2 Scanners

Contents of this module

1. Recognizing words
2. Regular expressions
3. From regular expression to scanner
    - Thompson's construction
    - Subset construction
    - Hopcroft's minimization algorithm
4. Implementing scanners
    - Table-driven scanners
    - Direct-coded scanners
    - Hand-coded scanners

Scanner

The scanner, or lexical analyzer, reads a stream of characters and produces a stream of words or tokens

- applies rules that describe the **lexical structure** of the input programming language
- assigns each valid word a **syntactic category**, or “part of speech”

Scanners can be implemented very efficiently

- **scanner generator**: tool that processes mathematical description of language’s lexical syntax to produce a fast recognizer
- **hand-crafted scanner**: can be faster by avoiding a portion of the overhead that cannot be avoided in a generated scanner

Both types of scanners can be implemented to require just $O(1)$ time per character, *i.e.*, they run in time proportional to the number of characters in the input stream.

Microsyntax

The lexical structure of a language is also called the **microsyntax** of a programming language

- specifies how to **group** characters into words
- specifies how to **separate** words that run together

The set of valid words is typically specified by rules rather than by enumeration in a dictionary.

Keywords

- a word that is reserved for a particular syntactic purpose
- cannot be used as an identifier

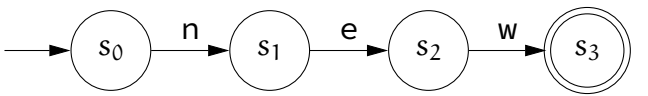
To recognize keywords, the scanner can either use dictionary lookup or encode the keywords directly into its microsyntax rules.

Recognizing Words

Consider the problem of recognizing the keyword `new`. We can formulate an algorithm that processes the input stream character by character.

```
1  c ← NextChar()
2  if c = 'n' then
3      c ← NextChar()
4      if c = 'e' then
5          c ← NextChar()
6          if c = 'w' then
7              return success
8          else
9              try something
                else
10     else
11         try something else
12 else
13     try something else
```

This code fragment can be represented using a simple **transition diagram**



The transition diagram represents a **recognizer**

- circles represent abstract **states** in the computation
- s_0 is the **initial state**, or start state
- s_3 is an **accepting state**, drawn with double circles

Recognizing Words

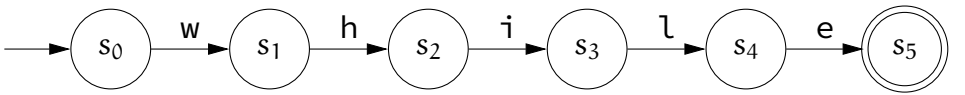
What happens to any other input that is **not** the keyword `new`, *e.g.*, `not`?

- the `n` takes the recognizer into state s_1, but the `o` does not match the edge leaving s_1
- in the code, `if` cases that do not match `new` use an `else` branch to *try something else*
- in the recognizer, this action can be represented as a transition to an **error state**.

Note When we draw the transition diagram of a recognizer, we usually omit transitions to the error state. Each state has a transition to the error state on each unspecified input.

Recognizing Words

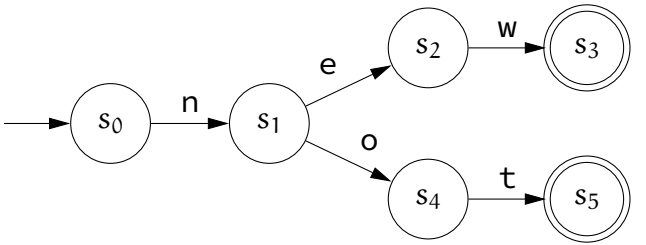
Using the same approach to build a recognizer for the keyword `while` would produce the following transition diagram.



Translating this recognizer to code would involve **five** nested `if-then-else` statements.

Recognizing Words

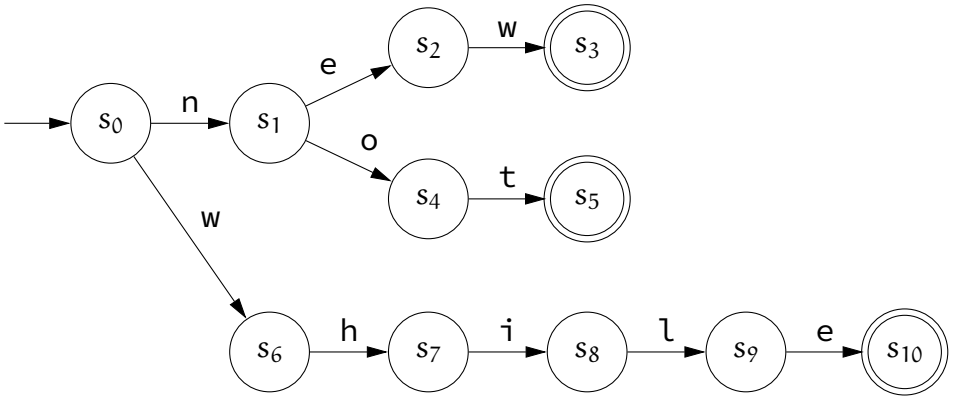
To recognize multiple words such as `new` and `not`, we can create **multiple edges** that leave a given state.



