CS 126 Lecture T1: Pattern Matching

Outline

- **Introduction**
- Pattern matching in Unix
- Regular expressions in Unix
- Regular expressions as formal languages
- Finite State Automata
- Conclusions

Introduction to Theoretical Computer Science

- Two fundamental questions:
  - **Power**? What are the things a computer can and cannot do?
  - **Speed**? How quickly can a computer solve different classes of problems?
- Approach:
  - We don't talk about specific physical machines or specific problems, instead
  - We reduce computers to **general minimalist abstract mathematical** entities
  - We talk about **general classes** of problems
- Today: the simplest machine (an FSA) and the class of problems it can solve

Why Learn Theory?

- In theory...
  - Deeper understanding of what a computer or computing is
  - Pure science: some of the most challenging “holy grails” (why climb a mountain? because it’s there!)
  - Philosophical implications
- In practice... (some examples)
  - A sequential circuit: theory of finite state automata
  - Compilers: theory of context free grammar
  - Cryptography: complexity theories

Outline

- ~~Introduction~~
- **Pattern matching in Unix**
- Regular expressions in Unix
- Regular expressions as formal languages
- Finite State Automata
- Conclusions

Unix Tools

- Remember what we said about the success of Unix?
  - A large number of very simple small tools
  - Unix provides “glue” that allows you to connect them together to perform useful tasks effortlessly
- Some of the most important tools have to do with pattern matching:
  - **`grep`**
  - **`awk`**
  - **`sed`**
  - **`more`**
  - **`emacs`**
  - **`perl`**

Demos

- Words and partial words
- Which files have the pattern
- Interaction with other commands

grep

general regular expression pattern matching

- filter
- stdout gets only those lines from stdin that "match" argument string

- Elementary examples:

- Does a file contain a string?

```
grep Smith classlist
```

- Which file contains a string?

```
grep grep *.sl
```

Any file names that end with ".sl": "Wildcard" file name matching ("glob style"): Unix shell feature, not to be confused with grep syntax

- Just give me the data of interest...

```
a.out | grep -v DEBUG
```

Dictionary

- Crossword puzzle or Scrabble too time consuming?

usr/dict/words is list of words in dictionary
25,486 words

Grep and similar tools can be effective in "finding" words

```
grep hh /usr/dict/words
beachhead
highhanded
withheld
withhold
```

```
grep .a.a.a usr/dict/words | wc
34
```

```
grep .u.u.u usr/dict/words
cumulus
```

A dot matches any character, part of grep syntax, not to be confused with the dots in file names

man grep excerpts

Name

grep - search file for regular expression

Syntax

grep [option...] expression [file...]

Description

Commands of the grep family search the input files (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output.

Take care when using special characters in the expression because they are also mea to the Shell. It is safest to enclose the enti expression argument in single quotes ' '.

Options

-c Produces count of matching lines only.
-e Produces count of matching lines only.
-i Considers upper and lowercase letter ide
-n Precedes each matching line with its line
-v Displays all lines that do not match.

Restrictions

Lines are limited to 256 chars;
longer lines are truncated.

See Also

Outline

- ~~Introduction~~
- ~~Pattern matching in Unix~~
- **Regular expressions in Unix**
- Regular expressions as formal languages
- Finite State Automata
- Conclusions

grep pattern conventions

conventions for grep:

c	any non-special char matches itself
^	beginning of line
$	end of line
.	any single character
[...]	any character in [a-z]
[^...]	any character not in [a-z]
r*	zero or more occurrences of r
r+	one or more occurrences of r

egrep or grep -E only

- "regular expression"
  - name for grep patterns
  - specific technical meaning in theoretical CS
    [stay tuned for precise definition]

