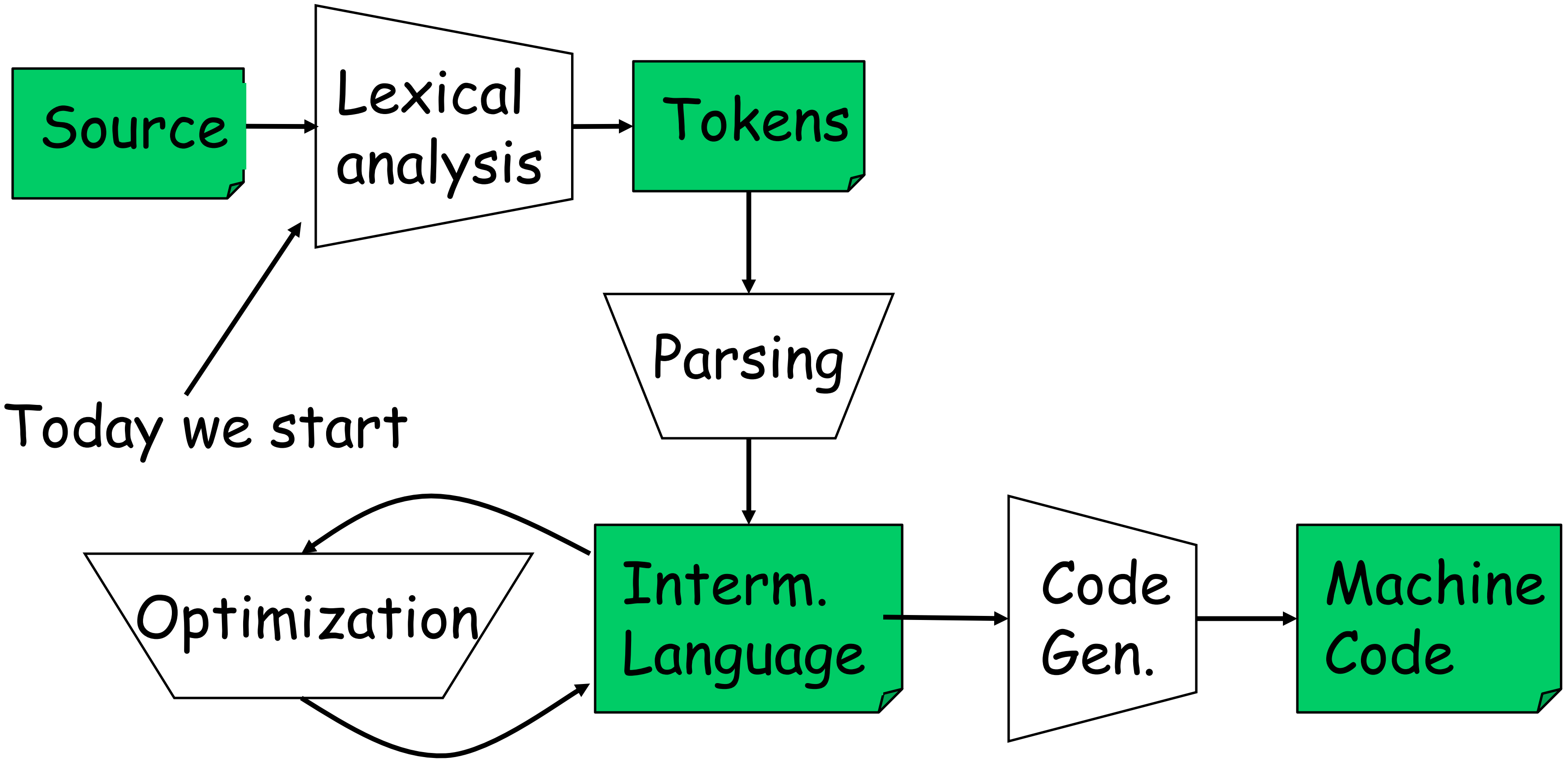
Lexical Analysis

Lecture 2-3

Recall: The Structure of a Compiler



Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    z = 0;
else
    z = 1;
```

- The input is just a sequence of characters:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Goal: Partition input string into substrings
  - And classify them according to their role

What's a Token?

- Output of lexical analysis is a stream of tokens
- A token is a syntactic category
  - In English:

    noun, verb, adjective, …
  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …
- Parser relies on the token distinctions:
  - E.g., identifiers are treated differently than keywords

Tokens

- Tokens correspond to sets of strings.

- Identifier: *strings of letters or digits, starting with a letter*
- Integer: *a non-empty string of digits*
- Keyword: *"else" or "if" or "begin" or …*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*
- OpenPar: *a left-parenthesis*

Lexical Analyzer: Implementation

- An implementation must do two things:

1. Recognize substrings corresponding to tokens

2. Return the value or <u>lexeme</u> of the token
   - The lexeme is the substring

Example

- Recall:

  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- Token-lexeme pairs returned by the lexer:
  - (Whitespace, “\t”)
  - (Keyword, “if”)
  - (OpenPar, “(“)
  - (Identifier, “i”)
  - (Relation, “==“)
  - (Identifier, “j”)
  - …

Lexical Analyzer: Implementation

- The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.
- Examples: Whitespace, Comments
- Question: What happens if we remove all whitespace and all comments prior to lexing?

Lookahead.

- Two important points:

1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time

2. "Lookahead" may be required to decide where one token ends and the next token begins

- Even our simple example has lookahead issues

    i vs. if

    = vs. ==

Next

- We need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if a keyword or an identifier
    - Is if two variables i and f?
    - Is == two equal signs = =?

