Lexical Analysis
Part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Grades for first homework are out.

• Third homework has been posted. Due date: Tuesday, February 26

• First project has been posted. Due dates: CODE Monday March 4; REPORT Wednesday March 6

• Midterm: Wednesday, March 13, in class; closed book, closed notes, 80 minutes
Regular expressions can be used to specify the words to be translated to parts of speech (tokens) by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions.

⇒ We study REs and associated theory to automate scanner construction!
Goal

- We will show how to construct a finite state automaton to recognize any RE

Overview:
- Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Requires $\varepsilon$-transitions to combine regular subexpressions
- Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Use a set-of-states construction
- Minimize the number of states
  - Hopcroft state minimization algorithm
- Generate the scanner code
  - Additional specifications needed for details
More Regular Expressions Examples

• 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

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

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

• 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 → (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:

\[(1^* (\varepsilon | 01 | 001 )1^*)^* (\varepsilon | 0 | 00)\]
Each RE corresponds to a *deterministic finite automaton* (DFA)

- May be hard to directly construct the right DFA

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

This is a little different

- \(S_0\) has a transition on \(\epsilon\)
- \(S_1\) has two transitions on \(a\)

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

- An NFA accepts a string $x$ iff there exists a path through 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 step.
  - 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.
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

\[ \rightarrow \text{Obviously} \]

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 and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser (define all parts of speech)
- You could build one in a weekend!
Automating Scanner Construction

RE $\rightarrow$ NFA (Thompson’s construction)
- Build an NFA for each term
- Combine them with $\epsilon$-moves

NFA $\rightarrow$ DFA (subset construction)
- Build the simulation

DFA $\rightarrow$ Minimal DFA
- Hopcroft’s algorithm

DFA $\rightarrow$ 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

The Cycle of Constructions
Key idea
• NFA pattern for each symbol and each operator
• Each NFA has a single start and accept state
• Join them with ε moves in precedence order

NFA for a

NFA for ab

NFA for a | b

NFA for a*

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

Let’s try $a \ (b \ | \ c )^*$

1. $a, b, \ & \ c$

2. $b \ | \ c$

3. $(b \ | \ c )^*$
4. \( a ( b \mid c )^* \)

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

But, we can automate production of the more complex one ...
Need to build a simulation of the NFA

Two key functions

- $\text{move}(s_i, a)$ is set of states reachable from $s_i$ by $a$
- $\varepsilon$-closure($s_i$) is set of states reachable from $s_i$ by $\varepsilon$

The algorithm (sketch):

- Start state derived from $s_0$ of the NFA
- Take its $\varepsilon$-closure $S_0 = \varepsilon$-closure($s_0$)
- For each state $S$, compute $\text{move}(S, a)$ for each $a \in \Sigma$, and take its $\varepsilon$-closure
- Iterate until no more states are added

*Sounds more complex than it is...*
NFA → DFA with Subset Construction

The algorithm:

\[
s_0 \leftarrow \varepsilon\text{-closure}(q_0)
\]

add \( s_0 \) to \( S \)

while ( \( S \) is still changing )

for each \( s_i \in S \)

for each \( a \in \Sigma \)

\[
s_? \leftarrow \varepsilon\text{-closure}(\text{move}(s_i, a))
\]

if ( \( s_? \notin S \) ) then

add \( s_? \) to \( S \) as \( s_j \)

\( T[s_i, a] \leftarrow s_j \)

else

\( T[s_i, a] \leftarrow s_? \)

Let’s think about why this works

The algorithm halts:

1. \( S \) contains no duplicates
   (test before adding)
2. \( 2^\Sigma \) is finite
3. while loop adds to \( S \), but does not remove from \( S \) (monotone)

\( \Rightarrow \) the loop halts

\( S \) contains all the reachable NFA states

It tries each symbol in each \( s_i \).

It builds every possible NFA configuration.

\( \Rightarrow S \) and \( T \) form the DFA
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

Other fixed-point computations
• Canonical construction of sets of LR(1) items
  → Quite similar to the subset construction
• Classic data-flow analysis
  → Solving sets of simultaneous set equations
• DFA minimization algorithm (coming up!)