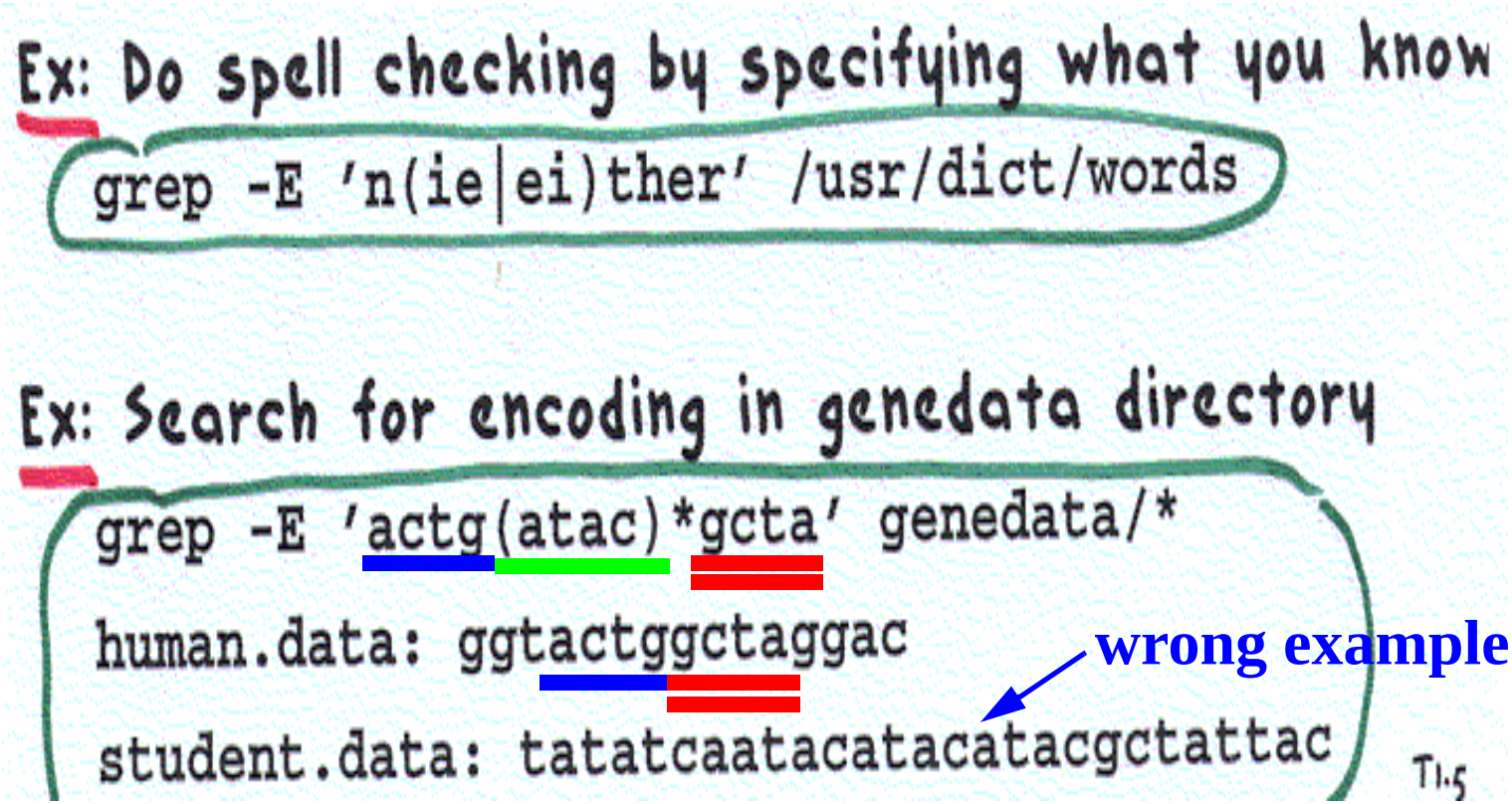
- "extended" regular expressions (grep -E) or egrep

(r)	grouping
r1 | r2	logical or

More Demos

- regular expressions
- egrep or grep -E features
- escape characters
- command line options

Examples



taactgatacatacatacatacgctaat

More Examples

- Find all references to Java

```
grep '[Jj]ava' text
grep -E 'java|Java' text
```

grep -i java

Unix command displaying disk usage

- Find all big files

```
du -a | grep -E "^[1-9][0-9][0-9][0-9]"
```

- Find all lines with dollar amounts on them

```
grep -E '[$][0-9]+.[0-9]*' myfile
```

$7.96

BUG: matches $7A46.

Ex: fix this bug

How to say it if you want a "real" dot? use an "escape character" in front...

- Find all words with no vowels and 6 or more lett

```
grep -v '[aeiou]' /usr/dict/words | grep '......'
```

rhythm

syzygy

"Escape" Character

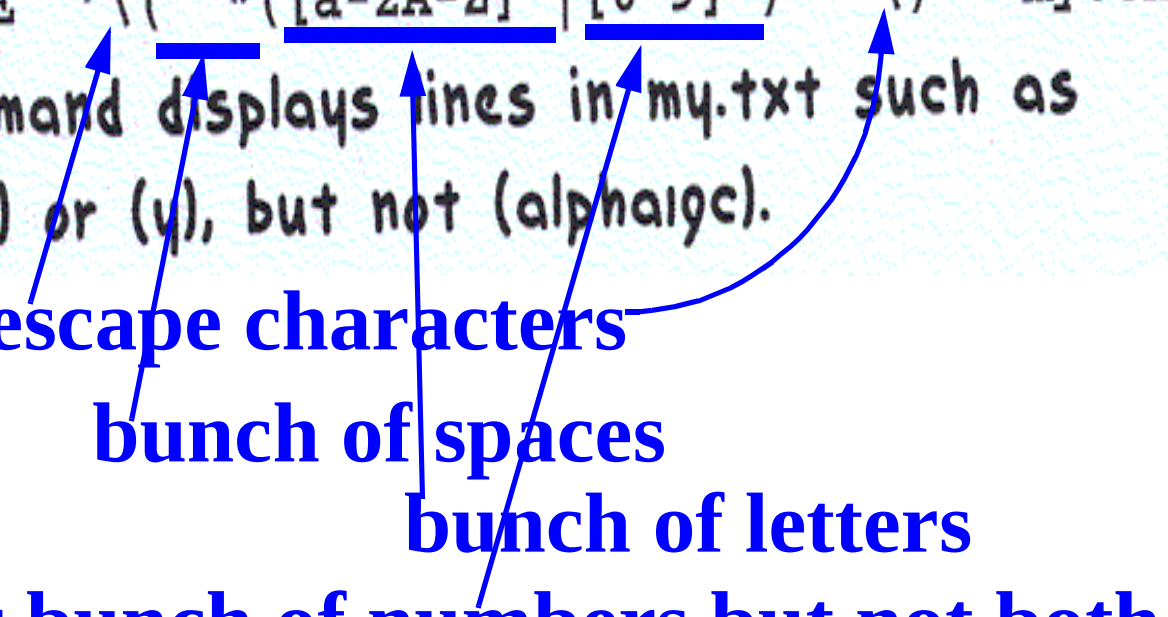
- Matches involving special chars can be complex

