Lexical Analysis — Part II:
Constructing a Scanner from Regular Expressions
Quick Review

Previous class:

→ The scanner is the first stage in the front end
→ Specifications can be expressed using regular expressions
→ Build tables and code from a DFA
Goal

- We will show how to construct a finite state automaton to recognize any RE
- This Lecture
  → Convert RE to a **nondeterministic finite automaton (NFA)**
    - Requires $\epsilon$-**transitions** to combine regular subexpressions
  → Convert an NFA to a **deterministic finite automaton (DFA)**
    - Use Subset construction

Next Lecture

→ Minimize the number of states
  - Hopcroft state minimization algorithm
→ Generate the scanner code
  - Additional code can be inserted
More Regular Expressions

• All strings of 1s and 0s ending in a 1

\[(0 \mid 1)^* 1\]

• All strings over lowercase letters where the vowels (a,e,i,o,u) occur exactly once, in ascending order

\[\text{Cons} \rightarrow (b \mid c \mid d \mid f \mid g \mid h \mid j \mid k \mid l \mid m \mid n \mid p \mid q \mid r \mid s \mid t \mid v \mid w \mid x \mid y \mid z)\]

\[\text{Cons}^* \ a \ \text{Cons}^* \ e \ \text{Cons}^* \ i \ \text{Cons}^* \ o \ \text{Cons}^* \ u \ \text{Cons}^*\]

• All strings of 1s and 0s that do not contain three 0s in a row:
More Regular Expressions

• All strings of 1s and 0s ending in a 1

\[(0 | 1)^* 1\]

• All strings over lowercase letters where the vowels (a,e,i,o,u) occur exactly once, in ascending order

\[Cons \rightarrow (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)\]
\[Cons^* a Cons^* e Cons^* i Cons^* o Cons^* u Cons^*\]

• All strings of 1s and 0s that do not contain three 0s in a row:

\[(1^* (\varepsilon | 01 | 001 ) 1^*)^* (\varepsilon | 0 | 00 )\]
Non-deterministic Finite Automata

Each RE corresponds to a *deterministic finite automaton* (DFA)

- May be hard to directly construct the right DFA

What about an RE such as \(( a \mid b )^* \text{abb}\)?

This is a little different
- \(S_1\) has two transitions on \(a\)

This is a *non-deterministic finite automaton* (NFA)
Non-deterministic Finite Automata

Each RE corresponds to a *deterministic finite automaton* (DFA)
- May be hard to directly construct the right DFA

What about an RE such as \((a | b)^* \text{abb}\)?

\[
\begin{array}{c}
S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_2 \xrightarrow{b} S_3 \xrightarrow{b} S_4 \\
S_1, \quad S_2, \quad S_3, \quad S_4
\end{array}
\]

This is a little different
- \(S_1\) has two transitions on \(a\)
- \(S_0\) has a transition on \(\varepsilon\)

This is a *non-deterministic finite automaton* (NFA)
Nondeterministic Finite Automata

- An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$
- Transitions on $\varepsilon$ consume no input
- To “run” the NFA, start in $s_0$ and guess the right transition at each choice point with multiple possibilities
  - Always guess correctly
  - If some sequence of correct guesses accepts $x$ then accept

Why study NFAs?
- They are the key to automating the RE→DFA construction
- We can paste together NFAs with $\varepsilon$-transitions
Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no $\varepsilon$ transitions
- DFA’s transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

$\rightarrow$ Obviously

NFA can be simulated with a DFA

(less obvious)

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream
Automating Scanner Construction

To convert a specification into code:
1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators
- Lex, Flex, and JLex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser (define all parts of speech)
Automating Scanner Construction

**RE → NFA (Thompson’s construction)**
- Build an NFA for each term
- Combine them with $\varepsilon$-transitions

**NFA → DFA (subset construction)**
- Build the simulation

**DFA → Minimal DFA**
- Hopcroft’s algorithm

**DFA → RE (Not part of the scanner construction)**
- All pairs, all paths problem
- Take the union of all paths from $s_0$ to an accepting state
RE $\rightarrow$ NFA using Thompson’s Construction

Key idea

- NFA pattern for each symbol & each operator
- Join them with $\varepsilon$ transitions in precedence order

**Concatenation**

NFA for $ab$

**Closure**

NFA for $a^*$

**Alternation**

NFA for $a \mid b$

Ken Thompson, CACM, 1968
Example of Thompson’s Construction

Let’s try \(a (b \mid c)^*\)

1. \(a, b, \& c\)

2. \(b \mid c\)

3. \((b \mid c)^*\)
Example of Thompson’s Construction  (cont’d)

4. $a (b | c)^*$

Of course, a human would design something simpler ...

