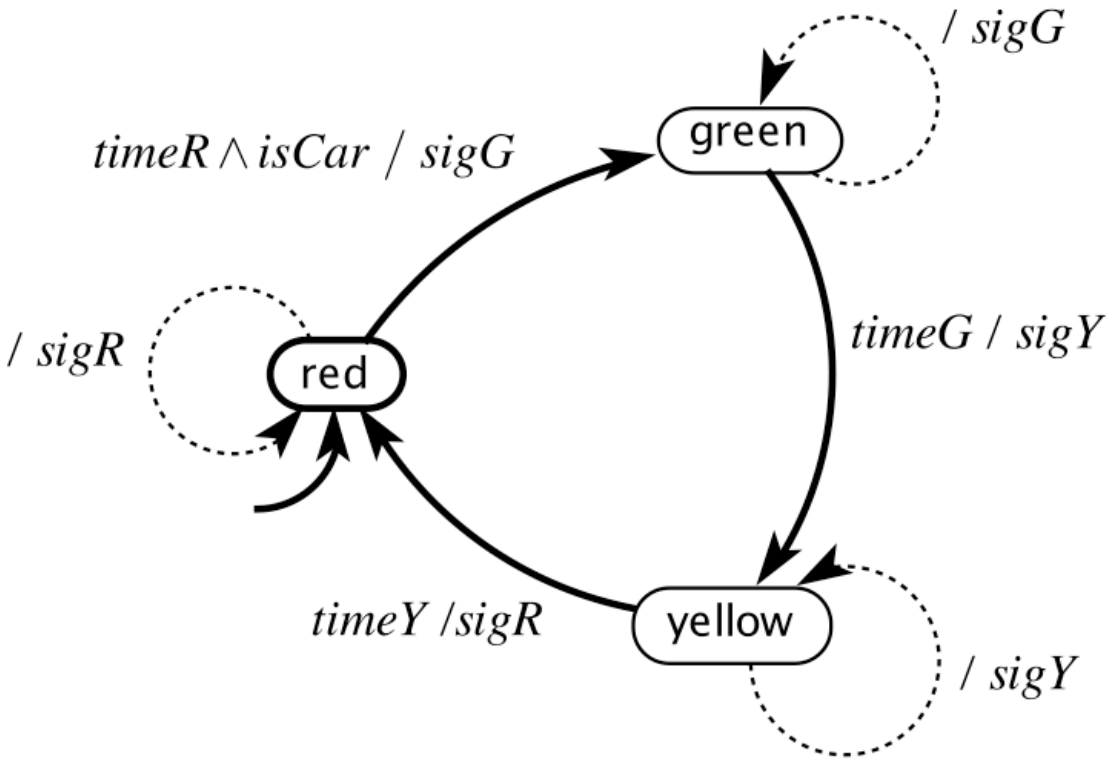
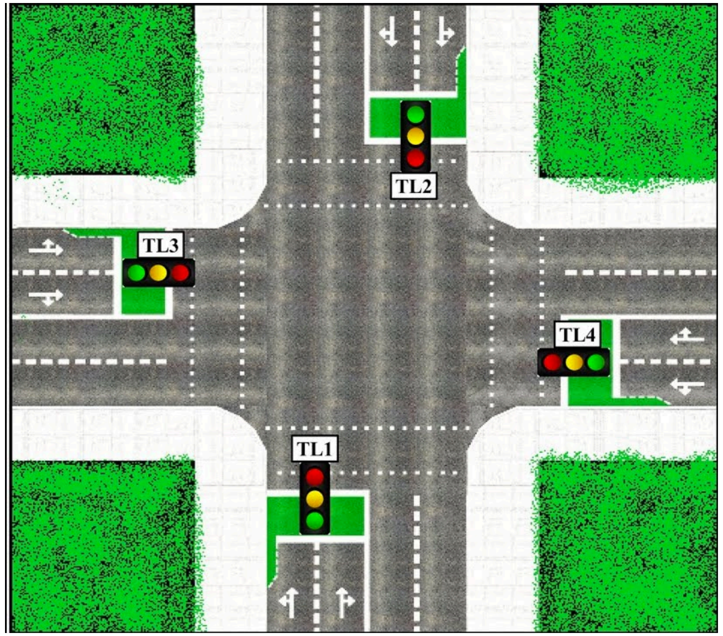
# COP290

# Finite State Machines

- 4 way Crossing
  - Traffic Light Controller

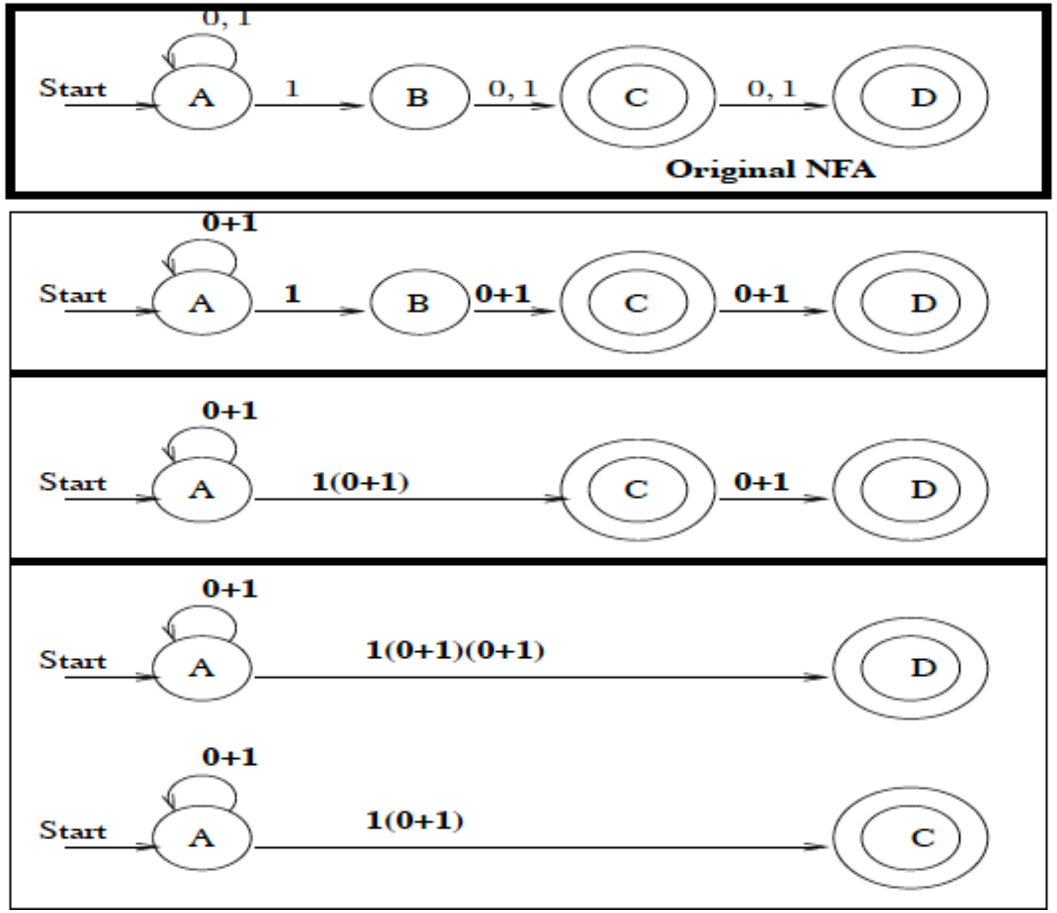




# *Finite Automaton*

- A machine that accepts strings of 0 and 1's such that either second or third position from the end has a 1.
  - e.g. 0001010, 011, etc