Regular Languages

- There are several formalisms for specifying tokens

- *Regular languages* are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

Def. Let Σ be a set of characters. A <u>language over</u> Σ is a set of strings of characters drawn from Σ

(Σ is called the <u>alphabet</u>)

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string on English characters is an English sentence

- Alphabet = ASCII
- Language = C programs

- Note: ASCII character set is different from English character set

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- For lexical analysis we care about *regular languages*, which can be described using *regular expressions.*

Regular Expressions and Regular Languages

- Each regular expression is a notation for a regular language (a set of words)

- If *A* is a regular expression then we write L(*A*) to refer to the language denoted by *A*

Atomic Regular Expressions

- Epsilon: ε

  $L(\varepsilon) = \{ \text{“”} \}$

- Single character: ‘c’

  $L(\text{‘c’}) = \{ \text{“c”} \}$ (for any $c \in \Sigma$)

Compound Regular Expressions

- Concatenation: AB (where A and B are reg. exp.)

  L(AB) = { ab | a ∈ L(A) and b ∈ L(B) }

- Example: L('i' 'f') = { "if" }

  (we will abbreviate 'i' 'f' as 'if')

Compound Regular Expressions

- Union

  L(A | B) = { s | s ∈ L(A) or s ∈ L(B) }

- Examples:

  ‘if’ | ‘then‘ | ‘else’ = { “if”, “then”, “else”}

  ‘0’ | ‘1’ | … | ‘9’ = { “0”, “1”, …, “9” }

  (note the … are just an abbreviation)

- Another example:

  (‘0’ | ‘1’) (‘0’ | ‘1’) = { “00”, “01”, “10”, “11” }

More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: A^*

  $L(A^*) = \{$ “” $\} \cup L(A) \cup L(AA) \cup L(AAA) \cup \ldots$
- Examples:

  ‘0’ * = { “”, “0”, “00”, “000”, …}

  ‘1’ ‘0’ * = { strings starting with 1 and followed by 0’s }

Example: Keyword

- Keyword: *“else”* *or* *“if”* *or* *“begin”* *or …*

‘else’ | *‘if’* | *‘begin’* | …

(Recall: ‘else’ abbreviates ‘e’ ‘l’ ‘s’ ‘e’)

Example: Integers

Integer: *a non-empty string of digits*

digit =

number =

Abbreviation: $A^{+} = A\ A^{*}$

Example: Integers

Integer: *a non-empty string of digits*

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

number = digit digit*

Abbreviation: $A^+ = A\ A^*$

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

letter =

identifier =

Is (letter* | digit*) the same as (letter | digit) * ?

Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

letter = 'A' | … | 'Z' | 'a' | … | 'z'

identifier = letter (letter | digit) *

Is (letter* | digit*) the same as (letter | digit) * ?

Example: Whitespace

Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

$(\text{' '} \mid \text{'\textbackslash t'} \mid \text{'\textbackslash n'})^{+}$

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 642-2420

Σ = { 0, 1, 2, 3, …, 9, (,), - }

area =

exchange =

phone =

number =

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 642-2420

  Σ = { 0, 1, 2, 3, …, 9, (,), - }

  area = digit^3

  exchange = digit^3

  phone = digit^4

  number = '(' area ')' exchange '-' phone

Example: Email Addresses

- Consider *ksen@cs.berkeley.edu*

Σ = letters [{ ., @ }

name =

address =

Example: Email Addresses

- Consider *ksen@cs.berkeley.edu*

Σ = letters [{ ., @ }

name = letter$^+$

address = name '@' name ('.' name)*

Summary

- Regular expressions describe many useful languages
- Next: *Given a string s and a rexp R,* is

$$s \in L(R)?$$

- But a yes/no answer is not enough !
- Instead: partition the input into lexemes

- We will adapt regular expressions to this goal

Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
    - Number, Keyword, Identifier, ...

2. Write a R.E. for the lexemes of each token
    - Number = digit^+
    - Keyword = ‘if’ | ‘else’ | ...
    - Identifier = letter (letter | digit)*
    - OpenPar = ‘(‘
    - ...

Regular Expressions => Lexical Spec. (2)

3. Construct R, matching all lexemes for all tokens

$$
\begin{aligned}
R &= \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \ldots \\
&= R_1 \mid R_2 \mid R_3 \mid \ldots
\end{aligned}
$$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_i)$ for some "i"
- This "i" determines the token that is reported

Regular Expressions => Lexical Spec. (3)

4. Let the input be $x_1 \ldots x_n$

   ($x_1 \ldots x_n$ are characters in the language alphabet)

   - For $1 \leq i \leq n$ check

     $x_1 \ldots x_i \in L(R)$?

5. It must be that

   $x_1 \ldots x_i \in L(R_j)$ for some i and j

6. Remove $x_1 \ldots x_i$ from input and go to (4)

Lexing Example

R = Whitespace | Integer | Identifier | '+'

- Parse "f+3 +g"
  - "f" matches R, more precisely Identifier
  - "+" matches R, more precisely '+'
  - …
  - The token-lexeme pairs are (Identifier, "f"), ('+' , "+"), (Integer, "3") (Whitespace, " "), ('+' , "+"), (Identifier, "g")
- We would like to drop the Whitespace tokens
  - after matching Whitespace, continue matching

Ambiguities (1)

- There are ambiguities in the algorithm
- Example:

  R = Whitespace | Integer | Identifier | '+'
- Parse "foo+3"
  - "f" matches R, more precisely Identifier
  - But also "fo" matches R, and "foo", but not "foo+"
- How much input is used? What if
    - $x_1 \ldots x_i \in L(R)$ and also $x_1 \ldots x_K \in L(R)$
  - "Maximal munch" rule: Pick the longest possible substring that matches R

More Ambiguities

R = Whitespace | 'new' | Integer | Identifier

- Parse "new foo"
  - "new" matches R, more precisely 'new'
  - but also Identifier, which one do we pick?
