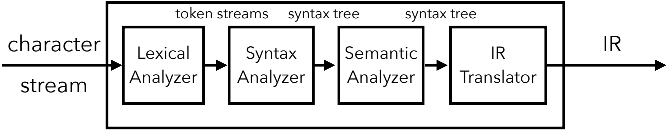
COSE312: Compilers

Lecture 2 — Lexical Analysis

Hakjoo Oh
2025 Spring

Lexical Analysis



ex) Given a C program

```
float match0 (char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.0;
}
```

the lexical analyzer returns the stream of tokens:

FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN LBRACE IF LPAREN BANG ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF

Specification, Recognition, and Automation

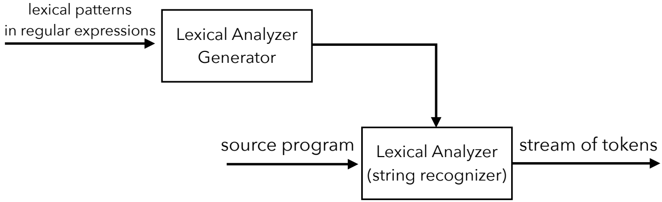
1. **Specification**: how to specify lexical patterns?
   - In C, identifiers are strings like `x`, `xy`, `match0`, and `_abc`.
   - Numbers are strings like `3`, `12`, `0.012`, and `3.5E4`.

   $\Rightarrow$ *regular expressions*
2. **Recognition**: how to *recognize* the lexical patterns?
   - Recognize `match0` as an identifier.
   - Recognize `512` as a number.

   $\Rightarrow$ *deterministic finite automata*.
3. **Automation**: how to automatically generate string recognizers from specifications?

   $\Rightarrow$ *Thompson's construction* and *subset construction*

cf) Lexical Analyzer Generator



- `lex`: a lexical analyzer generator for C
- `jlex`: a lexical analyzer generator for Java
- `ocamllex`: a lexical analyzer generator for OCaml

Part 1: Specification

- Preliminaries: alphabets, strings, languages
- Syntax and semantics of regular expressions
- Extensions of regular expressions

Alphabet

An alphabet $\mathbf{\Sigma}$ is a finite, non-empty set of symbols. E.g,

- $\boldsymbol{\Sigma = \{0, 1\}}$
- $\boldsymbol{\Sigma = \{a, b, \ldots, z\}}$

Strings

A string is a finite sequence of symbols chosen from an alphabet, e.g., 1, 01, 10110 are strings over $\Sigma = \{0, 1\}$. Notations:

- ϵ: the empty string.
- wv: the concatenation of w and v.
- w^R: the reverse of w.
- $|w|$: the length of string w:

$$\begin{aligned} |\epsilon| &= 0 \\ |va| &= |v| + 1 \end{aligned}$$

- If $w = vu$, then v is a *prefix* of w, and u is a *suffix* of w.
- Σ^k: the set of strings over Σ of length k
- Σ^*: the set of all strings over alphabet Σ:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{i \in \mathbb{N}} \Sigma^i$$

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots = \Sigma^* \setminus \{\epsilon\}$

Languages

A language L is a subset of Σ^*: $L \subseteq \Sigma^*$.

- $L_1 \cup L_2, \quad L_1 \cap L_2, \quad L_1 - L_2$
- $L^R = \{w^R \mid w \in L\}$
- $\overline{L} = \Sigma^* - L$
- $L_1 L_2 = \{xy \mid x \in L_1 \land y \in L_2\}$
- The *power* of a language, L^n:

$$\begin{array}{rcl} L^0 & = & \{\epsilon\} \\ L^n & = & L^{n-1} L \end{array}$$

- The *star-closure* (or *Kleene closure*) of a language, L^*:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \bigcup_{i \geq 0} L^i$$

- The *positive closure* of a language, L^+:

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \cdots = \bigcup_{i \geq 1} L^i$$

Regular Expressions

A regular expression is a notation to denote a language.

