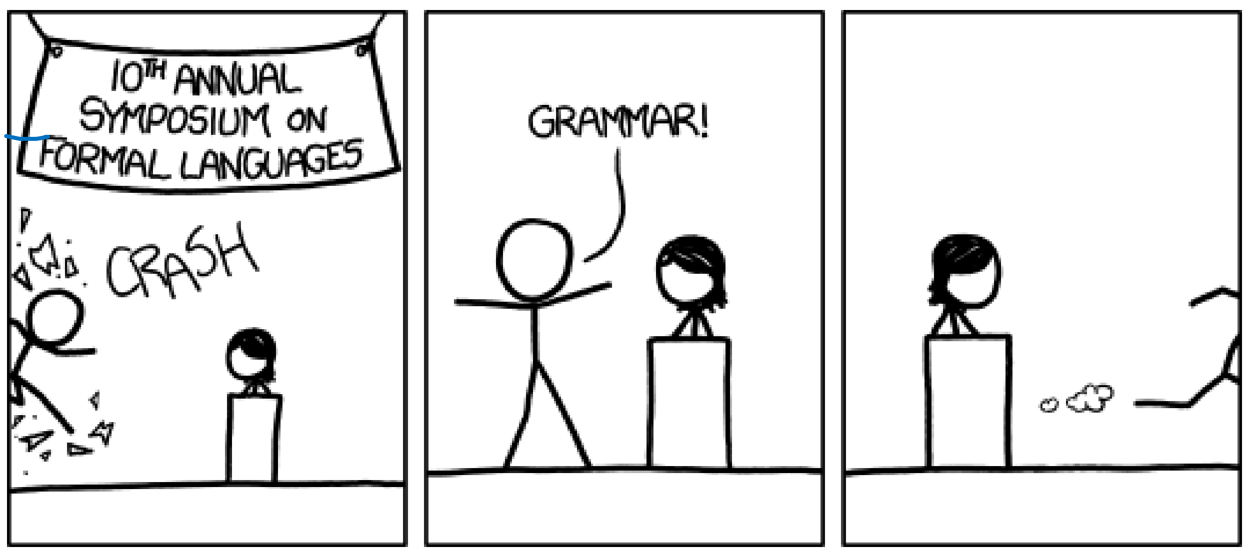
Fall 2015

Lecture 21: Context-free grammars and finite state machines

$1^*(0 \cup 1)0$



- All binary strings that have at least one 1.

$(0 \cup 1)^* \, 1 \, (0 \cup 1)^*$

- All binary strings that have an even # of 1's

$(0^* \, 1 \, 0^* \, 1)^* \, 0^*$

- All binary strings that *don't* contain 101

$0^* (1 \, 1^* \, 00 \, 0^*)^* \, 1^* \, 0^*$

limitations of regular expressions

- Not all languages can be specified by regular expressions

$L = \{0^n 1^n : n \geq 0\}$

- Even some easy things like
  - Palindromes
  - Strings with equal number of 0's and 1's

- But also more complicated structures in programming languages
  - Matched parentheses

    ( ) (( ))
  - Properly formed arithmetic expressions
  - etc.

- A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
  - A finite set **V** of *variables* that can be replaced
  - Alphabet Σ of *terminal symbols* that can't be replaced
  - One variable, usually **S**, is called the *start symbol*

- The rules involving a variable **A** are written as

$$\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$$

where each w_i is a string of variables and terminals:

$$w_i \in (\mathbf{V} \cup \Sigma)^*$$

how CFGs generate strings

- Begin with start symbol **S**

- If there is some variable **A** in the current string you can replace it by one of the w's in the rules for **A**
  - $\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$ $B \rightarrow w_{k+1} \mid w_{k+2}$
  - Write this as $x\mathbf{A}y \Rightarrow xw_1y$
  - Repeat until no variables left

- The set of strings the CFG generates are all strings produced in this way that have no variables

Example: $S \to 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

$S \to 0S0 \to 01S10 \to 011S110 \to 0110110$

L = palindroms

Example: $S \to 0S \mid S1 \mid \varepsilon$

$L = \{0^*1^*\}$.

example

Grammar for $\{0^n 1^n : n \geq 0\}$

(all strings with same # of 0's and 1's with all 0's before 1's)

$S \to 0S1 \mid \varepsilon$

Example: Grammar for Matched Paranthesis $\Sigma = \{(,)\}$.

