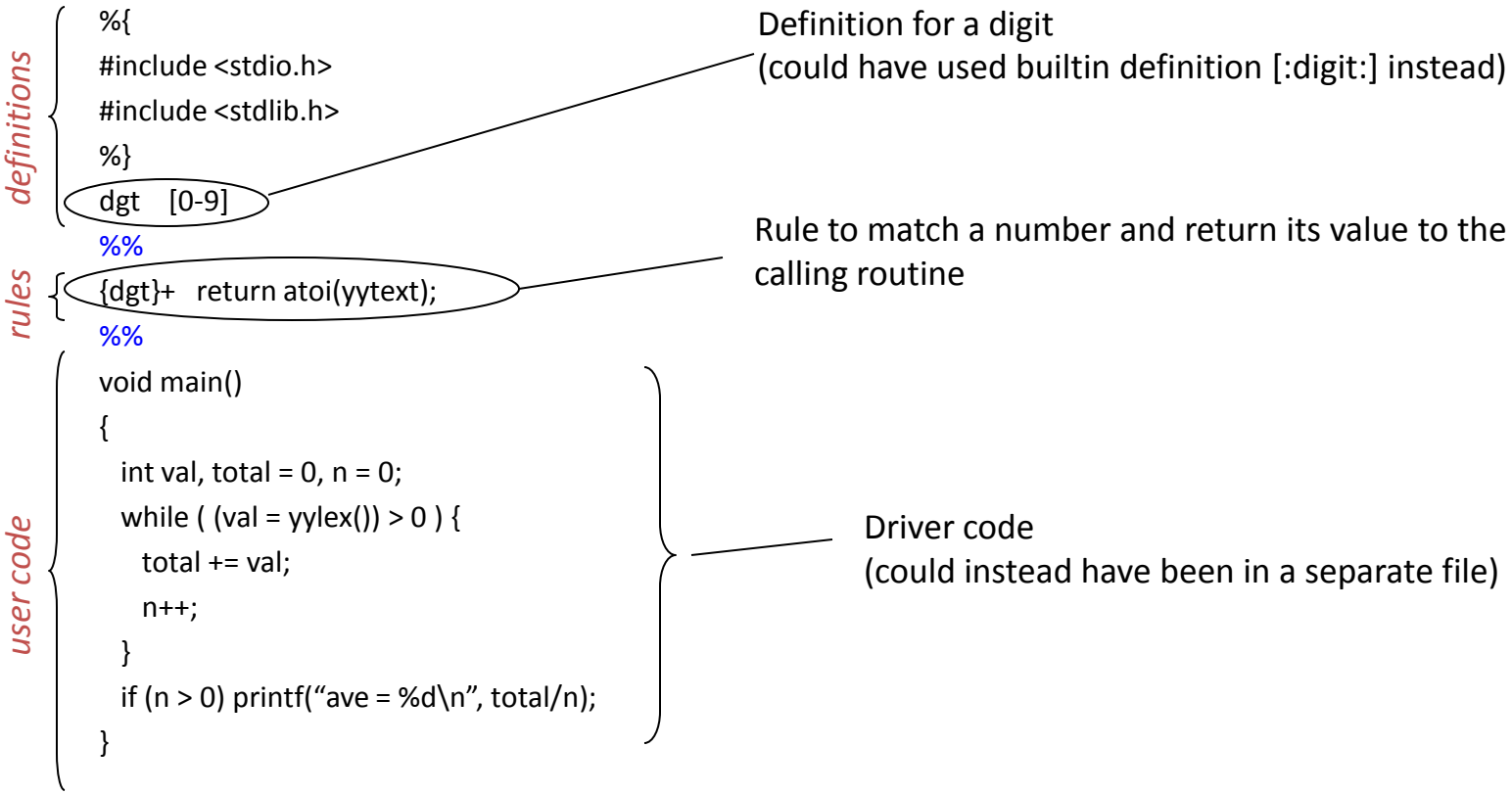
Example

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);
}
