

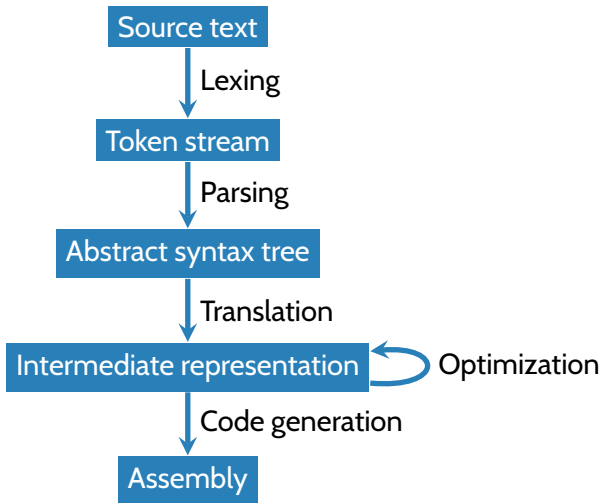
COS320: Compiling Techniques

Zak Kincaid

January 29, 2026

Lexing

Compiler phases (simplified)



- The *lexing* (or *lexical analysis*) phase of a compiler breaks a stream of characters (source text) into a stream of *tokens*.
 - Whitespace and comments often discarded
- A *token* is a sequence of characters treated as a unit (a *lexeme*) along with an *token type*:
 - *identifier tokens*: `x`, `y`, `foo`, ...
 - *integer tokens*: `0`, `1`, `-14`, `512`, ...
 - *if tokens*: `if`
 - ...
- Algebraic datatypes are a convenient representation for tokens

```
type token = IDENT of string
           | INT of int
           | IF
           | ...
```

```
// compute absolute value
if ( $x < 0$ ) {
    return  $-x$ ;
} else {
    return  $x$ ;
}
```

↓**Lexer**

*IF, LPAREN, IDENT "x", LT, INT 0, RPAREN, LBRACE,
RETURN, MINUS, IDENT "x", SEMI,
RBRACE, ELSE, LBRACE,
RETURN, IDENT "x", SEMI,
RBRACE*

Implementing a lexer

- Option 1: write by hand
- Option 2: use a *lexer generator*
 - Write a *lexical specification* in a domain-specific language
 - Lexer generator compiles specification to a lexer (in language of choice)
- Many lexer generators available
 - lex, flex, ocamllex, jflex, ...

Formal Languages

- An *alphabet* Σ is a finite set of symbols (e.g., $\{0, 1\}$, ASCII, unicode, tokens).
- A *word* (or *string*) over Σ is a finite sequence $w = w_1 w_2 w_3 \dots w_n$, with each $w_i \in \Sigma$.
 - The *empty word* ϵ is a word over any alphabet
 - The set of all words over Σ is typically denoted Σ^*
 - E.g., $01001 \in \{0, 1\}^*$, *embiggen* $\in \{a, \dots, z\}^*$
- A *language* over Σ is a set of words over Σ
 - Integer literals form a language over $\{0, \dots, 9, -\}$
 - The keywords of OCaml form a (finite) language over ASCII
 - Syntactically-valid Java programs forms an (infinite) language over Unicode

Regular expressions (regex)

- Regular expressions are one mechanism for describing languages
 - E.g., $0|(1(0|1)^*)$ recognizes the language of all binary sequences without leading zeros
- Abstract syntax of regular expressions:

$\langle \text{RegExp} \rangle ::=$	ϵ	Empty word
	$ \Sigma$	Letter
	$ \langle \text{RegExp} \rangle \langle \text{RegExp} \rangle$	Concatenation
	$ \langle \text{RegExp} \rangle \langle \text{RegExp} \rangle$	Alternative
	$ \langle \text{RegExp} \rangle^*$	Repetition

Regular expressions (regex)

- Regular expressions are one mechanism for describing languages
 - E.g., $0|(1(0|1)^*)$ recognizes the language of all binary sequences without leading zeros
- Abstract syntax of regular expressions:

$\langle \text{RegExp} \rangle ::=$	ϵ	Empty word
	$ \Sigma$	Letter
	$ \langle \text{RegExp} \rangle \langle \text{RegExp} \rangle$	Concatenation
	$ \langle \text{RegExp} \rangle \langle \text{RegExp} \rangle$	Alternative
	$ \langle \text{RegExp} \rangle^*$	Repetition

