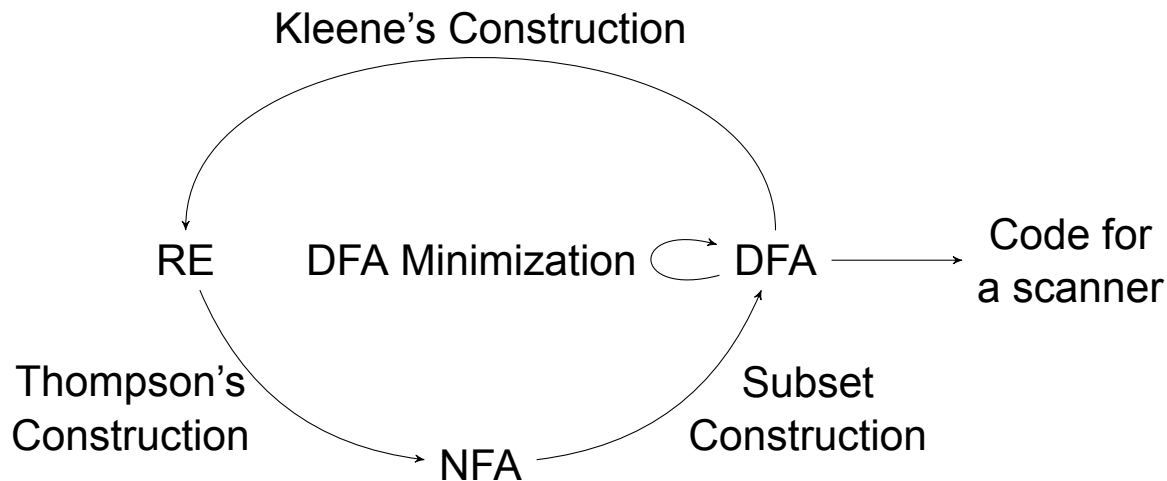


From Regular Expression to Scanner



From Regular Expression to Scanner

The construction of a Deterministic FA (DFA) from an RE follows three steps.

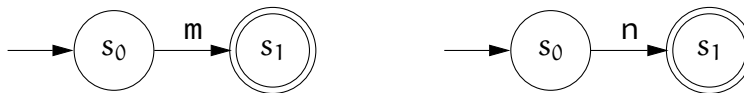
1. **Thompson's construction** derives a Nondeterministic FA (NFA) from a RE
2. **Subset construction** builds a DFA that simulates the NFA
3. **Hopcroft's algorithm** minimizes the DFA

Nondeterministic Finite Automata

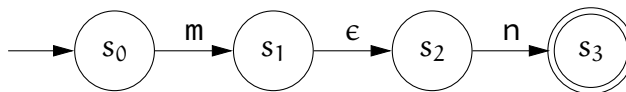
A **Nondeterministic Finite Automaton (NFA)** is an FA that allows...

- transitions on the empty string, *i.e.*, ϵ -transitions that do not advance the input, and
- states that have multiple transitions on the same character.

We can use transitions on ϵ to combine FAs and form FAs for more complex REs.

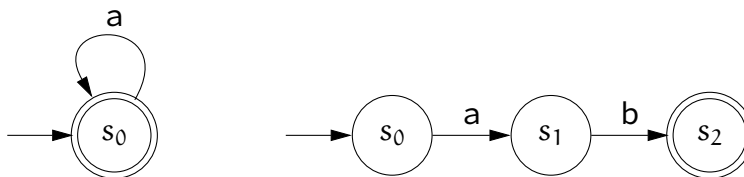


For example, using a ϵ -transition, the FAs for the REs m and n can be combined into an FA for the RE mn as follows.

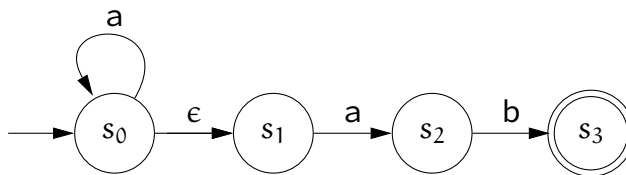


Nondeterministic Finite Automata

Merging two FAs with an ϵ -transition can complicate our model of how FAs work. Consider the FAs for the languages a^* and ab .



We can combine them with an ϵ -transition to form a FA for a^*ab .



Nondeterministic Finite Automata

The ϵ -transition gives the FA two distinct transitions out of s_0 on the letter a.

- it can take the transition $s_0 \xrightarrow{a} s_0$, or
- the two transitions $s_0 \xrightarrow{\epsilon} s_1$ and $s_1 \xrightarrow{a} s_2$

Which one is correct? Consider the strings aab and ab, which the NFA should both accept.

aab $s_0 \xrightarrow{a} s_0, s_0 \xrightarrow{\epsilon} s_1, s_1 \xrightarrow{a} s_2, s_2 \xrightarrow{b} s_3$

ab $s_0 \xrightarrow{\epsilon} s_1, s_1 \xrightarrow{a} s_2, s_2 \xrightarrow{b} s_3$

→ The correct transition out of s_0 depends on the characters that follow a.