```

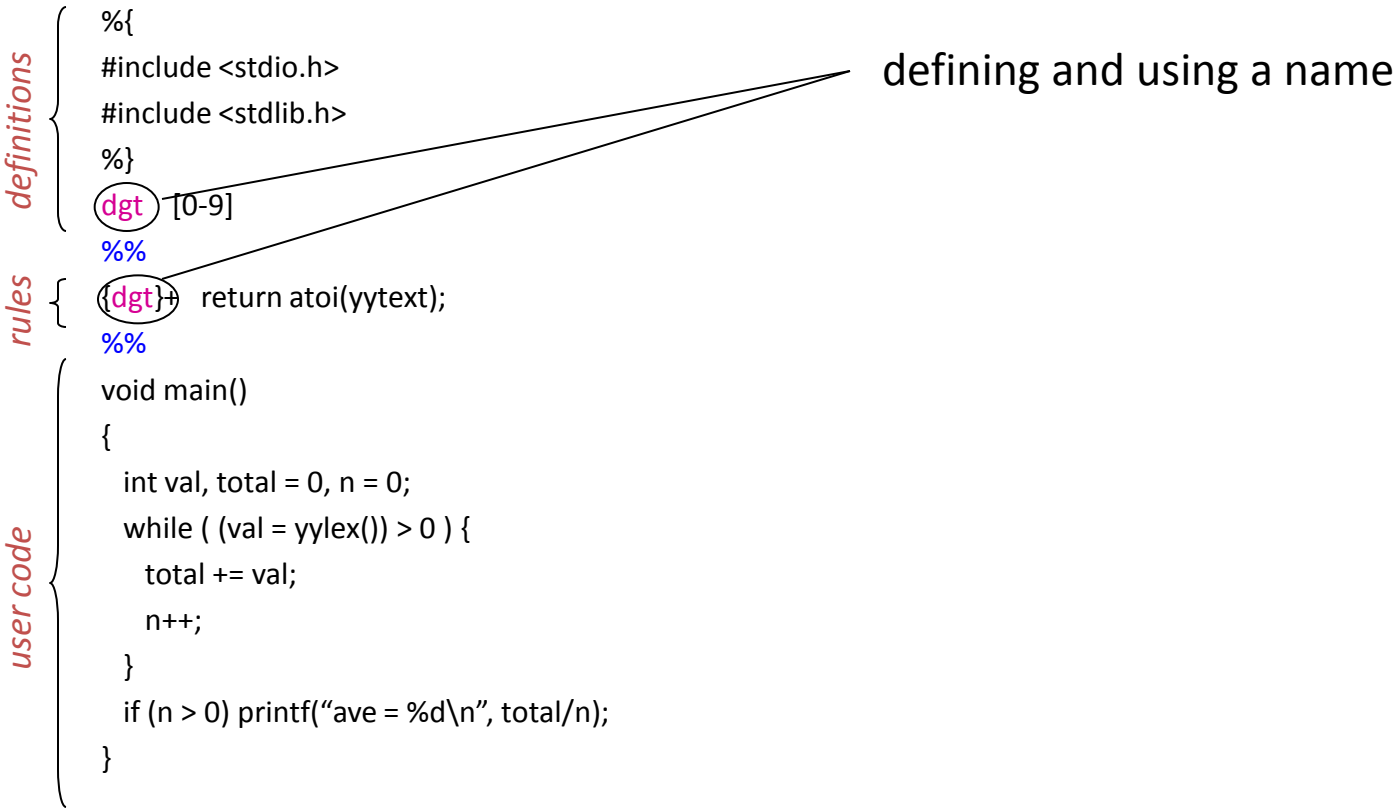
Example

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.