But, we can automate production of the more complex one ...
NFA $\rightarrow$ DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

- **$\text{Delta}(q_i, a)$** is set of states reachable from each state in $q_i$ by $a$
  
  $\rightarrow$ Returns a set of states, for each $n \in q_i$ of $\delta_i(n, a)$

- **$\varepsilon$-closure($s_i$)** is set of states reachable from $s_i$ by $\varepsilon$ transitions

The algorithm:

- Start state derived from $n_0$ of the NFA
- Take its $\varepsilon$-closure $q_0 = \varepsilon$-closure($n_0$)
- Compute $\text{Delta}(q, \alpha)$ for each $\alpha \in \Sigma$, and take its $\varepsilon$-closure
- Iterate until no more states are added

*Sounds more complex than it is...*
NFA $\rightarrow$ DFA with Subset Construction

The algorithm:

$q_0 \leftarrow \varepsilon$-closure($n_0$)
$Q \leftarrow \{q_0\}$

WorkList $\leftarrow \{q_0\}$

while (WorkList $\neq \emptyset$)
remove $q$ from WorkList
for each $\alpha \in \Sigma$

$t \leftarrow \varepsilon$-closure($\Delta(q, \alpha)$)
$T[q, \alpha] \leftarrow t$
if ($t \notin Q$) then
add $t$ to $Q$ and WorkList

Let's think about why this works

The algorithm halts:

1. $Q$ contains no duplicates (test before adding)
2. $2^Q$ is finite
3. while loop adds to $Q$, but does not remove from $Q$ (monotone)
$\Rightarrow$ the loop halts

$Q$ contains all the reachable NFA states

It tries each character in each $q$. It builds every possible NFA configuration.
$\Rightarrow Q$ and $T$ form the DFA
NFA $\rightarrow$ DFA with Subset Construction

Example of a *fixed-point* computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

*We will see many more fixed-point computations*
NFA $\rightarrow$ DFA with Subset Construction

\[ a \ (b \ | \ c)^* : \]

Applying the subset construction:

<table>
<thead>
<tr>
<th></th>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$q_0$</td>
<td>$q_1, q_2, q_3,$</td>
<td>$\text{none}$</td>
<td>$\text{none}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q_4, q_6, q_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$q_1, q_2, q_3,$</td>
<td>$\text{none}$</td>
<td>$q_5, q_8, q_9, $</td>
<td>$q_7, q_8, q_9, $</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q_4, q_6, q_9$</td>
<td>$q_3, q_4, q_6$</td>
<td>$q_3, q_4, q_6$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$q_5, q_8, q_9,$</td>
<td>$\text{none}$</td>
<td>$s_2$</td>
<td>$s_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q_3, q_4, q_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3$</td>
<td>$q_7, q_8, q_9,$</td>
<td>$\text{none}$</td>
<td>$s_2$</td>
<td>$s_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q_3, q_4, q_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Final states**
NFA $\rightarrow$ DFA with Subset Construction

The DFA for $a (b \mid c)^*$

- Ends up smaller than the NFA
- All transitions are deterministic
- Use same code skeleton as before

\[
\begin{array}{c|c|c|c}
\delta & a & b & c \\
\hline
s_0 & s_1 & - & - \\
\hline
s_1 & - & s_2 & s_3 \\
\hline
s_2 & - & s_2 & s_3 \\
\hline
s_3 & - & s_2 & s_3 \\
\end{array}
\]
Where are we? Why are we doing this?

**RE → NFA (Thompson’s construction) ✔**
- Build an NFA for each term
- Combine them with ε-moves

**NFA → DFA (subset construction) ✔**
- Build the simulation

**DFA → Minimal DFA**
- Hopcroft’s algorithm

**DFA → RE**
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state

*Enough theory for today*
What we expect of the Scanner

• **Report errors** for lexicographically malformed inputs
  → reject illegal characters, or meaningless character sequences
  → E.g., ‘#’ or “floop” in COOL

• Return an **abstract representation** of the code
  → character sequences (e.g., “if” or “loop”) turned into **tokens**.

• Resulting sequence of tokens will be used by the parser

• **Makes the design of the parser** a lot easier.
How to specify a scanner

- A scanner specification (e.g., for JLex), is list of (typically short) regular expressions.

- Each regular expressions has an action associated with it.

- Typically, an action is to return a token.

- On a given input string, the scanner will:
  - find the longest prefix of the input string, that matches one of the regular expressions.
  - will execute the action associated with the matching regular expression highest in the list.

- Scanner repeats this procedure for the remaining input.

- If no match can be found at some point, an error is reported.
Example of a Specification

- Consider the following scanner specification.
  1. `aaa` { return T1 }
  2. `a*b` { return T2 }
  3. `b` { return S }

- Given the following input string into the scanner
  aaabbaaa
  
  the scanner as specified above would output
  
  T2 T2 T1

- Note that the scanner will report an error for example on the string ‘aa’.
Special Return Tokens

- Sometimes one wants to extract information out of what prefix of the input was matched.
- Example:
  
  ```
  \[a-zA-Z0-9]*\] { return STRING(yytext()) }
  ```

- Above RE matches every string that
  - starts and ends with quotes, and
  - has any number of alpha-numerical chars between them.
- Associated action returns a string token, which is the exact string that the RE matched.
- Note that yytext() will also include the quotes.
- Furthermore, note that this regular expression does not handle escaped characters.