Nondeterministic Finite Automata

To understand how a NFA operates, we need a set of rules that describe its behavior. Historically, two distinct models have been given for the behavior of a NFA.

1. **NFA guesses correct transition at each point**

- NFA is omniscient
- follow transition that leads to an accepting state, if such a transition exists

2. **NFA pursues all paths concurrently**

- NFA clones itself to pursue each possible transition
- specific set of states in which the NFA is active is called its configuration
- NFA accepts the string if reaches a configuration in which it has exhausted the input and one or more of the clones has reached an accepting state

Equivalence of NFA and DFA

NFAs and DFAs are **equivalent** in their expressive power

- any DFA is a special case of an NFA, *i.e.*, an NFA is at least as powerful as a DFA
- ← any NFA can be simulated by a DFA by subset construction (*cf.* Slide 80)

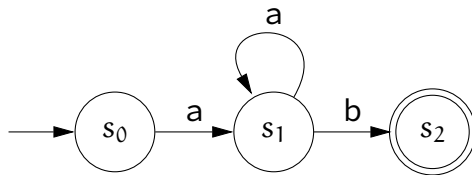
The intuition behind **subset construction** is simple: to simulate the behavior of the NFA, we need a DFA with a state for each configuration of the NFA.

- under the second model of NFA behavior, the NFA has a finite number of clones
- this number can be bounded: an NFA with n states produces at most $|\Sigma|^n$ configurations
- to simulate the NFA, we need a DFA with a state for each configuration of the NFA

Note The DFA that simulates the NFA still runs in time proportional to the length of the input string, *i.e.*, we have a potential space problem, but not a time problem.

Equivalence of NFA and DFA

Since NFAs and DFAs are equivalent, we can construct the following DFA for a^*ab .



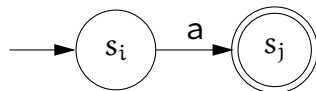
It relies on the observation that a^*ab specifies the same regular language as aa^*b .

From RE to NFA: Thompson's Construction

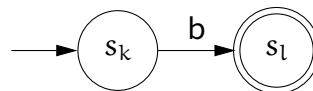
As a first step in moving from an RE to an implemented scanner, we must derive an NFA from the RE.

Thompson's construction builds an NFA from an RE in a straightforward way

- one template for building each NFA that corresponds to a single-letter RE
- NFA transformations for RE operators: concatenation, alternation, and closure

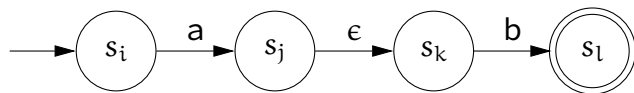


NFA for a

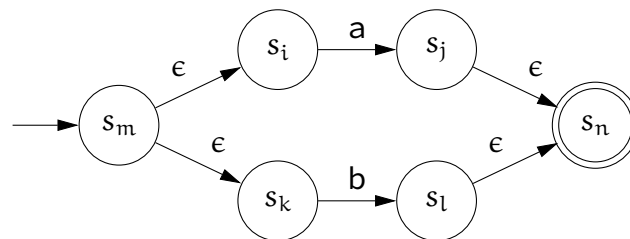


NFA for b

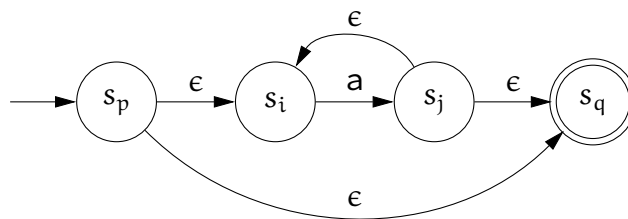
From RE to NFA: Thompson's Construction



NFA for ab



NFA for $a|b$

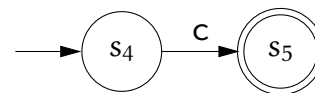
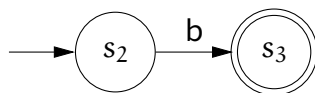
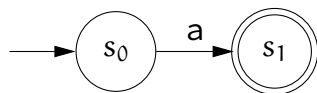


NFA for a^*

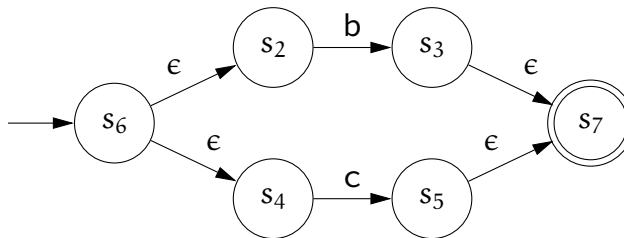
Example: Thompson's Construction

As an example, we use Thompson's Construction to derive the NFA for the RE $a(b \mid c)^*$.