(0+1)*1(1+0) + (0+1)*1(1+0) (0+1)



# *Pattern Matching & Regular Expressions*

- Common examples
  - ls *.c
  - grep –lR “cop*”
- A pattern of special characters used to match strings in a search
  - Typically made up from special characters called metacharacters
- Regular expressions are used thoughout UNIX:
  - Editors: ed, ex, vi
  - Utilities: grep, egrep, sed, and awk

# Metacharacters

| RE Metacharacter | Matches… |
|---|---|
| **.** | **Any one character, except new line** |
| **[a-z]** | **Any one of the enclosed characters (e.g. a-z)** |
| ***** | **Zero or more of preceding character** |
| **? or \?** | **Zero or one of the preceding characters** |
| **+ or \+** | **One or more of the preceding characters** |

- any non-metacharacter matches itself

# The grep Utility

- “grep” command:
  searches for text in file(s)

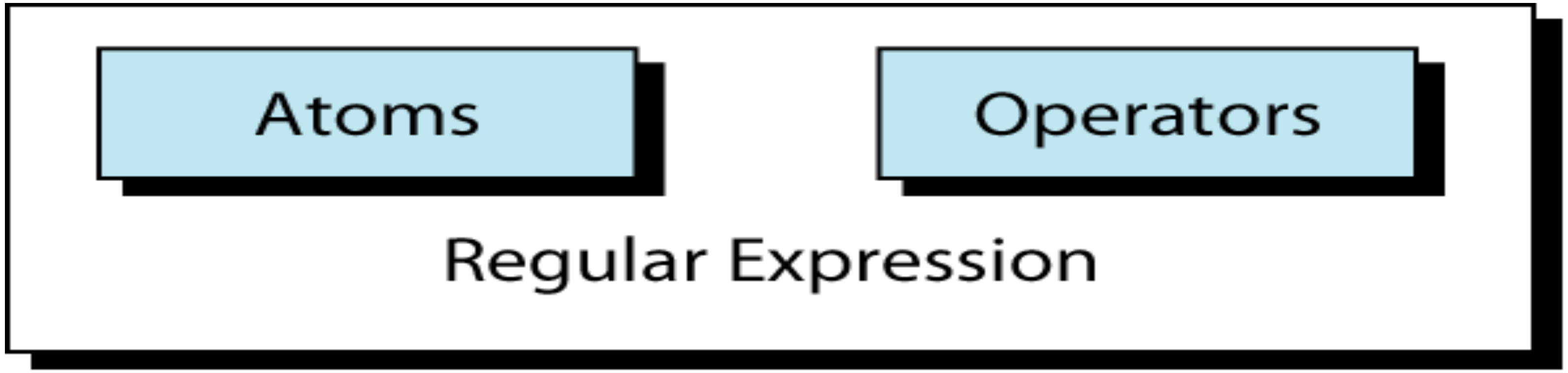
<u>Examples:</u>

```
% grep root mail.log
% grep r..t mail.log
% grep ro*t mail.log
% grep ‘ro*t’ mail.log
% grep ‘r[a-z]*t’ mail.log
```

# more Metacharacters

<table>
<tr><th>RE Metacharacter</th><th>Matches…</th></tr>
<tr><td>^</td><td>beginning of line</td></tr>
<tr><td>$</td><td>end of line</td></tr>
<tr><td>\<i>char</i></td><td>Escape the meaning of <i>char</i> following it</td></tr>
<tr><td>[^]</td><td>One character not in the set</td></tr>
<tr><td>\<</td><td>Beginning of word anchor</td></tr>
<tr><td>\></td><td>End of word anchor</td></tr>
<tr><td>( ) or \( \)</td><td>Tags matched characters to be used later (max = 9)</td></tr>
<tr><td>| or \|</td><td>Or grouping</td></tr>
<tr><td>x\{m\}</td><td>Repetition of character x, m times (x,m = integer)</td></tr>
<tr><td>x\{m,\}</td><td>Repetition of character x, at least m times</td></tr>
<tr><td>x\{m,n\}</td><td>Repetition of character x between m and m times</td></tr>
</table>

# Regular Expression

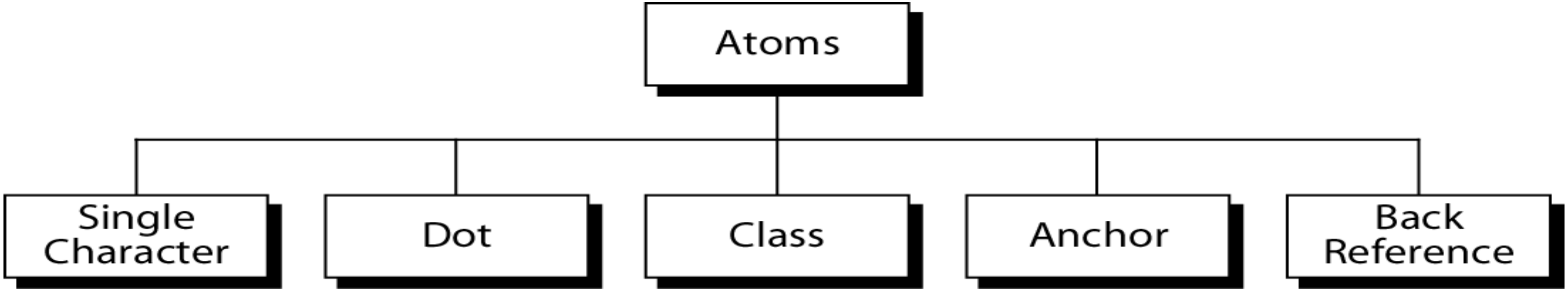


An atom specifies what text is to be matched and where it is to be found.

An operator combines regular expression atoms.