- Syntax

$$\begin{array}{rcl} R & \to & \emptyset \\ & | & \epsilon \\ & | & a \in \Sigma \\ & | & R_1 \mid R_2 \\ & | & R_1 \cdot R_2 \\ & | & R_1^* \\ & | & (R) \end{array}$$

- Semantics

$$\begin{array}{rcl} L(\emptyset) & = & \emptyset \\ L(\epsilon) & = & \{\epsilon\} \\ L(a) & = & \{a\} \\ L(R_1 \mid R_2) & = & L(R_1) \cup L(R_2) \\ L(R_1 \cdot R_2) & = & L(R_1)L(R_2) \\ L(R^*) & = & (L(R))^* \\ L((R)) & = & L(R) \end{array}$$

Example

$$
\begin{aligned}
L(a^* \cdot (a \mid b)) &= L(a^*)L(a \mid b) \\
&= (L(a))^*(L(a) \cup L(b)) \\
&= (\{a\})^*(\{a\} \cup \{b\}) \\
&= \{\epsilon, a, aa, aaa, \ldots\}(\{a, b\}) \\
&= \{a, aa, aaa, \ldots, b, ab, aab, \ldots\}
\end{aligned}
$$

Exercises

Write regular expressions for the following languages:

- The set of all strings over $\Sigma = \{a, b\}$.
- The set of strings of a's and b's, terminated by ab.
- The set of strings with an even number of a's followed by an odd number of b's.
- The set of C identifiers.

Regular Definitions

Give names to regular expressions and use the names in subsequent expressions, e.g., the set of C identifiers:

$$\begin{array}{rcl} \textit{letter} & \rightarrow & \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ \textit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ \textit{id} & \rightarrow & \textit{letter}(\textit{letter} \mid \textit{digit})^* \end{array}$$

Formally, a *regular definition* is a sequence of definitions of the form:

$$\begin{array}{rcl} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \cdots & \\ d_n & \rightarrow & r_n \end{array}$$

1. Each d_i is a new name such that $d_i \not\in \Sigma$.
2. Each r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$.

Example

Unsigned numbers (integers or floating point), e.g., `5280`, `0.01234`, `6.336E4`, or `1.89E-4`:

$$\begin{array}{rcl}
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
digits & \rightarrow & digit\ digit^{*} \\
optionalFraction & \rightarrow & \textbf{.}\ digits \mid \epsilon \\
optionalExponent & \rightarrow & (\texttt{E}\ (\texttt{+} \mid \texttt{-} \mid \epsilon)\ digits) \mid \epsilon \\
number & \rightarrow & digits\ optionalFraction\ optionalExponent
\end{array}$$

Extensions of Regular Expressions

1. R^+: the positive closure of R, i.e., $L(R^+) = L(R)^+$.
2. $R?$: zero or one instance of R, i.e., $L(R?) = L(R) \cup \{\epsilon\}$.
3. $[a_1 a_2 \cdots a_n]$: the shorthand for $a_1 \mid a_2 \mid \cdots \mid a_n$.
4. $[a_1\text{-}a_n]$: the shorthand for $[a_1 a_2 \cdots a_n]$, where $a_1, \ldots, a_n$ are consecutive symbols.
   - $[abc] = a \mid b \mid c$
   - $[a\text{-}z] = a \mid b \mid \cdots \mid z$.

Examples

- C identifiers:

$$\begin{array}{rcl} letter & \rightarrow & [\texttt{A-Za-z\_}] \\ digit & \rightarrow & [\texttt{0-9}] \\ id & \rightarrow & letter\ (letter|digit)^* \end{array}$$

- Unsigned numbers:

$$\begin{array}{rcl} digit & \rightarrow & [\texttt{0-9}] \\ digits & \rightarrow & digit^+ \\ number & \rightarrow & digits\ (.\ digits)?\ (\texttt{E}\ [\texttt{+-}]?\ digits)? \end{array}$$

Summary

1. **Specification**: how to specify lexical patterns?
   - In C, identifiers are strings like `x`, `xy`, `match0`, and `_abc`.
   - Numbers are strings like `3`, `12`, `0.012`, and `3.5E4`.

   $\Rightarrow$ *regular expressions*
2. **Recognition**: how to *recognize* the lexical patterns?
   - Recognize `match0` as an identifier.
   - Recognize `512` as a number.

   $\Rightarrow$ *deterministic finite automata*.
3. **Automation**: how to automatically generate string recognizers from specifications?

