A scanner separates input into *tokens* based on lexical analysis.

Legal tokens are usually specified by regular expressions (REs).

Regular expressions specify *regular languages*. 

Advantages of regular languages

Regular languages can be recognized by *deterministic finite automata (dfa).*

Deterministic finite automata

- have simple implementations
- character based transitions are $O(1)$
- word recognition is $O(|\text{input}|)$

$\Rightarrow$ *dfas* can be fast

- fast implementations are easy
- asymptotic behavior is linear
- choices per state is irrelevant

$\Rightarrow$ *dfas* can be built automatically

- *Desiderata:* write specification, not code
- *Reality:* for scanners, this works
Limits of regular languages

Not all languages are regular.

You cannot construct \(dfa\)'s to recognize these languages:

- \(L = \{ p^k q^k \}\)
- \(L = \{ wcw'' \mid w \in \Sigma^* \}\)

Note: neither of these is a regular expression!
\((dfa\)'s cannot count!\)

But, this is a little subtle. You can construct \(dfa\)'s for:

- alternating 0’s and 1’s
  \((\epsilon \mid 1)(01)^*(\epsilon \mid 0)\)
- sets of pairs of 0’s and 1’s
  \((01 \mid 10)^+\)
More regular languages

Let’s look at another regular language — the set of strings containing an even number of zeros and an even number of ones

The regular expression is

$$(00 \mid 11)^* ((01 \mid 10)(00 \mid 11)^*(01 \mid 10)(00 \mid 11)^*)^*$$

(This is similar to parts of problem 3.6.)
Nondeterministic finite automata

What about the regular expression \((a \mid b)^*abb\)?

State Start has \(\epsilon\) transition to S1.
State S1 has multiple transitions on a!

\[ \Rightarrow \text{nondeterministic finite automaton (nfa)} \]

Different definition for accept

A \textit{nfa accepts} \(x\) if and only if there is some path through the transition graph from the start state to an accepting state such that the labels along the edges spell \(x\).
nfas versus dfas

What is the relationship between a nfa and a dfa?

dfa is special case of nfa

1. no ε transitions
2. single-valued transition function

dfa can be simulated on a nfa

- obviously

nfa can be simulated on a dfa

- simulate sets of simultaneous states
- possible exponential blowup
Constructing a \textit{dfa} from a regular expression

regular expression (RE) \(\rightarrow\) \textit{nfa} w/\(\epsilon\) moves

build \textit{nfa} for each term

connect them with \(\epsilon\) moves

\textit{nfa} w/\(\epsilon\) moves to \textit{nfa}

coalesce states

\textit{nfa} \(\rightarrow\) \textit{dfa}

construct the simulation

the “subset” construction

\textit{dfa} \(\rightarrow\) regular expression

construct \(R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}\)
Converting regular expressions to *nfas*

Build two-state automaton for atomic regular expression a, with a as the edge.

Compose automata as follows:

- Kleene closure

- concatenate

- union
Converting regular expressions to nfas (cont)

Apply subset construction algorithm

Input: \( nfa \ N \)
Output: A \( dfa \ D \) with \( D\text{states} \) and \( D\text{tran} \) that accepts the same language
Method: let \( s \) be a state in \( nfa \) and \( T \) a set of states, using the following definitions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon\text{-closure}(s) )</td>
<td>Set of ( nfa ) states reachable from ( nfa ) state ( s ) on ( \epsilon)-transitions alone.</td>
</tr>
<tr>
<td>( \epsilon\text{-closure}(T) )</td>
<td>Set of ( nfa ) states reachable from some ( nfa ) state ( s ) in ( T ) on ( \epsilon)-transitions alone.</td>
</tr>
<tr>
<td>( move(T,a) )</td>
<td>Set of ( nfa ) states to which there is a transition on input symbol ( a ) from some ( nfa ) state ( s ) in ( T ).</td>
</tr>
</tbody>
</table>
Subset construction (cont)

state $Start = \varepsilon$-closure($s_0$)
add $Start$ unmarked to $Dstates$

while $\exists$ an unmarked state $T$ in $Dstates$
    mark $T$
    for each input symbol $a$ do
        $U = \varepsilon$-closure(move($T,a$))
        if $U$ is not in $Dstates$ then
            add $U$ to $Dstates$ unmarked
            $Dtran[T,a] = U$
    endfor
endwhile

Each state in $D$ corresponds to a set of states in $N$.

Up to $2^{|N|}$ possible states in $D$.

$\varepsilon$-closure($s_0$) is the start state of $D$.

A state is an accepting state in $D$, if one or more of the states it represents in $N$ is accepting.
Building minimum-state dfas

Important theoretical result

Every regular language is recognized by a minimum-state dfa that is unique up to state names.

Look for states that can be distinguished from each other (i.e., end up in accepting/nonaccepting state for identical input).

*dfa* state minimization algorithm

- construct initial partition of states into accepting and non-accepting states
- successively refine partition by splitting a group $G$ into smaller groups if states in $G$ have transitions to different groups
- update transition edges, remove dead states

See proof of theorem 3.10, pages 67–71 in Hopcroft and Ullman's book

*Introduction to Automata Theory, Languages, and Computation*
Converting *dfas* to regular expressions

Method:

For a DFA $M = (\{s_0, \ldots, s_n\}, \Sigma, \delta, s_0, F)$

- Let $R_{i,j}^k$ denote the set of all strings $x$ such that $\delta(s_i, x) = s_j$ and if $y$ is a prefix of $x$ then $\delta(s_i, y) = s_l$, where $l \leq k$.
- Let $R_{i,j}$ be the set of all strings that take $M$ from state $s_i$ to state $s_j$ without going through a state $s_l$ where $l > k$
- “through $s_k$” means both entering and leaving $s_k$

Then, $\mathcal{L}(M) = \bigcup_{s_j \in F(M)} R_{0,j}^n$

More formally

(1) $R_{i,j}^k = R_{i,k}^{k-1}(R_{k,k}^{k-1})^*R_{k,j}^{k-1} \cup R_{i,j}^{k-1}$

(2) if $i \neq j$, $R_{i,j}^0 = \{a | \delta(s_i, a) = s_j\}$

(3) if $i = j$, $R_{i,j}^0 = \{a | \delta(s_i, a) = s_j\} \cup \{\epsilon\}$

See proof of Theorem 2.4, pages 33-34 in Hopcroft and Ullman’s book

*Introduction to Automata Theory, Languages, and Computation*
Summary

Scanners

- break up input into tokens
- catch lexical errors
- difficulty affected by language design

Issues

- input buffering
- lookahead
- error recovery

Scanner generators

- tokens specified by regular expressions
- construct DFA to recognize language
- highly efficient in practice