- In general, if $x_1 \ldots x_i \in L(R_j)$ and $x_1 \ldots x_i \in L(R_k)$
  - Rule: use rule listed first (j if $j < k$)

- We must list 'new' before Identifier

Error Handling

R = Whitespace | Integer | Identifier | '+'

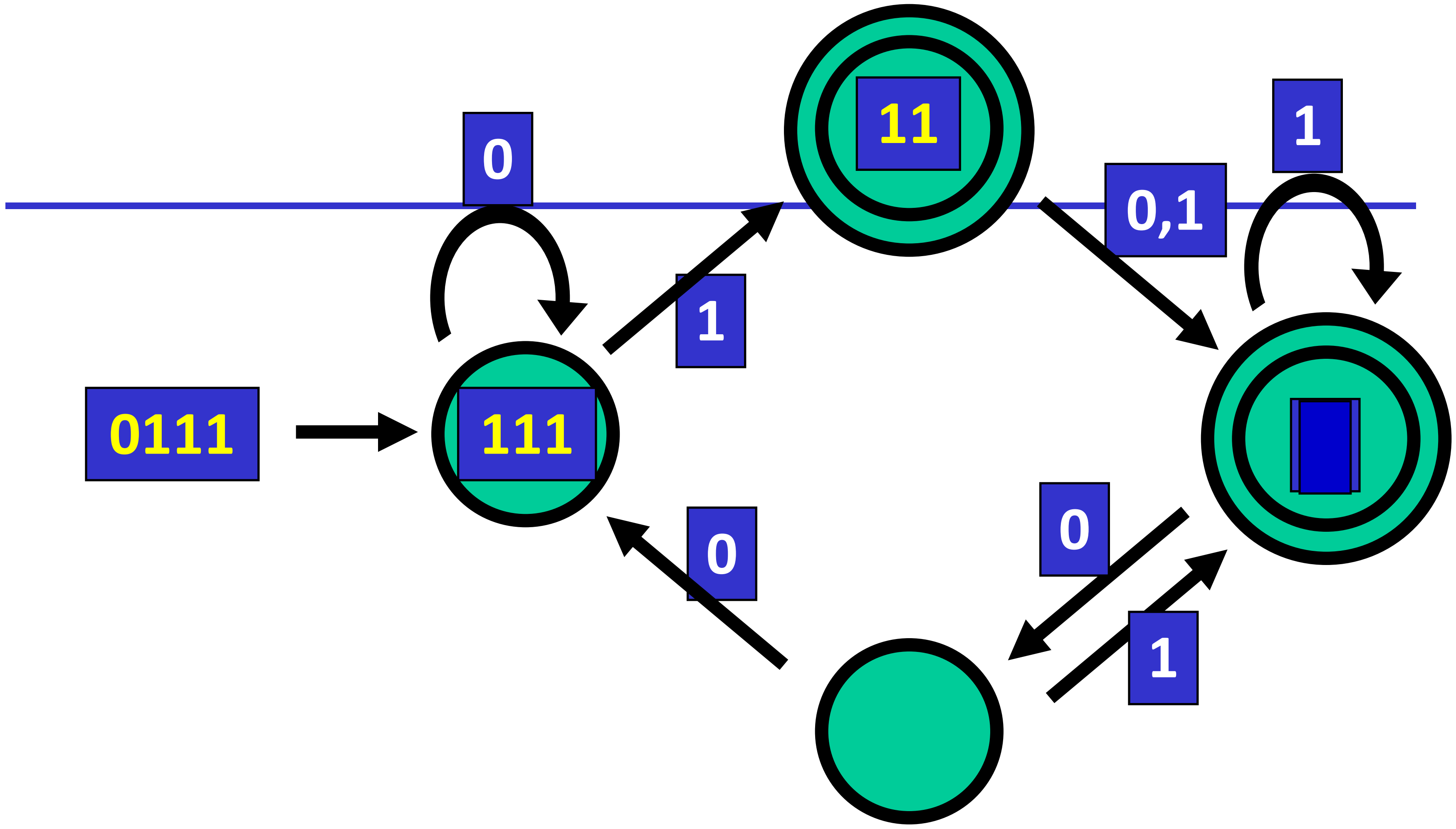
- Parse "=56"
  - No prefix matches R: not "=", nor "=5", nor "=56"
- Problem: Can't just get stuck …
- Solution:
  - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:

  $R = R_1 \mid \ldots \mid R_n \mid \text{Error}$

  - Token Error matches if nothing else matches
  - Pick the **shortest** non-empty string that matches Error.