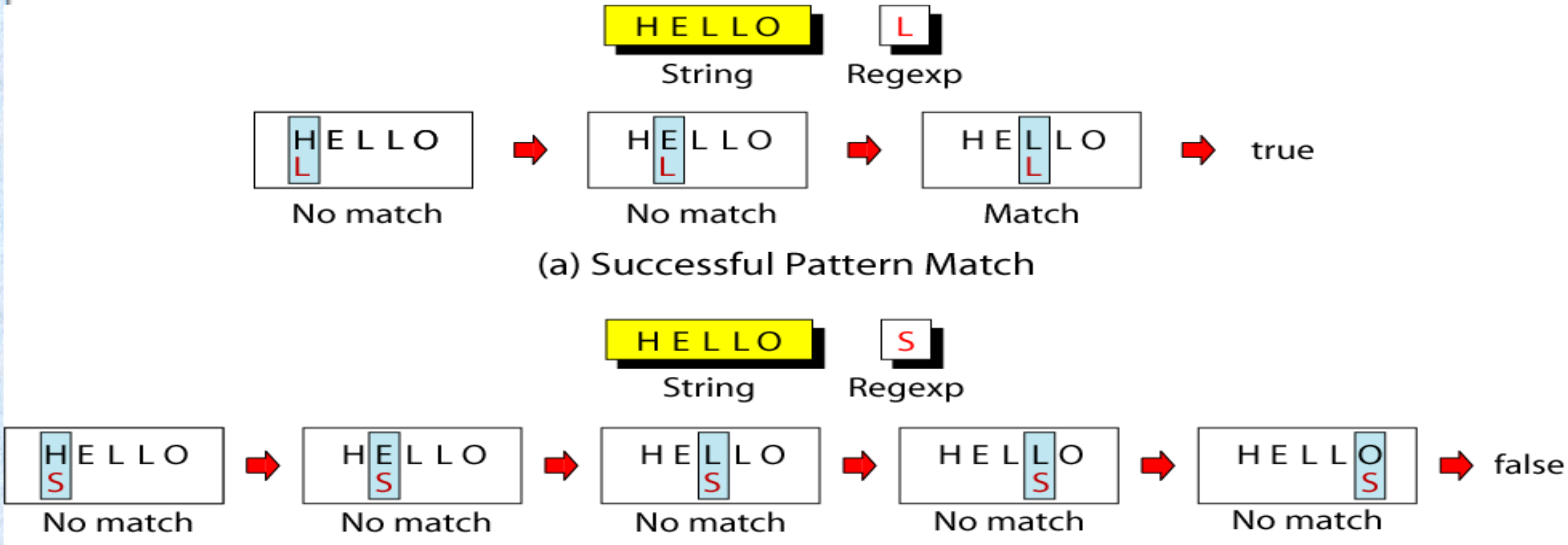
# Atoms

An atom specifies what text is to be matched and where it is to be found.



# Single-Character Atom

A single character matches itself



HELLO

String

L

Regexp

HELLO L No match → HELLO L No match → HELLO L Match → true

(a) Successful Pattern Match

HELLO

String

S

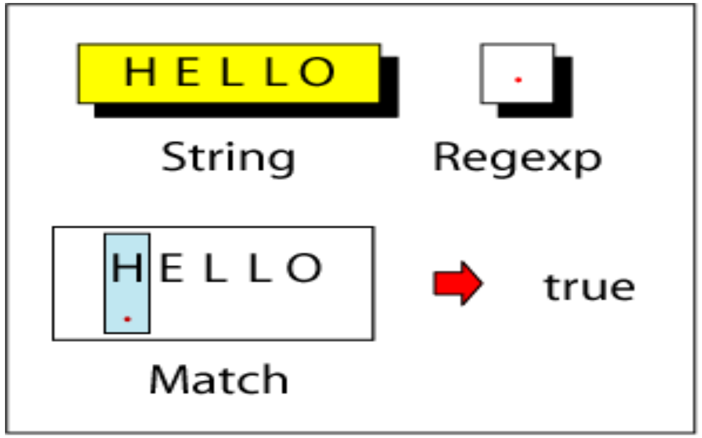
Regexp

HELLO S No match → HELLO S No match → HELLO S No match → HELLO S No match → HELLO S No match → false

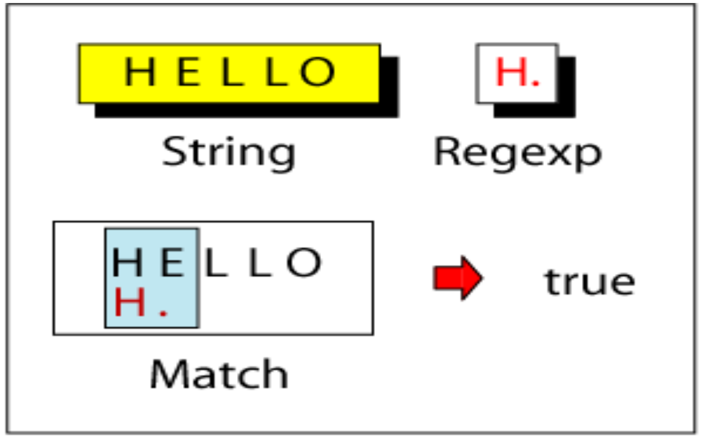
(b) Unsuccessful Pattern Match

# Dot Atom

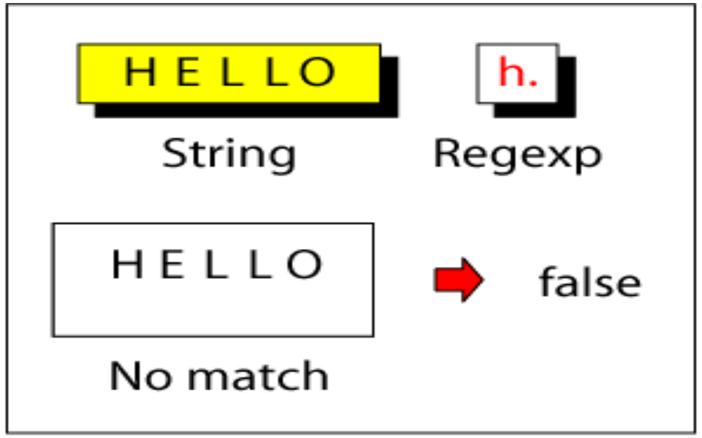
matches any single character except for a new line character (\n)



(a) Single-Character



(b) Combination–True

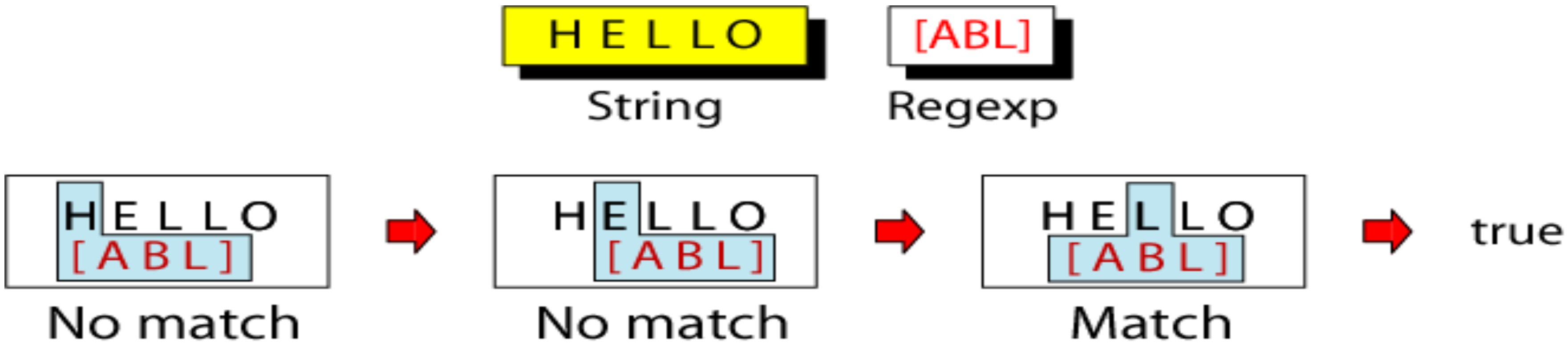


(c) Combination–False

# Class Atom

matches only single character that can be any of the characters defined in a set:

Example: [ABC] matches either A, B, or C.