*We will see many more fixed-point computations*
a \ (b \mid c)^* : 

Applying the subset construction:
Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>$\varepsilon$-closure (move(s, *))</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$ $q_0$</td>
<td>$q_1, q_2, q_3, q_4, q_6, q_9$</td>
</tr>
<tr>
<td>$s_1$ $q_1, q_2, q_3, q_4, q_6, q_9$</td>
<td>$\text{none}$</td>
</tr>
<tr>
<td>$s_2$ $q_5, q_8, q_9, q_3, q_4, q_6$</td>
<td>$\text{none}$</td>
</tr>
<tr>
<td>$s_3$ $q_7, q_8, q_9, q_3, q_4, q_6$</td>
<td>$\text{none}$</td>
</tr>
</tbody>
</table>
The DFA for $a (b | c)^*$

• Ends up smaller than the NFA
• All transitions are deterministic
Automating Scanner Construction

RE $\rightarrow$ NFA (Thompson’s construction)
• Build an NFA for each term
• Combine them with $\varepsilon$-moves

NFA $\rightarrow$ DFA (subset construction)
• Build the simulation

DFA $\rightarrow$ Minimal DFA
• Hopcroft’s algorithm

DFA $\rightarrow$ RE (not really part of scanner construction)
• All pairs, all paths problem
• Union together paths from $s_0$ to a final state

The Cycle of Constructions

---

cs415, spring 19  Lecture 8  21
The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state
The Big Picture
• Discover sets of equivalent states
• Represent each such set with just one state

Two states are equivalent if and only if:
• \( \forall a \in \Sigma, \) transitions on \( a \) lead to equivalent states \((\text{DFA})\)
• if \( a \text{-transitions to different sets} \Rightarrow \) two states must be in different sets, i.e., cannot be equivalent
DFA Minimization

The Big Picture
- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:
- \( \forall a \in \Sigma, \text{ transitions on } a \text{ lead to equivalent states} \) (DFA)
- if \( a \)-transitions to different sets \( \Rightarrow \) two states must be in different sets, i.e., cannot be equivalent

A partition \( P \) of \( S \)
- Each state \( s \in S \) is in exactly one set \( p_i \in P \)
- The algorithm iteratively partitions the DFA’s states
Details of the algorithm

- Group states into maximal size sets, **optimistically**
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, $P_0$, has two sets: $\{F\} \& \{Q-F\}$ \hspace{4cm} (D=($Q, \Sigma, \delta, q_0, F$))

Splitting a set ("partitioning a set by $a$")

- Assume $q_a, q_b \in s$, and $\delta(q_a, a) = q_x$, & $\delta(q_b, a) = q_y$
- If $q_x$ & $q_y$ are not in the same set, then $s$ must be split
  $\Rightarrow q_a$ has transition on $a$, $q_b$ does not $\Rightarrow a$ splits $s$
The algorithm

\[ P \leftarrow \{ F, \{Q - F}\} \]

while (\( P \) is still changing)

\[ T \leftarrow \{ \} \]

for each set \( S \in P \)

\[ T \leftarrow T \cup \text{split}(S) \]

\[ P \leftarrow T \]

\text{split}(S):

for each \( a \in \Sigma \)

if \( a \) splits \( S \) into \( S_1, S_2, \ldots \) then

return \( \{S_1, S_2, \ldots\} \)

else return \( S \)

Why does this work?

- Partition \( P \in 2^Q \)
- Start off with 2 subsets of \( Q \) \{\( F \)\} and \{\( Q - F \)\}
- While loop takes \( P_i \rightarrow P_{i+1} \) by splitting 1 or more sets
- \( P_{i+1} \) is at least one step closer to the partition with \( |Q| \) sets
- Maximum of \( |Q| \) splits

Note that

- Partitions are never combined

This is a fixed-point algorithm!
Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 ) ( {s_1, s_2, s_3} {s_0} )</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

To produce the minimal DFA

We observed that a human would design a simpler automaton than Thompson’s construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!
Wrap-up Lexical Analysis

Syntax Analysis (top-down)

Read EaC: Chapter 3.1 - 3.3