1. The construction begins by first building NFAs for a , b , and c

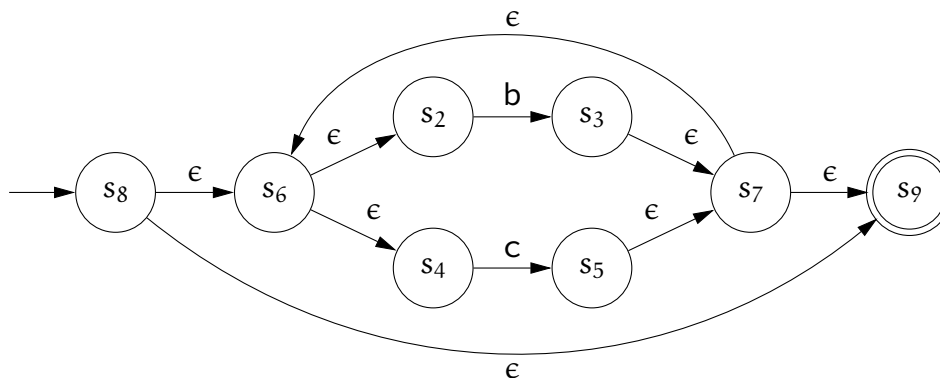


2. Since parentheses have highest precedence, it then builds the NFA for the expression enclosed in parentheses, $b \mid c$



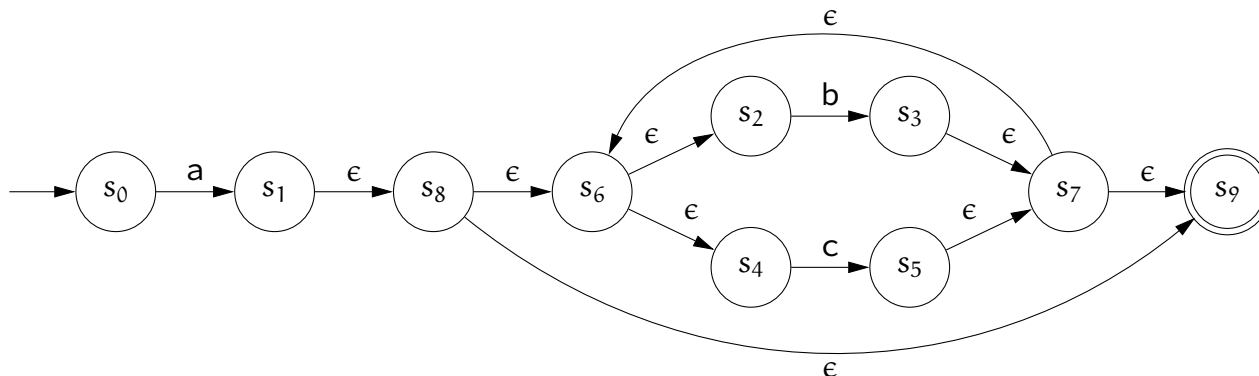
Example: Thompson's Construction

3. Since closure has higher precedence than concatenation, the construction next builds the closure, $(b \mid c)^*$



Example: Thompson's Construction

4. Finally, it concatenates the NFA for a to the NFA for $(b \mid c)^*$

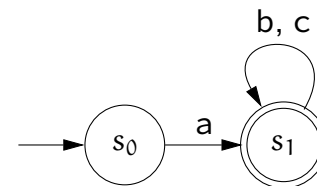


From RE to NFA: Thompson's Construction

The NFAs derived from Thompson's construction have **several specific** properties that simplify an implementation.

- each NFA has one start state and one accepting state
- no transition, other than the initial transition, enters the start state
- no transition leaves the accepting state
- an ϵ -transition always connects two states that were, earlier in the process, the start state and the accepting state of NFAs for some component REs
- each state has at most two entering and two exiting ϵ -moves, and at most one entering and one exiting move on a symbol in the alphabet

Not to rain on your parade, but in general Thompson's construction builds very large NFAs with many ϵ -moves that are unneeded.



From NFA to DFA: Subset Construction

Since DFA execution is much easier to simulate than NFA execution, we convert the NFA built by Thompson's construction into a DFA that recognizes the **same** language.

The **Subset Construction** takes as input an NFA $(N, \Sigma, \delta_N, n_0, N_A)$ and produces a DFA $(D, \Sigma, \delta_D, d_0, D_A)$

- the NFA and the DFA use the same alphabet Σ
- the DFA's start state d_0 and its accepting states D_A will emerge from the construction
- the **complex part** of the construction is
 - the derivation of the set of DFA states D from the NFA states N , and
 - the derivation of the DFA transition function δ_D

From NFA to DFA: Subset Construction

```

1  $q_0 \leftarrow \epsilon\text{-closure}(\{n_0\})$ 
2  $Q \leftarrow q_0$ 
3  $WorkList \leftarrow \{q_0\}$ 
4 while  $WorkList \neq \emptyset$  do
5   remove  $q$  from  $WorkList$ 
6   foreach character  $c \in \Sigma$  do
7      $t \leftarrow \epsilon\text{-closure}(\Delta(q, c))$ 
8      $T[q, c] \leftarrow t$ 
9     if  $t \notin Q$  then
10       add  $t$  to  $Q$ 
11       add  $t$  to  $WorkList$ 