Notes:

1) A range of characters is indicated by a dash, e.g. [A-Q]
2) Can specify characters to be excluded from the set, e.g. [^0-9] matches any character other than a number.

# Example: Classes

| RegExpr | Means | RegExpr | Means |
|---|---|---|---|
| [A-H] | [ABCDEFGH] | [^AB] | Any character except A or B |
| [A-Z] | Any uppercase alphabetic | [A-Za-z] | Any alphabetic |
| [0-9] | Any digit | [^0-9] | Any character except a digit |
| [[a] | [ or a | []a] | ] or a |
| [0-9\-] | digit or hyphen | [^\^] | Anything except^ |

# Repetition Operator: \{…\}

The repetition operator specifies that the atom or expression immediately before the repetition may be repeated.

`\{m , n\}`

matches previous character m to n times.

`A\{3 , 5\}` → matches "AAA" , "AAAA", or "AAAAA"

`BA\{3 , 5\}` → matches "BAAA" , "BAAAA", or "BAAAAA"

# Grep detail and examples

- grep is family of commands
  - grep

    common version
  - egrep

    understands extended REs

    (| + ? ( ) don’t need backslash)
  - fgrep

    understands only fixed strings, i.e. is faster
  - rgrep

    will traverse sub-directories recursively

# *COMMONLY USED “GREP” OPTIONS:*

-c Print only a count of matched lines.

-i Ignore uppercase and lowercase distinctions.

-l List all files that contain the specified pattern.

-n Print matched lines and line numbers.

-s Work silently; display nothing except error messages. Useful for checking the exit status.

-v Print lines that do not match the pattern.

# Example: grep with pipe

Pipe the output of the “ls –l” command to grep and list/ select only directory entries.

```
% ls -l | grep '^d'
drwxr-xr-x   2 krush    csci         512 Feb  8 22:12 assignments
drwxr-xr-x   2 krush    csci         512 Feb  5 07:43 feb3
drwxr-xr-x   2 krush    csci         512 Feb  5 14:48 feb5
drwxr-xr-x   2 krush    csci         512 Dec 18 14:29 grades
drwxr-xr-x   2 krush    csci         512 Jan 18 13:41 jan13
drwxr-xr-x   2 krush    csci         512 Jan 18 13:17 jan15
drwxr-xr-x   2 krush    csci         512 Jan 18 13:43 jan20
drwxr-xr-x   2 krush    csci         512 Jan 24 19:37 jan22
drwxr-xr-x   4 krush    csci         512 Jan 30 17:00 jan27
drwxr-xr-x   2 krush    csci         512 Jan 29 15:03 jan29
% ls -l | grep -c '^d'
10
```

Display the number of lines where the pattern was found.  This does not mean the number of occurrences of the pattern.

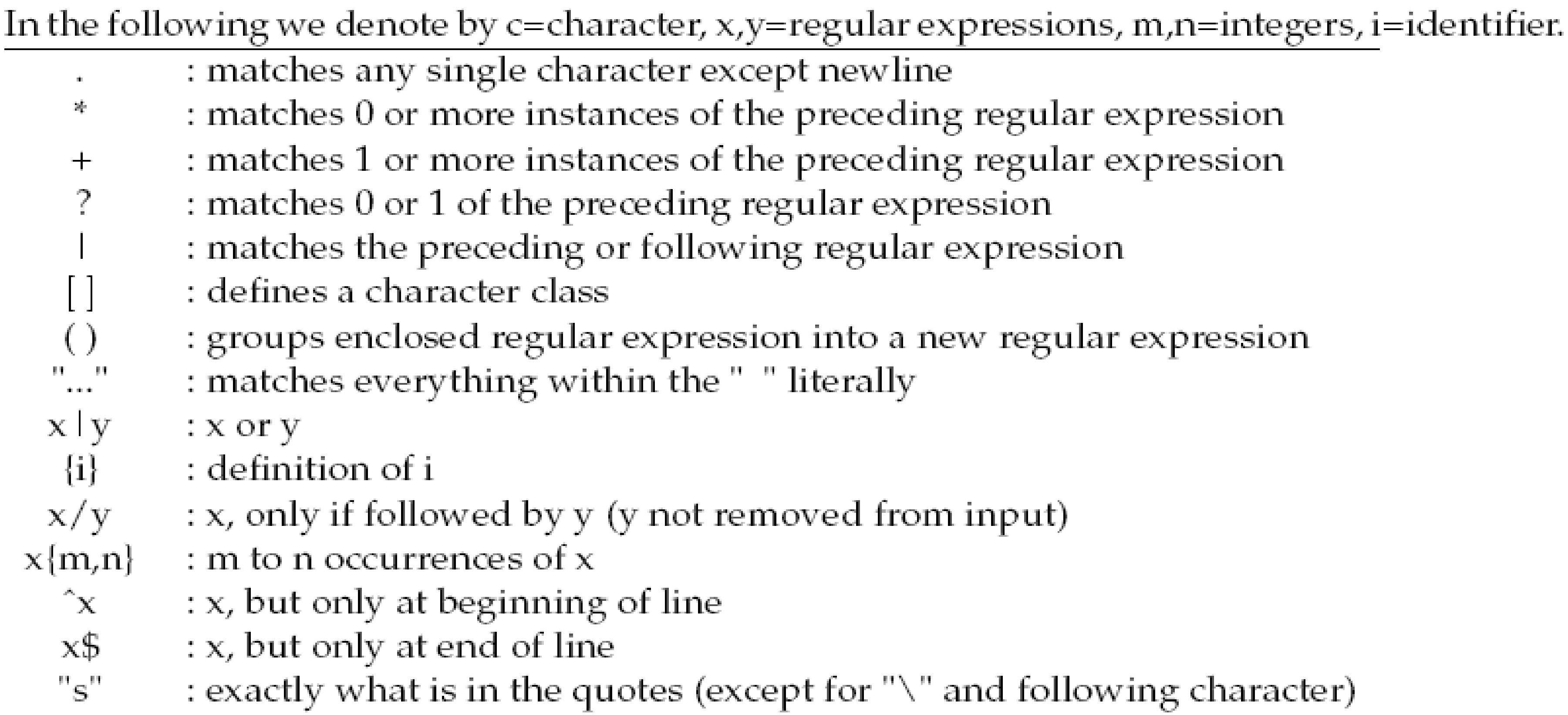
# Example: grep with \< \>

```
% cat grep-datafile
northwest       NW      Charles Main            300000.00
western         WE      Sharon Gray             53000.89
southwest       SW      Lewis Dalsass           290000.73
southern        SO      Suan Chin               54500.10
southeast       SE      Patricia Hemenway       400000.00
eastern         EA      TB Savage               440500.45
northeast       NE      AM Main Jr.             57800.10
north           NO      Ann Stephens            455000.50
central         CT      KRush                   575500.70
Extra [A-Z]****[0-9]..$5.00
```

Print the line if it  contains the word  “north”.

