Announcements

• Homework #2 deadline extension?

• Homework #3 has been posted. Due: Friday, February 23

• My office hours: Not too many students show up. Conflict with other classes?
Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;
Lexical patterns form a *regular language*

*** any finite language is regular ***

Regular expressions (REs) describe regular languages.

Regular Expression (over alphabet $\Sigma$)

- $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
- If “a” is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
- If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
  - $x | y$ is an RE denoting $L(x) \cup L(y)$
  - $xy$ is an RE denoting $L(x)L(y)$
  - $x^*$ is an RE denoting $L(x)^*$
  - $(x)$ is an RE denoting $L(x)$

Precedence is closure, then concatenation, then alternation

Ever type “rm *.o a.out”?
Review: NFA

- 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 \textit{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
- \textit{We can paste together} NFAs with \( \varepsilon \)-transitions
DFA is a special case of an NFA
- DFA has no ε 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 \( (\text{less obvious}) \)
- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream
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 $\varepsilon$-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

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 (con’t)

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

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

• $move(s_i, a)$ 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 $move(S, a)$ for each $a \in \Sigma$, and take its $\varepsilon$-closure
• Iterate until no more states are added

$Sounds more complex than it is...$
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. \( 2^Q \) 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
**NFA → DFA with Subset Construction**

### a (b | c)*:

**Applying the subset construction:**

<table>
<thead>
<tr>
<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>$q_1$, $q_2$, $q_3$, $q_4$, $q_6$, $q_9$</td>
<td>none</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$q_1$, $q_2$, $q_3$, $q_4$, $q_6$, $q_9$</td>
<td>none</td>
<td>$q_5$, $q_8$, $q_9$, $q_3$, $q_4$, $q_6$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$q_5$, $q_8$, $q_9$, $q_3$, $q_4$, $q_6$</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$q_7$, $q_8$, $q_9$, $q_3$, $q_4$, $q_6$</td>
<td>none</td>
<td>$s_2$</td>
</tr>
</tbody>
</table>

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

- 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 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:

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

• $\forall a \in \Sigma$, transitions on $a$ 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, \textit{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{1cm} (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 \( s \)
DFA Minimization

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!
Back to our DFA Minimization example

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>${s_1, s_2, s_3}$</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>${s_0}$</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!
Start with a regular expression

\( r0 \mid r1 \mid r2 \mid r3 \mid r4 \mid r5 \mid r6 \mid r7 \mid r8 \mid r9 \)

Abbreviated Register Specification

The Cycle of Constructions
Thompson’s construction produces

\[
\begin{align*}
&\text{The Cycle of Constructions} \\
&\text{RE} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{minimal DFA}
\end{align*}
\]
The subset construction builds

This is a DFA, but it has a lot of states ...

The Cycle of Constructions
The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

The Cycle of Constructions
Advantages of Regular Expressions
- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

\[
\begin{align*}
Term & \rightarrow [a-zA-Z] ([a-zA-Z] \mid [0-9])^* \\
Op & \rightarrow + \mid - \mid * \mid /
\end{align*}
\]

\[
Expr \rightarrow (Term\ Op)^*\ Term
\]

Of course, this would generate a DFA ...

If REs are so useful ...

*Why not use them for everything?*
Limits of Regular Languages

Not all languages are regular

RL’s ⊂ CFL’s ⊂ CSL’s

You cannot construct DFA’s to recognize these languages

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

Neither of these is a regular language

But, this is a little subtle. You can construct DFA’s for

- Strings with alternating 0’s and 1’s
  \( (\epsilon | 1)(01)^*(\epsilon | 0) \)
- Strings with and even number of 0’s and 1’s
- Strings of bit patterns that represent binary numbers which are divisible by 5 (homework)
Poor language design can complicate scanning

- Reserved words are important
  
  if then then then = else; else else = then  
  (PL/I)

- Insignificant blanks
  
  do 10 i = 1,25  
  do 10 i = 1.25  
  (Fortran & Algol68)

- String constants with special characters
  
  newline, tab, quote, comment delimiters, ...  
  (C, C++, Java, ...)

- Limited identifier “length”  
  (Fortran 66 & PL/I)
More Lexical Analysis

Syntax Analysis

Read EaC: 3.1 - 3.3