CS415 Compilers

Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
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
  - $x \mid y$ 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 $L(x)$

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 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 $\rightarrow$ DFA construction

• 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

→ *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 $\varepsilon$-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 \(a ( b | c )^*\)

1. \(a, b, & c\)
   
   ![Diagram 1](image1)

2. \(b | c\)
   
   ![Diagram 2](image2)

3. \(( b | c )^*\)
   
   ![Diagram 3](image3)
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
- $\text{move}(s_i, a)$ 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:

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

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 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)^

![NFA Diagram](image)

Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>$\varepsilon$-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>

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

- 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:

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

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

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

Why does this work?

- Partition \( P \in 2^Q \)
- Start off with 2 subsets of \( Q \) \{F\} and \{Q-F\}
- 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>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>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

\[ r_0 \mid r_1 \mid r_2 \mid r_3 \mid r_4 \mid r_5 \mid r_6 \mid r_7 \mid r_8 \mid r_9 \]
Thompson’s construction produces

The Cycle of Constructions
Abbreviated Register Specification

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
More Lexical Analysis

Syntax Analysis

Read EaC: 3.1 - 3.3