To translate this recognizer to code, we need to elaborate the *try something else* paths.

Recognizing Words

We can combine the recognizer for `new` or `not` with the one for `while` by **merging** their initial states and **relabeling** all the states.



The recognizer has three accepting states, s_3, s_5, and s_{10}. If any state encounters an input character that does not match one of its transitions, the recognizer moves to an error state.

A Formalism for Recognizers

Towards formalizing recognizers…

- transition diagrams are a concise **abstraction** of the code that would be required to implement the corresponding recognizer
- they can also be described as formal mathematical objects, called **finite automata**

In the following, we will use finite automata as a **formalism** to specify recognizers.

A Formalism for Recognizers

A **Finite Automaton** (FA) is a five-tuple $(S, \Sigma, \delta, s_0, S_A)$, where

- S is the finite set of states in the recognizer, along with an error state s_e.
- Σ is the finite alphabet used by the recognizer. Typically, Σ is the union of the edge labels in the transition diagram.
- $\delta(s, c)$ is the recognizer's transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition $s_i \xrightarrow{c} \delta(s_i, c)$.
- $s_0 \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

A Formalism for Recognizers

Example

The transition diagram of the recognizer for `new`, `not`, and `while` can be formalized as the following finite automaton.

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$$

$$\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{\mathtt{n}} s_1, s_0 \xrightarrow{\mathtt{w}} s_6, s_1 \xrightarrow{\mathtt{e}} s_2, s_1 \xrightarrow{\mathtt{o}} s_4, s_2 \xrightarrow{\mathtt{w}} s_3, \\ s_4 \xrightarrow{\mathtt{t}} s_5, s_6 \xrightarrow{\mathtt{h}} s_7, s_7 \xrightarrow{\mathtt{i}} s_8, s_8 \xrightarrow{\mathtt{l}} s_9, s_9 \xrightarrow{\mathtt{e}} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

For all other combinations of state s_i and input character c, we define $\delta(s_i, c) = s_e$, where s_e is the designated error state.

A Formalism for Recognizers

A FA accepts a string x if and only if, starting in s_0, the sequence of characters takes the FA through a transition sequence that leaves it in an accepting state when the entire string has been consumed.

Examples

`new` $s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{e} s_2, s_2 \xrightarrow{w} s_3$

$\longrightarrow$ the FA **accepts** the word since $s_3 \in S_A$

`nut` $s_0 \xrightarrow{n} s_1, s_1 \xrightarrow{u} s_e$

$\longrightarrow$ once the FA enters s_e, it stays in s_e until it exhausts the input stream

A Formalism for Recognizers

More formally, if the string x is composed of characters $x_1x_2x_3 \ldots x_n$, then the FA $(S, \Sigma, \delta, s_0, S_A)$ accepts x if and only if

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3)\ldots, x_{n-1}), x_n) \in S_A.$$

Apart from ending up in an accepting state, two other cases are possible.

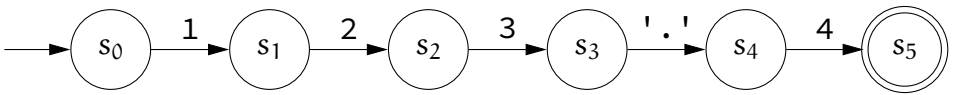
- a character x_j takes the FA into the error state s_e: this indicates a **lexical error**, *i.e.*, $x_1x_2x_3 \ldots x_j$ is not a valid prefix for any word in the accepted language
- the FA terminates in a non-accepting state other than s_e after exhausting its input: this also indicates an **error**, even though the input is a valid prefix of some accepted word

Note In any case, the FA takes one transition for each input character. Implemented efficiently, we expect the recognizer to run in time proportional to the length of the input.

Recognizing More Complex Words

The character-by-character model used so far can easily be extended to handle arbitrary collections of fully specified words. Can we use the same approach to recognize numbers?

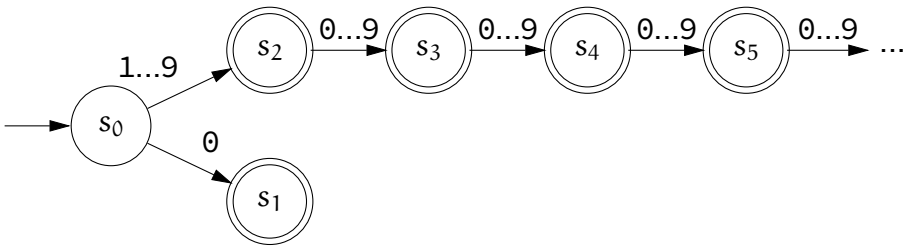
A **specific number**, such as 123.4, is straightforward.