```

Algorithm constructs a set Q whose elements q_i are each a **subset** of N , *i.e.*, each $q_i \in 2^N$

$\epsilon\text{-closure}(S)$ examines each state $s_i \in S$ and adds to S any state reachable by following one or more ϵ -transitions from s_i

$\Delta(q, c)$ applies the transition function of the NFA to each state s in q and returns

$$\bigcup_{s \in q} \delta_N(s, c)$$

From NFA to DFA: Subset Construction

Notice

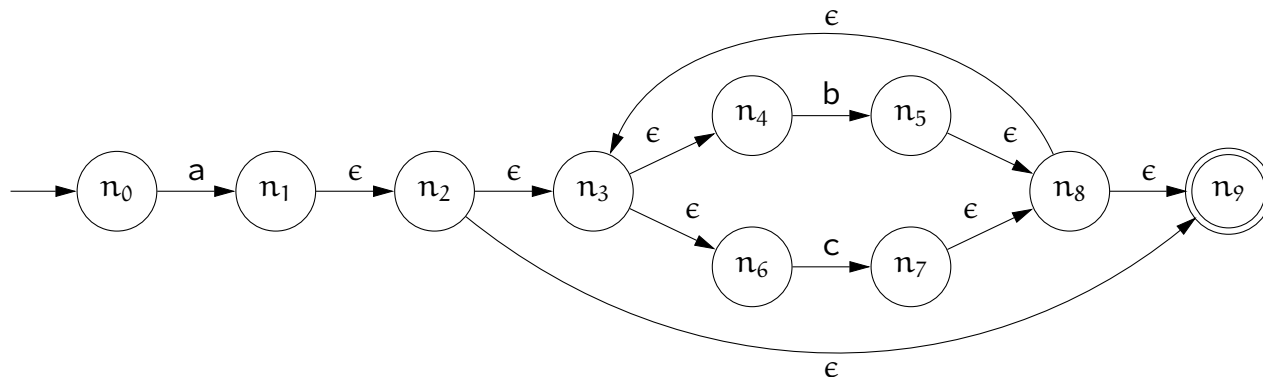
- each q represents a **valid configuration** of the original NFA
- the set Q grows **monotonically** and can become as large as $|2^N|$ distinct states
- when the algorithm **halts**...
 - Q contains **all** of the valid configurations of the NFA
 - each $q_i \in Q$ corresponds to **one** state $d_i \in D$ in the DFA
 - T holds **all** of the transitions between them

Building the DFA from Q and T is therefore straightforward

- each $q_i \in Q$ needs a state $d_i \in D$ to represent it
- if q_i contains an accepting state of the NFA, then d_i is an accepting state of the DFA
- the transition function δ_D is defined by the mapping from q_i to d_i in T
- the state constructed from q_0 becomes d_0 , the initial state of the DFA

Example: Subset Construction

Consider the NFA built for $a(b \mid c)^*$ and shown below with its states renumbered.



Example: Subset Construction

The algorithm takes the following steps

1. The initialization sets q_0 to $\epsilon\text{-closure}(\{n_0\})$, which is just n_0
2. The 1st iteration computes $\epsilon\text{-closure}(\text{Delta}(q_0, a))$, which contains six NFA states, and $\epsilon\text{-closure}(\text{Delta}(q_0, b))$ and $\epsilon\text{-closure}(\text{Delta}(q_0, c))$, which are empty
3. The 2nd iteration examines q_1 and produces two configurations, named q_2 and q_3
4. The 3rd iteration examines q_2 and constructs two configurations, which are identical to q_2 and q_3
5. The 4th iteration examines q_3 and it reconstructs q_2 and q_3 (like the 3rd iteration)

Example: Subset Construction

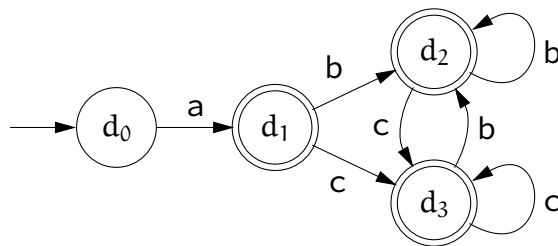
The following table sketches the steps that the subset construction algorithm follows.

Set Name	DFA States	NFA States	$\epsilon\text{-closure}(\text{Delta}(q, \circ))$		
			a	b	c
q_0	d_0	n_0	$\left\{ \begin{array}{l} n_1, n_2, n_3, \\ n_4, n_6, n_9 \end{array} \right\}$	\emptyset	\emptyset
q_1	d_1	$\left\{ \begin{array}{l} n_1, n_2, n_3, \\ n_4, n_6, n_9 \end{array} \right\}$	\emptyset	$\left\{ \begin{array}{l} n_5, n_8, n_9, \\ n_3, n_4, n_6 \end{array} \right\}$	$\left\{ \begin{array}{l} n_7, n_8, n_9, \\ n_3, n_4, n_6 \end{array} \right\}$
q_2	d_2	$\left\{ \begin{array}{l} n_5, n_8, n_9, \\ n_3, n_4, n_6 \end{array} \right\}$	\emptyset	q_2	q_3
q_3	d_3	$\left\{ \begin{array}{l} n_7, n_8, n_9, \\ n_3, n_4, n_6 \end{array} \right\}$	\emptyset	q_2	q_3

Example: Subset Construction

Finally, we can construct the resulting DFA as follows

- the states correspond to the DFA states from the table
- the transitions are given by the `DeLta` operations that generate those states
- since the sets q_1 , q_2 , and q_3 all contain n_9 (the accepting state of the NFA), all three become accepting states in the DFA



Digression: Fix-Point Computation

Subset construction is an example of a **fixed-point computation**, a style of computation that arises regularly in computer science.