$\cancel{S \to (S) \mid S() \mid ()S \mid \varepsilon}$

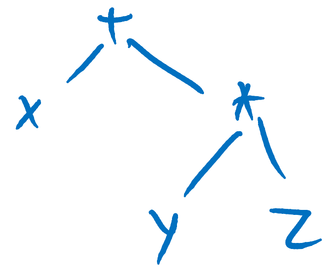
$()$ $(())$ ✓ $)($ X

$(())(())$

$S \to SS \mid (S) \mid \varepsilon$

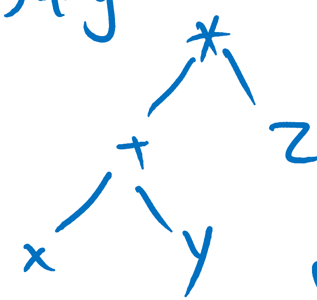
$S \to S(S)S \mid \varepsilon$

$$\mathbf{E} \rightarrow \mathbf{E+E} \mid \mathbf{E*E} \mid \mathbf{(E)} \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$



Generate $(2*x) + y$

$E \rightarrow E+E \rightarrow E+y \rightarrow (E)+y \rightarrow (E*E)+y$
$\rightarrow (2*E)+y \rightarrow (2*x)+y$



Generate $x+y*z$ in two fundamentally different ways

$E \rightarrow E+E \rightarrow x+E \rightarrow x+E*E \rightarrow x+y*E \rightarrow x+y*z.$
$E \rightarrow E*E \rightarrow E*z \rightarrow E+E*z \rightarrow x+E*z \rightarrow x+y*z$

parse trees

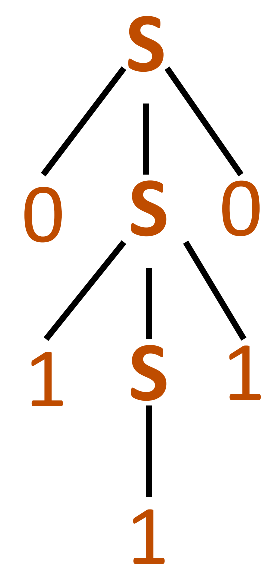
Suppose that grammar G generates a string x

A **parse tree** of x for G has

- Root labeled S (start symbol of G)
- The children of any node labeled **A** are labeled by symbols of w left-to-right for some rule **A** $\rightarrow$ w
- The symbols of x label the leaves ordered left-to-right

$$\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \varepsilon$$

Parse tree of 01110:



CFGs and recursively-defined sets of strings

- A CFG with the start symbol **S** as its only variable recursively defines the set of strings of terminals that **S** can generate

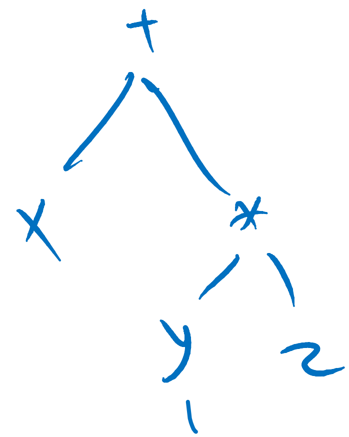
- A CFG with more than one variable is a simultaneous recursive definition of the sets of strings generated by *each* of its variables
  - Sometimes necessary to use more than one

building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

$$\mathbf{E} \rightarrow \mathbf{T} \mid \mathbf{E+T}$$
$$\mathbf{T} \rightarrow \mathbf{F} \mid \mathbf{F*T}$$
$$\mathbf{F} \rightarrow (\mathbf{E}) \mid \mathbf{I} \mid \mathbf{N}$$
$$\mathbf{I} \rightarrow x \mid y \mid z$$
$$\mathbf{N} \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$x + y * z$



$E \rightarrow E+T \rightarrow E + F*T \rightarrow E+I*T \rightarrow E+y*T$

$\rightarrow E+y*F \rightarrow E+y*I \rightarrow E+y*z \rightarrow T+y*z$

$\rightarrow F+y*z \rightarrow I+y*z \rightarrow x+y*z$

Backus-Naur form (same as CFG)

BNF (Backus-Naur Form) grammars

– Originally used to define programming languages
– Variables denoted by long names in angle brackets, e.g.
  <identifier>, <if-then-else-statement>,
  <assignment-statement>, <condition>
  $::=$ used instead of $\rightarrow$

