Implementation of Lexical Analysis

Lecture 3
Written Assignments

• WA1 assigned today

• Due in one week
  – By 11:59 pm
  – Turn in
    • In Quera
    • Electronically
Notation

• There is variation in regular expression notation

• At least one: \( A^+ \) \( \equiv AA^* \)
• Union: \( A | B \) \( \equiv A + B \)
• Option: \( A + \varepsilon \) \( \equiv A? \)
• Range: ‘a’+‘b’+…+‘z’ \( \equiv [a-z] \)
• Excluded range:
  complement of \([a-z]\) \( \equiv [^a-z] \)
Regular Expressions in Lexical Specification

- Last Lecture: a specification for the predicate
  \[ s \in L(R) \]
  Set of strings
- But a yes/no answer is not enough!
- Instead: partition the input into tokens
  \[ C_1 C_2 C_3 | C_4 C_5 C_6 C_7 | \ldots \]
- We adapt regular expressions to this goal
Regular Expressions => Lexical Spec. (1)

1. Write a rexp for the lexemes of each token
   - Number = digit +
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - OpenPar = '('
   - ...

Regular Expressions => Lexical Spec. (2)

2. Construct $R$, matching all lexemes for all tokens

\[ R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots \]
\[ = R_1 + R_2 + \ldots \]
3. Let input be $x_1...x_n$
   For $1 \leq i \leq n$ check
   $x_1...x_i \in L(R)$

4. If success, then we know that
   $x_1...x_i \in L(R_j)$ for some $j$

5. Remove $x_1...x_i$ from input and go to (3)
Ambiguities (1)

• There are ambiguities in the algorithm

• How much input is used? What if
  • $x_1 \ldots x_i \in L(R)$ and also
  • $x_1 \ldots x_k \in L(R)$ with $k \neq i$

• Rule: Pick longest possible string in $L(R)$
  - The “maximal munch”
Ambiguities (2)

• Which token is used? What if
  • $x_1...x_i \in L(R_j)$ and also
  • $x_1...x_i \in L(R_k)$
    \[ R = R_1 + R_2 + R_3 + ... \]
    \[ k \neq i \]

Keyword = ‘if’ + ‘else’ + ...
Identifier = letter (letter + digit)*

• Rule: use rule listed first ($j$ if $j < k$)
  - Treats “if” as a keyword, not an identifier

Prof. Aiken
Error Handling

• What if
  No rule matches a prefix of input?
  \( x_1...x_i \notin L(R_j) \)

• Problem: Can’t just get stuck …

• Solution:
  - Write a rule matching all “bad” strings
  - Put it last (lowest priority)
Summary

• Regular expressions provide a concise notation for string patterns

• Use in lexical analysis requires small extensions
  – To resolve ambiguities
  – To handle errors

• Good algorithms known
  – Require only single pass over the input
  – Few operations per character (table lookup)
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet $\Sigma$
  - A finite set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \rightarrow^{\text{input}} \text{state}$
Finite Automata

• Transition

\[ s_1 \rightarrow^a s_2 \]

• Is read

In state \( s_1 \) on input “a” go to state \( s_2 \)

• If end of input and in accepting state => accept

• Otherwise => reject

\[ \begin{align*}
\text{Terminates in a state } s \text{ that is} \\
\text{NOT an accepting state} \ (s \notin F) \\
\text{Gets stuck}
\end{align*} \]
Finite Automata State Graphs

- A state

- The start state

- An accepting state

- A transition

Prof. Aiken 14
A Simple Example

- A finite automaton that accepts only “1”

- Accepts ‘1’ : ↑1, 1↑
- Rejects ‘0’ : ↑0
- Rejects ’10’ : ↑1, 1↑0
Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: \{0,1\}

- Accepts ‘110’: \[ \rightarrow 110, 1\rightarrow 10, 11\rightarrow 0, \quad 110\rightarrow \]
- Rejects ‘100’: \[ \rightarrow 100, 1\rightarrow 00, \quad 10\rightarrow 0 \]
And Another Example

- Alphabet \(\{0,1\}\)
- What language does this recognize?
And Another Example

Select the regular language that denotes the same language as this finite automaton

- \((0 + 1)^*\)
- \((1^* + 0)(1 + 0)\)
- \(1^* + (01)^* + (001)^* + (000^*1)^*\)
- \((0 + 1)^*00\)
And Another Example

Select the regular language that denotes the same language as this finite automaton

- $(0 + 1)^*$
- $(1^* + 0)(1 + 0)$
- $1^* + (01)^* + (001)^* + (000^*1)^*$
- $(0 + 1)^*00$
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

- Machine can move from state $A$ to state $B$ without reading input
Deterministic and Nondeterministic Automata

• Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No $\varepsilon$-moves

• Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
Execution of Finite Automata

• A DFA can take only one path through the state graph
  - Completely determined by input

• NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take
Acceptance of NFAs

• An NFA can get into multiple states

• Input:

• Possible States:

Rule: NFA accepts if it can get to a final state
Acceptance of NFAs

• An NFA can get into multiple states

1

0

A

0

• Input: 1

• Possible States:

Rule: NFA accepts if it can get to a final state
Acceptance of NFAs

- An NFA can get into multiple states

- Input: 1
- Possible States: \{A\}

Rule: NFA accepts if it can get to a final state
Acceptance of NFAs

• An NFA can get into multiple states

• Input: \[ \begin{array}{c} 1 \\ 0 \end{array} \]