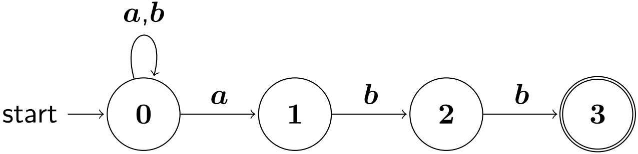
   $\Rightarrow$ *Thompson's construction* and *subset construction*

Part 2: String Recognition by Finite Automata

- Non-deterministic finite automata
- Deterministic finite automata

String Recognizer in NFA

An NFA that recognizes strings $(a|b)^*abb$:



Non-deterministic Finite Automata

Definition (NFA)

A *nondeterministic finite automaton* (or *NFA*) is defined as,

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q: a finite set of *states*
- Σ: a finite set of *input symbols* (or input alphabet). We assume that $\epsilon \notin \Sigma$.
- $q_0 \in Q$: the *initial state*
- $F \subseteq Q$: a set of *final states* (or *accepting states*)
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$: *transition function*

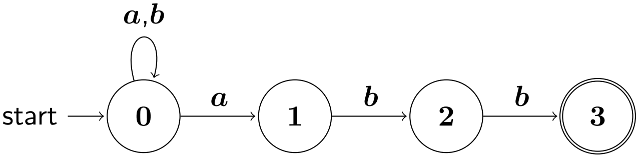
Example

Definition of an NFA:

$$(\{0,1,2,3\},\{a,b\},\delta,0,\{3\})$$

$$\begin{array}{ll}
\delta(0,a)=\{0,1\} & \delta(0,b)=\{0\} \\
\delta(1,a)=\emptyset & \delta(1,b)=\{2\} \\
\delta(2,a)=\emptyset & \delta(2,b)=\{3\} \\
\delta(3,a)=\emptyset & \delta(3,b)=\emptyset
\end{array}$$

The *transition graph*:



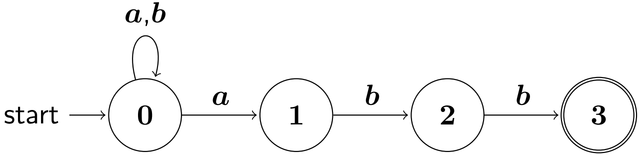
Example

The *transition table*:

State	a	b	ϵ
$\mathbf{0}$	$\{0, 1\}$	$\{0\}$	$\emptyset$
$\mathbf{1}$	$\emptyset$	$\{2\}$	$\emptyset$
$\mathbf{2}$	$\emptyset$	$\{3\}$	$\emptyset$
$\mathbf{3}$	$\emptyset$	$\emptyset$	$\emptyset$

String Recognition

- An NFA recognizes a string w if there is a path in the transition graph labeled by w.



String $aabb$ is accepted because

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

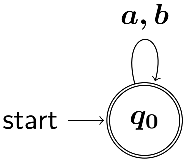
In general, the automaton recognizes any strings that end with abb:

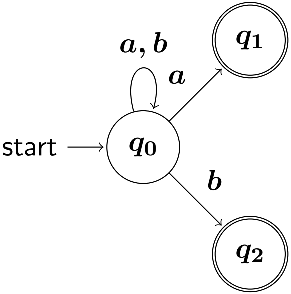
$$L = \{wabb \mid w \in \{a, b\}^*\}$$

- The *language* of an NFA is the set of recognizable strings.

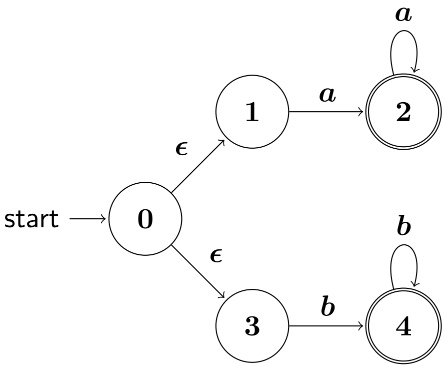
Exercises

Find the languages of the NFAs:

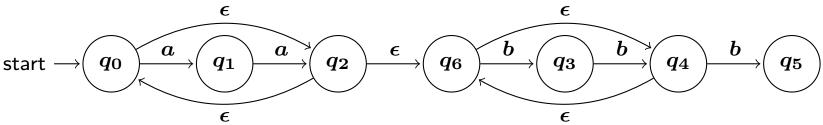
- 

- 

Exercises

- 

start
0
1
2
3
4
ϵ
ϵ
a
a
b
b

- 

start
q_0
q_1
q_2
q_6
q_3
q_4
q_5
a
a
b
b
b
ϵ
ϵ
ϵ
ϵ
ϵ

Deterministic Finite Automata (DFA)

A DFA is a special case of an NFA, where

1. there are no moves on ϵ, and
2. for each state and input symbol, the next state is unique.

