CS412/413
Introduction to Compilers
Radu Rugina

Lecture 4: Lexical Analyzers
30 Jan 06

Finite Automata
• Finite automata:
  – States, transitions between states
  – Initial state, set of final states
• DFA = deterministic
  – Each transition consumes an input character
  – Each transition is uniquely determined by the input character
• NFA = non-deterministic
  – ε-transitions, multiple transitions from the same state on the same input character

From RE to DFA
• Two steps:
  – Convert the regular expression to an NFA
  – Convert the resulting NFA to a DFA
• The generated DFAs may have a large number of states
• State Minimization = optimization that converts a DFA to another DFA that recognizes the same language and has a minimum number of states

State Minimization
• Example:
  – DFA1:
  – DFA2:
  – Both DFAs accept: b*a*b*a

State Minimization
• Step 1: Partition states of original DFA into maximal-sized groups of “equivalent” states
  \[ S = G_1 \cup \ldots \cup G_i \]
  \[
  \begin{array}{c}
  1 \\
  2 \\
  3 \\
  4 \\
  5 \\
  6
  \end{array}
  \]
  \[
  \begin{array}{c}
  a \\
  b \\
  a \\
  b \\
  a \\
  b
  \end{array}
  \]
  \[
  \begin{array}{c}
  1 \\
  2 \\
  3 \\
  4 \\
  5 \\
  6
  \end{array}
  \]
  \[
  \begin{array}{c}
  a \\
  b \\
  a \\
  b \\
  a \\
  b
  \end{array}
  \]

Optimized Acceptor
Regular Expression \( R \)
\[
\begin{array}{c}
\text{RE} \Rightarrow \text{NFA} \\
\text{NFA} \Rightarrow \text{DFA} \\
\text{Minimize DFA} \\
\text{DFA Simulation}
\end{array}
\]

Input String \( w \)
\[
\begin{array}{c}
\text{Yes, if } w \in L(R) \\
\text{No, if } w \notin L(R)
\end{array}
\]
Lexical Analyzers vs Acceptors

- Lexical analyzers use the same mechanism, but they:
  - Have multiple RE descriptions for multiple tokens
  - Return a sequence of matching tokens at the output
  - Always return the longest matching token
  - For multiple longest matching tokens use rule priorities

Lexical Analyzers

REs for Tokens $R_1, R_2, \ldots, R_n$  
Character Stream -> DFA -> Program -> Token stream (and errors)

Handling Multiple REs

- Combine the NFAs of all the regular expressions into a single finite automata

Lexical Analyzers

- Token stream at the output
  - Associate tokens with final states
  - Output the corresponding token when reaching a final state
- Longest match
  - When in a final state, look if there is a further transition; otherwise return the token for the current final state
- Rule priority
  - Same longest matching token when there is a final state corresponding to multiple tokens
  - Associate that final state to the token with the highest priority

Longest Matching Sequence

- Problem: lexer goes past a final state of a short token, but then doesn’t find a longer matching token
- Consider $R = 0 \mid 00 \mid 10 \mid 0011$ and input: 0010

Automating Lexical Analysis

- All of the lexical analysis process can be automated!
  - $RE \rightarrow NFA \rightarrow DFA \rightarrow$ Minimized DFA
  - Minimized DFA $\rightarrow$ Lexical Analyzer (DFA Simulation Program)

- We only need to specify:
  - Regular expressions for the tokens
  - Rule priorities for multiple longest match cases
Lexical Analyzer Generators

<table>
<thead>
<tr>
<th>REs for Tokens</th>
<th>JFlex</th>
<th>Laser</th>
<th>jJava</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Stream</td>
<td>program</td>
<td>Token stream (and errors)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JFlex Specification File

- **JFlex** = Lexical analyzer generator
  - written in Java
  - generates a Java lexical analyzer

- Has three parts:
  - Preamble, which contains package/import declarations
  - Definitions, which contains regular expression abbreviations
  - Regular expressions and actions, which contains:
    - the list of regular expressions for all the tokens
    - Corresponding actions for each token (Java code to be executed when the token is returned)

Example Specification File

```java
package FrontEnd;
import Error.LexicalError;
%
digits = 0|[1-9][0-9]*
letter = [a-zA-Z]
identifier = [letter][letter][0-9,]*
whitespace = [\t\n\r]*
{whitespace} { /* discard */ }
(digits) { return new Token(INT, Integer.valueOf(yytext())); }
'if' { return new Token(IF, null); }
'while' { return new Token(WHILE, null); }
(identifier) { return new Token(ID, yytext()); }
.
{ throw new LexicalError("illegal character"); }
```

Start States

- Mechanism that specifies state in which to start the execution of the DFA

- Define states in the second section
  - `STATE`

- Use states as prefixes of regular expressions in the third section:
  - `<STATE>` regex {action}

- Set current state in the actions
  - `yybegin(STATE)`

- There is a predefined initial state: YYINITIAL

Example

```
STATE COMMENT
%
<YYINITIAL> /\* \* \* yybegin(COMMENT); }
<COMMENT> /\* \* \* yybegin(YYINITIAL); }
<COMMENT> . ( }
```

Start States and REs

- The use of states allow the lexer to recognize more than regular expressions
  - Reason: the lexer can jump across different states in the semantic actions using `yybegin(STATE)`

- Example: nested comments
  - Increment a global variable on open parentheses and decrement it on close parentheses
  - When the variable gets to zero, jump to YYINITIAL
  - This models an infinite number of states
Conclusion

- The way lexical analyzers work:
  - Convert REs to NFA
  - Convert NFA to DFA
  - Minimize DFA
  - Use the minimized DFA to recognize tokens in the input
  - Use priorities, longest matching rule

- Lexical analyzer generators automate the process
  - Programmer writes regular expression descriptions of tokens
  - Automatically gets a lexical analyzer program that reads
    characters from the input stream and generates tokens