- Meaning of regular expressions:

$$\mathcal{L}(\epsilon) = \{\epsilon\}$$

$$\mathcal{L}(a) = \{a\}$$

$$\mathcal{L}(R_1 R_2) = \{uv : u \in \mathcal{L}(R_1) \wedge v \in \mathcal{L}(R_2)\}$$

$$\mathcal{L}(R_1 | R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$$

$$\mathcal{L}(R^*) = \{\epsilon\} \cup \mathcal{L}(R) \cup \mathcal{L}(RR) \cup \mathcal{L}(RRR) \cup \dots$$

ocamllex regex concrete syntax

- 'a': letter
- "abc": string (equiv. 'a"b"c')
- R+: one or more repetitions of R (equiv. RR*)
- R?: zero or one R (equiv. R| ϵ)
- ['a'-'z']: character range (equiv. 'a'|'b'|...|'z')
- R as x: bind string matched by R to variable x

Lexer generators

Lexer generators take as input a lexical specification, and output code that tokenizes a character stream w.r.t. that specification

Example lexical specification:

$$\begin{array}{lcl} \text{token type} & & \text{pattern} \\ \hline \text{identifier} & = & [a - zA - Z][a - zA - Z0 - 9]^* \\ \text{integer} & = & [1 - 9][0 - 9]^* \\ \text{plus} & = & + \end{array}$$

Lexer generators

Lexer generators take as input a lexical specification, and output code that tokenizes a character stream w.r.t. that specification

Example lexical specification:

$$\begin{aligned} \text{token type} & \qquad \text{pattern} \\ \text{identifier} &= [a - zA - Z][a - zA - Z0 - 9]^* \\ \text{integer} &= [1 - 9][0 - 9]^* \\ \text{plus} &= + \end{aligned}$$

- “foo+42+bar” \rightarrow $\underbrace{\text{identifier}}_{\text{token type}}$ “foo”, $\underbrace{\text{plus}}_{\text{token type}}$ “+”, $\underbrace{\text{integer}}_{\text{token type}}$ “42”, $\underbrace{\text{plus}}_{\text{token type}}$ “+”, $\underbrace{\text{identifier}}_{\text{token type}}$ “bar”
token type lexeme

Lexer generators

Lexer generators take as input a lexical specification, and output code that tokenizes a character stream w.r.t. that specification

Example lexical specification:

$$\begin{array}{lcl} & \text{token type} & \text{pattern} \\ \text{identifier} & = & [a - zA - Z][a - zA - Z0 - 9]^* \\ \text{integer} & = & [1 - 9][0 - 9]^* \\ \text{plus} & = & + \end{array}$$

- “foo+42+bar” \rightarrow $\underbrace{\text{identifier}}_{\text{token type}}$ “foo”, $\underbrace{\text{plus}}_{\text{lexeme}}$ “+”, integer “42”, plus “+”, identifier “bar”
- Typically, lexical spec associates an *action* to each token type, which is code that is evaluated on the lexeme (often: produce a token value)

Disambiguation

- May be more than one way to lex a string:

$IF = \text{if}$

$IDENT = [a-zA-Z][a-zA-Z0-9]^*$

$INT = [1-9][0-9]^*$

$LT = <$

...

- Input string `ifx<10`: IDENT "ifx", LT, INT 10 or IF, IDENT "x", LT, INT 10?
- Input string `if x<9`: IF, IDENT "x", LT, INT 9 or IDENT "if", IDENT "x", LT, INT 9?

Disambiguation

- May be more than one way to lex a string:

$IF = \text{if}$

$IDENT = [a-zA-Z][a-zA-Z0-9]^*$

$INT = [1-9][0-9]^*$

$LT = <$

...

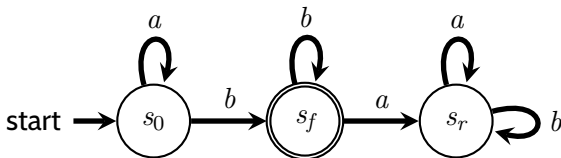
- Input string `ifx<10`: `IDENT "ifx", LT, INT 10` or `IF, IDENT "x", LT, INT 10`?
- Input string `if x<9`: `IF, IDENT "x", LT, INT 9` or `IDENT "if", IDENT "x", LT, INT 9`?
- Two rules sufficient to disambiguate (remember these!)
 - 1 **The lexer is greedy**: always prefer longest match
 - 2 **Order matters**: prefer earlier patterns

How do lexer generators work?

Lexer generator pipeline

- Lexical specification is compiled to a *deterministic finite automaton* (DFA), which can be executed efficiently
- Typical pipeline: lexical specification \rightarrow *nondeterministic* FA \rightarrow DFA
- Kleene's theorem: regular expressions, NFAs, and DFAs describe the same class of languages
 - A language is *regular* if it is accepted by a regular expression (equiv., NFA, DFA).

