CS415 Compilers

Lexical Analysis
Syntax Analysis

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

• Homework #2 due tonight

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

• My new office hours time: Wednesdays, 11:00am – 12:30pm
Automating Scanner Construction

RE $\rightarrow$ NFA \textit{(Thompson’s construction)}
\begin{itemize}
  \item Build an NFA for each term
  \item Combine them with $\varepsilon$-moves
\end{itemize}

NFA $\rightarrow$ DFA \textit{(subset construction)}
\begin{itemize}
  \item Build the simulation
\end{itemize}

DFA $\rightarrow$ Minimal DFA
\begin{itemize}
  \item Hopcroft’s algorithm
\end{itemize}

DFA $\rightarrow$ RE \textit{(Not part of the scanner construction)}
\begin{itemize}
  \item All pairs, all paths problem
  \item Take the union of all paths from $s_0$ to an accepting state
\end{itemize}
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

\[ S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_2 \xrightarrow{\varepsilon} S_3 \]  
NFA for \( a \)

\[ S_0 \xrightarrow{a} S_1 \xrightarrow{\varepsilon} S_2 \xrightarrow{b} S_4 \]  
NFA for \( ab \)

\[ S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a^*} S_2 \xrightarrow{\varepsilon} S_3 \xrightarrow{\varepsilon} S_4 \]  
NFA for \( a^* \)

Ken Thompson, CACM, 1968
a ( b | c )^*
NFA $\rightarrow$ DFA with Subset Construction

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...
The algorithm:

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

\textit{add } s_0 \textit{ to } S

\textit{while ( } S \textit{ is still changing ) }

\textit{for each } s_i \in S

\textit{for each } a \in \Sigma

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

\textit{if ( } s_? \notin S \textit{ ) then}

\textit{add } s_? \textit{ to } S \textit{ as } s_j

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

\textit{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

\textit{It tries each symbol in each } s_i.

\textit{It builds every possible NFA configuration.}

\( \Rightarrow S \text{ and } T \text{ form the DFA} \)
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*
### Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>ε-closure (move(s,*))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>(q_0)</td>
</tr>
<tr>
<td>(s_1)</td>
<td>(q_1, q_2, q_3, q_4, q_6, q_9)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(q_5, q_8, q_9, q_3, q_4, q_6)</td>
</tr>
<tr>
<td>(s_3)</td>
<td>(q_7, q_8, q_9, q_3, q_4, q_6)</td>
</tr>
</tbody>
</table>

For \(a\): \(q_1, q_2, q_3, q_4, q_6, q_9\)

For \(b\): \(q_5, q_8, q_9, q_3, q_4, q_6\)

For \(c\): \(q_7, q_8, q_9, q_3, q_4, q_6\)

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

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

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

NFA → DFA (subset construction)
• Build the simulation

DFA → Minimal DFA
• Hopcroft’s algorithm

DFA → 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:

• \( \forall \ a \in \Sigma \), transitions on \( a \) lead to equivalent states \((\text{DFA})\)
• if \( a \)-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:

• \( \forall a \in \Sigma, \) 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, optimistically
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, $P_0$, has two sets: \{F\} & \{Q-F\}  \quad (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

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

\[ \text{while (P is still changing)} \]
\[ \quad T \leftarrow \{\} \]
\[ \quad \text{for each set } S \in P \]
\[ \quad \quad T \leftarrow T \cup \text{split}(S) \]
\[ \quad P \leftarrow T \]

\text{split}(S):
\[ \quad \text{for each } a \in \Sigma \]
\[ \quad \quad \text{if } a \text{ splits } S \text{ into } \]
\[ \quad \quad \quad \quad S_1, S_2, \ldots \text{ then} \]
\[ \quad \quad \quad \quad \quad \text{return } \{S_1, S_2, \ldots\} \]
\[ \quad \quad \text{else return } S \]

Why does this work?

- Partition \( P \in 2^Q \)
- Start off with 2 subsets of \( Q \) \{\( F \)\} and \( \{Q - F\} \)
- \textit{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 \textit{never} combined

\textit{This is a fixed-point algorithm!}
Then, apply the minimization algorithm

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

\[
\begin{align*}
\text{Term} & \rightarrow [a-zA-Z] ([a-zA-Z] | [0-9])^* \\
\text{Op} & \rightarrow \pm | \cdot | \ast | / \\
\text{Expr} & \rightarrow (\text{Term} \text{ Op})^* \text{ Term}
\end{align*}
\]

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
\( \text{RL's} \subset \text{CFL's} \subset \text{CSL's} \)

You cannot construct DFA's to recognize these languages

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

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
  \( (\varepsilon \mid 1)(01)^*(\varepsilon \mid 0) \)
- 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)
What can be so hard?

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)
Syntax Analysis (top-down)

Read EaC: Chapter 3.1 - 3.3