Ex: excerpt from "man grep":

```
grep -E  '\(  *([a-zA-Z]*|[0-9]*)  *\)'  my.txt
```

This command displays lines in my.txt such as

(783902) or (y), but not (alpha19c).



Pattern Matching alternatives in UNIX

grep

- -E "extended" regular expressions
- -f search for multiple patterns

more: (Try it!)

Substitution (editing), not just matching

emacs, ex (various ways)

- interactive

sed

- filter
- line-by-line editing

```
sed 's/apples/oranges/g' file
```

awk, perl: Pattern matching "languages"

- matching
- substitution
- pattern manipulation
- variables
- numeric capabilities
- control and logic

Testament to Flexibility and Power of Unix Philosophy

- Simple general tools + glue (scripting, and shell)
- The advantages are being magnified in the age of web

Outline

- ~~Introduction~~
- ~~Pattern matching in Unix~~
- ~~Regular expressions in Unix~~
- **Regular expressions as formal language**
  - **Regular expression generator**
- Finite State Automata
- Conclusions

Unix vs. Theory

- Specifying "pattern" for grep can be complex

```
^[^aeiou]*a[^aeiou]*e[^aeiou]*i
          [^aeiou]*o[^aeiou]*u[^aeiou]*$
```

- What kinds of patterns can be specified?
  - match all lines containing an even number?
  - match all lines containing a prime number?
- Which aspects are essential?

- Unix regular expressions are **useful**
- But more **complex** than the theoretical minimum
- But are they any more **powerful**? no.

Formal Languages

- Formal definitions
  - An **alphabet**: a finite set of symbols
  - A **string**: a finite sequence of symbols from the alphabet
  - A **language**: a (potentially infinite) set of strings over an alphabet
- Intriguing topic: **finite representation** of a language
  - How?
    - + language **generators** (a set of rules for producing strings)
    - + language **recognizers**
  - We will study different **classes of languages**, their generators, and their recognizers, each more powerful than the previous ones
  - There are even strange languages that fail all these finite representational methods!

Why Study Formal Languages

- Can cast any computation as a language problem
- Start by trying to understand simple languages
- Do so by building a machine specifically designed for the task

(Bare Minimum) Regular Expression: Generator Rules

Regular Expression

0 or 1	symbols
(a)	grouping
ab	concatenation
a+b	logical or
a*	closure (0 or more replications)

where a and b are regular expressions

Ex:

(10)*

(0+011+101+110)*

(01*01*01*)*

Regular Languages

Every regular expression (RE) describes a languag
(the set of all strings that "match")

- Regular Language:
  - any language that can be described by an RE

What languages are regular?

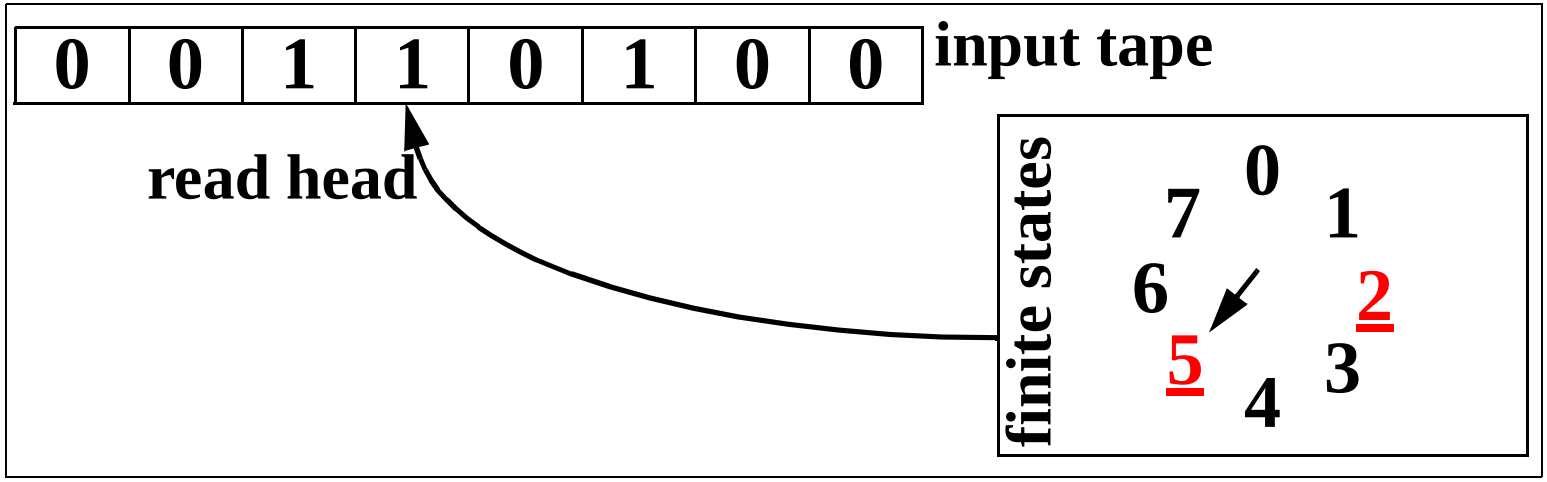
Examples (all but one of the following are regular)

all bit strings $0(0+1)^*1$
that begin with 0 and end with 1
whose number of 0's is a multiple of 5
with more 1's than 0's
with no consecutive 1's