Deterministic finite automata (DFA)

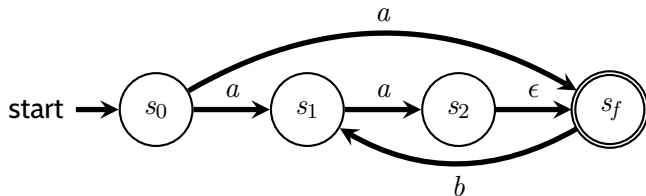


A **deterministic finite automaton** (DFA) $A = (Q, \Sigma, \delta, s, F)$ consists of

- Q : finite set of states
- Σ : finite alphabet
- $\delta : Q \times \Sigma \rightarrow Q$: transition function
 - Every state has *exactly* one outgoing edge per letter
- $s \in Q$: initial state
- $F \subseteq Q$: final (accepting) states

DFA accepts a string $w = w_1 \dots w_n \in \Sigma^*$ iff $\delta(\dots \delta(\delta(s, w_1), w_2), \dots, w_n) \in F$.

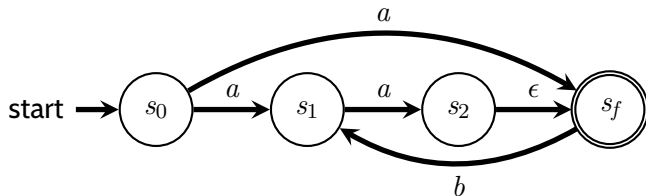
Non-deterministic finite automata



A **non-deterministic finite automaton** (NFA) $A = (Q, \Sigma, \Delta, s, F)$ generalization of a DFA, where

- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: transition *relation*
 - A state can have *more than one* outgoing edge for a given letter
 - A state can have *no* outgoing edges for a given letter
 - A state can have ϵ -transitions (read no input, but change state)

Non-deterministic finite automata



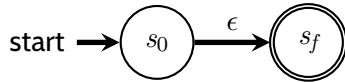
A **non-deterministic finite automaton** (NFA) $A = (Q, \Sigma, \Delta, s, F)$ generalization of a DFA, where

- $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$: transition *relation*
 - A state can have *more than one* outgoing edge for a given letter
 - A state can have *no* outgoing edges for a given letter
 - A state can have ϵ -transitions (read no input, but change state)

NFA accepts a string $w = w_1 \dots w_n \in \Sigma^*$ iff there exists a w -labeled path from s to an final state (i.e., there is some sequence $(q_0, u_1, q_1), (q_1, u_2, q_2), \dots, (q_{m-1}, u_m, q_m)$ with $q_0 = s, q_m \in F$, and $u_1 u_2 \dots u_m = w$).

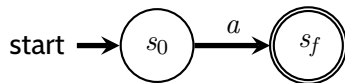
Regex \rightarrow NFA

Case: ϵ (empty word)



Regex \rightarrow NFA

Case: a (letter)



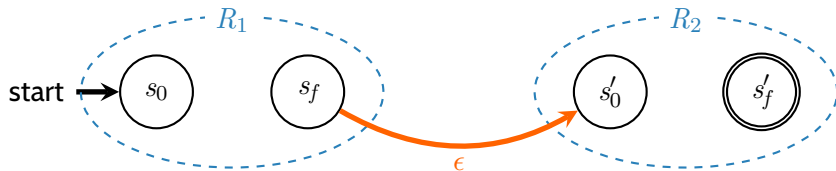
Regex \rightarrow NFA

Case: $R_1 R_2$ (concatenation)



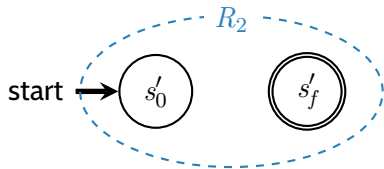
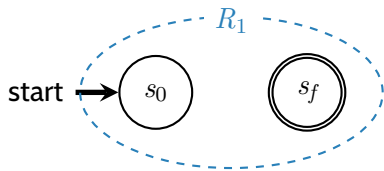
Regex \rightarrow NFA

Case: $R_1 R_2$ (concatenation)



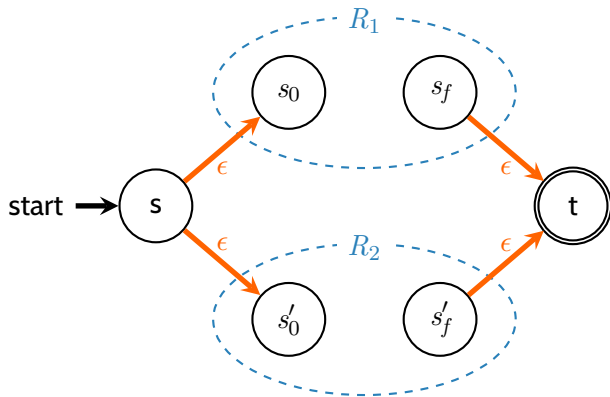
Regex \rightarrow NFA

Case: $R_1|R_2$ (alternative)



