Lecture 24: Finite State Machines, Languages and Grammars

Jingjin Yu | Computer Science @ Rutgers

RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

December 4, 2020

Algorithmic Robotics & Control Lab

Outline

- Lecture 23 review
- Finite state machines
  - Designing a vending machine
  - Deterministic finite automaton and regular language
- Languages and grammars
  - DFA and regular language
  - Pushdown automata and context-free language

- A repeating note: **make sure you read the textbook**

L23: Boolean Functions

- **Def^n**: A Boolean function is a function that maps a vector of Boolean values to Boolean values, i.e., $f: \{0,1\}^n \to \{0,1\}$
  - **Ex**: $f(x,y) = x\bar{y}, g(x,y,z) = xy + \bar{z}$
  - **Ex**: $F(x,y,z) = f(x,y)g(x,yz) = x\bar{y}(xy + \bar{z})$
- # of Boolean functions for n variables: 2^{2^n}
- Identities
- Duality: $f = g \Leftrightarrow d(f) = d(g)$

TABLE 5 Boolean Identities.

Identity	*Name*
$\bar{\bar{x}} = x$	Law of the double complement
$x + x = x$ $x \cdot x = x$	Idempotent laws
$x + 0 = x$ $x \cdot 1 = x$	Identity laws
$x + 1 = 1$ $x \cdot 0 = 0$	Domination laws
$x + y = y + x$ $xy = yx$	Commutative laws
$x + (y + z) = (x + y) + z$ $x(yz) = (xy)z$	Associative laws
$x + yz = (x + y)(x + z)$ $x(y + z) = xy + xz$	Distributive laws
$\overline{(xy)} = \bar{x} + \bar{y}$ $\overline{(x + y)} = \bar{x}\,\bar{y}$	De Morgan's laws
$x + xy = x$ $x(x + y) = x$	Absorption laws
$x + \bar{x} = 1$	Unit property
$x\bar{x} = 0$	Zero property

L23: Representing Boolean Functions

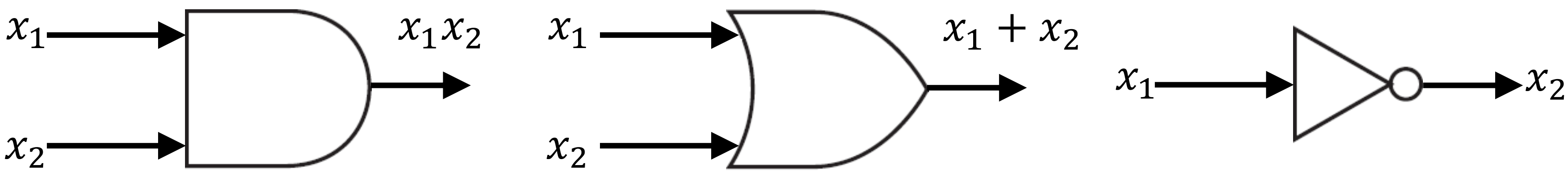
- Given inputs and outputs of Boolean functions, it can be written as a **sum-of-product**
  - **Def^n**: A **literal** for a Boolean variable x is x or $\bar{x}$.
  - **Def^n**: A **minterm** of Boolean variables $x_1, \dots, x_n$ is a **Boolean product** $y_1 \dots y_n$ where y_i is a literal for the variable x_i . That is, $y_i = x_i$ or $y_i = \overline{x_i}$.
  - **Def^n**: A **sum-of-product** Boolean function is composed of sums of minterms
- **Algorithm**: Set $F = 0$. For each entry $1 \le k \le m$ of function F, if $F^k = 1$, add an additive minterm to F such that if $x_i^k = 1$, the literal x_i^k is added to the minterm; otherwise, the literal $\overline{x_i^k}$ is added to the minterm