Outline

- ~~Introduction~~
- ~~Pattern matching in Unix~~
- ~~Regular expressions in Unix~~
- ~~Regular expressions as formal languages~~
- **Finite State Automata**
  - **Regular expression recognizer and beyond**
- Conclusions

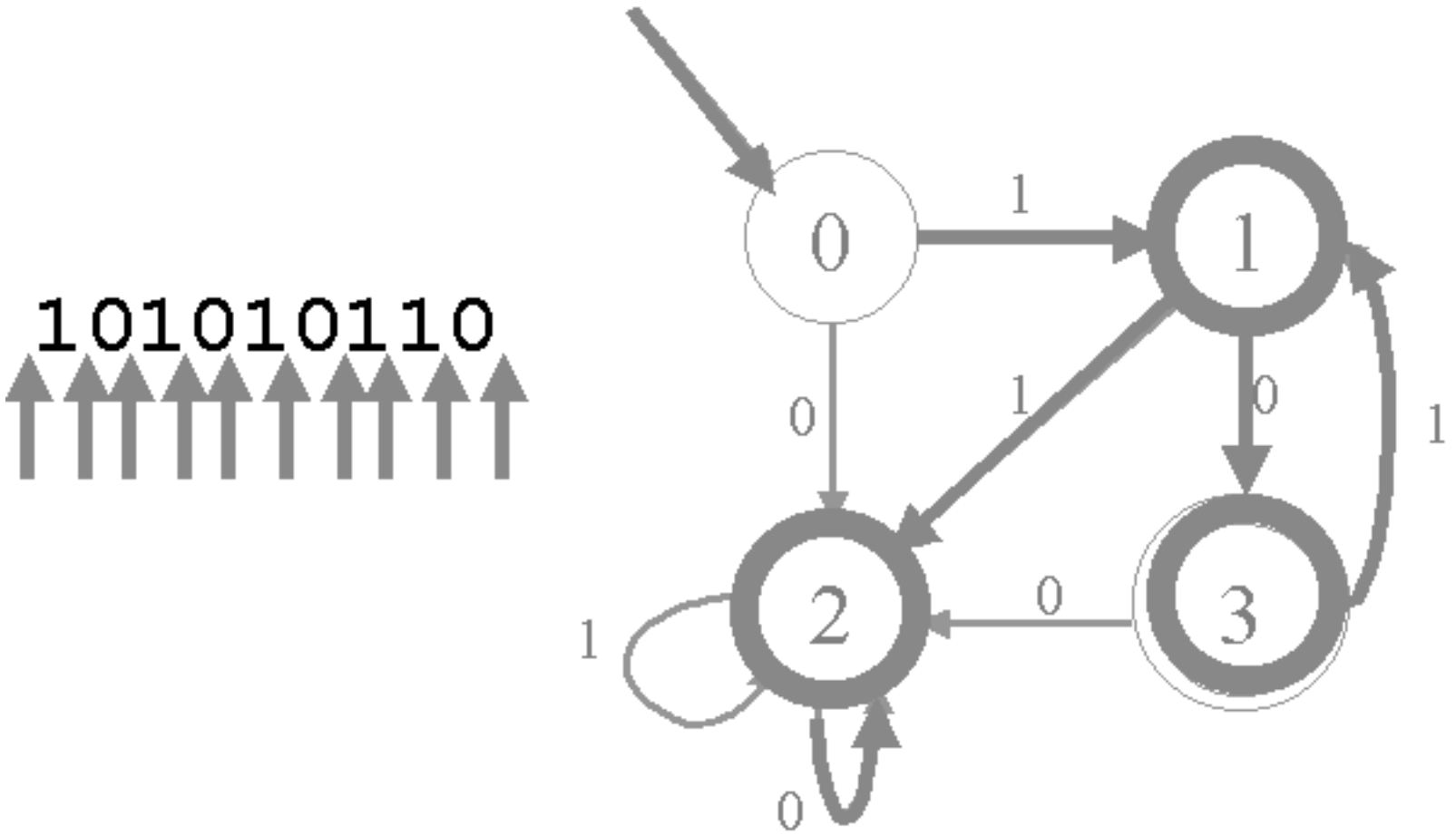
Finite State Automata: Regular Language Recognizers



Simple machine with N states FSA

- start in state 0
- read a bit
- move to new state
  (depends on bit, current state)
- stop when last bit read
  ACCEPT if in specified state X
  REJECT otherwise