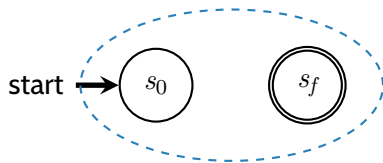
Regex \rightarrow NFA

Case: $R_1 | R_2$ (alternative)



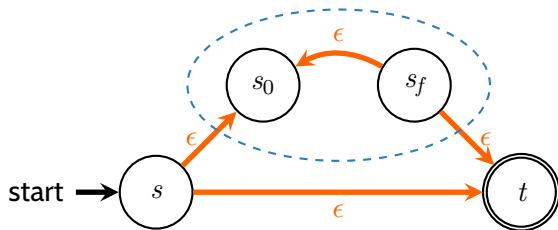
Regex \rightarrow NFA

Case: R^* (iteration)



Regex \rightarrow NFA

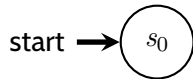
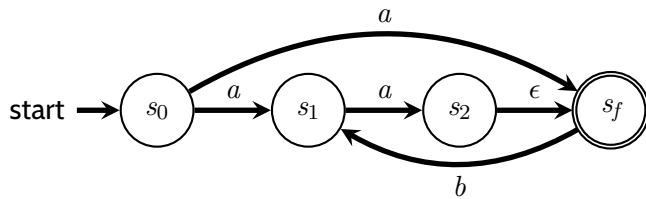
Case: R^* (iteration)



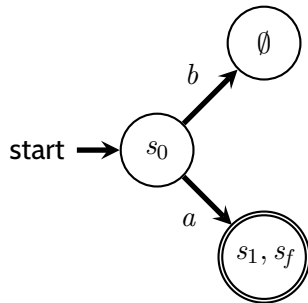
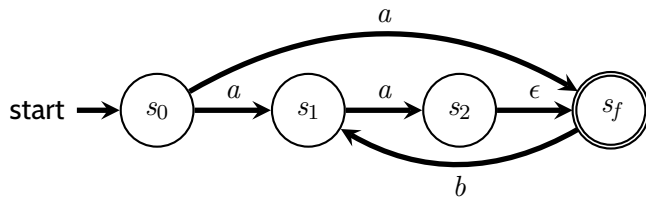
NFA \rightarrow DFA

- For any NFA, there is a DFA that recognizes the same language
- **Intuition:** the DFA simulates all possible paths of the NFA simultaneously
 - There is an unbounded number of paths *but* we only care about the “end state” of each path, not its history
 - States of the DFA track the set of possible states the NFA could be in
 - DFA accepts when *some* path accepts