  - Merge multiple consecutive Error tokens to a single Error Token

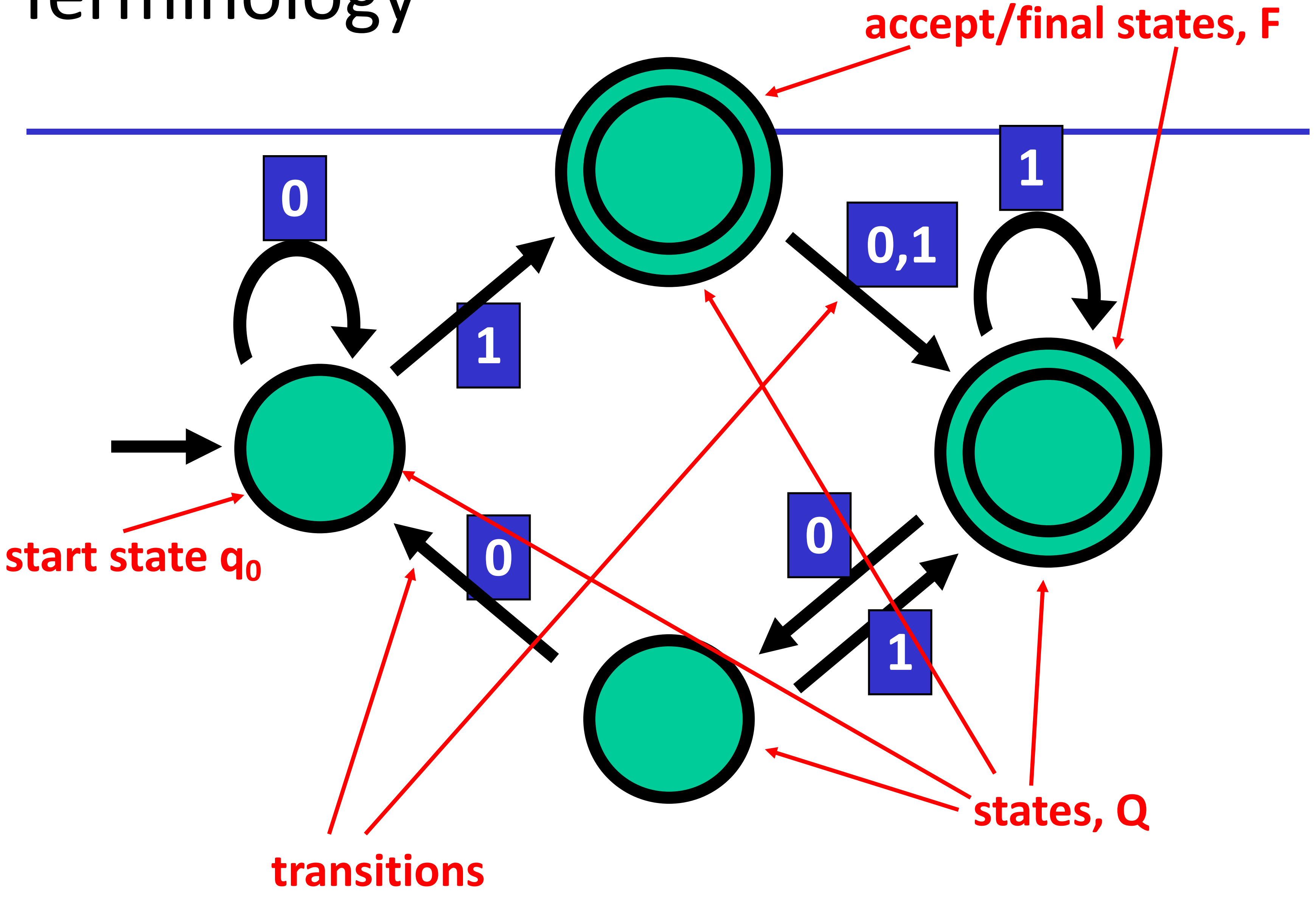
Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)



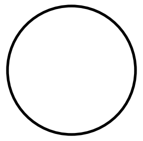
The machine **accepts** a string if the process ends in a double circle

Terminology

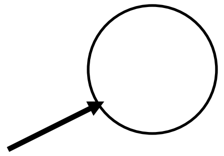


Finite Automata State Graphs

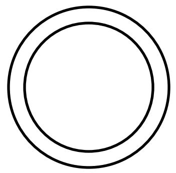
- A state



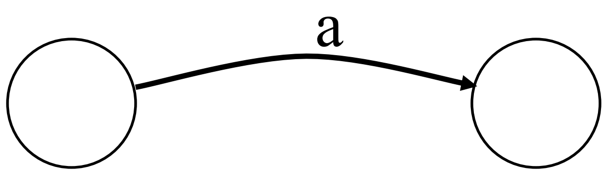
- The start state



- An accepting state

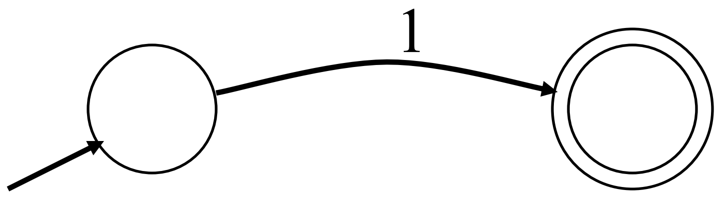


- A transition



A Simple Example

- A deterministic finite automaton that accepts only "1"



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

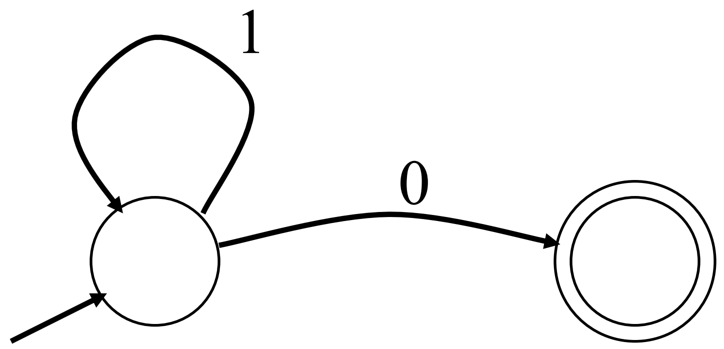
Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

- Check that "1110" is accepted but "110…" is not

Another Simple Example

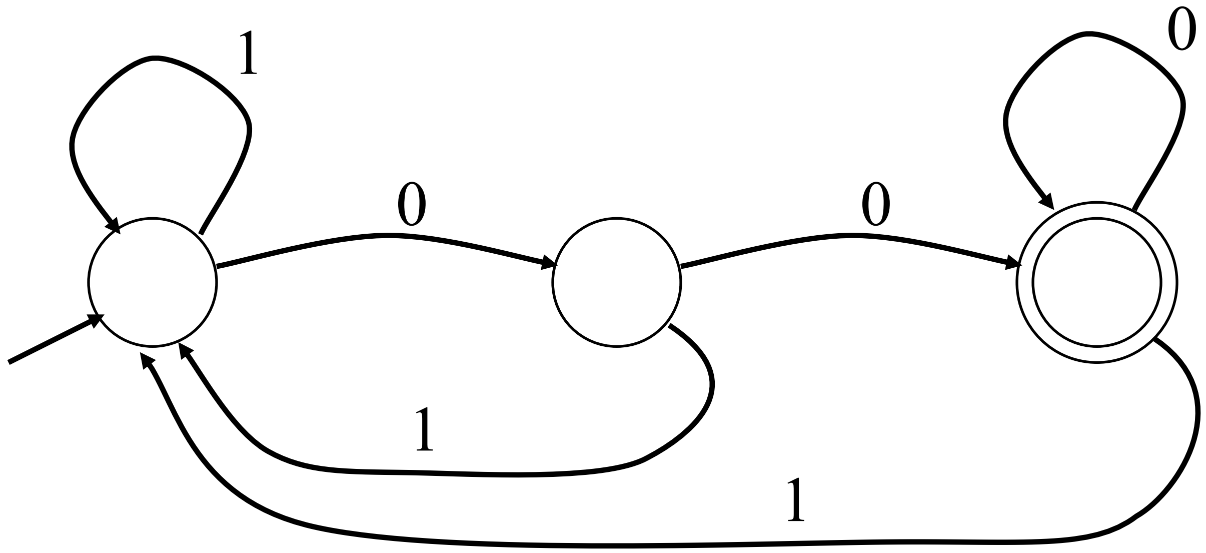
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

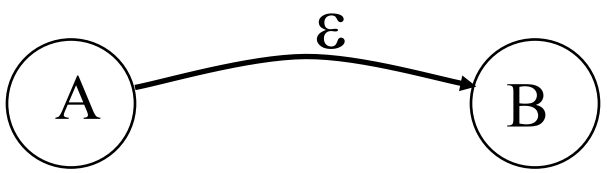
And Another Example

- Alphabet {0,1}
- What language does this recognize?