FSA Example Demo



FSA Example

beginning state

read a 1, and the string still has a chance

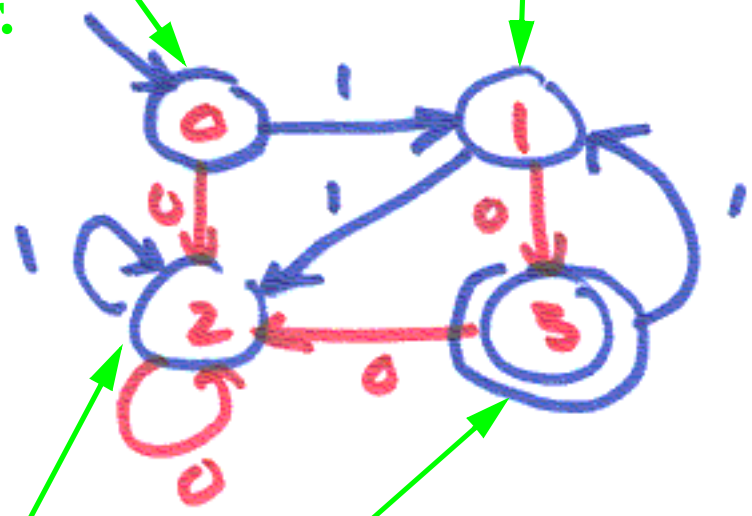
Can kill any number of these “ears”, and the string will still be accepted!
Important implication later.

Ex:

10(10)*

input 10101010?

state 013131313

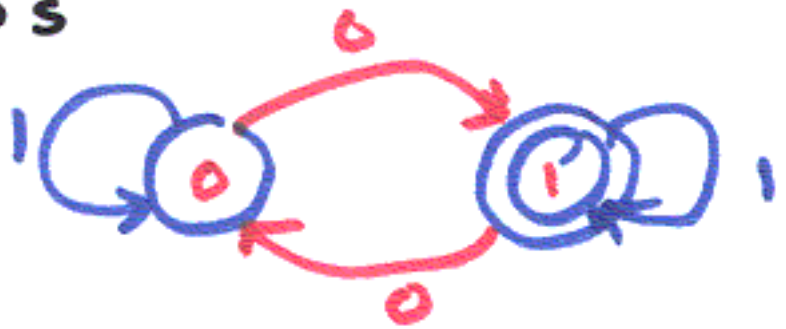


dead state

read a 0, and the string is accepted if we stop now

Second FSA Example

Ex: odd number of 0's



00011110?

01011110 ✗

FSAs and REs are equivalent

[stay tuned]

An Application

"Bounce" filter to remove noise from data

- remove isolated 0's and 1's in a bitstream

input:

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

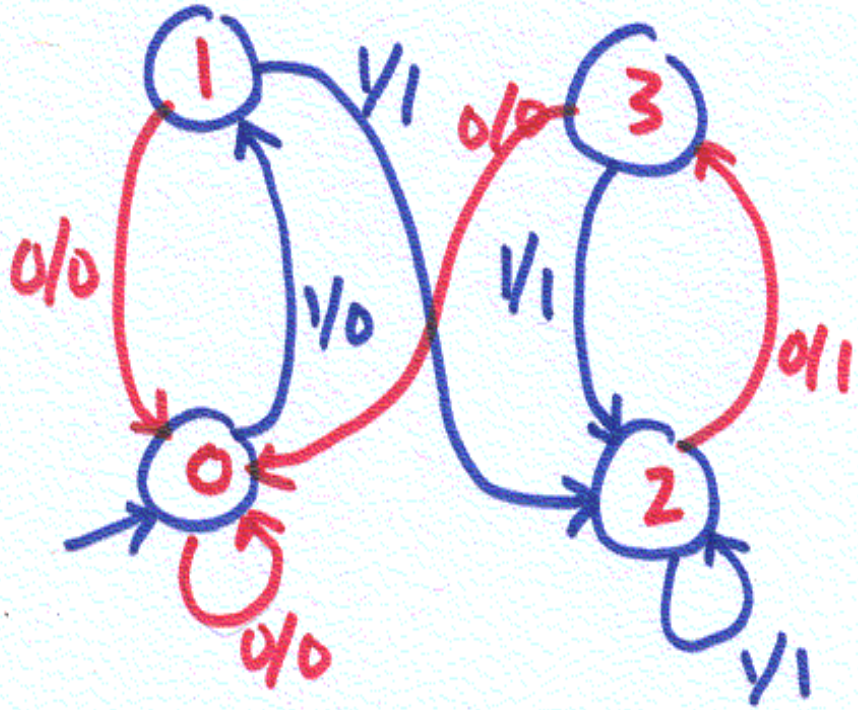
output (one-bit delay)