BNF for C

```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```

parse trees

Back to middle school:

<sentence>::=<noun phrase><verb phrase>

<noun phrase>::==<article><adjective><noun>

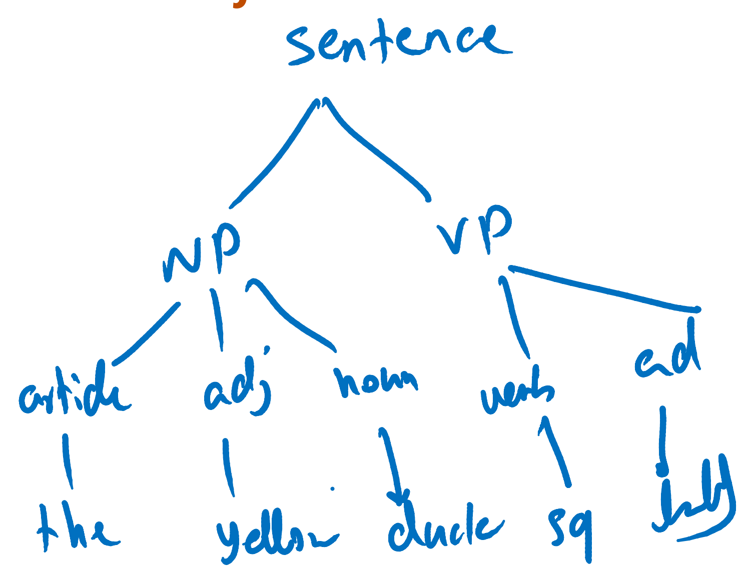
<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

Parse:

The yellow duck squeaked loudly

The red truck hit a parked car

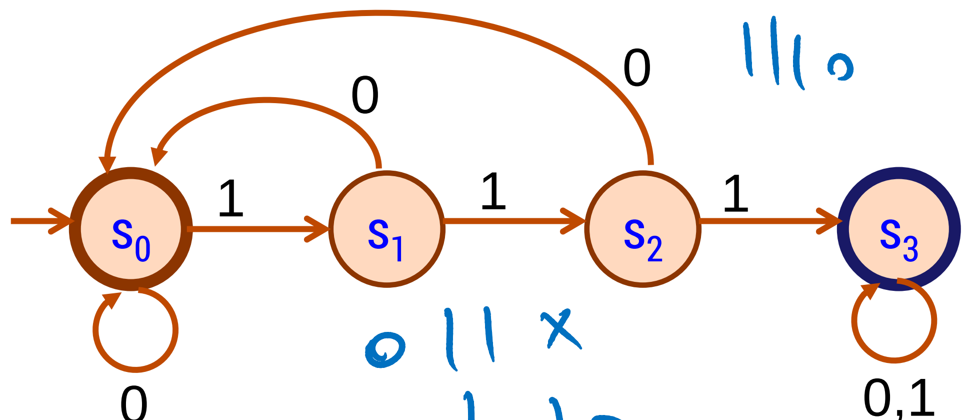


finite state machines

- States
- Transitions on inputs
- Start state and final states
- The language recognized by a machine is the set of strings that reach a final state