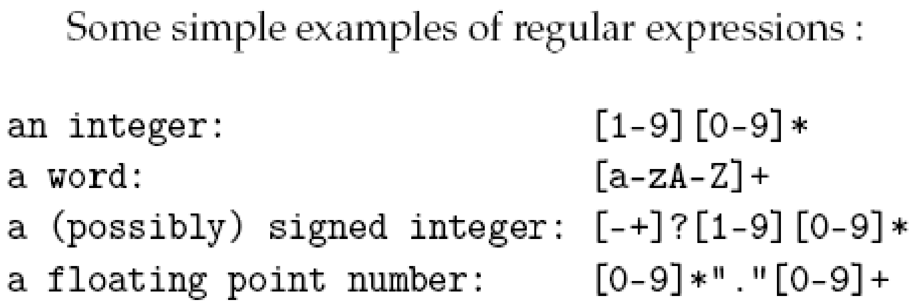
```
% grep '\<north\>' grep-datafile
north           NO      Ann Stephens            455000.50
```

# Regular Expressions



In the following we denote by c=character, x,y=regular expressions, m,n=integers, i=identifier.

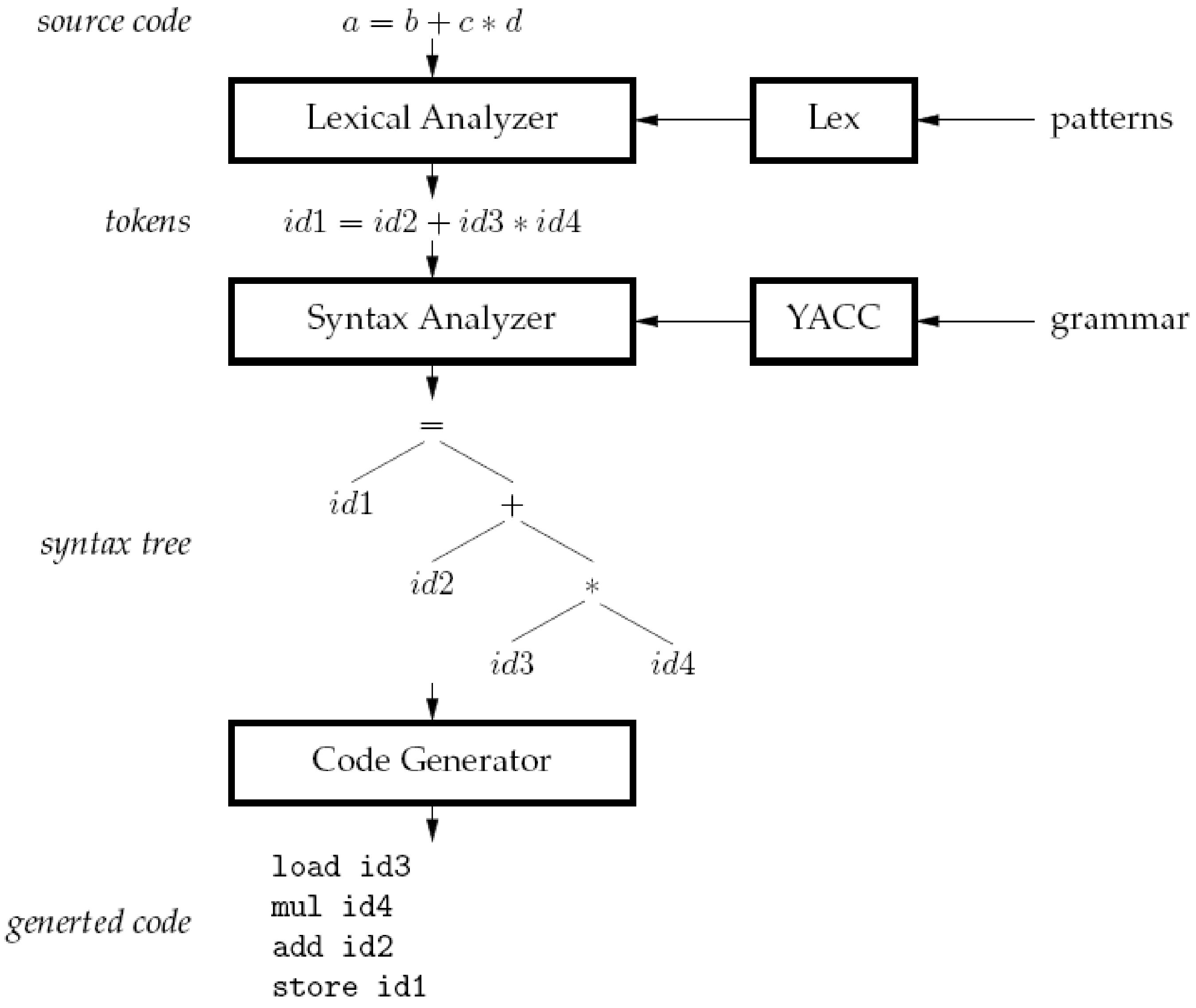
| | |
|---|---|
| . | : matches any single character except newline |
| * | : matches 0 or more instances of the preceding regular expression |
| + | : matches 1 or more instances of the preceding regular expression |
| ? | : matches 0 or 1 of the preceding regular expression |
| \| | : matches the preceding or following regular expression |
| [ ] | : defines a character class |
| ( ) | : groups enclosed regular expression into a new regular expression |
| "..." | : matches everything within the " " literally |
| x \| y | : x or y |
| {i} | : definition of i |
| x/y | : x, only if followed by y (y not removed from input) |
| x{m,n} | : m to n occurrences of x |
| ˆx | : x, but only at beginning of line |
| x$ | : x, but only at end of line |
| "s" | : exactly what is in the quotes (except for "\" and following character) |



Some simple examples of regular expressions :

| | |
|---|---|
| `an integer:` | `[1-9][0-9]*` |
| `a word:` | `[a-zA-Z]+` |
| `a (possibly) signed integer:` | `[-+]?[1-9][0-9]*` |
| `a floating point number:` | `[0-9]*"."[0-9]+` |