0* 0 0 0 0 0 1 1 1 1 1 1 1

x/y

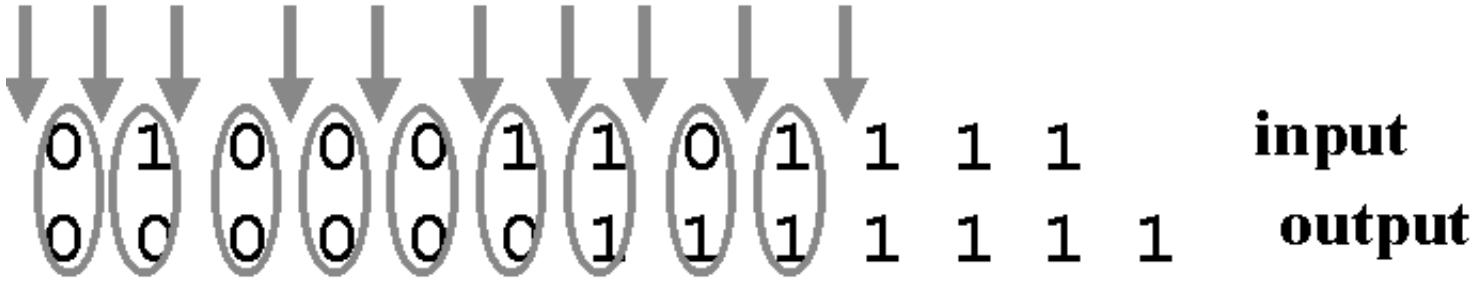
Third FSA Example: Add Outputs

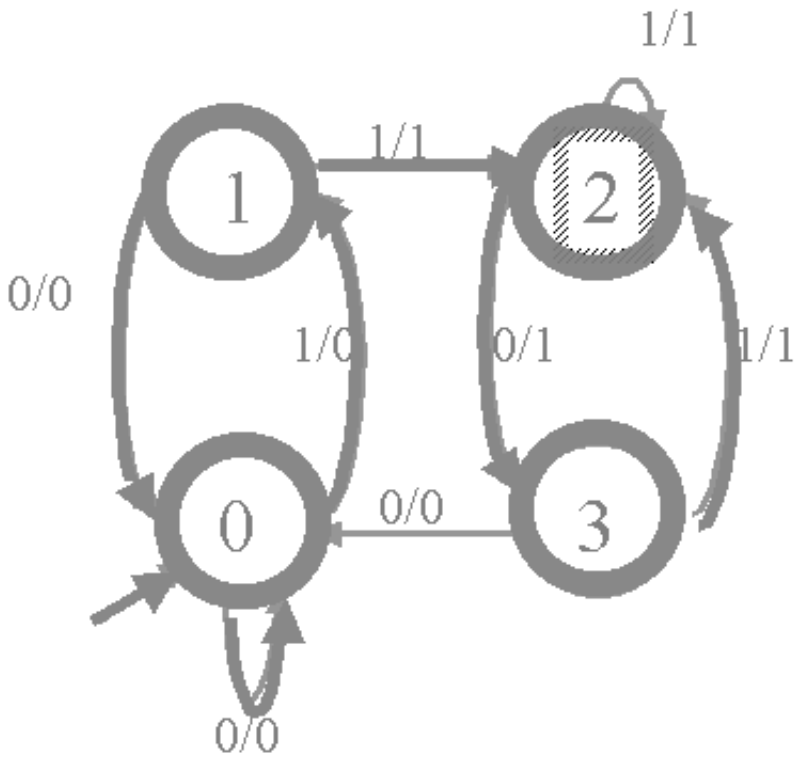
if input is x, change state and output y



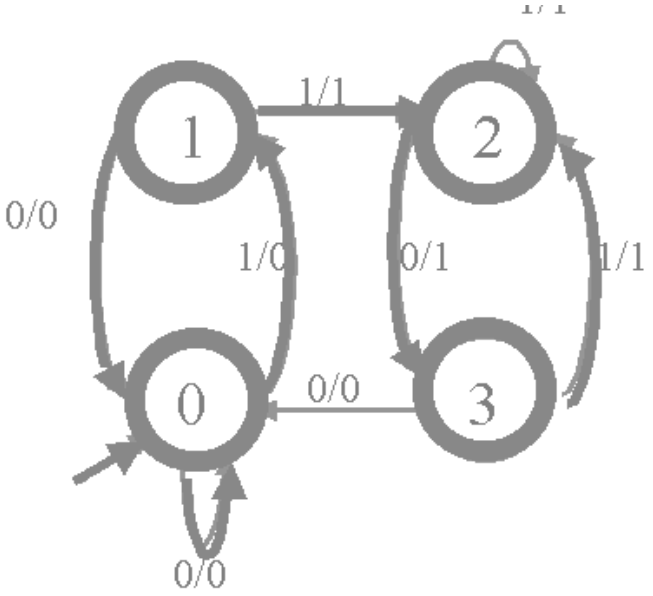
state	in	out	next
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	2
2	0	1	3
2	1	1	2
3	0	0	0
3	1	1	2