Definition (DFA)

A *deterministic finite automaton* (or *DFA*) is defined by a tuple of five components:

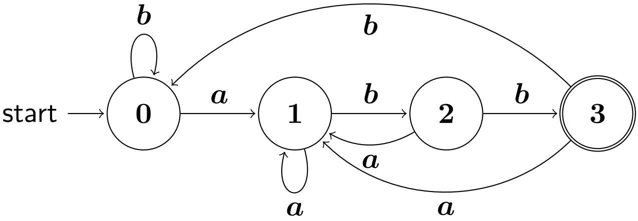
$$M = (Q, \Sigma, \delta, q_0, F)$$

where

- Q: a finite set of *states*
- Σ: a finite set of *input symbols* (or input alphabet)
- $\delta : Q \times \Sigma \to Q$: a *total* function called *transition function*
- $q_0 \in Q$: the *initial state*
- $F \subseteq Q$: a set of *final states*

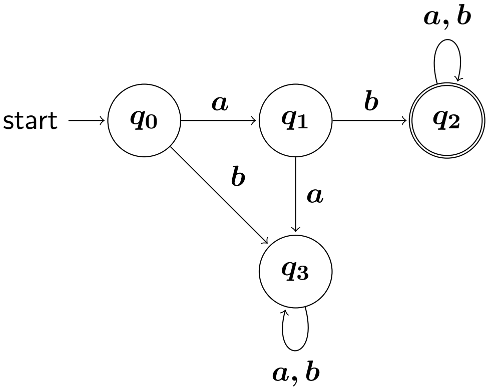
Example

A DFA that accepts $(a \mid b)^*abb$:



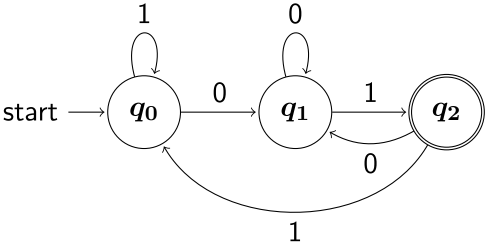
Exercise 1

What is the language of the DFA?



Exercise 2

What is the language of the DFA?



Summary

NFAs and DFAs are string recognizers.

- DFAs provide a concrete algorithm for recognizing strings.
- NFAs bridge the gap between REs and DFAs:
  - REs are descriptive but not executable.
  - DFAs are executable but not descriptive.
  - NFAs are in-between the REs and DFAs.

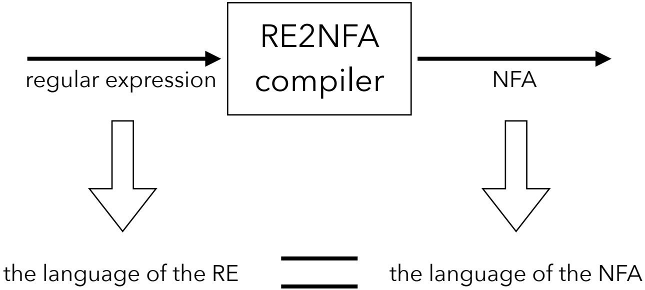
Part 3: Automation

Transform the lexical specification into an executable string recognizers:

$$\text{RE} \xrightarrow[\substack{\text{Thompson's} \\ \text{construction}}]{} \text{NFA} \xrightarrow[\substack{\text{subset} \\ \text{construction}}]{} \text{DFA}$$

From REs to NFAs

Transform a given regular expression into a semantically equivalent NFAs:



An instance of compilation, where

- source language is regular expressions and target language is NFAs
- the correctness is defined by the equivalence of the denoted languages

Principles of Compilation

Every automatic compilation

1. is done "compositionally", and
2. maintains some "invariants" during compilation.

Compilation of regular expressions, e.g., $R_1|R_2$:

1. The compilation of $R_1|R_2$ is defined in terms of the compilation of R_1 and R_2.
2. Compiled NFAs for R_1 and R_2 satisfy the invariants:
   - an NFA has exactly one accepting state,
   - no arcs into the initial state, and
   - no arcs out of the accepting state.