TABLE 1

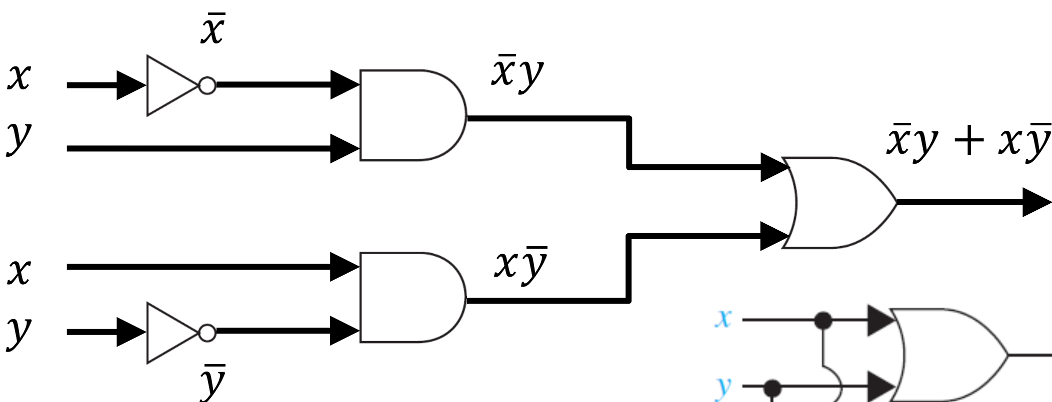
	x	y	z	F	G
1	1	1	1	0	0
2	1	1	0	0	1
3	1	0	1	1	0
4	1	0	0	0	0
5	0	1	1	0	0
6	0	1	0	0	1
7	0	0	1	0	0
8	0	0	0	0	0

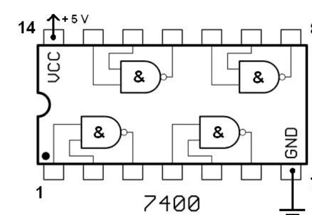
L23: Logic Gates & Circuits

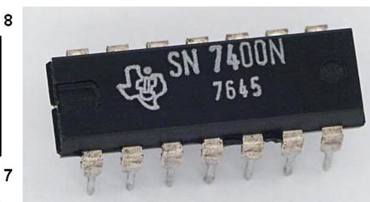
- NOT (invertor), OR, and AND gates.

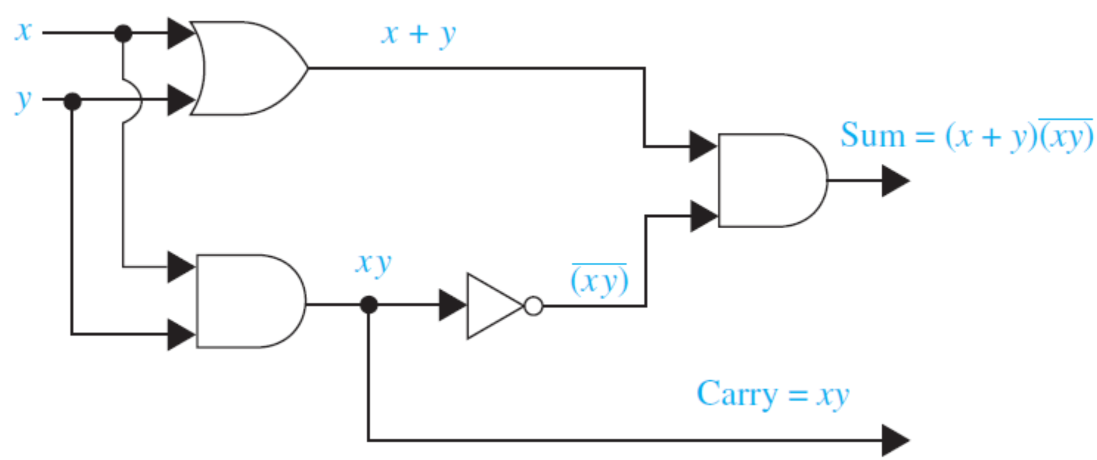


AND OR NOT









The "Useless" Machine

- A useless machine: a machine that turns itself off
  - The attributed inventor is Marvin Minsky