Fix-Point Computation

- characterized by the iterated application of a **monotone function** to some collection of sets drawn from a domain whose structure is known
- computation terminates when they reach a state where further iteration produces the **same answer**, *i.e.*, if a “fixed point” is reached

→ Fixed-point computations play an important and recurring role in compiler construction

DFA to Minimal DFA: Hopcroft's Algorithm

The DFA that emerges from the subset construction can have a **large** set of states

- we can use an algorithm to minimize the number of states in the DFA
- this does not decrease the time needed to scan a string
- but it does decrease the size of the recognizer in memory

To minimize the number of states in a DFA $(D, \Sigma, \delta, d_0, D_A)$, we need an algorithm to detect if two states are **equivalent**, *i.e.*, if they produce the same behavior on any input string.

DFA to Minimal DFA: Hopcroft's Algorithm

```

1  $T \leftarrow \{D_A, \{D - D_A\}\}$ 
2  $P \leftarrow \emptyset$ 
3 while  $P \neq T$  do
4    $P \leftarrow T$ 
5    $T \leftarrow \emptyset$ 
6   foreach set  $p \in P$  do
7      $T \leftarrow T \cup \text{Split}(p)$ 

1 Split( $S$ )
2   foreach  $c \in \Sigma$  do
3     if  $c$  splits  $S$  into  $s_1$  and  $s_2$  then
4       return  $\{s_1, s_2\}$ 
5   return  $S$ 

```

The algorithm constructs a set partition $P = \{p_1, p_2, \dots, p_m\}$ of the DFA states by grouping together DFA states that have the **same behavior**.

Two DFA states $d_i, d_j \in p_s$ have the same behavior in response to all input characters

$$\forall p_s \in P; d_i, d_j \in p_s; c \in \Sigma : d_i \xrightarrow{c} d_x, d_j \xrightarrow{c} d_y \implies d_x = d_y$$

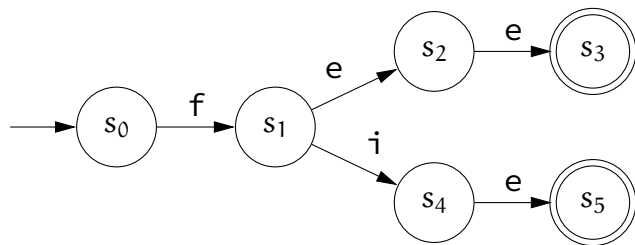
DFA to Minimal DFA: Hopcroft's Algorithm

Observations

- To minimize a DFA, each set $p_s \in P$ should be as large as possible, within the constraint of behavioral equivalence
- The initial partition contains two sets: $p_0 \in D_A$ and $p_1 \in \{D - D_A\}$
- Since the algorithm never combines two partitions, this separation ensures that no set in the final partition contains both accepting and nonaccepting states
- To construct the new DFA from the final partition P , we
 - create a single state to represent each set $p \in P$
 - add the appropriate transitions between these new representative states
- The resulting DFA is minimal (the proof is beyond our scope)
- This algorithm is another example of a fixed-point computation
- The worst-case behavior occurs when each state in the DFA has different behavior

Example: Hopcroft's Algorithm

As a first example, consider a DFA that recognizes the language $fee \mid fie$, shown below.



By inspection, we can see that states s_3 and s_5 serve the same purpose

- both are accepting states entered only by a transition on the letter e
- neither has a transition that leaves the state

→ We would expect the DFA minimization algorithm to discover this fact and replace them with a **single state**.

Example: Hopcroft's Algorithm

The algorithm takes the following steps

1. Initialization separates accepting from nonaccepting states: $\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$
2. Step 1 examines $\{s_3, s_5\}$: neither state has an exiting transition, *i.e.*, no split occurs
3. Step 2 examines $\{s_0, s_1, s_2, s_4\}$: on character *e*, it splits $\{s_2, s_4\}$ out of the set
4. Step 3 examines $\{s_0, s_1\}$: on character *f*, it splits $\{s_1\}$ out of the set
5. Step 4 makes a final pass over the current partition $\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$: no more splits occur and, therefore, the fix-point is reached

Note We assume that the various loops in the algorithm iterate over the sets of P and over the characters in $\Sigma = \{e, f, i\}$ in order.