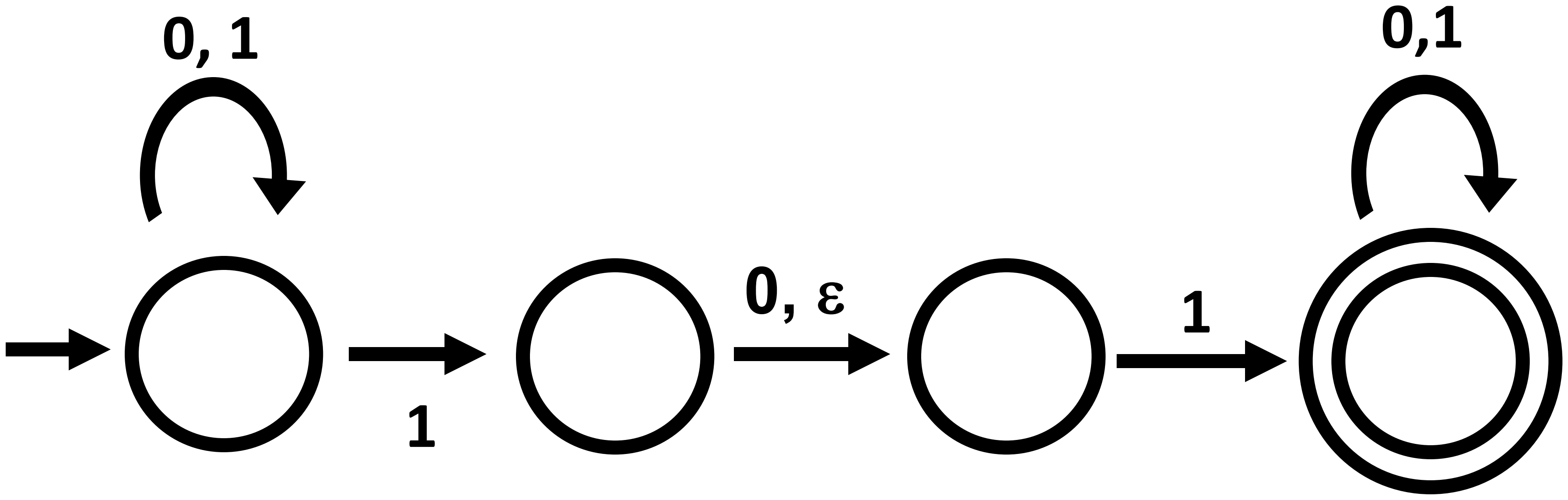
Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Non Deterministic Finite Automaton (NFA)



Deterministic and Nondeterministic Automata

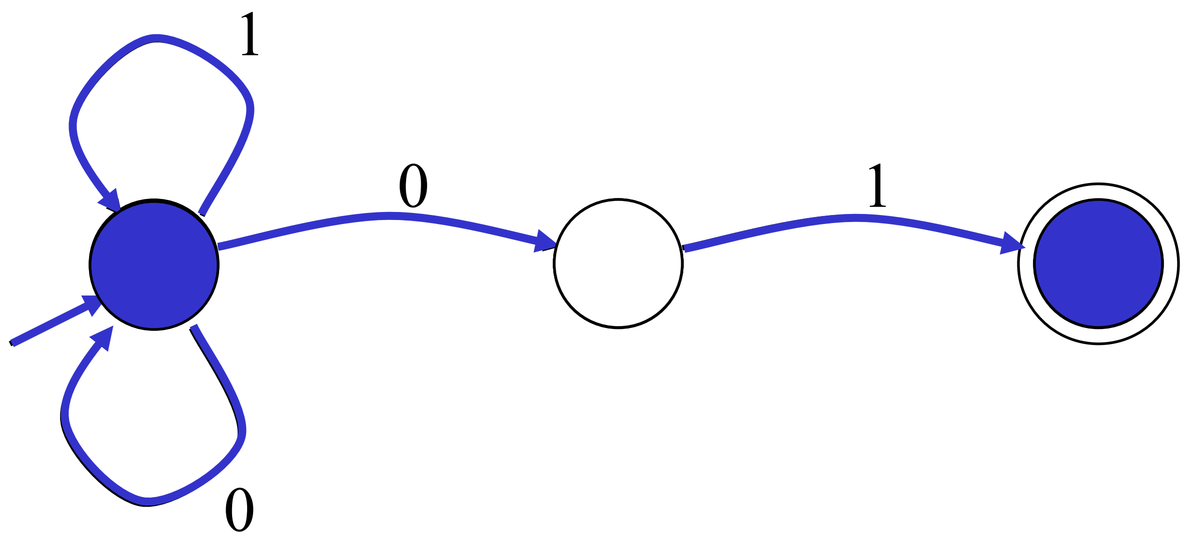
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- Finite automata have finite memory
  - Need only to encode the current state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

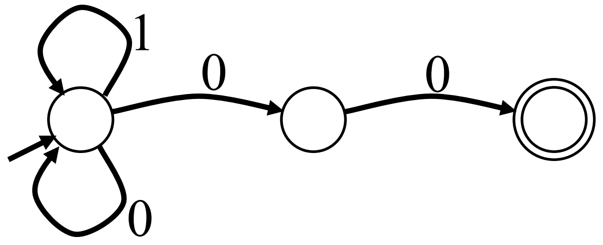
- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider

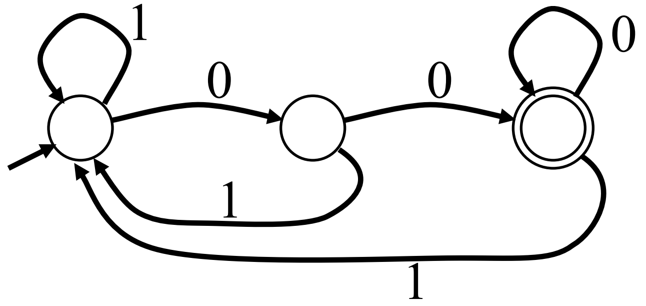
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA

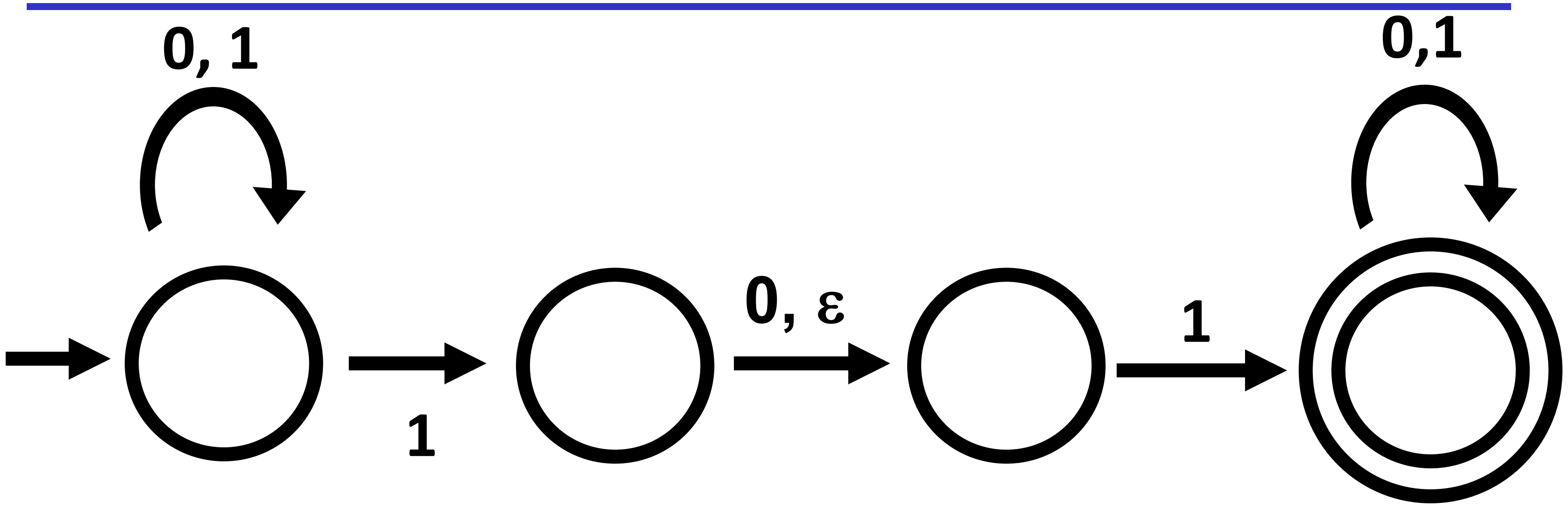


DFA



- DFA can be exponentially larger than NFA

NFA to DFA

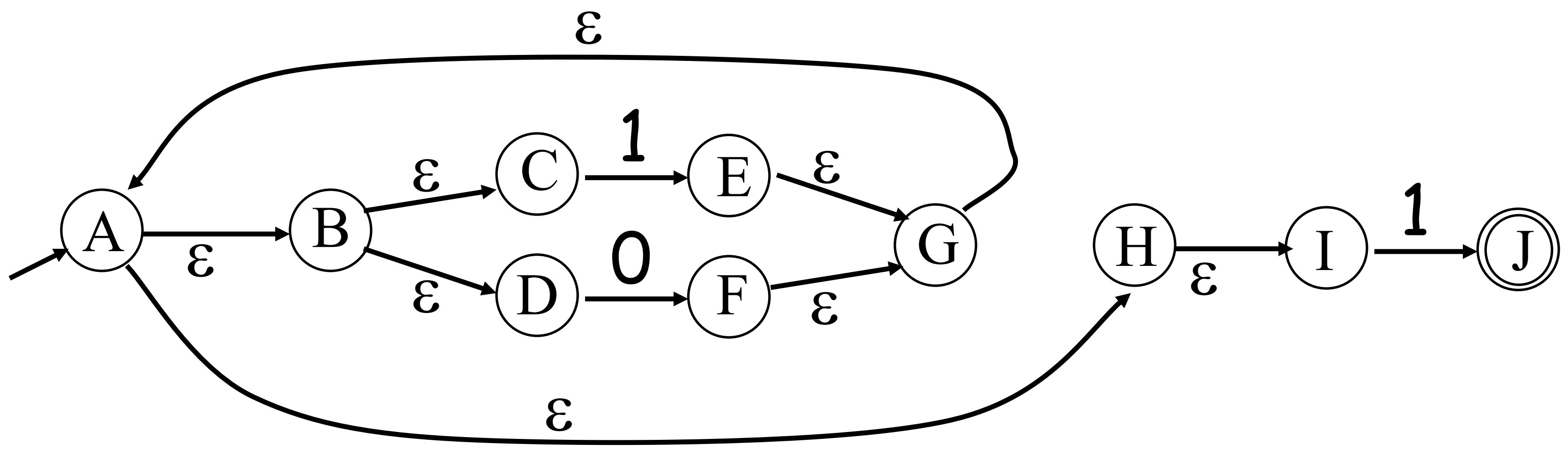


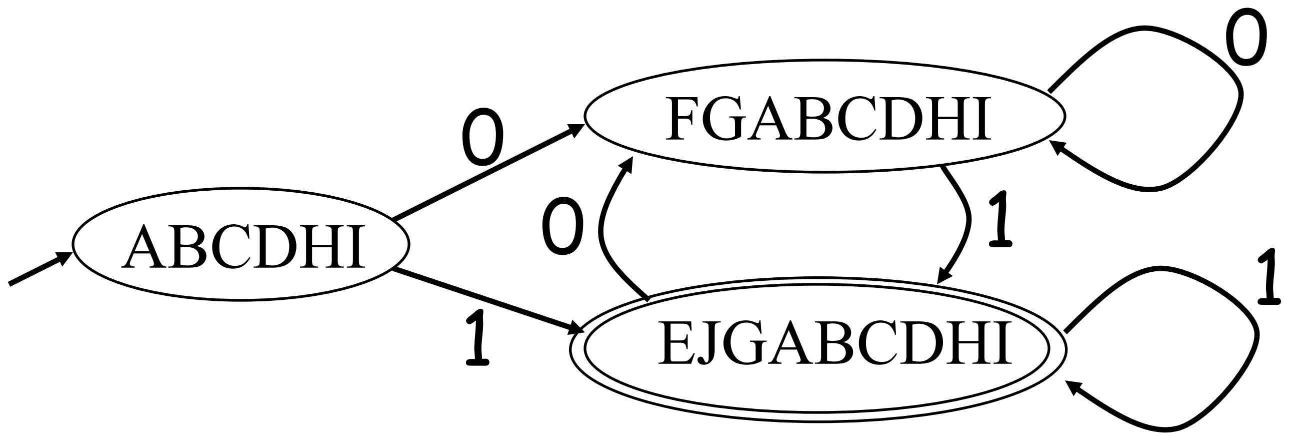
Does it accept 010110?