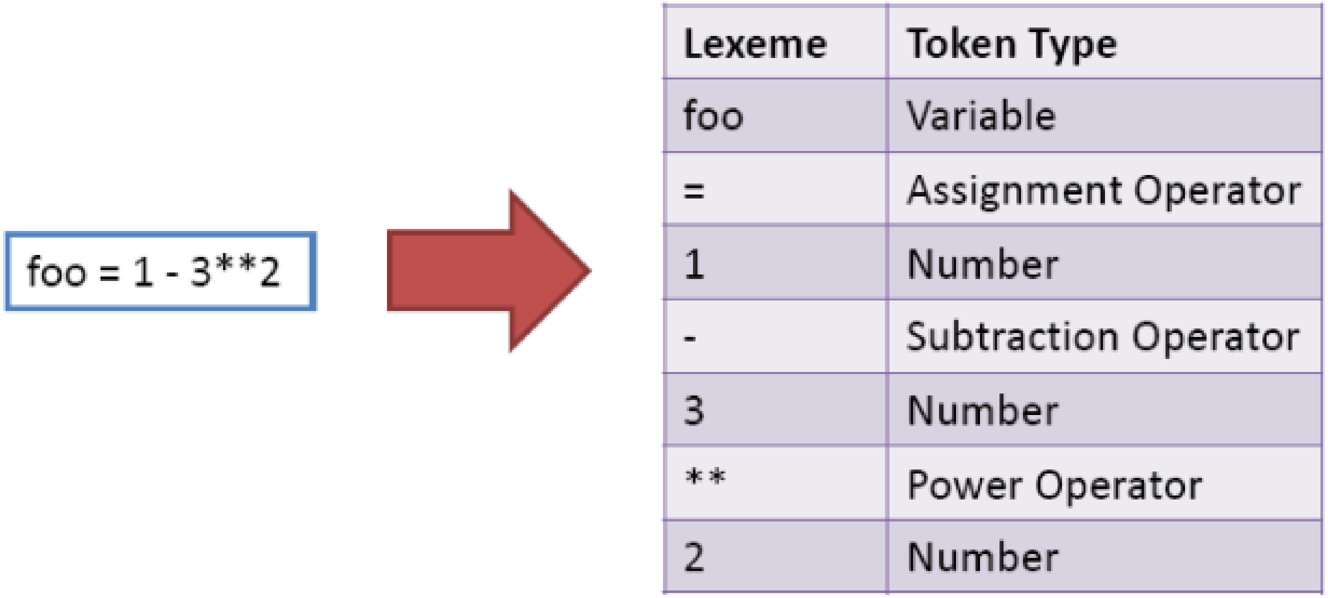
# Compilation



```
load id3
mul id4
add id2
store id1
```

# *Lexical Analysis*

- Lexical Analysis –
  - Process of converting a sequence of characters into a sequence of tokens
- Lex
  - Is a tool to generator lexical analyzers.
  - It was written by Mike Lesk and Eric Schmidt (the Google guy)



```
foo = 1 - 3**2
```

| Lexeme | Token Type |
|---|---|
| foo | Variable |
| = | Assignment Operator |
| 1 | Number |
| - | Subtraction Operator |
| 3 | Number |
| ** | Power Operator |
| 2 | Number |

# Lex

- Look for Patterns

```
<pattern>  <action to take when matched>
<pattern>  <action to take when matched>
```

Example

```
[A-Za-z]+  printf("this is a word");
[0-9]+  printf("this is a number");
```

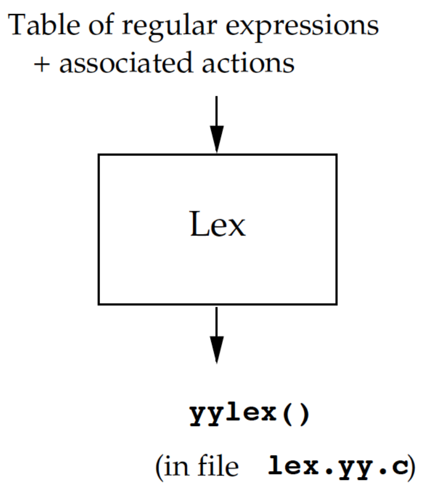


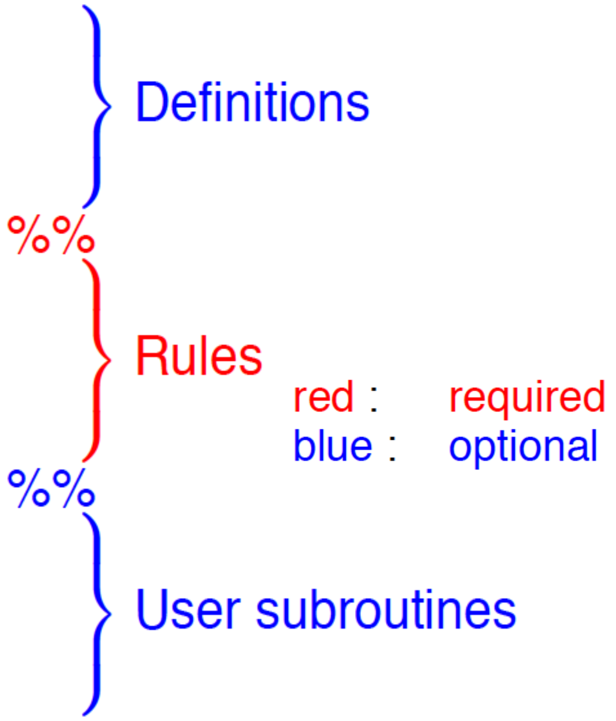
Table of regular expressions
+ associated actions

Lex

`yylex()`

(in file **`lex.yy.c`**)

| scanner.l | lex.yy.c is generated |
|---|---|
| %{<br>< C global variables, prototypes, comments ><br>%} | *This part will be embedded into lex.yy.c* |
| DEFINITION SECTION | *substitutions, code and start states; will be copied into *.c* |
| %% | |
| RULES SECTION | *define how to scan and what action to take for each token* |
| %% | |
| <auxiliary C subroutines> | *any user code. For example, a main function to call the scanning function yylex().* |