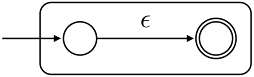
The Source Language

$$
\begin{array}{rcl}
R & \rightarrow & \emptyset \\
& | & \epsilon \\
& | & a \in \Sigma \\
& | & R_1 \mid R_2 \\
& | & R_1 \cdot R_2 \\
& | & R_1^* \\
& | & (R)
\end{array}
$$

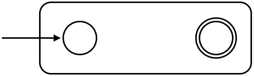
Compilation

Base cases:

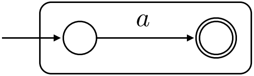
- $R = \epsilon$:



- $R = \emptyset$



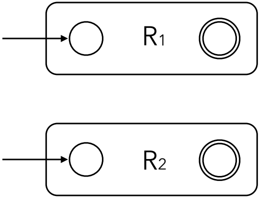
- $R = a \ (\in \Sigma)$



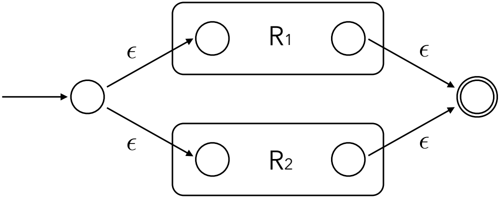
Compilation

Inductive cases:

- $R = R_1 | R_2$:
  1. Compile R_1 and R_2:

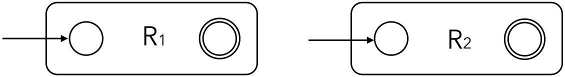


  2. Compile $R_1 | R_2$ using the results:

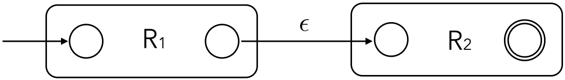


Compilation

- $R = R_1 \cdot R_2$:
  1. Compile R_1 and R_2:

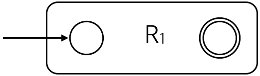


  2. Compile $R_1 \cdot R_2$ using the results:

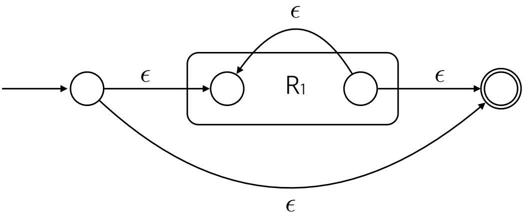


Compilation

- $R = R_1^*$:
  1. Compile R_1:



  2. Compile R_1^* using the results:



Examples

- $\mathbf{0} \cdot \mathbf{1}^*$:
- $(\mathbf{0}|\mathbf{1}) \cdot \mathbf{0} \cdot \mathbf{1}$:
- $(\mathbf{0}|\mathbf{1})^* \cdot \mathbf{1} \cdot (\mathbf{0}|\mathbf{1})$:

From NFA to DFA

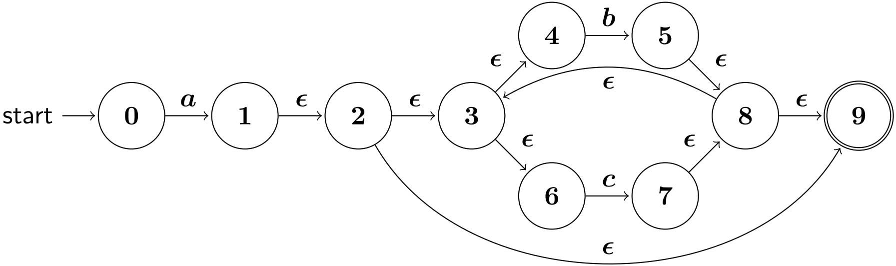
Transform an NFA

$$(N, \Sigma, \delta_N, n_0, N_A)$$

into an equivalent DFA

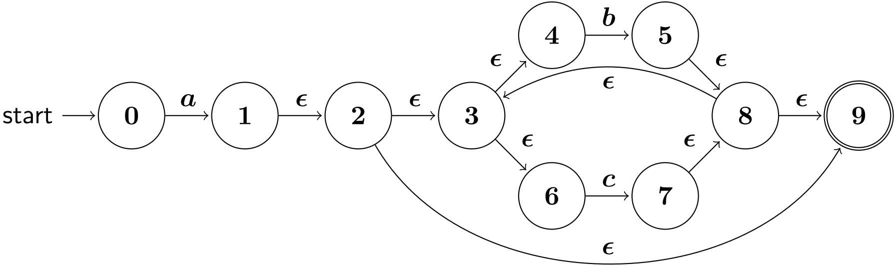
$$(D, \Sigma, \delta_D, d_0, D_A).$$

Running example:



ϵ-Closures

ϵ-**closure**(I): the set of states reachable from I without consuming any symbols.