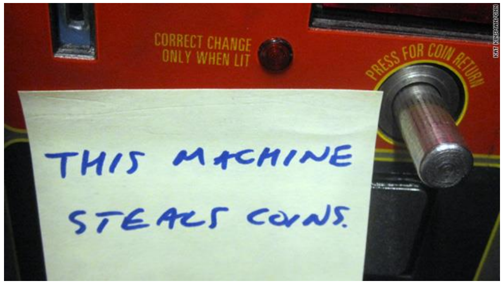
- The useless machine is a **finite state machine** or **automaton**

Vending Machine

- Our simplified vending machine: orange juice at 30 cents each
  - User input: nickels, dimes, quarters, OK button, coin return
  - Output: change return, juice

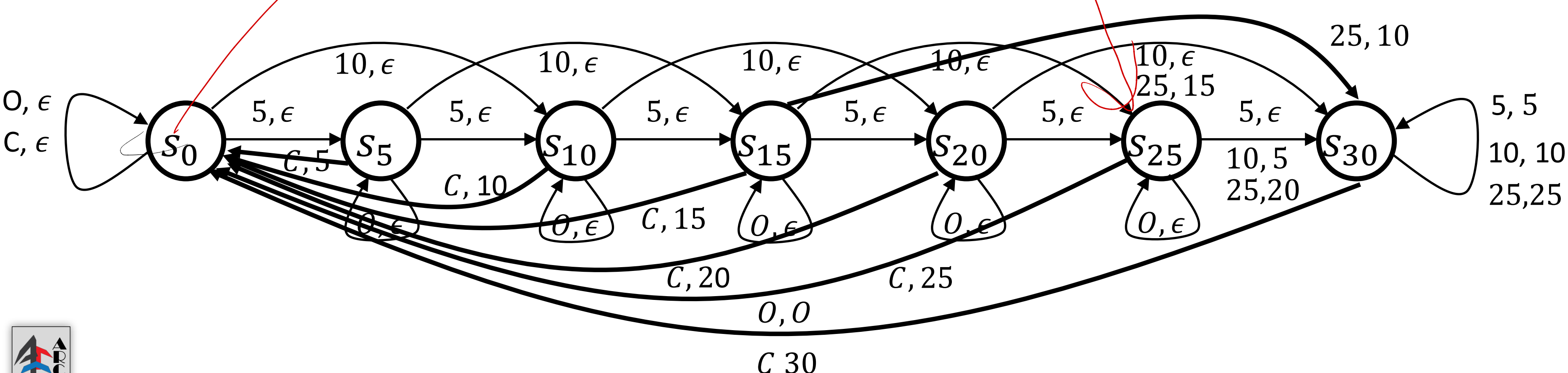
- Must have:
  - State/memory: current state of the machine
  - Correct output
  - Fast response
- These can be realized using **finite state machines**
- Also known as **automaton/automata**

- Basically, they are machines that “**eat**” user i
  the desired output.
  - Well, unless they malfunction