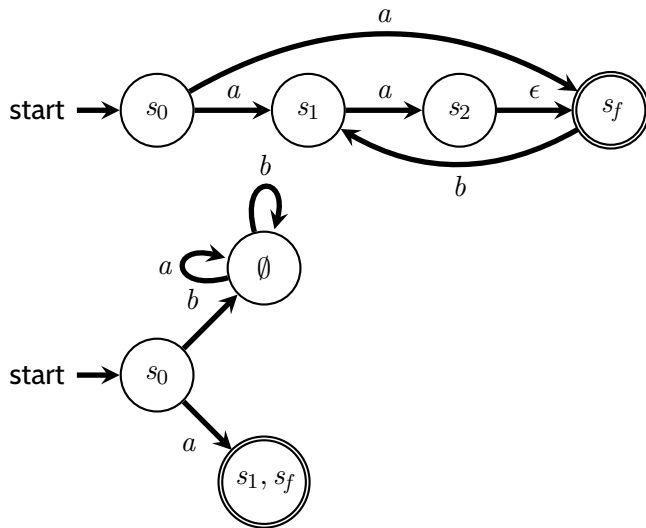
NFA \rightarrow DFA



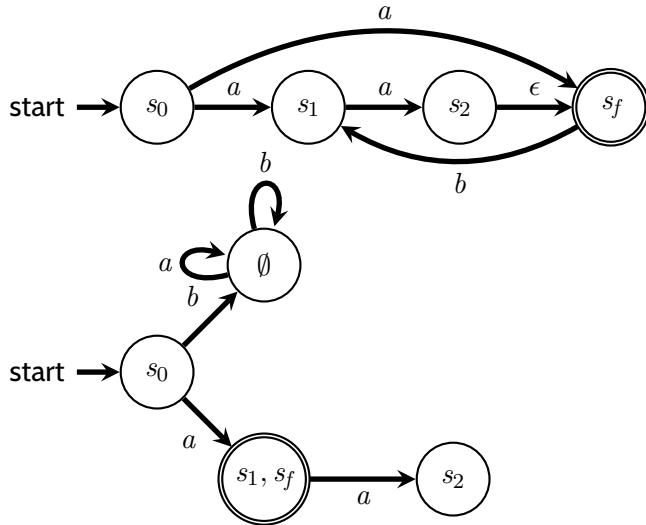
NFA \rightarrow DFA



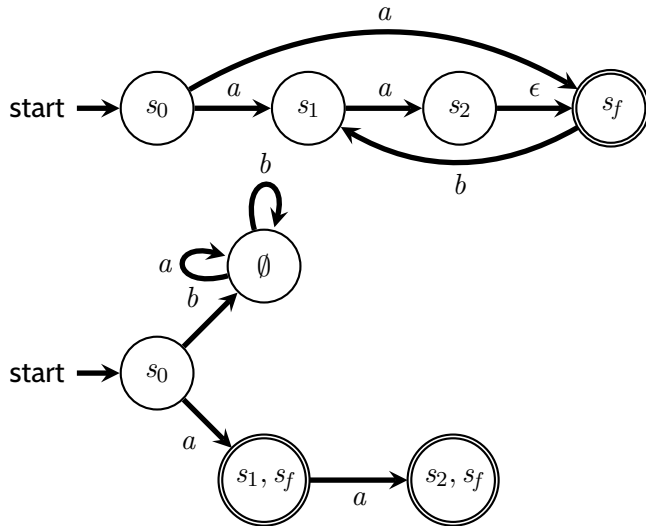
NFA \rightarrow DFA



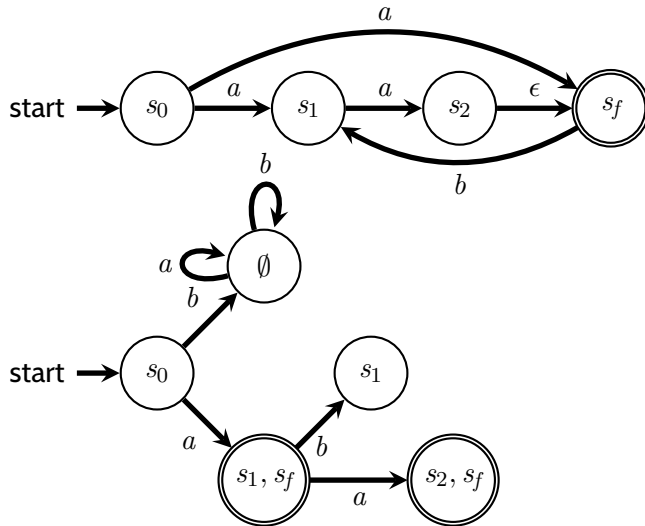
NFA \rightarrow DFA



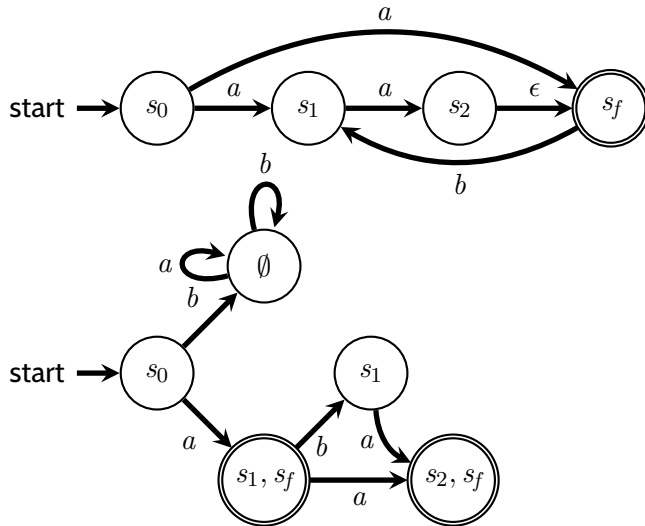
NFA \rightarrow DFA



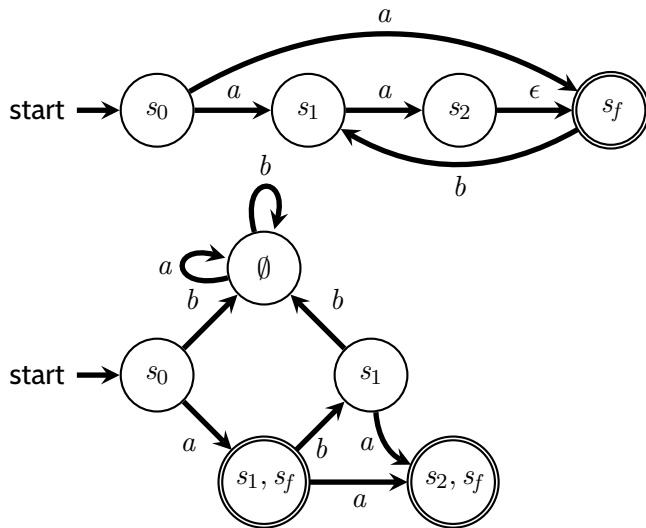
NFA \rightarrow DFA



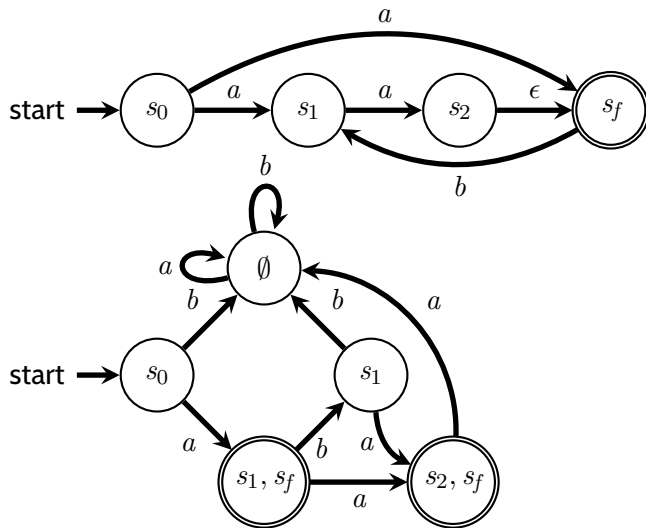
NFA \rightarrow DFA



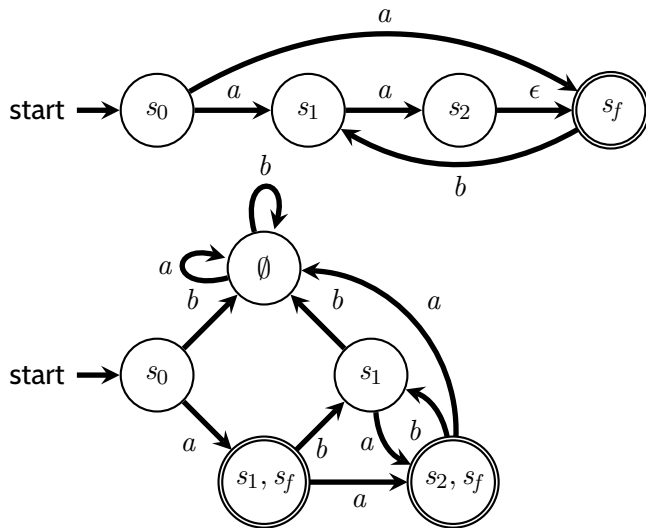
NFA \rightarrow DFA



NFA \rightarrow DFA



NFA \rightarrow DFA



NFA \rightarrow DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the ϵ -closure of S to be the set of states reachable from S by ϵ transitions (incl. S)
 $\epsilon\text{-cl}(S)$ = smallest set that contains S and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$

NFA \rightarrow DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the ϵ -closure of S to be the set of states reachable from S by ϵ transitions (incl. S)
 $\epsilon\text{-cl}(S)$ = smallest set that contains S and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
- Construct DFA as follows:
 - Q' = set of all ϵ -closed subsets of Q
 - $\delta'(S, a) = \epsilon\text{-closure of } \{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
 - $s' = \epsilon\text{-closure of } \{s\}$
 - $F' = \{S \in Q' : S \cap F \neq \emptyset\}$