L = { three consecutive ones }

State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
$\mathbf{s_3}$	s_3	s_3



1110

011 x

01010 x

0110010

applications of FSMs (aka finite automata)

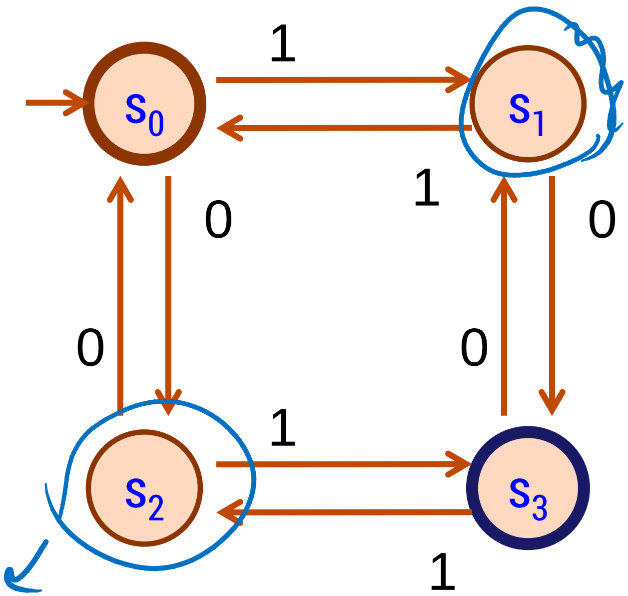
- Implementation of regular expression matching in programs like `grep`
- Control structures for sequential logic in digital circuits
- Algorithms for communication and cache-coherence protocols
  - Each agent runs its own FSM
- Design specifications for reactive systems
  - Components are communicating FSMs

applications of FSMs (aka finite automata)