But how would we draw a transition diagram (and design the corresponding code fragment) that can recognize **any number**, for example, unsigned integers?

Recognizing More Complex Words

An **unsigned integer** is either zero, or it is a series of one or more digits, where the first digit is from one to nine, and the subsequent digits are from zero to nine.



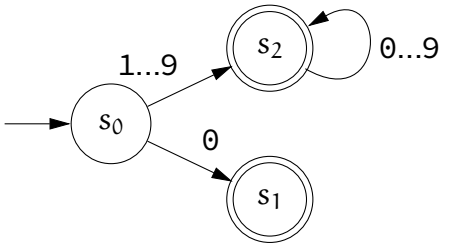
Problems

- path from s_0 to s_2, to s_3, and so on does not end, *i.e.*, this is not a **finite** automaton
- states $s_2, s_3, \ldots$ are all **equivalent**, *i.e.*, accepting states with same outgoing labels

Recognizing More Complex Words

We can fix both of these problems by allowing the transition diagram to have **cycles**

→ replace entire chain of states beginning at s_2 with a single transition from s_2 back to itself



Note This FA recognizes a class of strings with a common property, *i.e.*, unsigned integer

- class “unsigned integer” is a **syntactic category**
- text of a specific unsigned integer is its **lexeme**

$$S = \{s_0, s_1, s_2\}$$
$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$
$$\delta = \left\{ \begin{array}{l} s_0 \xrightarrow{0} s_1, s_0 \xrightarrow{1-9} s_2, \\ s_2 \xrightarrow{0-9} s_2, s_1 \xrightarrow{0-9} s_e \end{array} \right\}$$
$$s_0 = s_0$$
$$S_A = \{s_1, s_2\}$$

Recognizing More Complex Words

The introduction of cycles in the transition graph creates the need for **cyclic control flow**

- can be implemented with a `while`-loop
- δ can be specified efficiently as a table

δ	0	1	2	3	4	5	6	7	8	9	**other**
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

```
1 c ← NextChar()
2 s ← s_0
3 while c ≠ eof ∧ s ≠ s_e do
4 |   s ← δ(s, c)
5 |   c ← NextChar()
6 if s ∈ S_A then
7 |   report acceptance
8 else
9 |   report failure
```

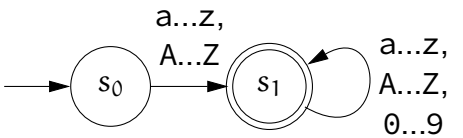
Note The last row of the δ-table can be omitted, since failure is reported as soon as the FA enters the error state s_e.

Recognizing More Complex Words

Using the same technique, we can develop FAs to recognize strings from other syntactic categories, such a signed integers, real numbers, and complex numbers.

An important syntactic category are **identifier names**

→ an alphabetic character followed by zero or more alphanumeric characters



Note Most programming languages extend the notion of “alphabetic character” to include designated special characters, such as the underscore.

Regular Expressions

To simplify and automate scanner implementation, we need…

- a concise notation for specifying the lexical structure of words,
- a way of turning those specifications into an FA, and
- a mechanism to produce code that implements the FA.

In the following, we look at these three problems in turn, beginning with the first.

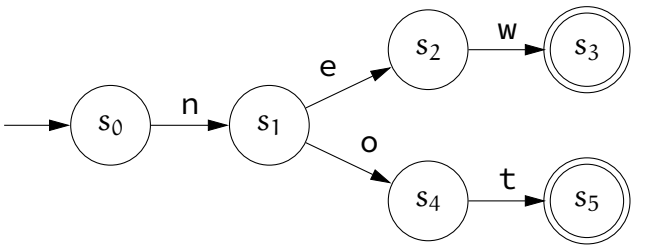
Note The set of words accepted by a FA, $\mathcal{F}$, forms a language, denoted $\mathrm{L}(\mathcal{F})$. We can also describe its language using a notation called a **regular expression** (RE). The language described by an RE is called a **regular language**.

Regular Expressions

Examples

- The language consisting of the single word `new` is described by an RE written as *new*
- The language consisting of the two words `new` or `while` is written as *new* | *while*
- The language consisting of `new` or `not` can be written as *new* | *not* or as *n (ew* | *ot)*

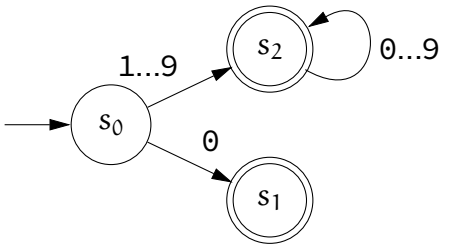
The RE *n (ew* | *ot)* suggests the structure of the FA that we drew earlier to recognize these two words.