NFA \rightarrow DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the ϵ -closure of S to be the set of states reachable from S by ϵ transitions (incl. S)
 $\epsilon\text{-cl}(S)$ = smallest set that contains S and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
- Construct DFA as follows:
 - Q' = set of all ϵ -closed subsets of Q
 - $\delta'(S, a) = \epsilon\text{-closure of } \{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
 - $s' = \epsilon\text{-closure of } \{s\}$
 - $F' = \{S \in Q' : S \cap F \neq \emptyset\}$
- **Crucial optimization:** only construct states that are reachable from s'

NFA \rightarrow DFA, formally

- Have: NFA $A = (Q, \Sigma, \delta, s, F)$. Want: DFA $A' = (Q', \Sigma, \delta', s', F')$ that accepts same language.
- For any $S \subseteq Q$, define the ϵ -closure of S to be the set of states reachable from S by ϵ transitions (incl. S)
 $\epsilon\text{-cl}(S)$ = smallest set that contains S and such that $\forall (q, \epsilon, q') \in \Delta, q \in S \Rightarrow q' \in S$
- Construct DFA as follows:
 - Q' = set of all ϵ -closed subsets of Q
 - $\delta'(S, a) = \epsilon\text{-closure of } \{q_2 : \exists q_1 \in S. (q_1, a, q_2) \in \Delta\}$
 - $s' = \epsilon\text{-closure of } \{s\}$
 - $F' = \{S \in Q' : S \cap F \neq \emptyset\}$
- **Crucial optimization:** only construct states that are reachable from s'
- Less crucial, still important: minimize DFA (Hopcroft's algorithm, $O(n \log n)$)