• Possible States: \{A\}

Rule: NFA accepts if it can get to a final state
Acceptance of NFAs

- An NFA can get into multiple states

Input: 1 0
Possible States: \{A\} \{A, B\}

Rule: NFA accepts if it can get to a final state.
Acceptance of NFAs

• An NFA can get into multiple states

• Input: 1 0 0

• Possible States: \{A\}, \{A, B\}

Rule: NFA accepts if it can get to a final state
Acceptance of NFAs

• An NFA can get into multiple states

• Input: 1 0 0

• Possible States: {A} {A, B} {A, B, C}

Rule: NFA accepts if it can get to a final state
NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute
  - There are no choices to consider

- NFAs are, in general, smaller
  - Sometimes exponentially smaller
NFA vs. DFA (2)

• For a given language NFA can be simpler than DFA

• DFA can be exponentially larger than NFA
Regular Expressions to Finite Automata

- High-level sketch

![Diagram showing the relationship between regular expressions, NFA, DFA, lexical specification, and table-driven implementation of DFA.]
Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp $M$

- For $\varepsilon$

- For input $a$
Regular Expressions to NFA (2)

• For $AB$

• For $A + B$
Regular Expressions to NFA (3)

- For $A^*$
Example of RegExp -> NFA conversion

- Consider the regular expression 
  \((1+0)^*1\)
- The NFA is
NFA to DFA: *The Trick*

- Simulate the NFA
- Each state of DFA
  - = a non-empty subset of states of the NFA
- Start state
  - = $\varepsilon$-closure of the start state of NFA
- Add a transition $S \xrightarrow{a} S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from any state in $S$ after seeing the input $a$, considering $\varepsilon$-moves as well
- Final states
  - Subsets that include at least one final state of NFA
\( \varepsilon\)-closure of a state

\( \varepsilon\)-closure(B) = \{B,C,D\}

\( \varepsilon\)-closure(G) = \{A,B,C,D,G,H,I\}
NFA -> DFA Example
NFA -> DFA Example
NFA -> DFA Example
NFA -> DFA Example
NFA -> DFA Example

Prof. Aiken
NFA -> DFA Example

ABCDHI

Prof. Aiken
NFA -> DFA Example
NFA -> DFA Example
NFA -> DFA Example

Prof. Aiken
NFA -> DFA Example
NFA -> DFA Example
NFA -> DFA Example

Prof. Aiken
NFA -> DFA Example
NFA -> DFA Example
Implementation

• A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

• DFA “execution”
  - If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  - Very efficient
Table Implementation of a DFA

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Implementation (Cont.)

- NFA $\rightarrow$ DFA conversion is at the heart of tools such as flex

- But, DFAs can be huge

- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations
DFA for recognizing two relational operators

We’ve accepted “>” and have read “other” character that must be unread. That is moving the input pointer one character back.
DFA of Pascal relational operators

```
start
0  <  1  =  2  return(SYMBOL, <=)
     |    |    |
     >  3  =  4  return(SYMBOL, <>)
     |    |
     other  5  *  6  return(SYMBOL, <)
     |
     >  7  =  8  return(SYMBOL, =)
     |
     other
```
DFA for recognizing id and keyword

If the token is an ID, its lexeme is inserted into the symbol table (only one record for each lexeme); and lexeme of the token is returned.

returns either a KEYWORD or ID based on the type of the token
DFA of Pascal Unsigned Numbers

return(NUM, lexeme of the number)
Lexical errors

• Some errors are out of power of lexical analyzer to recognize:
\[
fi (a == f(x)) \ldots
\]

• However, it may be able to recognize errors like:
\[
d = 2r
\]

• Such errors are recognized when no pattern for tokens matches a character sequence
Error recovery

• Panic mode: successive characters are ignored until we reach to a well formed token

• Delete one character from the remaining input
• Insert a missing character into the remaining input
• Replace a character by another character
• Transpose two adjacent characters
Consider the string \texttt{abbbaacc}. Which of the following lexical specifications produces the tokenization: \texttt{ab/bb/a/acc}

Choose all that apply

- \(c^*\)
- \(a(b + c^*)\)
- \(ab\)
- \(b^+\)
- \(b^+\)
- \(b^+\)
- \(ab^*\)
- \(ac^*\)
- \(ac^*\)

Prof. Aiken
Question?

Using the lexical specification below, how is the string “dictatorial” tokenized?

Choose all that apply

- dict (1)  ○ 1, 3
- dictator (2) ○ 3
- [a-z]* (3) ○ 4
- dictatorial (4) ○ 2, 3
Question?

Given the following lexical specification:

\[ a(ba)^* \]
\[ b^*(ab)^* \]
\[ abd \]
\[ d^+ \]

Which of the following statements is true?

Choose all that apply

- babad will be tokenized as: bab/a/d
- ababdddd will be tokenized as: abab/dddd
- dddabbabab will be tokenized as: ddd/a/bbabab
- ababddababa will be tokenized as: ab/abd/d/ababa
Question?

Given the following lexical specification:

\[(00)^*\]

\[01^+\]

\[10^+\]

Which strings are NOT successfully processed by this specification?

Choose all that apply

- 011110
- 01100100
- 01100110
- 01100110
- 0001101
Which of the following regular expressions generate the same language as the one recognized by this NFA?

Choose all that apply

- $(000)^*(01)^+$
- $0(000)^*1(01)^*$
- $(000)^*(10)^+$
- $0(00)^*(10)^*$
- $0(000)^*(01)^*$
Question?

Which of the following automata are DFA? Choose all that apply.
Question?

Which of the following automata are NFA? Choose all that apply