Regular Expressions

Recall the FA that we developed for unsigned integers

- uses a cycle to recognize arbitrary integers of any length, other than `0`
- to write it as a RE, we need a notation for this notion of “zero or more occurrences”



For a RE χ, the meaning “zero or more occurrences of χ” is written a χ^*. We call the $*$ operator **Kleene closure**, or **closure** for short.

Example Using the closure operator, we can write an RE for this FA:

*0 | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)**

Regular Expressions

An **Regular Expression** (RE) describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ϵ that represents the empty string. For a given RE, r, we denote the language that it specifies as $L(r)$. A RE is built up from three basic operations:

1. The **alternation**, or union, of two sets of strings, R and S, denoted $R \,|\, S$, is $\{x \,|\, x \in R \vee x \in S\}$.
2. The **concatenation** of two sets R and S, denoted RS, contains all strings formed by prepending an element of R onto one from S, or $\{xy \,|\, x \in R \wedge y \in S\}$.
3. The **Kleene closure** of a set R, denoted R^*, is $\bigcup_{i=0}^{\infty} R^i$. This is just the union of the concatenations of R with itself, zero or more times.

Regular Expressions

For convenience, we introduce the following notation (syntactic sugar).

- **Finite Closure** R^i: one to i occurrences of R, *e.g.*, $R^3 = (R \mid RR \mid RRR)$
- **Positive Closure** R^+: one or more occurrences of R, *i.e.*, RR^*
- **Ranges** $[x_0 \ldots x_n]$: abbreviate ranges *e.g.*, $[0 \ldots 3] = (0 \mid 1 \mid 2 \mid 3)$
- **Complement** $\hat{}c$: complement of c with respect to Σ, *i.e.*, $\{\Sigma - c\}$

Note To eliminate any ambiguity, parentheses have highest precedence, followed by complement, closure, concatenation, and alternation, in that order.

Regular Expressions

Exercise

Design a RE to match Java-style multi-line comments. Can you think of a FA that would implement this RE?

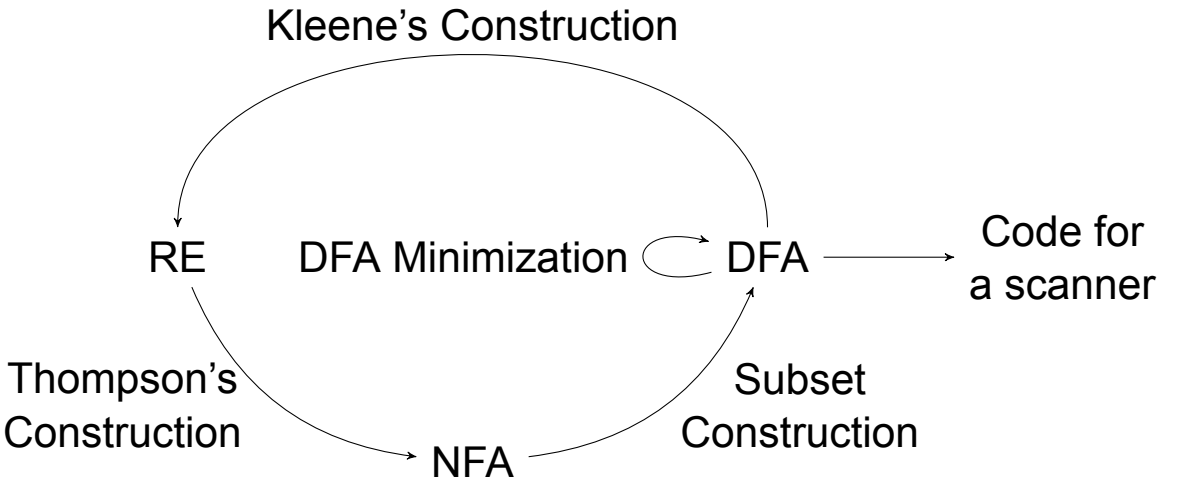
Regular Expressions

Regular expressions are **closed** under many operations. These closure properties play a critical role in the use of REs to build scanners.

- Closure under **union** implies that any finite language is a regular language: we can construct an RE for any finite collection of words by listing them in a large alternation.
- Closure under **concatenation** allows us to build complex res from simpler ones by concatenating them.
- Closure under both **Kleene closure** and the finite closures: we can specify particular kinds of large, or even infinite, sets with finite patterns.

The fact that REs are closed under alternation, concatenation, and closure is critical to the constructions that we will look at in the following.

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

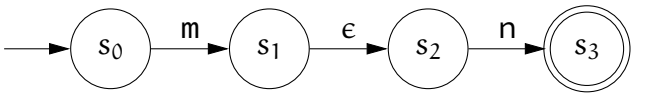
Nondeterministic Finite Automata

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.