Definitions

%%

Rules

%%

User subroutines

red : required
blue : optional

# A Sample lex Effort

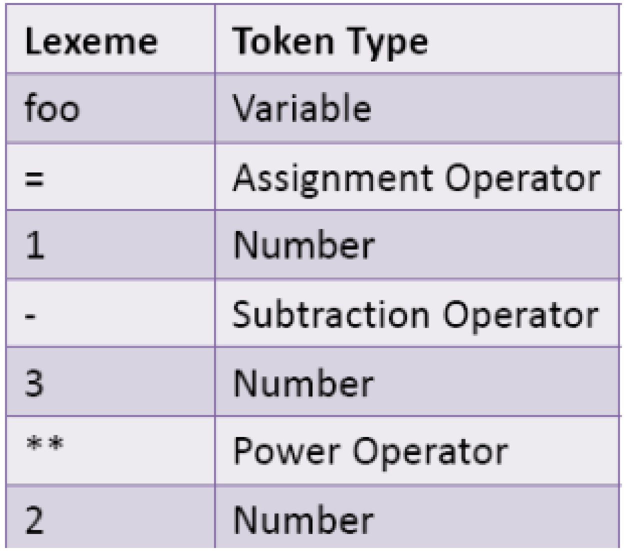
- Count Characters
  - Whenever a letter or . (dot) is encountered, charCount is incremented by one.

```
%{
int charCount;
%}
//This comment is to tell that following is the rule section
%%
[a-zA-Z.] { charCount++;}
%%
main()
{
yylex();
printf("%d \n", charCount);
}
```

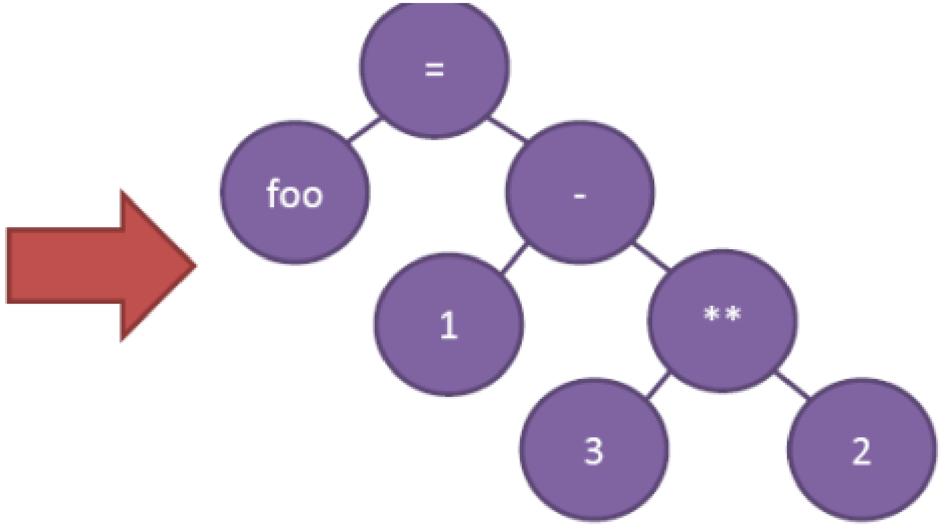
- *yylex()*
  - *reads each character in the input file and is matched against the patterns specified in the rule section*

# Parsing

- Syntactic Analysis
  - The process of analyzing a sequence of tokens to determine its grammatical structure.
  - Yacc
    - Is a tool to generate parsers (syntactic analyzers).
    - Generated parsers require a lexical analyzer

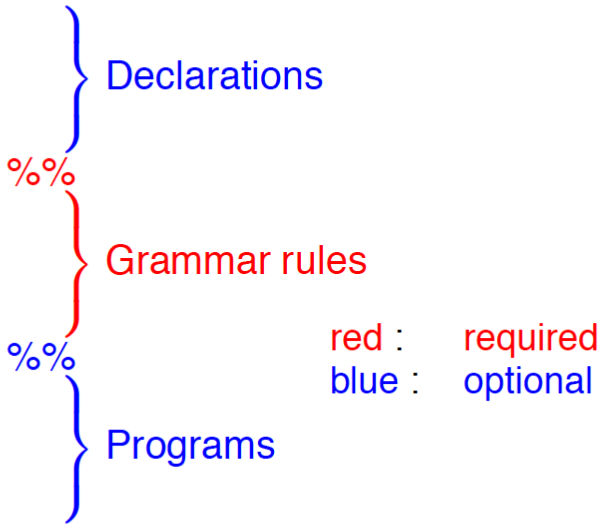


| Lexeme | Token Type |
|---|---|
| foo | Variable |
| = | Assignment Operator |
| 1 | Number |
| - | Subtraction Operator |
| 3 | Number |
| ** | Power Operator |
| 2 | Number |



# Yacc – Parser Generator

- Takes a specification from CFG and produces a LALR Parser
  - *The user must supply an integer-valued function yylex() that implements the lexical analyzer (scanner).*
    - *If there is a value associated with the token, it should be assigned to the external variable yylval.*
    - *The token error is reserved for error handling.*
    - *Token numbers : These may be chosen by the user if desired. The default is:*
      - *chosen by yacc [in a file y.tab.h]*
      - *the token no. for a literal is its ASCII value*
      - *other tokens are assigned numbers starting at 257*
      - *the endmarker must have a number zero or negative.*
  - *Generate y.tab.h using 'yacc -d'*