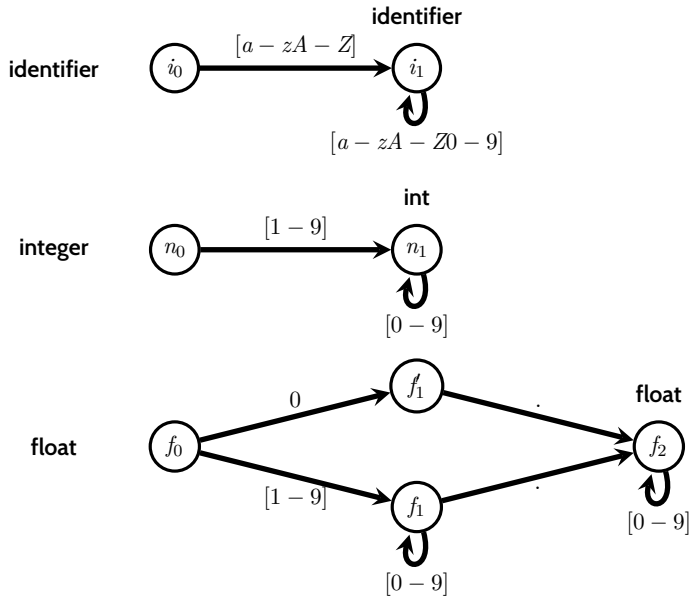
Lexical specification \rightarrow String classifier

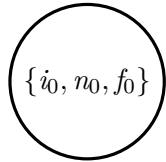
- Want: partial function *match* mapping strings to token types
 - $match(s)$ = highest-priority token type whose pattern matches s (undef otherwise)
- Process:
 - 1 Convert each pattern to an NFA. Label accepting states w/ token types.
 - 2 Take the union of all NFAs
 - 3 Convert to DFA
 - States of the DFA labeled with *sets* of token types.
 - Take highest priority.

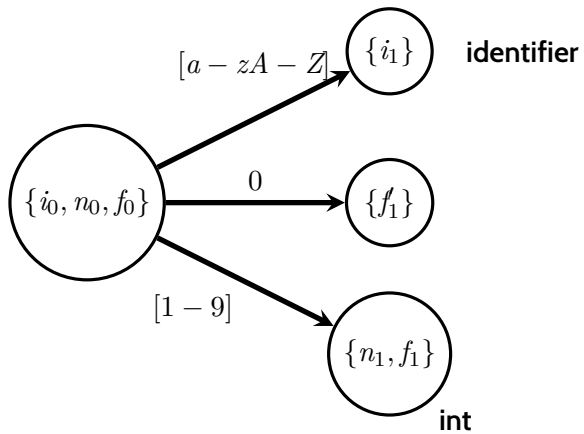
identifier = $[a - zA - Z][a - zA - Z0 - 9]^*$

