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: \( A + \emptyset \) \( \equiv \) \( A? \)
• Range: \( 'a'+'b'+\ldots+'z' \) \( \equiv \) \( [a-z] \)
• 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

$$c_1c_2c_3c_4c_5c_6c_7 \ldots$$

- 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

\[ R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots \]
\[ = R_1 + R_2 + \ldots \]
Regular Expressions => Lexical Spec. (3)

3. Let input be $x_1 \ldots x_n$
   For $1 \leq i \leq n$ check
   \[ x_1 \ldots x_i \in L(R) \]

4. If success, then we know that
   \[ x_1 \ldots x_i \in L(R_j) \text{ for some } j \]

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

• There are ambiguities in the algorithm

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

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

• Which token is used? What if
  • \(x_1 \ldots x_i \in L(R_j)\) and also
  • \(x_1 \ldots x_i \in L(R_k)\)
  \(R = R_1 + R_2 + R_3 + \ldots\)
  \(k \neq i\)
  Keyword = ‘if’ + ‘else’ + ...
  Identifier = letter (letter + digit)*

• Rule: use rule listed first (j if \(j < k\))
  - Treats “if” as a keyword, not an identifier
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
  \[ s_1 \xrightarrow{a} s_2 \]

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

- A state
- The start state
- An accepting state
- A transition
A Simple Example

- A finite automaton that accepts only “1”

- Accepts ‘1’ : \(1 \uparrow, \quad 1 \uparrow\)
- Rejects ‘0’ : \(0 \uparrow\)
- Rejects ’10’ : \(1 \uparrow, \quad 1 \uparrow 0\)
Another Simple Example

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

- Accepts ‘$110$’: $110, 110, 110, 110$
- Rejects ‘$100$’: $100, 100, 100, 100$
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

Rule: NFA accepts if it \textit{can} get to a final state
Acceptance of NFAs

• An NFA can get into multiple states

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

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

Input: 1 0

Possible States: \{A\}

Prof. Aiken
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

   ![Diagram of NFA]

• 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

NFA

DFA

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

• High-level sketch

- Regular expressions
- Lexical Specification
- Table-driven Implementation of DFA

NFA

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 Diagram](image-url)
NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state
  = $\epsilon$-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 $\epsilon$-moves as well
- Final states
  - Subsets that include at least one final state of NFA
\( \varepsilon \text{-closure of a state} \)

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

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

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

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

Prof. Aiken
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><strong>S</strong></td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td><strong>U</strong></td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Implementation (Cont.)

• NFA -> 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

- **State 0** (start)
  - Transition on `<` to **State 1**
  - Transition on `=` to **State 5**
  - Transition on `>` to **State 3**

- **State 1**
  - Transition on `=` to **State 2**
  - Transition on `>` to **State 3**
  - Transition on `>` to **State 8**

- **State 2**
  - Transition on `<=` (return(relop, <=))

- **State 3**
  - Transition on `<>` (return(relop, <>))

- **State 5**
  - Transition on `=` (return(relop, =))

- **State 6**
  - Transition on `=` to **State 7**

- **State 7**
  - Transition on `>=` (return(relop, >=))

- **State 8**
  - Transition on `>` (return(relop, >))

Other transitions:
- Transition on `<=` from **State 1**
- Transition on `<>` from **State 1**
- Transition on `<` from **State 1**
- Transition on `>=` from **State 6**
- Transition on `>` from **State 6**
- Transition on `>` from **State 5**
- Transition on `<` from **State 5**
- Transition on `>=` from **State 6**
- Transition on `>` from **State 6**
- Transition on `>` from **State 8**
- Transition on `<` from **State 8**

The DFA transitions on symbols `<=`, `<>`, `<`, `>=`, and `>` result in invoking `return(relop, ...)`. Other transitions label the corresponding operators.
DFA for recognizing id and keyword

returns either a keyword or an identifier based on the type of the token

Either returns pointer to the symbol table (if token is an id) or “0” (if token is a keyword)
DFA of Pascal Unsigned Numbers

return(num, install_num())
Lexical errors

• Some errors are out of power of lexical analyzer to recognize:
  \( f_i (a == f(x)) \) ...

• 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
Question?

Consider the string \textit{abbbaacc}. Which of the following lexical specifications produces the tokenization: \textit{ab/bb/a/acc}

Choose all that apply

\begin{itemize}
  \item $c^*$
  \item $a(b + c^*)$
  \item $ab$
  \item $b^+$
  \item $b^+$
  \item $b^+$
  \item $ab^*$
  \item $ac^*$
  \item $ac^*$
\end{itemize}
Using the lexical specification below, how is the string “dictatorial” tokenized?

Choose all that apply

- dict (1)
- dictator (2)
- [a-z]* (3)
- dictatorial (4)

- 1, 3
- 3
- 4
- 2, 3
Question?

Given the following lexical specification:

\[
\begin{align*}
    &a(ba)^* \\
    &b^*(ab)^* \\
    &abd \\
    &d^+
\end{align*}
\]

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
- dddabbababab will be tokenized as: ddd/a/bbabab
- ababddababa will be tokenized as: ab/abd/d/ababa
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
Question?

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