Declarations

%%

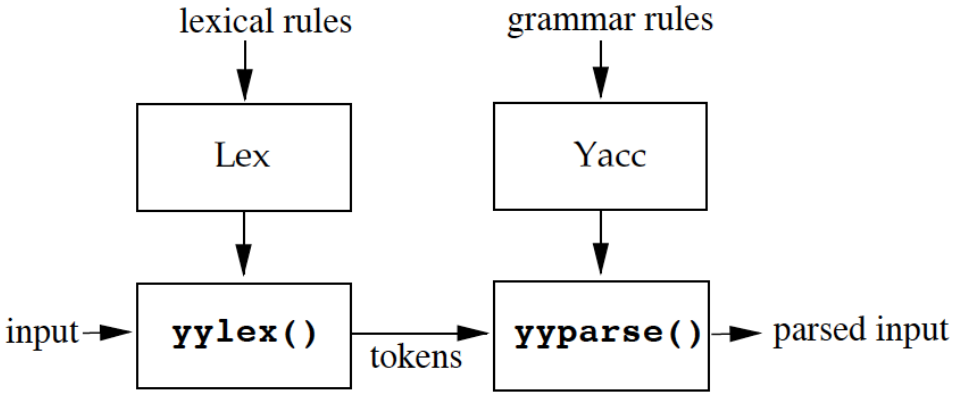
Grammar rules

%%

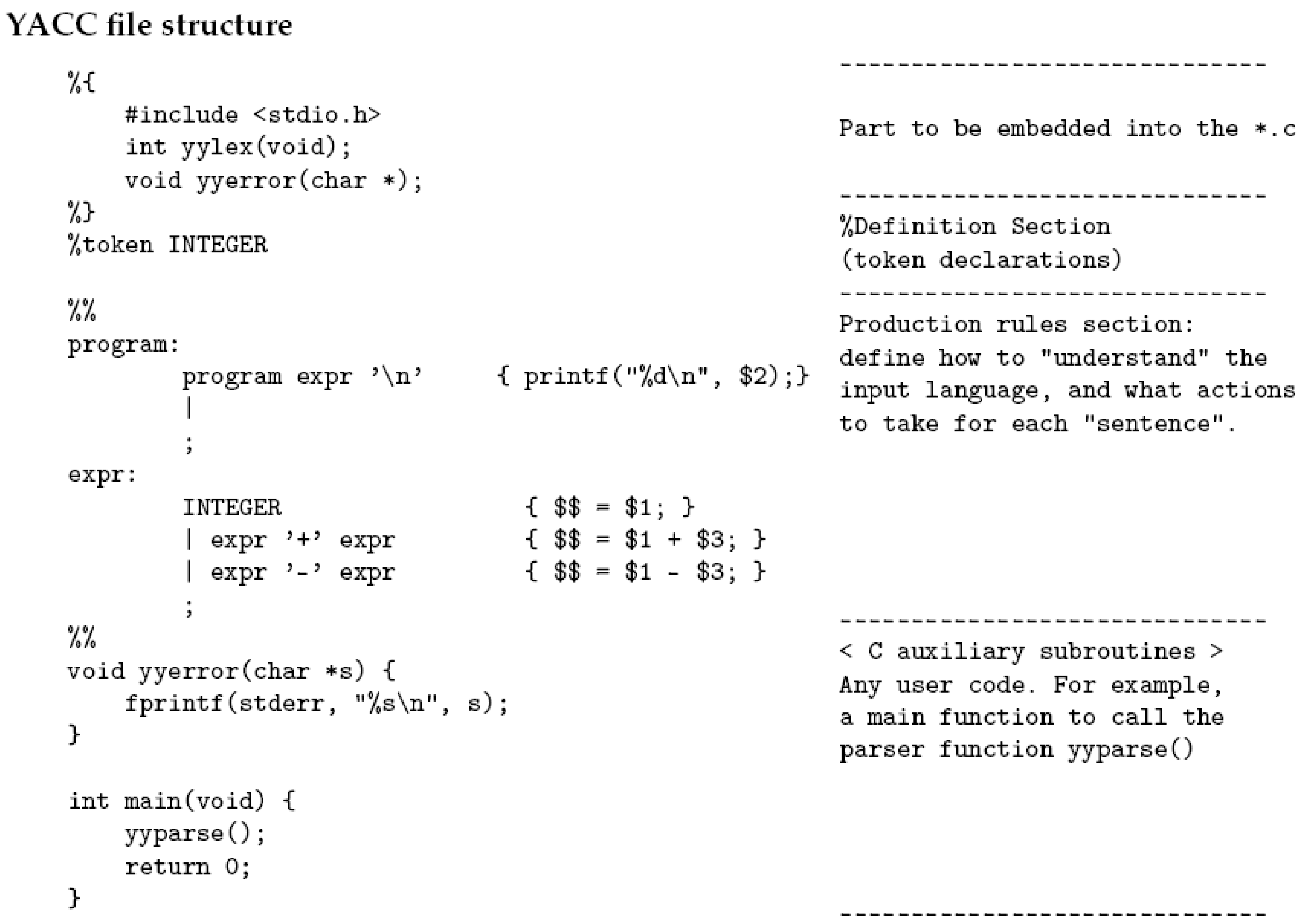
Programs

red : required

blue : optional



# Yacc File Structure

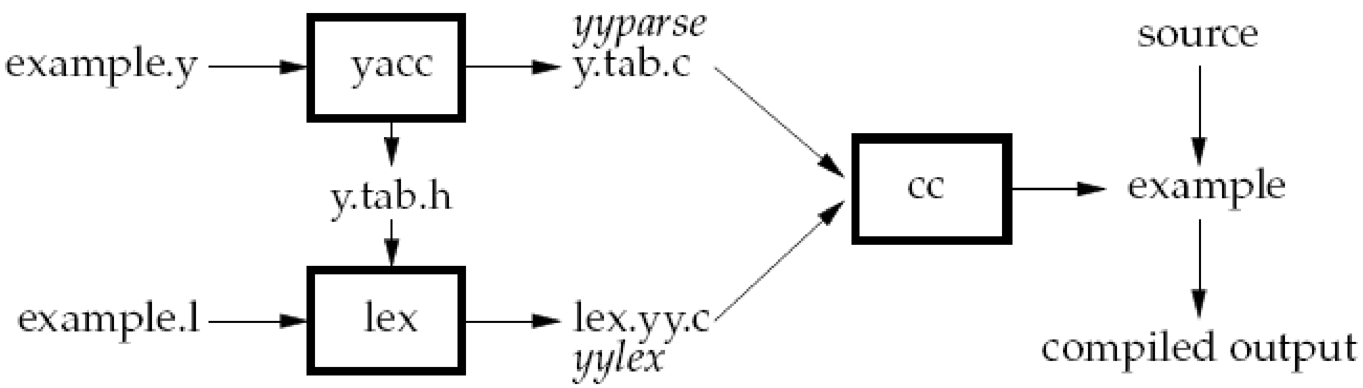


YACC file structure

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
%}
%token INTEGER

%%
program:
        program expr ’\n’      { printf("%d\n", $2);}
        |
        ;
expr:
        INTEGER                  { $$ = $1; }
        | expr ’+’ expr          { $$ = $1 + $3; }
        | expr ’-’ expr          { $$ = $1 - $3; }
        ;
%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

```
------------------------------

Part to be embedded into the *.c

------------------------------
%Definition Section
(token declarations)
------------------------------
Production rules section:
define how to "understand" the
input language, and what actions
to take for each "sentence".

------------------------------
< C auxiliary subroutines >
Any user code. For example,
a main function to call the
parser function yyparse()

------------------------------
```

# Combining Lex and Yacc



# Infix to Postfix

Input: a*b+c
Output: ab*c+

Input: a+b*d
Output: abd*+

```
%{
/* Definition section */
%}
ALPHA [A-Z a-z]
DIGIT [0-9]

/* Rule Section */
%%
{ALPHA}({ALPHA}|{DIGIT})* return ID;
{DIGIT}+                              {yylval=atoi(yytext); return ID;}
[\n \t]                               yyterminate();
.
return yytext[0];
%%
```

```
%{
/* Definition section */
#include <stdio.h>
#include <stdlib.h>
%}

%token ID
%left '+' '-'
%left '*' '/'
%left UMINUS

/* Rule Section */
%%

S : E
E : E'+'{A1();}T{A2();}
| E'-'{A1();}T{A2();}
| T
;
T : T'*'{A1();}F{A2();}
| T'/'{A1();}F{A2();}
| F
;
F : '('E{A2();}')'
| '-'{A1();}F{A2();}
| ID{A3();}
;

%%
```

```
#include"lex.yy.c"
char st[100];
int top=0;

//driver code
int main()
{
        printf("Enter infix expression: ");
        yyparse();
        printf("\n");
        return 0;
}
A1()
{
        st[top++]=yytext[0];
}

A2()
{
        printf("%c", st[--top]);
}

A3()
{
        printf("%c", yytext[0]);
}
```