Bounce Filter Demo





State Meaning



State interpretations

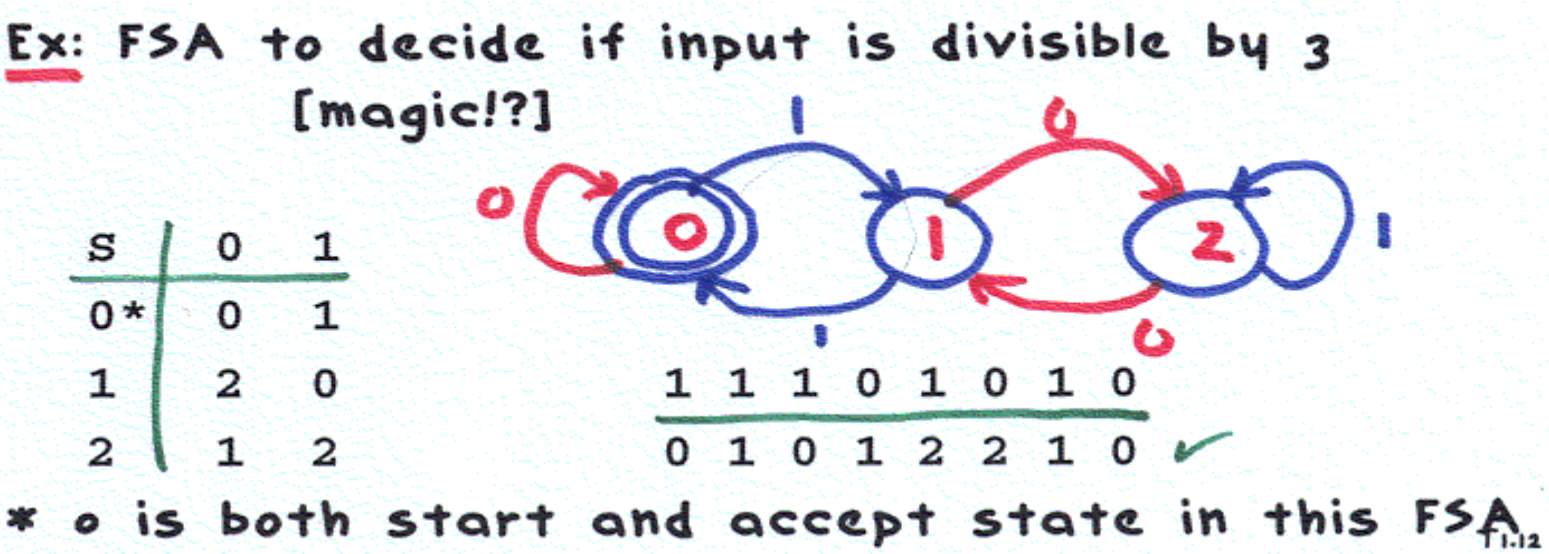
0: at least two consecutive 0's

1: seq. of 0's followed by a 1

2: at least two consecutive 1's

3: seq. of 1's followed by a 0

Fourth FSA Example



- How does it work?
  - Every time we scan one more digit: `x = x<<1 + y`
  - Equivalent to: `x = x*2 + y`
  - Three states: `x%3==0, x%3==1, x%3==2`
  - Six transitions:

```
(0*2+0)%3==0, (0*2+1)%3==1
(1*2+0)%3==2, (1*2+1)%3==0
(2*2+0)%3==1, (2*2+1)%3==2
```

C program to simulate FSAs

```
#include <stdio.h>

main(int argc, char*argv[])
   {
      int zero[100], one[100]; char c;
      FILE *fsa = fopen(argv[1], "r");
      int state, N, accept;
      fscanf(fsa, "%d ", &accept);
      for (N = 0; !feof(fsa); N++)
         fscanf(fsa, "%d  %d ", &zero[N], &one[N])
      state = 0;
      while ((c = getchar()) != EOF)
         if (c == '0') state = zero[state];
                  else state = one[state];
      if (state == accept) printf("Accepted ");
      else printf("Rejected ");
   }
```

Outline

- ~~Introduction~~
- ~~Pattern matching in Unix~~
- ~~Regular expressions in Unix~~
- ~~Regular expressions as formal languages~~
- ~~Finite State Automata~~
- **Conclusions**

Looking Ahead...