NFA to DFA. The Trick

- Simulate the NFA
- Each state of resulting DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example





Size of DFA

NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

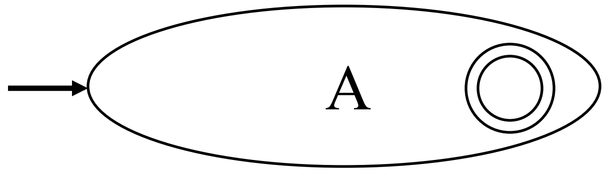
Regular Expressions to Finite Automata

- High-level sketch

$$\begin{tikzcd}
& \text{NFA} \arrow[dr] & \\
\text{Regular expressions} \arrow[ur] & & \text{DFA} \arrow[d] \\
\text{Lexical Specification} \arrow[u] & & \text{Table-driven Implementation of DFA}
\end{tikzcd}$$

Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A

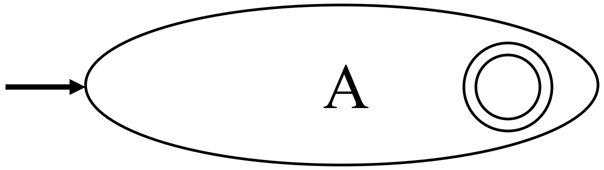


- For ε

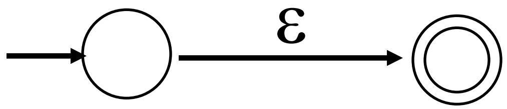
- For input a

Regular Expressions to NFA (1)

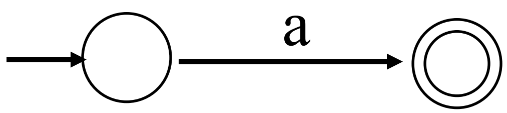
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp *A*