Designing a Vending Machine

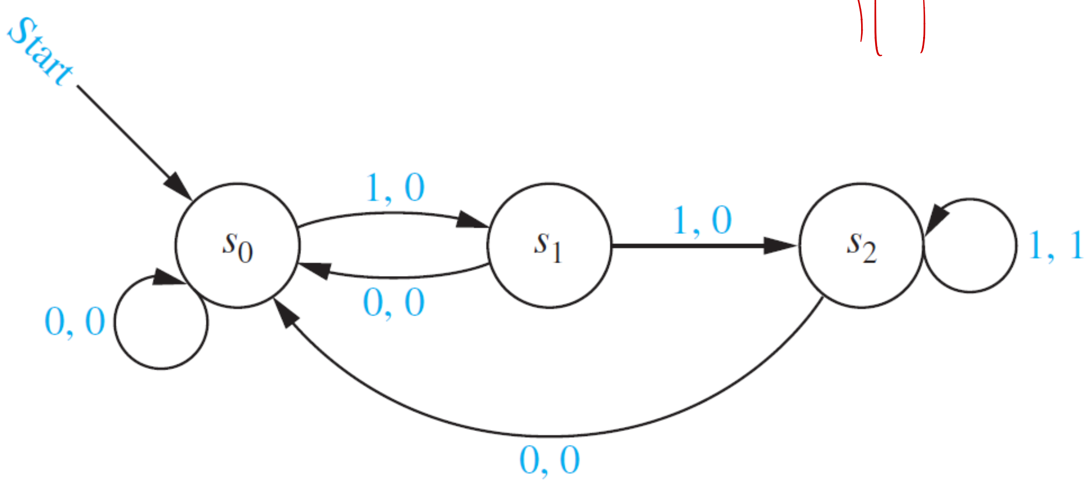
- Input: 5c, 10c, 25c, O, C (coin return)
- Memory must be finite (the smaller the better)
  - Translates to "states"
  - How many states?
  - We need at least memory for 0, 5, 10, 15, 20, 25, 30 cents.
  - And this is enough
- Now, let's construct it!
  - To construct, check what should happen for each input at each state



More Example: “111” Detector

- The following machine detects “111”

111



  - How does it “eat” the string 100111?
  - What about 1101101?
  - What about 11111111?

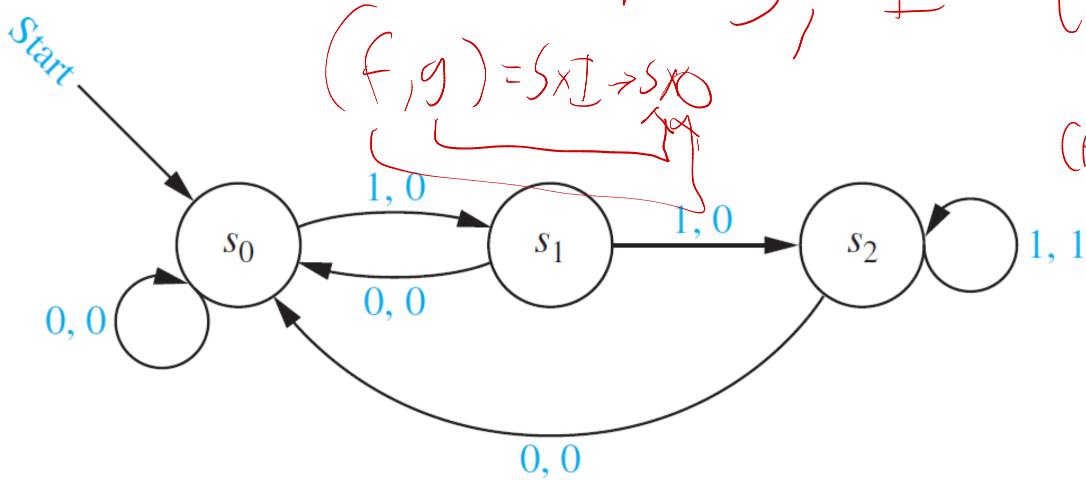
- Such machines can be used for string search in text – regular expressions

Finite State Machine: Formal Definition

- **Def^{n}**: A finite-state machine (FSM) $M = (S, I, O, f, g, s_0)$ consists of a finite set S of states, a finite input alphabet I, a finite output alphabet O, a transition function f that assigns to each state and input pair a new state, an output function g that assigns to each state and input pair an output, and an initial state s_0.