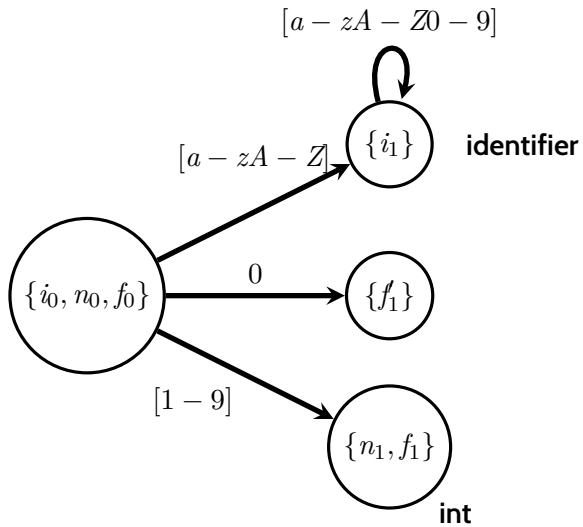
integer = $[1 - 9][0 - 9]^*$

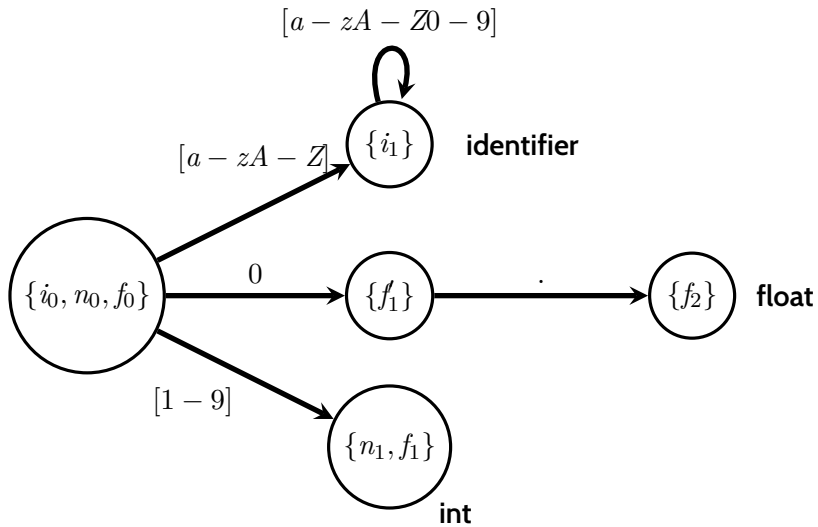
float = $([1 - 9][0 - 9]^*|0).[0 - 9]^+$

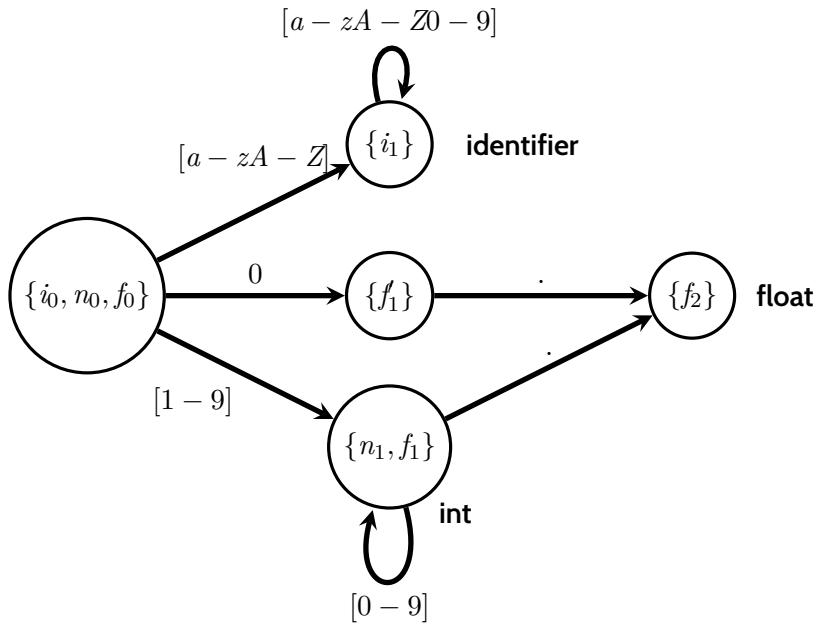


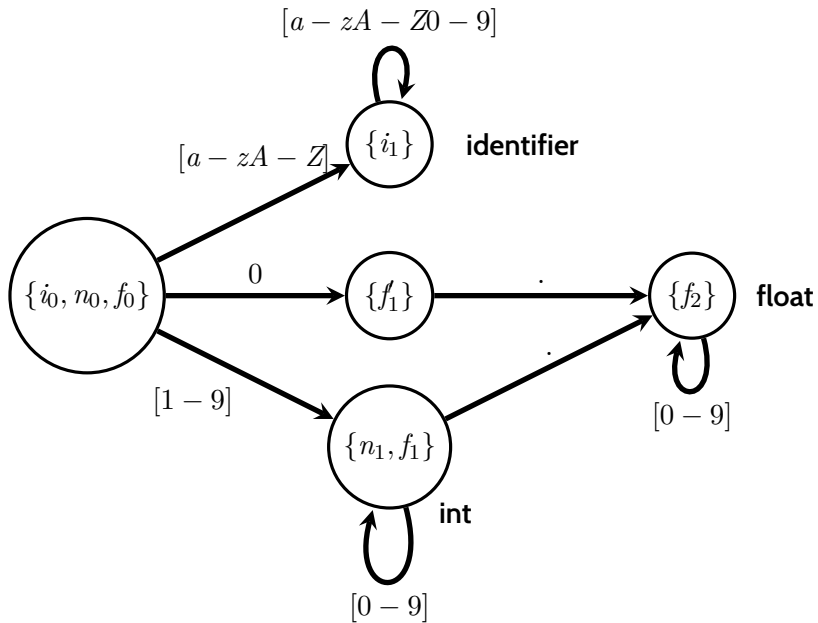












Compiler phases (simplified)