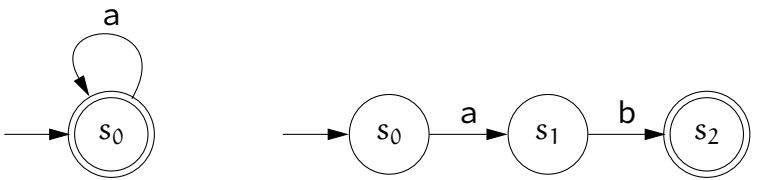
We can use transitions on ϵ to combine FAs and form FAs for more complex REs.

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.

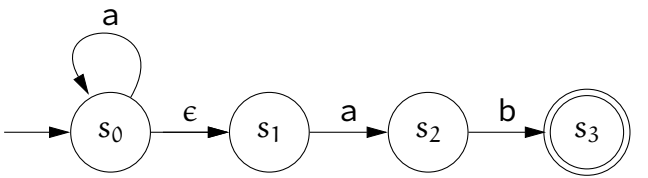


Nondeterministic Finite Automata

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

$\longrightarrow$ The correct transition out of s_0 depends on the characters that follow a.

Nondeteministic 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

$\rightarrow$ any DFA is a special case of an NFA, *i.e.*, an NFA is at least as powerful as a DFA

$\leftarrow$ any NFA can be simulated by a DFA by subset construction (*cf.* Slide 80)

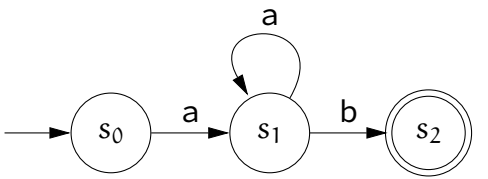
The intuition behind **subset construction** is simple: to simulate the behavior of the NFA, we need a DFA with a state for each configuration of the NFA.

- 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*.



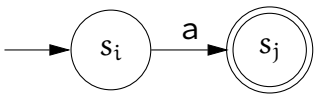
It relies on the observation that *a*ab* specifies the same regular language as *aa*b*.

From RE to NFA: Thompson's Construction

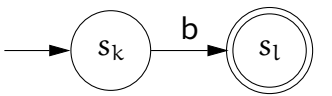
As a first step in moving from an RE to an implemented scanner, we must derive an NFA from the RE.