$S = \{s_0, s_1, s_2\}$, $I = \{0, 1\}$, $O = \{0, 1\}$

$(f, g) = S \times I \to S \times O$



$(f, g)(s_0, 1) = (s_1, 0)$
$(f, g)(s_0, 0) = (s_0, 0)$
$(s_1, 1) = (s_2, 0)$
$(s_1, 0) = (s_0, 0)$
$(s_2, 1) = (s_2, 1)$
$(s_2, 0) = (s_0, 0)$

Languages and Grammars

Sentences, Grammars, and Languages

A language (for our study) is **a set of sentences over some vocabulary**. It has **syntactic** and **semantic** rules for validating and interpreting sentences.

- Syntactic rules for deciding what sentences are valid ones
  - E.g., “Hops large quickly rabbit a” is not a syntactically valid sentence in English.

A large rabbit hops quickly.

article adjective noun verb adverb

decorated noun

verb phrase

noun phrase

sentence

- Semantic rules for interpreting the meaning
  - E.g., for “Monkeys are way smarter than humans” probably doesn’t make much sense
- This lecture discusses a bit about grammars of
  - Regular language
  - Context-free language (most computer languages)

Grammar of Propositional Logic as a Language

"can be" "or"

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{True} \mid \textit{False} \mid p \mid q \mid r \mid \ldots \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
 & \mid & \neg\ \textit{Sentence} \\
 & \mid & \textit{Sentence} \land \textit{Sentence} \\
 & \mid & \textit{Sentence} \lor \textit{Sentence} \\
 & \mid & \textit{Sentence} \ \Rightarrow\ \textit{Sentence} \\
 & \mid & \textit{Sentence} \ \Leftrightarrow\ \textit{Sentence}
\end{array}
$$

"propositions"

With grammars, we can "generate" sentences of languages, e.g.,

$$
\begin{aligned}
Sentence &\rightarrow ComplexSentence \rightarrow Sentence \leftrightarrow Sentence \\
&\rightarrow ComplexSentence \leftrightarrow ComplexSentence \rightarrow (Sentence) \leftrightarrow (Sentence) \\
&\rightarrow (ComplexSentence) \leftrightarrow (ComplexSentence) \\
&\rightarrow (Sentence \rightarrow Sentence) \leftrightarrow (Sentence \rightarrow Sentence) \\
&\rightarrow (AtomicSentence \rightarrow AtomicSentence) \leftrightarrow (AtomicSentence \rightarrow AtomicSentence) \\
&\rightarrow (p \rightarrow q) \leftrightarrow (q \rightarrow p)
\end{aligned}
$$

Note that the sentence can be wrong (not meaningful)!

Regular Language

Regular language (as a set of strings) over an alphabet Σ is defined (recursively) as

- Empty language {} and the empty string language $\{\varepsilon\}$ are regular languages.
- For each symbol $a \in \Sigma$, $\{a\}$ is a regular language.
- Given regular languages A and B, $A \cdot B$ (concatenation), $A \cup B$ (union), and A^* (Kleene star, or self concatenation) are regular languages.

A symbol in the alphabet Σ corresponds to a word in English

A string corresponds to a sentence in English

Concatenation example: $A = \{\varepsilon, a, b\}$, $B = \{b, ab, ca\}$

$$A \cdot B = \{b, ab, ca, ab, aab, aca, bb, bab, bca\}$$

Note that ab has duplicates and we only need to keep one.

Kleene star: $A^* = \{\varepsilon\} \cup \{w_1 w_2 \dots w_k \mid w_i \in A, k < \infty\}$

Examples of regular languages

- $\Sigma = \{a, b\}, L = \{ab, abab, ababab, \dots\} = \{ab(ab)^*\} = \{ab\} \cdot \{ab\}^*$
- $\Sigma = \{0, 1\}, L = \{0,1\}^* \cdot \{111\} \cdot \{0,1\}^*$ (binary strings containing “111”)

Yes, regular language is the language described by regular expressions

Deterministic Finite Automaton (DFA)