Example: Hopcroft's Algorithm

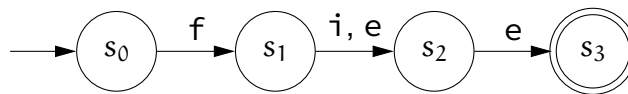
The following summarizes the significant steps that occur in minimizing this DFA.

Step	Current Partition	Set	Examines Char	Action
0	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	—	—	—
1	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_3, s_5\}$	<i>all</i>	<i>none</i>
2	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_0, s_1, s_2, s_4\}$	e	<i>split</i> $\{s_2, s_4\}$
3	$\{\{s_3, s_5\}, \{s_0, s_1\}, \{s_2, s_4\}\}$	$\{s_0, s_1\}$	f	<i>split</i> $\{s_1\}$
4	$\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$	<i>all</i>	<i>all</i>	<i>none</i>

Example: Hopcroft's Algorithm

To construct the new DFA, we must build...

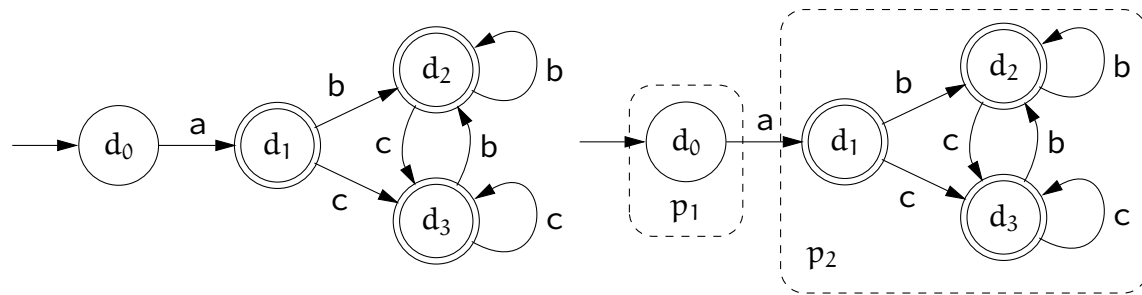
- a state to represent each set in the final partition,
- the appropriate transitions from the original DFA, and
- designate initial and accepting state(s)



Final DFA with renumbered states

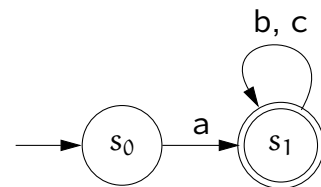
Example: Hopcroft's Algorithm

As a second example, consider the DFA for $a(b \mid c)^*$ that we built earlier using Thompson's construction and the subset construction.



Original DFA

Initial Partition



Minimal DFA

Using a DFA as a Recognizer

We now have the techniques to construct a DFA implementation from a **single** RE. But we need a **recognizer** that can handle **all** the REs for the language's microsyntax.

Given the REs for the various syntactic categories, $r_1, r_2, r_3, \dots, r_k$, we can construct a single RE for the entire collection by forming $(r_1|r_2|r_3|\dots|r_k)$.

However, since most real programs contain more than one word, we need to transform either the language or the recognizer.

- **at the language level**, we can insist that each word ends with an recognizable delimiter, *e.g.*, a blank or a tab
 - **at the recognizer level**, we can change the implementation of the DFA and its notion of acceptance
- As the first option is cumbersome, *e.g.*, “1+2” would not be recognized, whereas “1 + 2” would, we focus on the second option.

Using a DFA as a Recognizer

To find the **longest word** that matches one of the REs, the DFA should run until it reaches the point where the current state s has no outgoing transition on the next character.

Which RE was matched at this point?

1. if s is an **accepting state**: report the recognized word and its syntactic category
2. if s is an **nonaccepting state**
 - if possible, roll back to most recent accepting state to match **longest valid prefix**
 - otherwise, report an **error** since no prefix of the input string is a valid word

Note An accepting state in the DFA may represent several accepting states of the NFA. A keyword such as `new` might match two REs: one for keywords and one for identifiers.

→ A recognizer must have a strategy to break such ties, e.g., prioritize REs.

Using a DFA as a Recognizer

A compiler writer must also specify REs for parts of the input stream that do not form words in the program text.

Whitespace

- include an RE that matches blanks, tabs, end-of-line characters, *etc.*
- action on accepting whitespace: invoke the scanner, recursively, and return its result

Comments

- if comments are discarded, they are handled in a similar fashion as whitespace

Implementing Scanners

Scanner construction is a problem where the theory of formal languages has produced tools, so-called **scanner generators**, that can automate implementation.

Scanner generator

1. creates NFA for each RE defined by compiler writer for syntactic classes
2. joins all resulting NFAs by ϵ -transitions
3. translates NFA into a corresponding DFA
4. minimizes DFA
5. converts DFA into executable code

Having covered Steps 1 to 4 already, we now look at three implementation strategies for Step 5: **table-driven scanners**, **direct-coded scanners**, and **hand-coded scanners**.