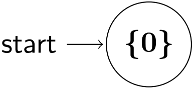
$$
\begin{aligned}
\epsilon\text{-}\mathbf{closure}(\{1\}) &= \{1,2,3,4,6,9\} \\
\epsilon\text{-}\mathbf{closure}(\{1,5\}) &= \{1,2,3,4,6,9\} \cup \{3,4,5,6,8,9\}
\end{aligned}
$$

Subset Construction

- Input: an NFA $(N, \Sigma, \delta_N, n_0, N_A)$.
- Output: a DFA $(D, \Sigma, \delta_D, d_0, D_A)$.
- Key Idea: the DFA simulates the NFA by considering every possibility at once. A DFA state $d \in D$ is a set of NFA state, i.e., $d \subseteq N$.

Running Example (1/5)

The initial DFA state $d_0 = \epsilon$-**closure**$(\{0\}) = \{0\}$.

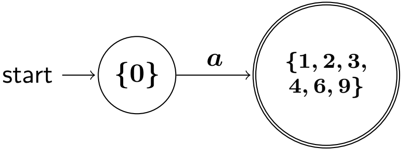


Running Example (2/5)

For the initial state S, consider every $x \in \Sigma$ and compute the corresponding next states:

$$\epsilon\text{-}\mathbf{closure}\Big(\bigcup_{s \in S} \delta(s, a)\Big).$$

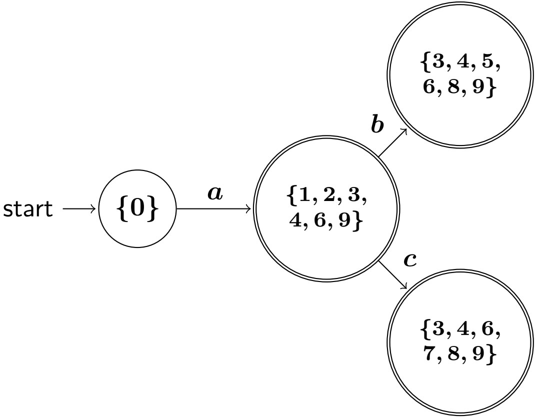
$$\begin{array}{rcl}
\epsilon\text{-}\mathbf{closure}(\bigcup_{s \in \{0\}} \delta(s, a)) & = & \{1, 2, 3, 4, 6, 9\} \\
\epsilon\text{-}\mathbf{closure}(\bigcup_{s \in \{0\}} \delta(s, b)) & = & \emptyset \\
\epsilon\text{-}\mathbf{closure}(\bigcup_{s \in \{0\}} \delta(s, c)) & = & \emptyset
\end{array}$$



Running Example (3/5)

For the state $\{1,2,3,4,6,9\}$, compute the next states:

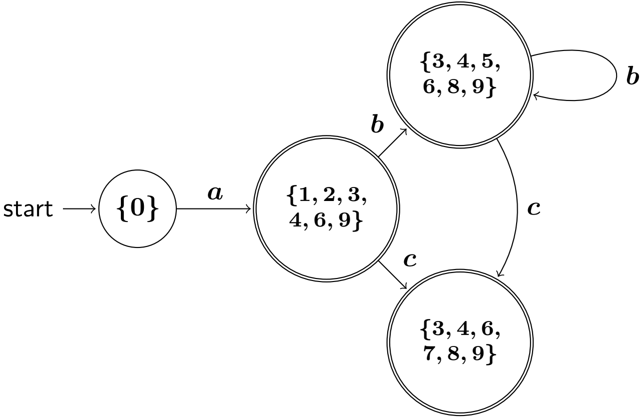
$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{1,2,3,4,6,9\}} \delta(s,a)) = \emptyset$$
$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{1,2,3,4,6,9\}} \delta(s,b)) = \{3,4,5,6,8,9\}$$
$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{1,2,3,4,6,9\}} \delta(s,c)) = \{3,4,6,7,8,9\}$$



Running Example (4/5)

Compute the next states of $\{3,4,5,6,8,9\}$:

