CS415 Compilers

Lexical Analysis
Part 3

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

- First quiz has been posted on sakai. You have two tries. Last one counts. Quiz is 50 minutes long. Open book, open notes. Deadline: 02/19 @ 11:55pm

- Second homework deadline extension: Sunday, 02/11

- First project deadline extension: Tuesday (03/02) for code, and Friday (03/05) for report.
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!
Consider the problem of recognizing ILOC register names

\[ \text{Register} \rightarrow r \ (0|1|2| \ldots | 9) \ (0|1|2| \ldots | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

**Example**

![Diagram](image)

Recognizer for *Register*

*Transitions on other inputs go to an error state, \( s_e \)*
DFA operation

- Start in state $S_0$ & take transitions on each input character
- DFA accepts a word $x$ iff $x$ leaves it in a final state ($S_2$)

So,

- $r_{17}$ takes it through $s_0$, $s_1$, $s_2$ and accepts
- $r$ takes it through $s_0$, $s_1$ and fails
- $a$ takes it straight to error state $s_e$ (not shown here)
To be useful, recognizer must turn into code

Char $\leftarrow$ *next character*
State $\leftarrow s_0$

while (Char $\neq$ EOF)
  State $\leftarrow \delta$(State,Char)
  Char $\leftarrow$ *next character*
if (State is a final state )
  then report success
else report failure

### Skeleton recognizer

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

### Table encoding RE

Example (continued)
To be useful, recognizer must turn into code

Char $\leftarrow$ next character
State $\leftarrow s_0$

while (Char $\neq$ EOF)
    State $\leftarrow \delta$(State,Char)
    perform specified action
    Char $\leftarrow$ next character

if (State is a final state)
    then report success
else report failure

### Table encoding RE

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$ start</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$ error</td>
<td>$s_2$ add</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$ error</td>
<td>$s_2$ add</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
</tr>
</tbody>
</table>

**Skeleton recognizer**

Lecture 10
r Digit Digit* allows arbitrary numbers
• Accepts r00000
• Accepts r99999
• What if we want to limit it to r0 through r31?

Write a tighter regular expression
→ Register → r ( (0|1|2) (Digit | e) | (4|5|6|7|8|9) | (3|30|31) )
→ Register → r0|r1|r2| ... |r31|r00|r01|r02| ... |r09

Produces a more complex DFA
• Has more states
• Same cost per transition
• Same basic implementation
The DFA for
\[ \text{Register} \rightarrow r \ ( (0|1|2) \ (\text{Digit} \mid \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) ) \]

- Accepts a more constrained set of registers
- Same set of actions, more states
Tighter register specification (continued)

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>( r )</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_e )</td>
<td>( s_2 )</td>
<td>( s_2 )</td>
<td>( s_5 )</td>
<td>( s_4 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_e )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_3 )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_e )</td>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
<tr>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
<td>( s_e )</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

 Runs in the same skeleton recognizer
Constructing a Scanner - Quick Review

- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
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
• 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

\[Cons \rightarrow (b\mid c\mid d\mid f\mid g\mid h\mid j\mid k\mid l\mid m\mid n\mid p\mid q\mid r\mid s\mid t\mid v\mid w\mid x\mid y\mid z)\]

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

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

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

- 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

  \(Cons \rightarrow (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)\)
  
  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:

  \(( 1^* ( \epsilon | 01 | 001 ) 1^* )^* ( \epsilon | 0 | 00 )\)
Each RE corresponds to a *deterministic finite automaton* (DFA)
- May be hard to directly construct the right DFA

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

This is a little different
- \(S_0\) has a transition on \(\varepsilon\)
- \(S_1\) has two transitions on \(a\)

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

• An NFA accepts a string \( x \) iff \( \exists \) 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→DFA construction
• We can paste together NFAs with \( \varepsilon \)-transitions

\[
\text{NFA} \xrightarrow{\varepsilon} \text{NFA} \quad \text{becomes an} \quad \text{NFA}
\]
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
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
- You could build one in a weekend!
Automating Scanner Construction

**RE → NFA (Thompson’s construction)**
- Build an NFA for each term
- Combine them with ε-moves

**NFA → DFA (subset construction)**
- Build the simulation

**DFA → Minimal DFA**
- Hopcroft’s algorithm

**DFA → 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
Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with $\varepsilon$ moves in precedence order

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

Let’s try $a \ (b \ | \ c)^*$

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

2. $b \ | \ c$

3. $(b \ | \ c)^*$
Example of Thompson’s Construction (cont’)

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

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

But, we can automate production of the more complex one ...
NFA $\rightarrow$ DFA with Subset Construction

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...
NFA \rightarrow DFA with Subset Construction

The algorithm:

\[ s_0 \leftarrow \varepsilon\text{-closure}(q_0) \]
\[ \text{add } s_0 \text{ to } S \]
\[ \text{while ( } S \text{ is still changing) } \]
\[ \text{for each } s_i \in S \]
\[ \text{for each } a \in \Sigma \]
\[ s_? \leftarrow \varepsilon\text{-closure(move}(s_i, a)) \]
\[ \text{if ( } s_? \notin S \text{ ) then} \]
\[ \text{add } s_? \text{ to } S \text{ as } s_j \]
\[ T[s_i, a] \leftarrow s_j \]
\[ \text{else} \]
\[ T[s_i, a] \leftarrow s_? \]

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)
\[ \Rightarrow \text{the loop halts} \]
\[ S \text{ contains all the reachable NFA states} \]
\[ \text{It tries each symbol in each } s_i. \]
\[ \text{It builds every possible NFA configuration.} \]
\[ \Rightarrow S \text{ and } T \text{ form the DFA} \]
NFA → DFA with Subset Construction

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:

\(a (b \cup c)^*:\)
a (b | c)*:

Applying the subset construction:

<table>
<thead>
<tr>
<th></th>
<th>NFA states</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, q₈, q₉, q₃, q₄, q₆</td>
<td>q₇, q₈, q₉, q₃, q₄, q₆</td>
</tr>
<tr>
<td>s₂</td>
<td>q₅, q₈, 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 \( a (b \mid c)^* \)

- Ends up smaller than the NFA
- All transitions are deterministic
Wrap-up Lexical Analysis

Syntax Analysis (top-down)

Read EaC: Chapter 3.1 – 3.3