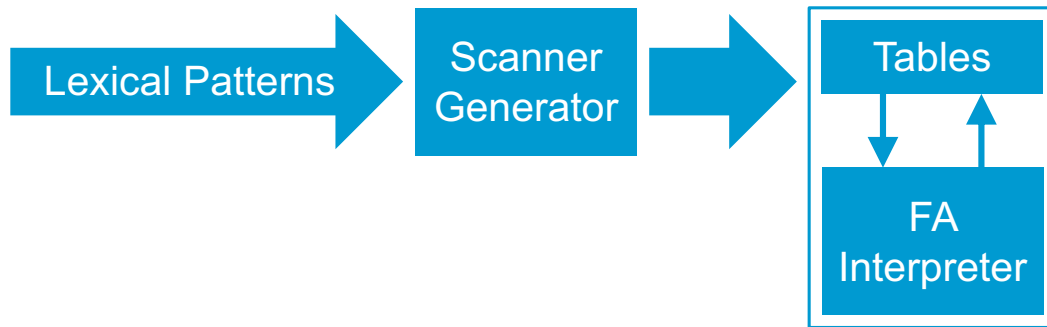
Implementing Scanners

The three implementation strategies, table-driven, direct-coded, and hand-coded, differ in

- how they encode the DFA's transition structure
- how they simulate its operation

Those differences, in turn, produce **different runtime costs**. However, they all have the **same asymptotic complexity**—constant cost per character, plus the cost of roll back.

Table-Driven Scanners

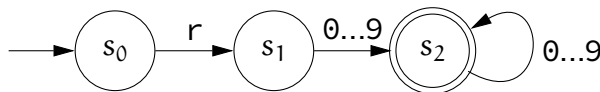


The table-driven approach uses a generic **skeleton scanner** for control and a set of **three generated tables** that encode language-specific knowledge

- classifier table CharCat maps each $c \in \Sigma$ to a category
- transition table δ encodes the transitions of the underlying DFA
- token type table Type maps each state s_i of the DFA to a token type

Table-Driven Scanners

As an example, we study the table-driven scanner for the RE $r[0\dots9]^+$, which is recognized by the DFA given below.



The code generated for the skeleton scanner, *i.e.*, algorithm `NextWord()` on Slide 103, is divided into **four** sections.

- initializations
- a scanning loop that models the DFA's behavior
- a roll back loop in case the dfa overshoots the end of the token
- a final section that interprets and reports the results

Table-Driven Scanners

Algorithm NextWord()

```

1 state  $\leftarrow s_0$ 
2 lexeme  $\leftarrow ""$ 
3 clear stack
4 push(bad)
5 while state  $\neq s_e$  do
6   NextChar(char)
7   lexeme  $\leftarrow$  lexeme + char
8   if state  $\in S_A$  then
9     clear stack
10  push(state)
11  cat  $\leftarrow$  CharCat[char]
12  state  $\leftarrow \delta[\text{state}, \text{cat}]$ 

```

```

13 while state  $\notin S_A \wedge$  state  $\neq \textit{bad}$  do
14   state  $\leftarrow$  pop()
15   truncate lexeme
16   RollBack()
17 if state  $\in S_A$  then
18   return Type[state];
19 else
20   return invalid

```

Notice the similarity between this code and the fragment shown on Slide 57.

Table-Driven Scanners

Tables CharCat and δ encode the DFA. To update the state, a **two-step translation** is performed.

1. character \rightarrow category
2. current state and category \rightarrow new state

This approach lets the scanner use a compressed transition table.

r	0, ..., 9	EOF	Other
Register	Digit	Digit	Other

Classifier Table CharCat

Larger character sets, e.g., Unicode, may need a more complex data structure to represent CharCat.

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Transition Table δ

s_0	s_1	s_2	s_e
invalid	invalid	register	invalid

Token Type Table Type

Table-Driven Scanners

In the algorithm shown on Slide 103, some regular expressions can produce **quadratic calls** to roll back.

Example

Consider a scanner built using our approach for the RE $ab \mid (ab)^* c$.

ababababc Scanner reads all characters and returns the entire string as one single word.

abababab Scanner must scan all characters before it can determine that the longest prefix is *ab*. On the next invocation, it will scan *ababab* to return *ab*, *etc*

Table-Driven Scanners

The so-called **Maximal Munch Scanner** avoids excess roll back by marking dead-end transitions as they are popped from the stack.

It differs from the scanner on Slide 103 in **three** important ways.

1. a global counter `InputPos` to record position in the input stream
2. a two-dimensional bit-array `Failed` to record dead-end transitions
3. additional routine `InitializeScanner()` to initialize these two variables

→ Over time, the maximal munch scanner records specific $\langle \text{state}, \text{input position} \rangle$ -pairs that cannot lead to an accepting state.

Direct-Coded Scanners

What could we do to **improve the performance** of a table-driven scanner?

- read a character
- compute the next DFA transition

Direct-coded scanners reduce the cost of computing DFA transitions by replacing the explicit representation of the DFA's state and transition graph with an implicit one.

- simplifies the two-step, table-lookup computation
- eliminates the memory references entailed in that computation
- allows other specializations

→ The resulting scanner has the same functionality as the table-driven scanner, but with a lower overhead per character.