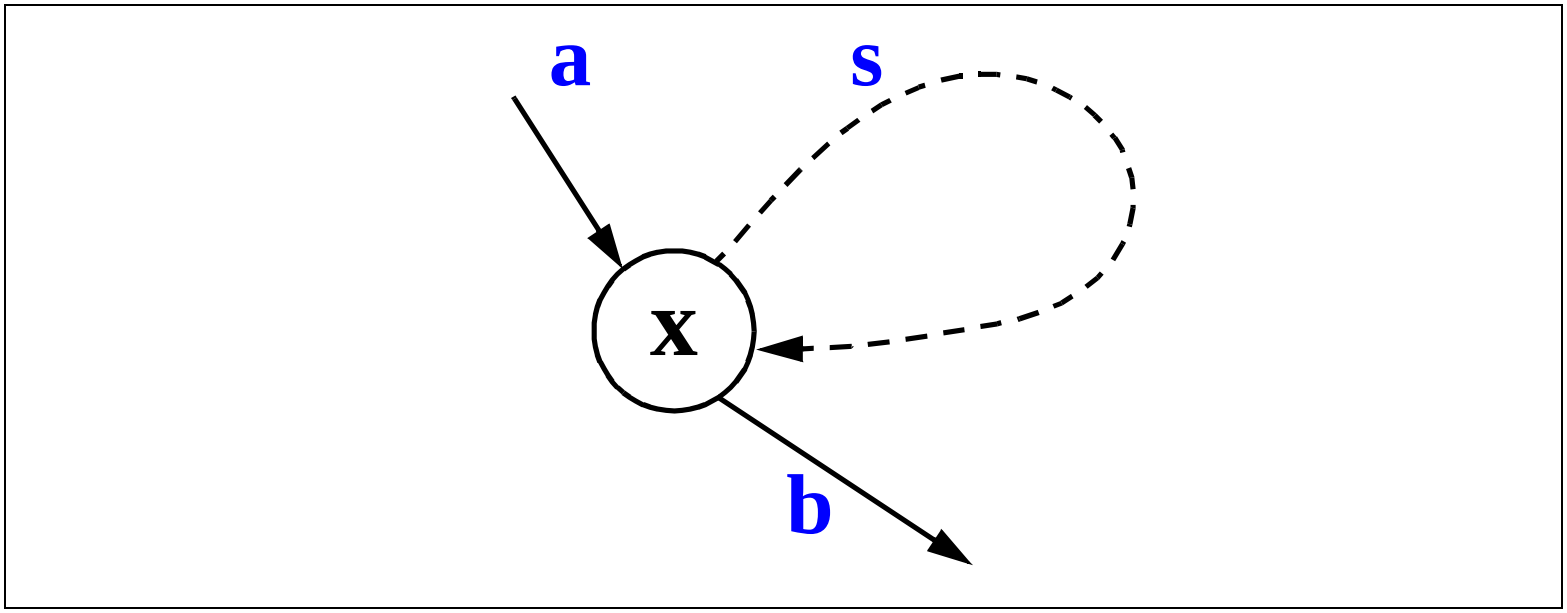
- Regular expressions are very simple languages, and FSAs are very simple machines
- What kind of languages cannot be expressed by regular expressions? What tasks can't be performed by FSAs?
- Basic idea: because the machine only has a finite number of states N, it can't remember more than N things
- So any language that requires remembering infinite number of things is not regular
- This is something that we will do a couple more times:
  - Define a machine, and understand its behavior
  - Find things it can't do
  - Define a more powerful machine
  - Repeat until we either run out of machines or problems
  - (Hmm... which will we run out first?)

A language that is not regular

FSAs can't "count"

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

A Warm-up Result



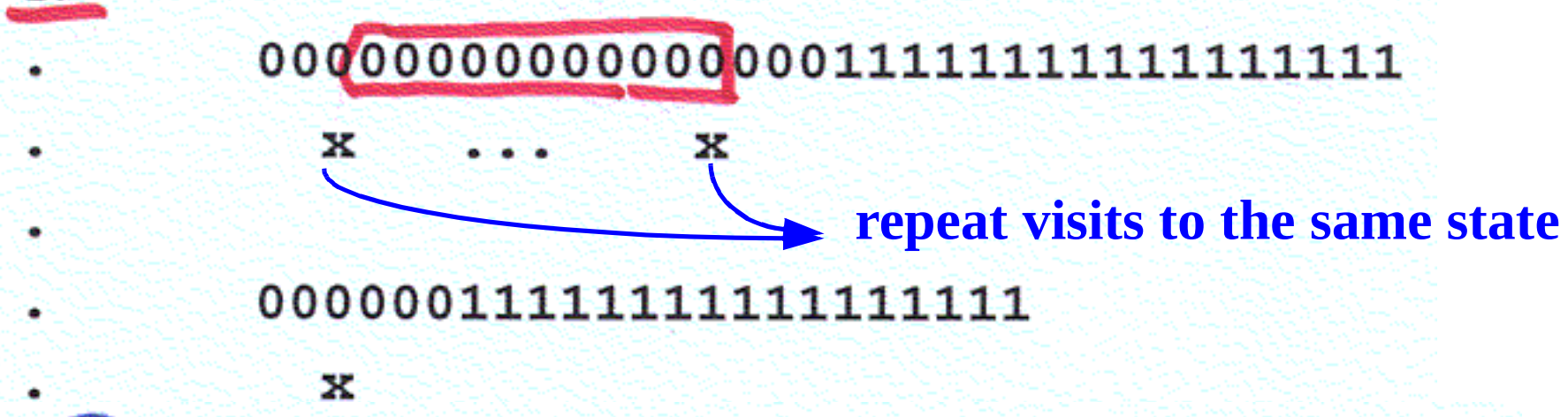
- Remember we said we could cut any ear when showing the first example of FSA?
- More formally, if **`a(s)*b`** is accepted, then **`ab`** is accepted

Proof:

- Suppose an N-state machine does it.
- Give it N+1 0's followed by N+1 1's
- Some state must be revisited
- Machine would accept the same string without the intervening 0's
- That string doesn't have the same number of 0's and 1's

Contradiction.

Ex:



Need to consider more powerful machines

What Have We Learned Today

- How to write Unix-style regular expressions
- How to use their associated Unix tools to perform useful and interesting tasks
- “Formal” regular expressions
- FSAs, how to trace their execution
- Constructing simple FSAs to solve problems
- Understanding the limits of REs and FSAs: being able to spot what problems they cannot solve (you’ll get better at this after a few more lectures...)