Lecture 21. Regular Expressions

- A ***regular expression*** describes a set of strings by giving a 'pattern' for them

c	Any nonspecial character matches itself	`A`
`.`	Any single character	`x`
c	Special character *c*	`\.`
[...]	Any character in ..., including ranges	`[a-z0-9]`
[^...]	Any character ***not*** in ..., including ranges	`[^0-9]`
$R_1 R_2$	Whatever matches R_1 followed by R_2	`[A-Z]_`
*R**	Zero or more occurrences of *R*	`[a-z][a-z]*`

- Tokens in most programming languages can be described by regular expressions

`[1-9][0-9]*`	Decimal constants in C
`0[0-7]*`	Octal constants in C
`[0-9][0-9]*\.[0-9]*`	Floating constants in C
`[A-Za-z_][A-Za-z_0-9]*`	C identifiers
`"[^"\n]*"` `'"[^"\n]*"'`	String literals in C (quoted for the shell)

egrep

- **Many UNIX tools support searching for patterns described by regular expressions**

`egrep`, `grep`, `fgrep`	**Search for lines matching regular expressions**
`ed`, `vi`, `emacs`	**Text editors**
`sed`	**Stream editor**
`awk`	**String-processing language**
More …	

- **`egrep` prints those lines that match the regular expression**

```
% cd /u/cs126/examples
% egrep emalloc *.c
compile.c:   Tree *t = emalloc(sizeof (Tree));
intlist.c:   struct intnode *p = emalloc(sizeof (struct intnode));
intlist.c:   struct intnode *p = emalloc(sizeof (struct intnode));
lookup.c:    ptr = emalloc(size*sizeof (char *));
lookup2.c:       struct node *p = emalloc(sizeof (struct node));
sort2.c:     ptr = emalloc(size*sizeof (int));
sort3.c:     ptr = emalloc(n*sizeof (int));
sublistn.c:  array = emalloc(size*sizeof (int));
sublistn2.c: array = emalloc(size*sizeof (int));
sublistn3.c: array = emalloc(size*sizeof (int));
```

egrep, cont'd

- **`/usr/dict/words` contains ≈ 25,143 words**

```
% egrep hh /usr/dict/words
beachhead
highhanded
withheld
withhold
```

How many words have 3 a's one letter apart?

```
% egrep .a.a.a /usr/dict/words | wc -l
50
% egrep .u.u.u /usr/dict/words
cumulus
```

- **`egrep` supports extended regular expressions**

^	**Beginning of line**	
$	**End of line**	
***R*+**	**One or more occurrences of *R***	`[0-9]+`
***R*?**	**Zero on one occurrence of *R***	`[0-9]*\.?[0-9]+`
$R_1 \| R_2$	**Whatever matches R_1 or R_2**	`[A-Z]|_+`
(*R*)	**Grouping**	

egrep, cont'd

- **egrep as a simple spelling checker: Specify plausible alternatives you know**

```
% egrep "n(ie|ei)ther" /usr/dict/words
neither
```

- **Find big files; du -ka prints file sizes in 1Kbyte blocks**

```
% du -ka /etc | egrep '^[5-9][0-9][0-9]'
552     /etc/fs/nfs/mount
553     /etc/fs/nfs
837     /etc/fs
850     /etc/lp/printers
883     /etc/lp
```

500 and up

- **Find all lines with signed numbers**

```
% egrep '[-+][0-9]+\.?[0-9]*' *.c
bsearch.c:           return -1;
compile.c:               strchr("+1-2*3", t->op)[1] - '0', dst,
convert.c:Print integers in a given base 2-16 (default 10)
convert.c:           sscanf(argv[i+1], "%d", &base);
...
strcmp.c:            return -1;
strcmp.c:            return +1;
```

- **egrep has its limits: It cannot match all lines that contain a number divisible by 5**

Formal Languages

- A ***language*** is a (possibly infinite) set of strings over a finite alphabet
- A regular expression describes a language: The set of all strings it 'matches'
- A ***regular language*** is any language that can be described by a regular expression
- Essential aspects of regular expressions can be specified with only

0 or 1	The alphabet
$R_1\ R_2$	R_1 followed by R_2
R_1+R_2	R_1 or R_2 (same as `egrep`'s |)
(R)	Grouping
R*	Kleene closure: 0 or more Rs (10)* (0+011+101+110)* (01*01*01*)*

- What languages over { 0 1 } are regular? All but one below are regular

Bit strings
  whose number of 0's is a multiple of 5
  that begin with 0 and end with 1
  with more 1's than 0's
  with no consecutive 1's
  for a binary number that is a multiple of 2
  for a binary number that is multiple of 5