Direct-Coded Scanners

At the heart of a direct-coded scanner is an alternate implementation of the central `while` loop of the table-driven scanner.

```

1 while state  $\neq$   $s_e$  do
2   ...
3   cat  $\leftarrow$  CharCat[char]
4   state  $\leftarrow$   $\delta$ [state, cat]

```

For each character, the table-driven scanner does **two** table lookups, involving address computations.

- **CharCat**: $@CharCat_0 + i \times w$
- **δ** : $\delta_0 + (state \times |columns\ in\ \delta| + cat) \times w$

Even though both lookups take $O(1)$ time, they impose **constant-cost overheads** that a direct-coded scanner can avoid.

→ Rather than representing the current DFA state and the transition diagram explicitly, a direct-coded scanner has a specialized code fragment to implement each state.

Direct-Coded Scanners

```

// State  $s_{init}$ 
1 lexeme  $\leftarrow$  ""
2 clear stack
3 push(bad)
4 goto 5
// State  $s_0$ 
5 NextChar(char)
6 lexeme  $\leftarrow$  lexeme + char
7 if state  $\in S_A$  then
8   | clear stack
9   push(state)
10 if char = 'r' then
11   | goto 14
12 else
13   | goto 32

```

```

// State  $s_1$ 
14 NextChar(char)
15 lexeme  $\leftarrow$  lexeme + char
16 if state  $\in S_A$  then
17   | clear stack
18   push(state)
19 if '0'  $\leq$  char  $\leq$  '9' then
20   | goto 23
21 else
22   | goto 32
// State  $s_2$ 
23 NextChar(char)
24 lexeme  $\leftarrow$  lexeme + char
25 if state  $\in S_A$  then
26   | clear stack

```

```

27 push(state)
28 if '0'  $\leq$  char  $\leq$  '9' then
29   | goto 23
30 else
31   | goto 32
// State  $s_{out}$ 
32 while state  $\notin S_A \wedge$ 
    state  $\neq$  bad do
33   | state  $\leftarrow$  pop()
34   | truncate lexeme
35   | RollBack()
36 if state  $\in S_A$  then
37   | return Type[state];
38 else
39   | return invalid

```

Direct-Coded Scanner

Observations

- Complicated address computations¹ are replaced by simple jumps (goto)
- Scanner can be further optimized
 - stack is not needed, since there is only one accepting state
 - jumps from s_0 and s_1 to s_{out} can be replaced by **return invalid**
 - stack is only needed in a DFA that has a transition from an accepting to a nonaccepting states
- Since all states follow a similar pattern, *i.e.*, assignments followed by branching logic, scanner code can easily be generated
 - unlike the table-driven scanner, the code changes for each set of REs
 - of course, generated code violates many precepts of structured programming

¹Omitting these computations also reduces register demand behind the scenes.

Hand-Coded Scanners

Many compilers use hand-coded scanners. A hand-coded scanner can further reduce the overhead of the interfaces between the scanner and the rest of the system.

In the following, we study two possible optimizations

- buffering the input stream
- generating lexemes on-the-fly

The Real World

Many popular open-source compilers rely on hand-coded scanners. For example, the *flex* scanner generator was ostensibly built to support the *gcc* project, but *gcc 4.0* uses hand-coded scanners in several of its front ends.

Hand-Coded Scanners

In order to reduce the I/O cost per character, we can use buffered I/O

- each read operation returns a longer string of characters or buffer
- the scanner then indexes through the buffer by maintaining a pointer into the buffer
- `NextChar` fills the buffer and tracks the current location in the buffer

The cost of reading a full buffer of characters has two components

- a large fixed overhead
- and a small per-character cost

A buffer and pointer scheme **amortizes** the fixed costs of the read over many single-character fetches.

Hand-Coded Scanners

The code shown for the table-driven and direct-coded scanners accumulated the input characters into a string `lexeme`.

However, for some words, the lexeme...

- is implicit and we can avoid building it
- is not a string, but a number or Boolean value

A more efficient way than accumulating the lexeme as a string and then to convert or discard it, is to build the appropriate lexeme one character at a time, if required at all.

Hand-Coded Scanner

Example

In the direct-coded scanner on Slide 109, we could insert the following line (before the goto 23 statement) to compute the register number on-the-fly.

$$\text{RegNum} \leftarrow \text{RegNum} \times 10 + (\text{char} - '0')$$

Note In the same way, we could hand-code states that recognize comments to avoid accumulating the lexeme altogether.

Handling Keywords

So far, we have assumed that keywords in the input language are recognized by including **explicit REs** for them in the description that generates the DFA and the recognizer.

Another approach is to recognize keywords as identifiers and then check a **lookup table** (e.g., hash table using *perfect hashing*) containing all keywords.

Trade-offs

- lookup table can lead to bad performance for nonkeywords, if it performs retry on miss
- extra states for keywords in DFA consume memory, but do not impact compile time

While the lookup table makes sense in the context of a completely hand-implemented scanner, in general it is better to fold keyword recognition into the DFA.