Recall the definition of finite state machines

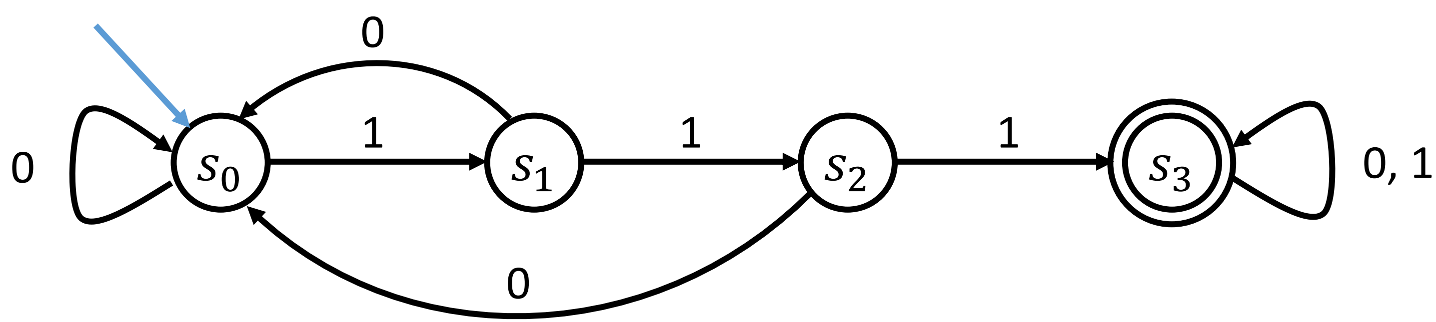
> A *finite-state machine* $M = (S, I, O, f, g, s_0)$ consists of a finite set S of *states*, a finite *input alphabet* I, a finite *output alphabet* O, a *transition function* f that assigns to each state and input pair a new state, an *output function* g that assigns to each state and input pair an output, and an *initial state* s_0.

Let's change the machine a bit

- Remove the output
- Add a set of "final states"
- We get a **deterministic finite automaton** (DFA)
- Written as $\boldsymbol{M = (\Sigma, S, \delta, s_0, F)}$
  - Σ: the alphabet
  - S: the set of states
  - δ: transition function
  - s_0: start state
  - F: final "accepting" states (may include s_0)
- If the machine "eats" a string and ends up at some $s \in F$, then the string is **accepted**. Otherwise, the string is **rejected** by the machine.
- All string accepted by the machine is the **language** of M or $L(M)$.

DFA and Regular Language

What does this DFA do?



Is 01110011010 accepted?
What about 110010?
The DFA recognizes binary strings containing “111”
The set of all strings accepted by the DFA forms the language of the DFA
Which is just $L = \{0,1\}^* \cdot \{111\} \cdot \{0,1\}^*$, or strings containing “111”

Languages for DFAs are exactly regular languages!

Regular Grammar

We can **generate** regular languages using **left-regular grammars**. For example, for strings containing “111”

$S \to A \mid 0A \mid 1A$
$A \to 1B \mid 0A \mid 1A$
$B \to 1C$
$C \to 1D$
$D \to \varepsilon \mid 0D \mid 1D$

S is the start symbol or the sentence symbol
Capital letters are “non-terminals” and lower cases ones are “terminals”
ε is the empty string
A sentence is generated through a **derivation** from S

E.g., $S \to A \to 1A \to 10A \to 101B \to 1011C \to 10111D \to 101110D \to 101110$

In a left-regular grammar

- Only non-terminals appear on the left of $\to$
- On the right side of $\to$, we must have a terminal, or a terminal followed by a non-terminal
- There are also right-regular grammars, which also generate regular languages

Limitation of Regular Languages

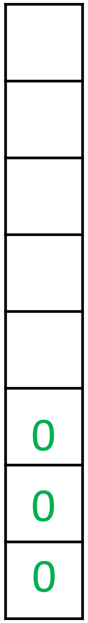
Note that DFAs has only limited memory – a state is essentially a memory cell. Therefore, it cannot recognize certain very simple languages,