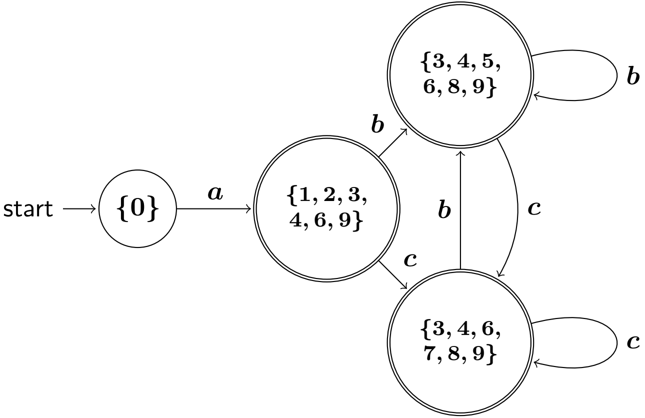
$$\epsilon\textbf{-closure}(\bigcup_{s\in\{3,4,5,6,8,9\}} \delta(s,a)) = \emptyset$$
$$\epsilon\textbf{-closure}(\bigcup_{s\in\{3,4,5,6,8,9\}} \delta(s,b)) = \{3,4,5,6,8,9\}$$
$$\epsilon\textbf{-closure}(\bigcup_{s\in\{3,4,5,6,8,9\}} \delta(s,c)) = \{3,4,6,7,8,9\}$$



Running Example (5/5)

Compute the next states of $\{3,4,6,7,8,9\}$:

$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{3,4,6,7,8,9\}}\delta(s,a)) = \emptyset$$
$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{3,4,6,7,8,9\}}\delta(s,b)) = \{3,4,5,6,8,9\}$$
$$\epsilon\text{-}\mathbf{closure}(\bigcup_{s\in\{3,4,6,7,8,9\}}\delta(s,c)) = \{3,4,6,7,8,9\}$$



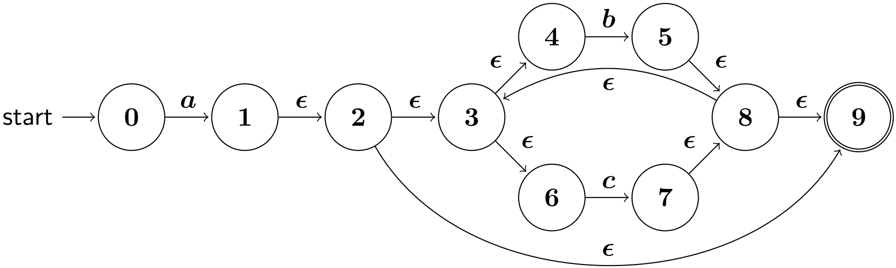
Subset Construction Algorithm

Algorithm 1 Subset construction

Input: An NFA $(N, \Sigma, \delta_N, n_0, N_A)$
Output: An equivalent DFA $(D, \Sigma, \delta_D, d_0, D_A)$

$d_0 = \epsilon\text{-closure}(\{n_0\})$
$D = \{d_0\}$
$W = \{d_0\}$
while $W \neq \emptyset$ **do**
  remove q from W
  for $c \in \Sigma$ **do**
    $t = \epsilon\text{-closure}(\bigcup_{s \in q} \delta(s, c))$
    $D = D \cup \{t\}$
    $\delta_D(q, c) = t$
    if t was newly added to D **then**
      $W = W \cup \{t\}$
    end if
  end for
end while
$D_A = \{q \in D \mid q \cap N_A \neq \emptyset\}$

Running Example (1/5)



The initial state $d_0 = \epsilon$-**closure**$(\{0\}) = \{0\}$. Initialize D and W:

$$D = \{\{0\}\}, \qquad W = \{\{0\}\}$$

Running Example (2/5)

Choose $q = \{0\}$ from W. For all $c \in \Sigma$, update δ_D:

	a	b	c
$\{0\}$	$\{1,2,3,4,6,9\}$	$\emptyset$	$\emptyset$

Update D and W:

$$D = \{\{0\}, \{1,2,3,4,6,9\}\}, \qquad W = \{\{1,2,3,4,6,9\}\}$$

Running Example (3/5)

Choose $q = \{1,2,3,4,6,9\}$ from W. For all $c \in \Sigma$, update δ_D:

	a	b	c
$\{0\}$	$\{1,2,3,4,6,9\}$	$\emptyset$	$\emptyset$
$\{1,2,3,4,6,9\}$	$\emptyset$	$\{3,4,5,6,8,9\}$	$\{3,4,6,7,8,9\}$

Update D and W:

$$\begin{array}{rcl} D & = & \{\{0\},\{1,2,3,4,6,9\},\{3,4,5,6,8,9\},\{3,4,6,7,8,9\}\} \\ W & = & \{\{3,4,5,6,8,9\},\{3,4,6,7,8,9\}\} \end{array}$$

Running Example (4/5)