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

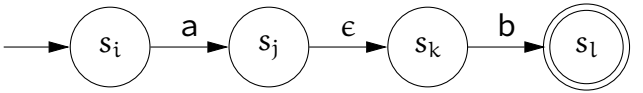


NFA for *a*

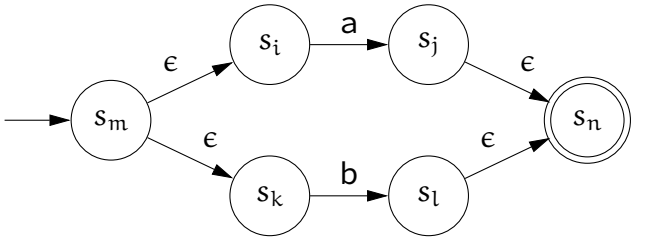


NFA for *b*

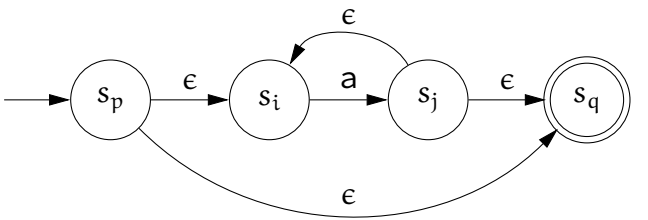
From RE to NFA: Thompson's Construction



NFA for *ab*



NFA for *a* | *b*

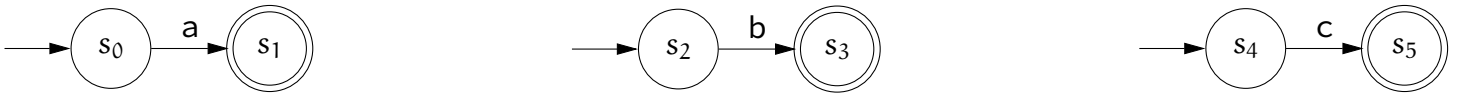


NFA for *a**

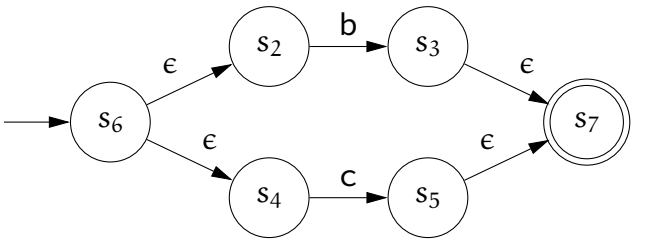
Example: Thompson's Construction

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

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

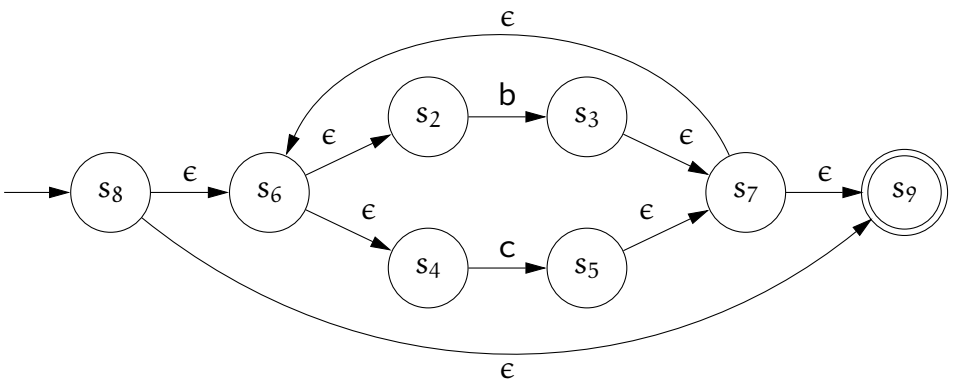


2. Since parentheses have highest precedence, it then builds the NFA for the expression enclosed in parentheses, *b* | *c*



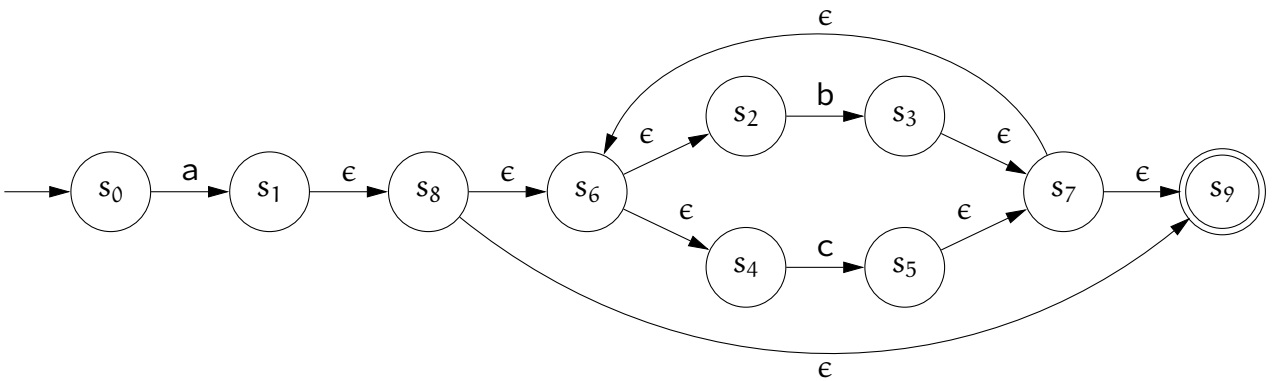
Example: Thompson's Construction

3. Since closure has higher precedence than concatenation, the construction next builds the closure, *(b | c)**



Example: Thompson's Construction

4. Finally, it concatenates the NFA for *a* to the NFA for *(b | c)**

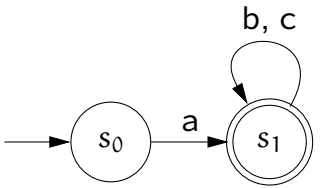


From RE to NFA: Thompson's Construction

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 convent 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  q0 ← ε-closure({n0})
2  Q ← q0
3  WorkList ← {q0}
4  while WorkList ≠ ∅ do
5      remove q from WorkList
6      foreach character c ∈ Σ do
7          t ← ε-closure(Delta(q,c))
8          T[q,c] ← t
9          if t ∉ 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 $q_i \in 2^N$

`ε-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

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

From NFA to DFA: Subset Construction

Notice

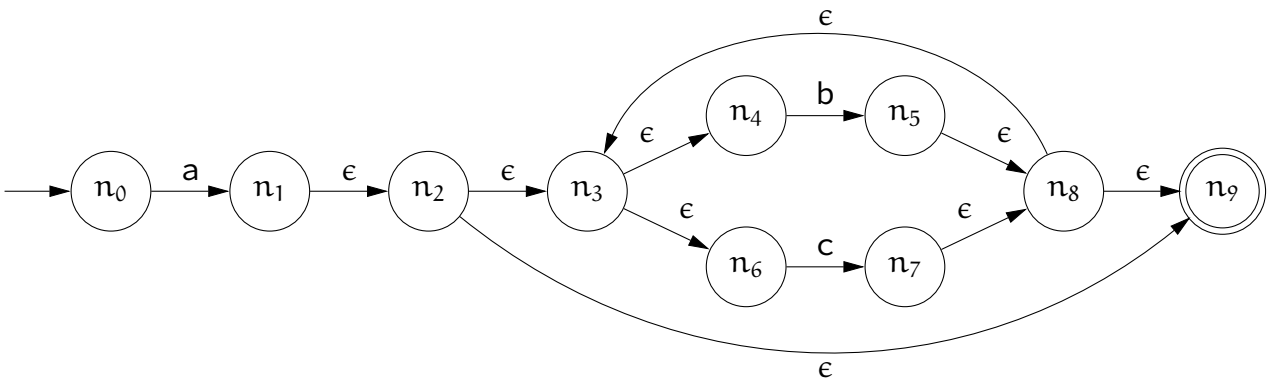
- each q represents a **valid configuration** of the original NFA
- the set Q grows **monotonically** and can become as large as $|2^N|$ distinct states
- when the algorithm **halts**…
  - Q contains **all** of the valid configurations of the NFA
  - each $q_i \in Q$ corresponds to **one** state $d_i \in D$ 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 | c)** and shown below with its states renumbered.



