CS415 Compilers

Lexical Analysis

Part 4

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
• Third homework has been posted. Due Monday, March 7

• First project (local instruction scheduler) NEW deadlines:
  code: March 9
  report: March 11
1. The order in which programs perform I/O has to be preserved. This requires a dependence notion on output instructions.

2. Benchmark codes have store and load memory accesses only, with r0 as the base register. You must use the offset to determine whether a dependence exists or not.

3. We will have some private tests that have store and load memory accesses. This is for extra credit.

4. Project reports should be around 6 pages long, with a max of 8 pages. We will not read your report beyond 8 pages. The report should include a short description of what you did, the outcome of your experiments, and how you interpret these outcomes. Use graphs/figures to show your results.
Example of Thompson’s Construction

Let’s try $a ( b \mid c )^*$

1. $a, b, \& c$

2. $b \mid c$

3. $( b \mid c )^*$
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 set of states $s_i$ by $a$
• $\varepsilon$-closure($s_i$) is set of states reachable from set of states $s_i$ 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:

\[ s_0 \rightarrow \varepsilon\text{-closure}(q_0) \]
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. \( 2^\mathcal{Q} \) is finite
3. while loop adds to \( S \), but does not remove from \( S \) (monotone)
\[ \Rightarrow \text{the loop halts} \]
\( S \) contains all the reachable NFA states
It tries each symbol in each \( s_i \).
It builds every possible NFA configuration.
\[ \Rightarrow S \text{ and } T \text{ form the DFA} \]
Example of a \textit{fixed-point} computation
\begin{itemize}
    \item Monotone construction of some finite set
    \item Halts when it stops adding to the set
    \item Proofs of halting & correctness are similar
    \item These computations arise in many contexts
\end{itemize}

Other fixed-point computations
\begin{itemize}
    \item \textit{Canonical} construction of sets of LR(1) items
        \begin{itemize}
            \item Quite similar to the subset construction
        \end{itemize}
    \item Classic data-flow analysis
        \begin{itemize}
            \item Solving sets of simultaneous set equations
        \end{itemize}
    \item DFA minimization algorithm (coming up!)
\end{itemize}

\textit{We will see many more fixed-point computations}
NFA → DFA with Subset Construction

Applying the subset construction:

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

\[ q_0 \overset{a}{\rightarrow} q_1 \overset{\varepsilon}{\rightarrow} q_2 \overset{\varepsilon}{\rightarrow} q_3 \overset{\varepsilon}{\rightarrow} q_4 \overset{b}{\rightarrow} q_5 \overset{\varepsilon}{\rightarrow} q_6 \overset{c}{\rightarrow} q_7 \overset{\varepsilon}{\rightarrow} q_8 \overset{\varepsilon}{\rightarrow} q_9 \]
### NFA → DFA with Subset Construction

**a (b | c)*:**

![Diagram](image)

**Applying the subset construction:**

<table>
<thead>
<tr>
<th>Sets of NFA states</th>
<th>ε-closure(move(s,*))</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>{q₀}</td>
<td>{q₁, q₂, q₃, q₄, q₆, q₉}</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>s₁</td>
<td>{q₁, q₂, q₃, q₄, q₆, q₉}</td>
<td>none</td>
<td>{q₅, q₆, q₉, q₃, q₄, q₆}</td>
<td>{q₇, q₈, q₉, q₃, q₄, q₆}</td>
</tr>
<tr>
<td>s₂</td>
<td>{q₅, q₆, q₉, q₃, q₄, q₆}</td>
<td>none</td>
<td>s₂</td>
<td>s₃</td>
</tr>
<tr>
<td>s₃</td>
<td>{q₇, q₈, q₉, q₃, q₄, q₆}</td>
<td>none</td>
<td>s₂</td>
<td>s₃</td>
</tr>
</tbody>
</table>

**Final states**
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
How do we know whether two states encode the same information?

Intuition: Two states are equivalent if for all sequences of input symbols “w” they both lead to an accepting state, or both end up in a non-accepting state.

$q_1$ and $q_2$ are not equivalent. “w” is a witness that they are not equivalent.
DFA Minimization

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 \hspace{1cm} \text{(DFA)}
• if $a$-transitions to different sets $\Rightarrow$ two states must be in different sets, i.e., cannot be equivalent
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 \hspace{1cm} \text{(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 $s$ 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, i.e., are considered equivalent, then $s$ must be split
  \[ \rightarrow q_a \text{ has transition on } a, q_b \text{ does not } \Rightarrow a \text{ splits } s \]
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?
\begin{itemize}
  \item Start off with 2 subsets of \( Q \) \{\( F \)\} and \( \{Q - F\} \)
  \item While loop takes \( P_i \rightarrow P_{i+1} \) by splitting 1 or more sets
  \item \( P_{i+1} \) is at least one step closer to the partition with \(|Q|\) sets
  \item Maximum of \(|Q|\) splits
\end{itemize}

Note that
\begin{itemize}
  \item Partitions are never combined
\end{itemize}

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

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
<th>( \varepsilon )</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>
<td></td>
</tr>
</tbody>
</table>

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!
Another Example Register Specification

Start with a regular expression

\( r_0 \mid r_1 \mid r_2 \mid r_3 \mid r_4 \mid r_5 \mid r_6 \mid r_7 \mid r_8 \mid r_9 \)

The Cycle of Constructions
Thompson’s construction produces

The Cycle of Constructions
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!
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

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

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
RL’s ⊂ CFL’s ⊂ 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
  $(\varepsilon | 1)(01)^*(\varepsilon | 0)$
• Strings with and even number of 0’s and 1’s
• Strings of bit patterns that represent binary numbers which are divisible by 5
Poor language design can complicate scanning

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

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

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

- **Limited identifier “length”**
  
  ```plaintext
  (Fortran 66 & PL/I)
  ```
Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.2