- E.g., to recognize $\{0^n 1^n\}$, a machine must remember how many 0s it has seen to match the 1's
- But this is important to have, for example, to parse things like

$$\Bigg(\bigg(\Big(\big(\quad\big)\big(\quad\big)\big(\quad\big)\Big)\big(\quad\big)\bigg)\big(\quad\big)\Bigg)$$

Because DFAs recognizes exactly regular languages, languages such as $\{0^n 1^n\}$ do not belong to regular languages.
So DFAs are not **powerful** enough as a machine and regular languages are not **expressive** enough as a language type
We need more powerful machines and more expressive languages!

Pushdown Automaton (PDA)



Let's revisit the language $\{0^n 1^n\}$: can we build a machine that recognize it?

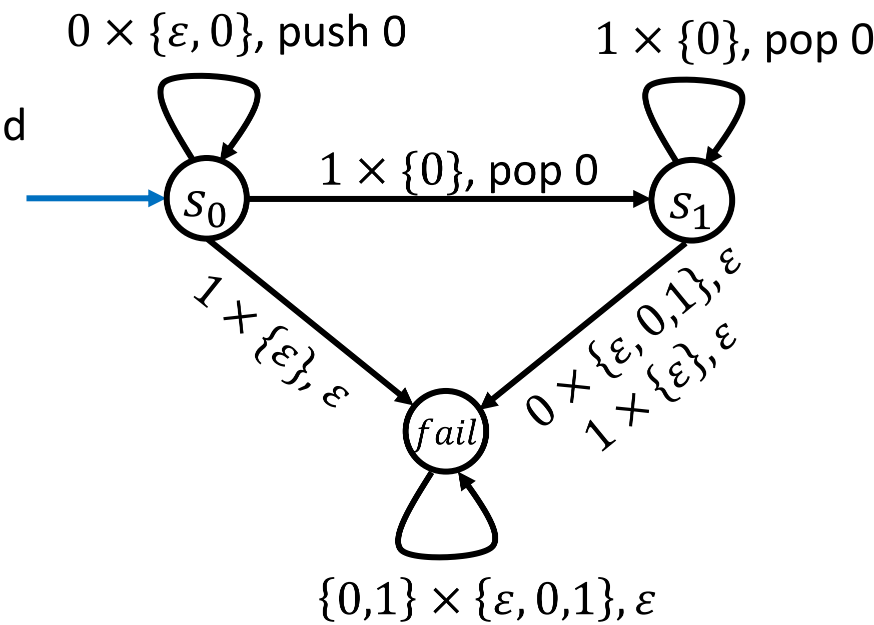
Yes, using pushdown automata, or DFA + infinite stack

At each step, look at the input symbol **and** the top of the stack for actions to take

- E.g., string 000111
- At the beginning, stack is empty, treated as ε
- $0 \times \varepsilon$: stay at s_0, push 0 onto stack
- 0×0: stay at s_0, push 0 onto stack
- 0×0: stay at s_0, push 0 onto stack
- 1×0: move to s_1, pop 0 from stack
- 1×0: stay at s_1, pop 0 from stack
- 1×0: stay at s_1, pop 0 from stack
- Accept if stack is empty when string is consumed
- Can also accept using final states
- So 000111 is accepted

Another example: 011001

- Not accepted



PDA and Context-Free Languages

With PDA, we can handle the parsing of languages with parenthesis!
In fact, PDA is powerful enough to handle complex programming languages

- For example, parsing propositional logic or C++

These languages are called **context-free languages**, generated with context free grammars
For example, for $\{0^n 1^n\}$, the grammar can be written as

$$S \to 0S1 \mid 01$$

Context-free languages are already somewhat complex, for example

- Given context-fee grammars A, B, computers cannot decide whether $L(A) = L(B)$
- Similarly, computers cannot decide whether $L(A) \cap L(B) = \emptyset$

But, we can parse context-free languages quickly, allowing the constructions of compilers.