Example: Subset Construction

The algorithm takes the following steps

1. The initialization sets q_0 to $\epsilon\texttt{-closure}(\{n_0\})$, which is just n_0
2. The 1st iteration computes $\epsilon\texttt{-closure}(\texttt{Delta}(q_0, a))$, which contains six NFA states, and $\epsilon\texttt{-closure}(\texttt{Delta}(q_0, b))$ and $\epsilon\texttt{-closure}(\texttt{Delta}(q_0, c))$, 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

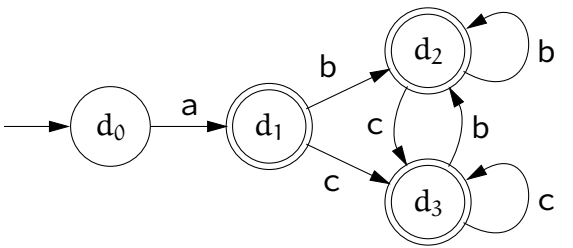
The following table sketches the steps that the subset construction algorithm follows.

Set Name	DFA States	NFA States	ϵ-**closure**(**Delta**(q, ○)) a	b	c
q_0	d_0	n_0	$\{ n_1, n_2, n_3, n_4, n_6, n_9 \}$	$\emptyset$	$\emptyset$
q_1	d_1	$\{ n_1, n_2, n_3, n_4, n_6, n_9 \}$	$\emptyset$	$\{ n_5, n_8, n_9, n_3, n_4, n_6 \}$	$\{ n_7, n_8, n_9, n_3, n_4, n_6 \}$
q_2	d_2	$\{ n_5, n_8, n_9, n_3, n_4, n_6 \}$	$\emptyset$	q_2	q_3
q_3	d_3	$\{ n_7, n_8, n_9, n_3, n_4, n_6 \}$	$\emptyset$	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 ← {D_A, {D − D_A}}
2 P ← ∅
3 while P ≠ T do
4     P ← T
5     T ← ∅
6     foreach set p ∈ P do
7         T ← T ∪ Split(p)
```

```
1 Split(S)
2     foreach c ∈ Σ 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 $P = \{p_1, p_2, \ldots, p_m\}$ 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

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

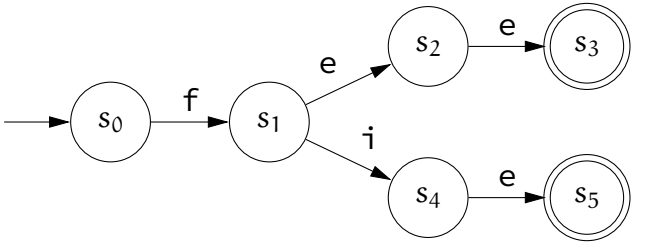
DFA to Minimal DFA: Hopcroft's Algorithm

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* | *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

$\longrightarrow$ 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 charachter `e`, it splits $\{s_2, s_4\}$ out of the set
4. Step 3 examines $\{s_0, s_1\}$: on charachter `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

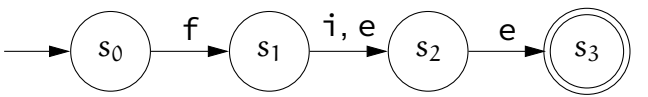
The following summarizes the significant steps that occur in minimizing this DFA.