- Formal verification of systems
  - Is an unsafe state reachable?
- Computer games
  - FSMs provide worlds to explore
- Minimization algorithms for FSMs can be extended to more general models used in
  - Text prediction
  - Speech recognition

what language does this machine recognize?

L_2 { all strings when they have an odd number of 0s and odd # of 1s }



odd ones and even zeros

odd zeros and even 1s

can we recognize these languages with DFAs?

- $\emptyset$

- Σ^*

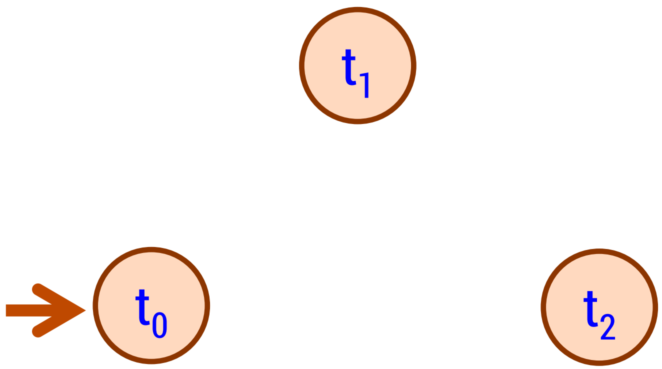
- $\{ x \in \{0,1\}^* : \text{len}(x) > 1\}$