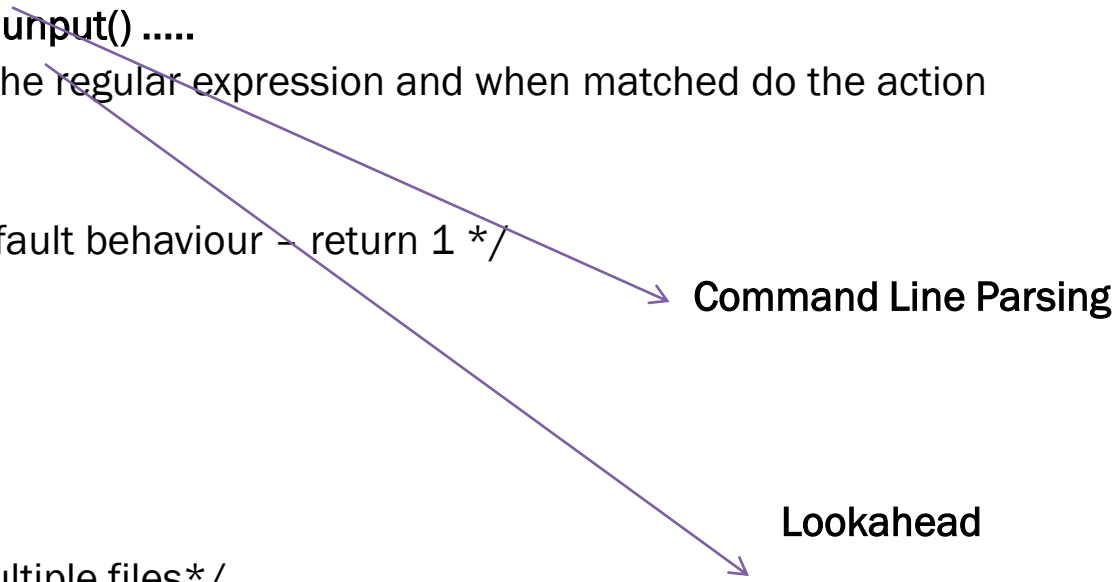
Matching the Input

- 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**.

```
(cs)  {printf("Department");}
(cs)[0-9]{3}  {printf("Course");}
[a-zA-Z]+[0-9]+ {printf("AnythingElse");}
Input: cs335
```

Control flow of lexer

```
yylex() {
      /*scan the file pointed to by yyin (default stdin)*/
            1.Repeated call of input() to get the next character from the input ……..
            2. Occatianal calls of unput() …..
            3. Try matching with the regular expression and when matched do the action
               part…..
       /*        got  EOF */
       Int status = yywrap(); /*default behaviour – return 1 */
       If(1 == status)
              exit() ;
       Else
            yylex() ;
}
/*Redefine yywrap to handle multiple files*/
Int yywrap() {
 …..
If(exists other files to process )  { yyin = nextFilePtr; return 0; }
Else  { return 1; }
}
```



Start Conditions

- Used to activate rules conditionally.
  - Any rule prefixed with <*S*> will be activated only when the scanner is in start condition *S*.
    - %s MAGIC ←-------Inclusive start condition
    - %%
    - <MAGIC>.+ {BEGIN 0; printf("Magic: "); ECHO; }
    - magic {BEGIN MAGIC}

    Input: magic two three

    Warning: A rule without an explicit start state will match regardless of what start state is active.
    - %s MAGIC ←-------Inclusive start condition
    - %%
    - magic {BEGIN MAGIC}
    - .+ ECHO;
    - <MAGIC>.+ {BEGIN 0; printf("Magic: "); ECHO; }

Start Conditions (cont'd)

WayOut:

- Use of exlpicit start state using %x MAGIC
- For versions that lacks %x
  - %s NORMAL MAGIC
  - %%
  - %{
    - » BEGIN NORMAL;
  - %}
  - <NORMAL>magic {BEGIN MAGIC}
  - <NORMAL>.+ ECHO;
  - <MAGIC>.+ {BEGIN 0; printf("Magic: "); ECHO; }

if(first_time == 1) { BEGIN NORMAL; first_time = 0 ; }

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 "-ll"