- It is possible to cast ***any*** computation as a language problem

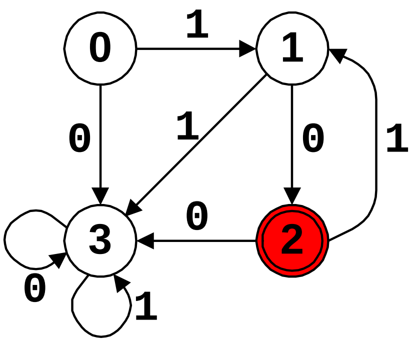
Finite State Automata

- A ***finite state automata***, an FSA, is another representation for regular languages
- A FSA is a simple machine with *N* states (0 to *N*–1)

  Start in state 0
  Read a bit
  Move to a new state depending on the bit and the current state
  Stop after reading last bit
  Accept if FSA is in one of its ***final states***, *Reject* otherwise

- An FSA 'recognizes' its input: 'Decides' if the input is in the FSA's regular language

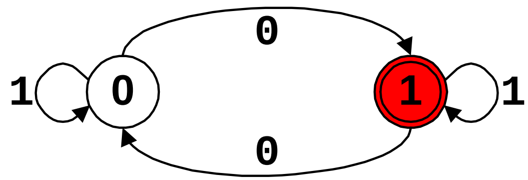
10(10)*



Transition table

	0	1
0	3	1
1	2	3
2	3	1
3	3	3

Odd number of 0s



10101010?

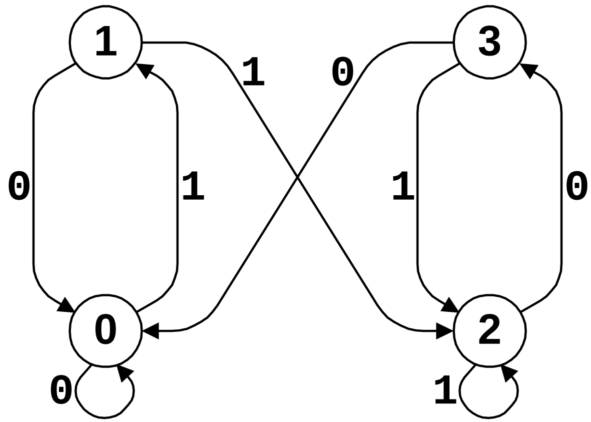
0001110?

- There is a one-to-one correspondence between FSAs and regular expressions
- It is possible to construct FSAs ***automatically*** from regular expressions

‘Bounce’ Filter

- **Flip isolated 0s and 1s in a bitstream**

  Input: `0 1 0 0 0 1 1 0 1 1`
  Output: `0 0 0 0 0 1 1 1 1 1`



- **State interpretations**

  1. **At least two consecutive 0s**
  2. **Sequence of 0s followed by a single 1**
  3. **At least two consecutive 1s**
  4. **Sequence of 1s followed by a single 0**

- **Do ‘output’ by monitoring the state transitions**

Simulating FSAs

```
int main(int argc, char *argv[]) {
    int i = 0, zero[100], one[100], final[100];
    for (i = 0; i < 100; i++)
        if (scanf("%d%d%d", &zero[i], &one[i], &final[i]) != 3)
            break;
    for (i = 1; i < argc; i++) {
        int state = 0;
        char *input = argv[i];
        for ( ; *input != '\0'; input++)
            if (*input == '0')
                state = zero[state];
            else
                state =  one[state];
        if (final[state])
            printf("%s: accepted\n", argv[i]);
        else
            printf("%s: rejected; ended in state %d\n",
                argv[i], state);
    }
    return 0;
}
```

```
% cat fsainput
3 1 0
2 3 0
3 1 1
3 3 0
```

```
% lcc fsa.c
% a.out 10101010 10 101011 <fsainput
10101010: accepted
10: accepted
101011: rejected; ended in state 3
```

FSAs Can't 'Count'

- **Theorem: No finite state machine can decide whether or not its input has the same number of 0s and 1s**
- **Proof**

  Suppose an *N*-state machine can determine if its input has equal number of 0s 1s

  Give it *N*+1 0s followed by *N*+1 1s

  Some state *must* be visited a least twice

  So, the machine would accept the same string *without* the intervening 0s

  And that string doesn't have the same number of 0s and 1s. Contradiction ∎

  0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1

- **Need more powerful machines than FSAs**

  How much more powerful? Language hierarchy

Regular	**Finite-state automata**
Context-free	**Pushdown automata (can count 2 things)**
Context-sensitive	**Linear-bounded automata**
Type 0	**Turing machines**

  Take COS 487, Theory of Automata and Computation