Choose $q = \{3, 4, 5, 6, 8, 9\}$ from W. For all $c \in \Sigma$, update δ_D:

	a	b	c
$\{0\}$	$\{1, 2, 3, 4, 6, 9\}$	$\emptyset$	$\emptyset$
$\{1, 2, 3, 4, 6, 9\}$	$\emptyset$	$\{3, 4, 5, 6, 8, 9\}$	$\{3, 4, 6, 7, 8, 9\}$
$\{3, 4, 5, 6, 8, 9\}$	$\emptyset$	$\{3, 4, 5, 6, 8, 9\}$	$\{3, 4, 6, 7, 8, 9\}$

D and W:

$$\begin{array}{rcl} D & = & \{\{0\}, \{1, 2, 3, 4, 6, 9\}, \{3, 4, 5, 6, 8, 9\}, \{3, 4, 6, 7, 8, 9\}\} \\ W & = & \{\{3, 4, 6, 7, 8, 9\}\} \end{array}$$

Running Example (5/5)

Choose $q = \{3,4,6,7,8,9\}$ from W. For all $c \in \Sigma$, update δ_D:

	a	b	c
$\{0\}$	$\{1,2,3,4,6,9\}$	$\emptyset$	$\emptyset$
$\{1,2,3,4,6,9\}$	$\emptyset$	$\{3,4,5,6,8,9\}$	$\{3,4,6,7,8,9\}$
$\{3,4,5,6,8,9\}$	$\emptyset$	$\{3,4,5,6,8,9\}$	$\{3,4,6,7,8,9\}$
$\{3,4,6,7,8,9\}$	$\emptyset$	$\{3,4,5,6,8,9\}$	$\{3,4,6,7,8,9\}$

D and W:

$$\begin{array}{rcl} D & = & \{\{0\},\{1,2,3,4,6,9\},\{3,4,5,6,8,9\},\{3,4,6,7,8,9\}\} \\ W & = & \emptyset \end{array}$$

The while loop terminates. The accepting states:

$$D_A = \{\{1,2,3,4,6,9\},\{3,4,5,6,8,9\},\{3,4,6,7,8,9\}\}$$

Algorithm for computing ϵ-Closures

- The definition

  ϵ-**closure**(I) is the set of states reachable from I
  without consuming any symbols.

  is neither formal nor constructive.

- A formal definition:
  $T = \epsilon$-**closure**(I) is the smallest set such that

$$I \cup \bigcup_{s \in T} \delta(s, \epsilon) \subseteq T.$$

- Alternatively, T is the smallest solution of the equation

$$F(X) \subseteq (X)$$

  where

$$F(X) = I \cup \bigcup_{s \in X} \delta(s, \epsilon).$$

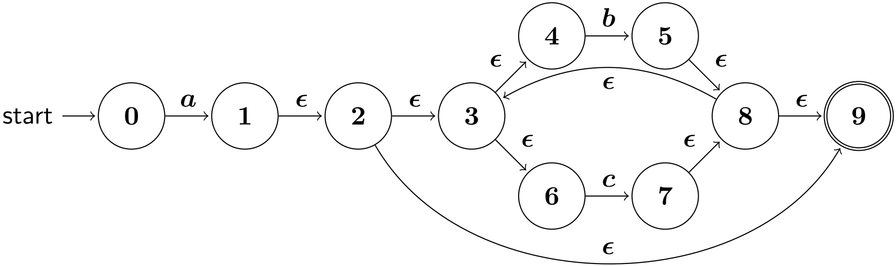
  Such a solution is called the least fixed point of F.

Fixed Point Iteration

The least fixed point of a function can be computed by the *fixed point iteration*:

$$
\begin{array}{l}
\boldsymbol{T = \emptyset} \\
\textbf{repeat} \\
\quad \boldsymbol{T' = T} \\
\quad \boldsymbol{T = T' \cup F(T')} \\
\textbf{until } \boldsymbol{T = T'}
\end{array}
$$

Example



ϵ-**closure**$(\{1\})$:

Iteration	T'	T
1	$\emptyset$	$\{1\}$
2	$\{1\}$	$\{1, 2\}$
3	$\{1, 2\}$	$\{1, 2, 3, 9\}$
4	$\{1, 2, 3, 9\}$	$\{1, 2, 3, 4, 6, 9\}$
5	$\{1, 2, 3, 4, 6, 9\}$	$\{1, 2, 3, 4, 6, 9\}$

Summary

Key concepts in lexical analysis:

- Specification: Regular expressions
- Implementation: Deterministic Finite Automata