FSM that accepts binary strings with a 1 three positions from the end

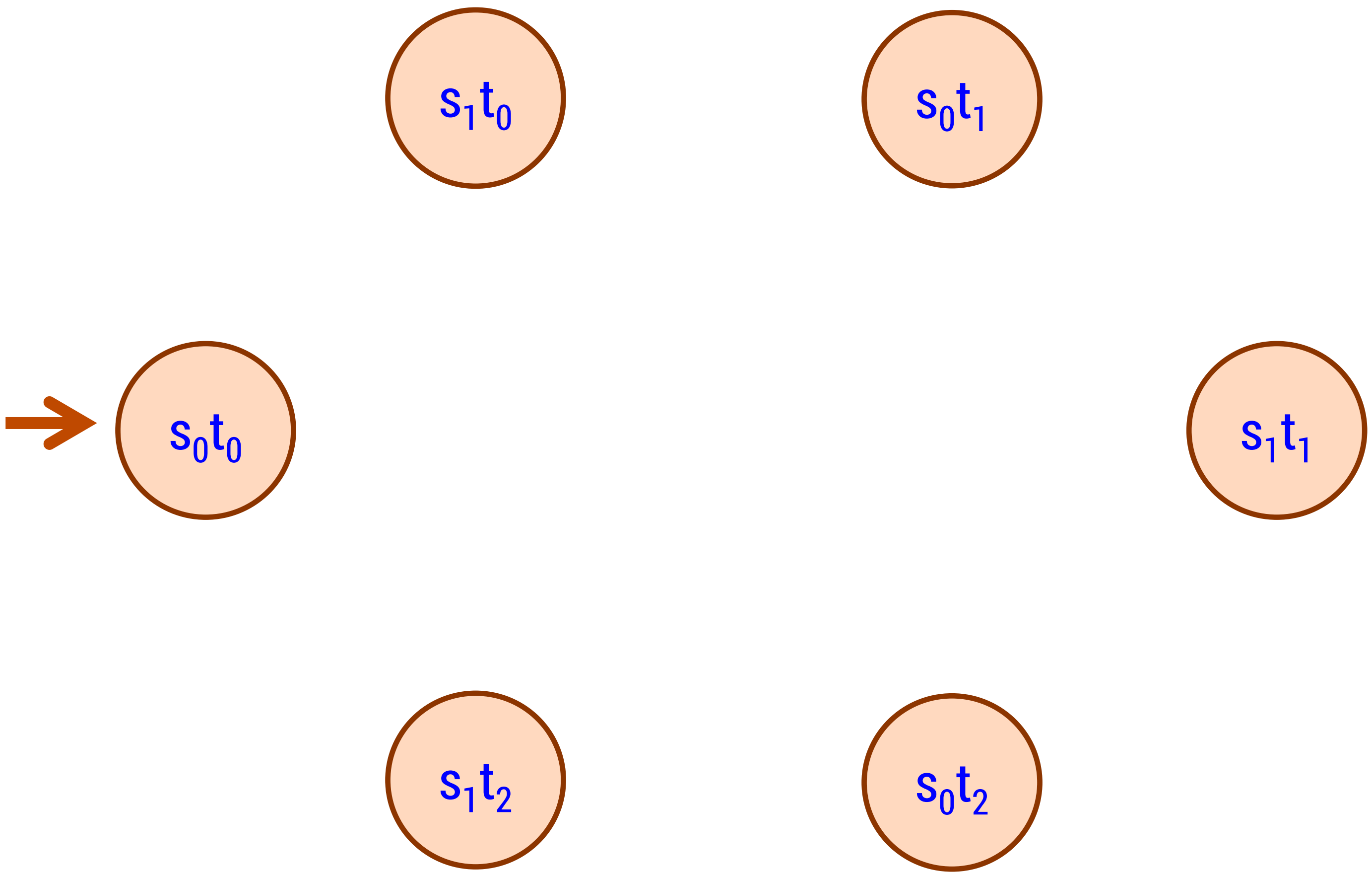
strings over {0, 1, 2}*

M_1: Strings with an even number of 2's

M_2: Strings where the sum of digits mod 3 is 0



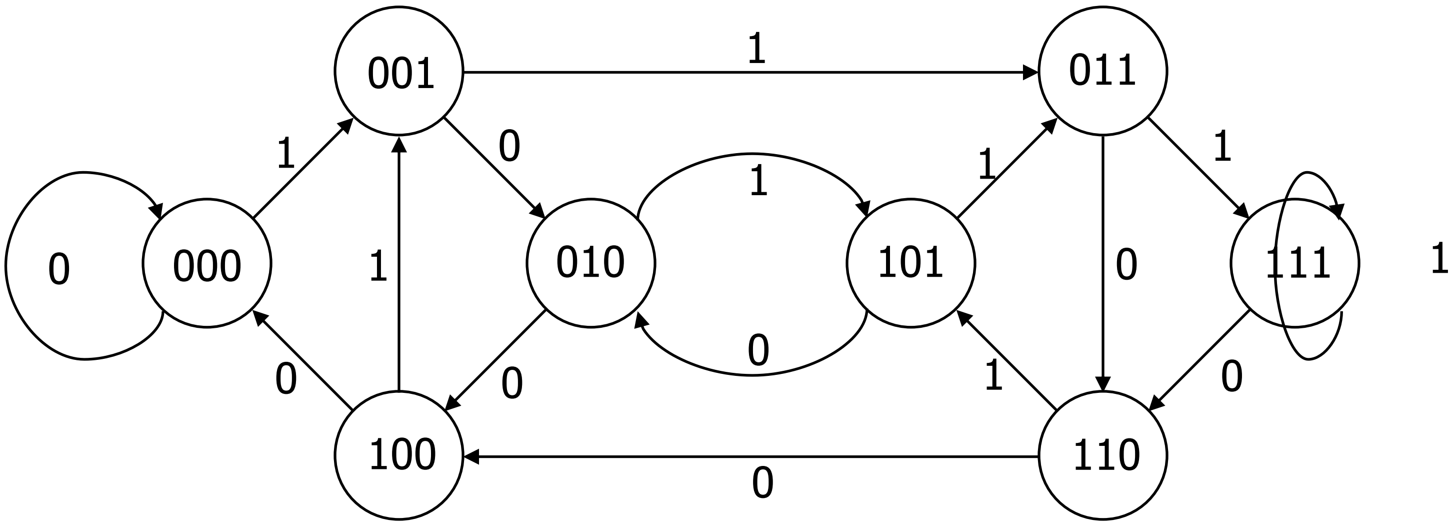
both: even number of 2's and sum mod 3 = 0

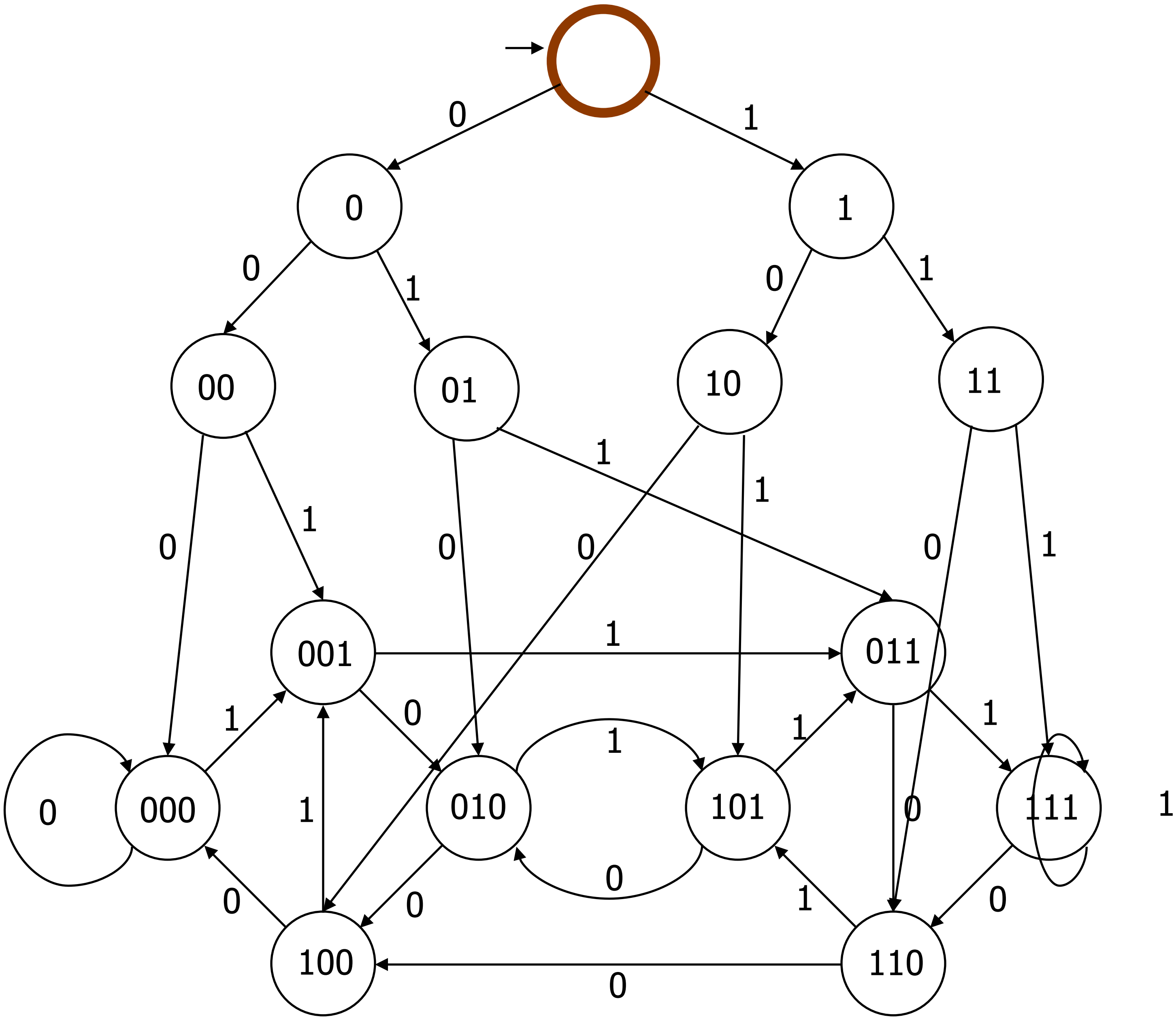


DFA that accepts strings of a’s, b’s, c’s with no more than 3 a’s

“Remember the last three bits”

3 bit shift register





0
1
0
0
1
0
1
00
01
10
11
1
1
0
1
0
0
0
1
001
1
011
1
0
1
1
0
000
1
010
101
0
111
1
0
0
0
1
0
100
0
110