- For ε



- For input a

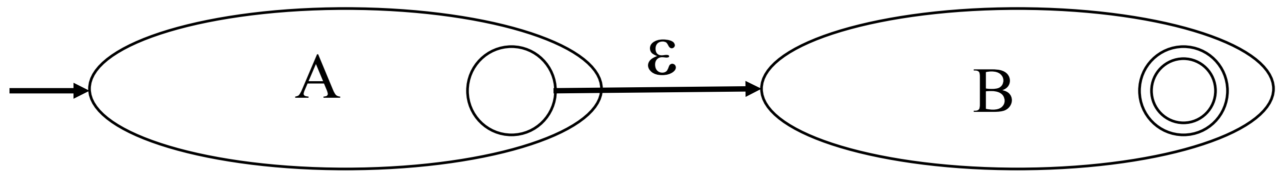


Regular Expressions to NFA (2)

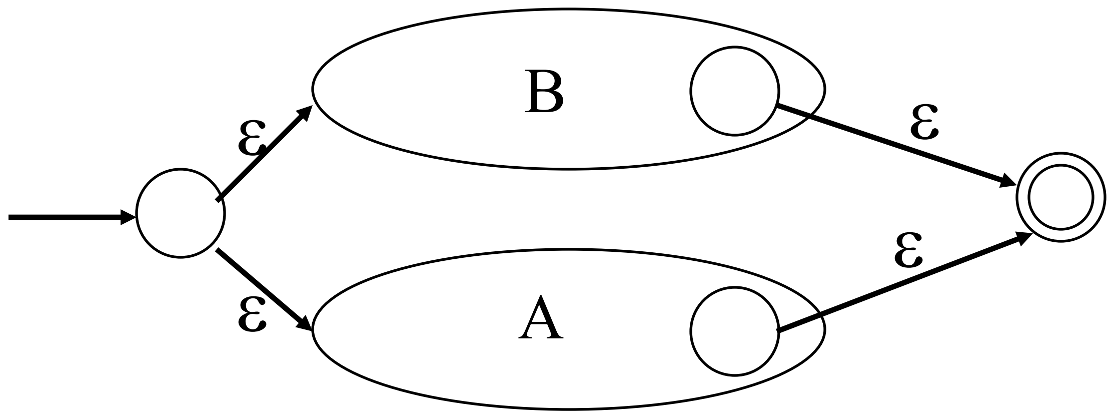
- For AB
- For A | B

Regular Expressions to NFA (2)

- For *AB*



- For *A* | B

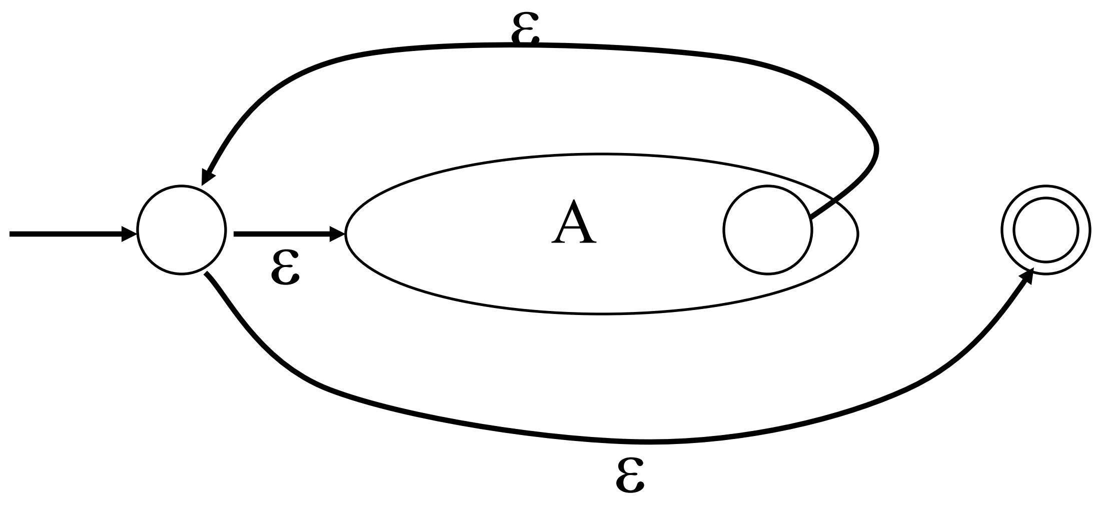


Regular Expressions to NFA (3)

- For *A**

Regular Expressions to NFA (3)

- For *A**



Example of RegExp -> NFA conversion

- Consider the regular expression

  (1 | 0)*1

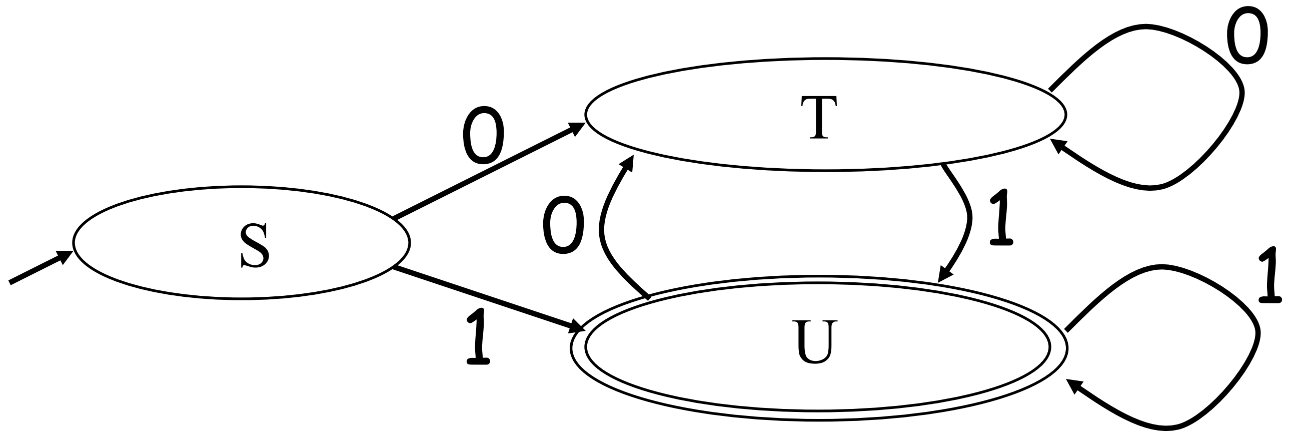
- The NFA is

$$\begin{tikzcd}
 & & & C \arrow[r, "1"] & E \arrow[dr, "\varepsilon"] & & & & \\
{} \arrow[r] & A \arrow[r, "\varepsilon"'] \arrow[rrrrr, bend right=40, "\varepsilon"'] & B \arrow[ur, "\varepsilon"] \arrow[dr, "\varepsilon"'] & & & G \arrow[llll, bend right=60, "\varepsilon"'] & H \arrow[r, "\varepsilon"'] & I \arrow[r, "1"] & J \\
 & & & D \arrow[r, "0"] & F \arrow[ur, "\varepsilon"'] & & & &
\end{tikzcd}$$

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA


$$\begin{tikzcd}
\arrow[r] & S \arrow[r, "0"] \arrow[dr, "1"'] & T \arrow[d, bend left, "1"] \arrow[loop right, "0"] \\
& & U \arrow[u, bend left, "0"] \arrow[loop right, "1"]
\end{tikzcd}$$

	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jlex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

PA1: Lexical and Syntactic Analysis of ChocoPy

- Correctness is job #1.
  - And job #2 and #3!
- Use piazza to find project partner
  - Search “Search for Teammates!” on piazza
- Tips on building large systems:
  - Keep it simple
  - Design systems that can be tested
  - Don’t optimize prematurely
  - It is easier to modify a working system than to get a system working