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

Example: Hopcroft's Algorithm

To construct the new DFA, we must build…

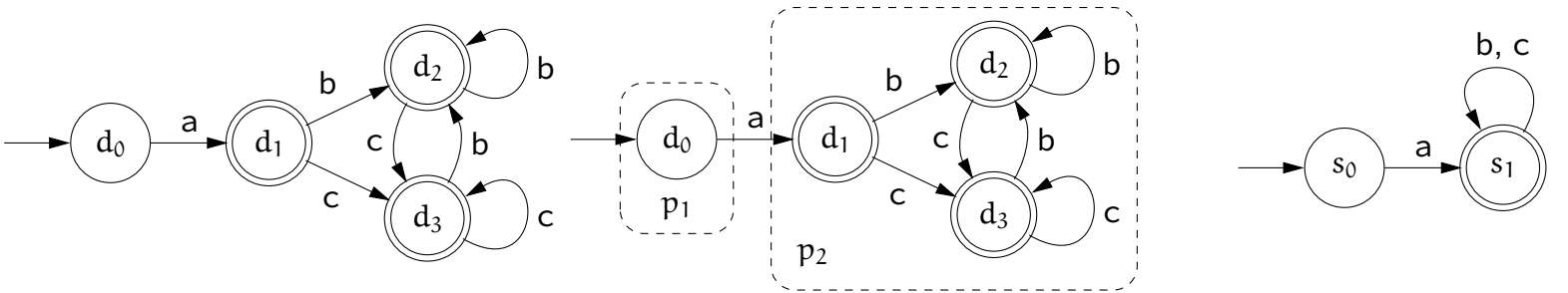
- a state to represent each set in the final partition,
- the appropriate transitions from the original DFA, and
- designate initial and accepting state(s)



Final DFA with renumbered states

Example: Hopcroft's Algorithm

As a second example, consider the DFA for *a (b | 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, $r_1, r_2, r_3, \ldots, r_k$, we can construct a single RE for the entire collection by forming $(r_1|r_2|r_3|\ldots|r_k)$.

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

$\longrightarrow$ 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.

$\longrightarrow$ 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**.

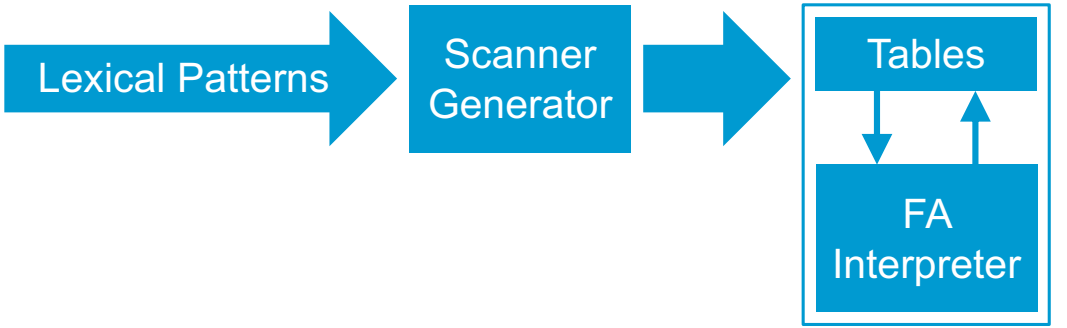
Implementing Scanners

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

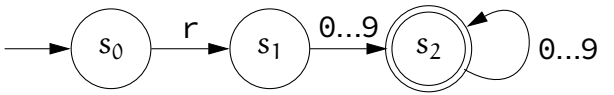


The table-driven approach uses a generic **skeleton scanner** for control and a set of **three generated tables** that encode language-specific knowledge

- 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…9]*+, 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 ← s_0
2  lexeme ← ""
3  clear stack
4  push(bad)
5  while state ≠ s_e do
6  |  NextChar(char)
7  |  lexeme ← lexeme + char
8  |  if state ∈ S_A then
9  |  |  clear stack
10 |  push(state)
11 |  cat ← CharCat[char]
12 |  state ← δ[state,cat]
13 while state ∉ S_A ∧ state ≠ bad do
14 |  state ← pop()
15 |  truncate lexeme
16 |  RollBack()
17 if state ∈ 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`

	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`

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

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* | *(ab)* c*.

`abababab c`	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

$\longrightarrow$ 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 ≠ s_e do
2 |   ...
3 |   cat ← CharCat[char]
4 |   state ← δ[state,cat]
```

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

- `CharCat`: $@\texttt{CharCat}_0 + \texttt{i} \times w$
- δ: $\delta_0 + (\texttt{state} \times |\text{columns in } \delta| + \texttt{cat}) \times w$

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

$\longrightarrow$ 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 ← ""
 2 clear stack
 3 push(bad)
 4 goto 5
   // State s_0
 5 NextChar(char)
 6 lexeme ← lexeme + char
 7 if state ∈ 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 ← lexeme + char
16 if state ∈ S_A then
17 |  clear stack
18 push(state)
19 if '0' ≤ char ≤ '9' then
20 |  goto 23
21 else
22 |  goto 32
   // State s_2
23 NextChar(char)
24 lexeme ← lexeme + char
25 if state ∈ S_A then
26 |  clear stack
27 push(state)
28 if '0' ≤ char ≤ '9' then
29 |  goto 23
30 else
31 |  goto 32
   // State s_out
32 while state ∉ S_A ∧
          state ≠ bad do
33 |  state ← pop()
34 |  truncate lexeme
35 |  RollBack()
36 if state ∈ S_A then
37 |  return Type[state];
38 else
39 |  return invalid
```

Direct-Coded Scanner

Observations

- Complicated address computations[1] 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

[1] Omitting 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.

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

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.