CS412/413

Introduction to Compilers
Radu Rugina

Lecture 4: Lexical Analyzers
27 Jan 03

Outline

• DFA state minimization
• Lexical analyzers
• Automating lexical analysis
• Jlex lexical analyzer generator

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
  – There may be e-transitions, which do not consume input characters
  – There may be multiple transitions from the same state on the same input character

From Regexp 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 which converts a DFA to another DFA which recognizes the same language and has a minimum number of states

State Minimization

• Example:
  – DFA1:
    ![Diagram]
  – DFA2:
    ![Diagram]
  – Both DFAs accept: b*ab*a

State Minimization

• Idea: find groups of equivalent states
  – all transitions from states in one group \( G_i \) go to states in the same group \( G_j \)
  – construct the minimized DFA such that there is one state for each group of states from the initial DFA
  – ![Diagram]
DFA Minimization Algorithm

Step 1: Construct a partition P of the set of states having two groups:
P = the set of final (accepting) states
S/P = the set of non-final states

Step 2:
Repeat
Let P = G₀ U ... U Gₚ, the current partition
Partition each group Gᵢ into subgroups:
Two states s and t are in the same subgroup Pᵢ, for each
symbol a there are transitions s → s' and t → t' and s', t'
belong to the same group Gᵢ
Combine all the computed subgroups into a new partition Pᵢ
Until P = Pᵢ
Step 3: Construct a DFA with one state for each group of states in the
final partition P

Optimized Acceptor

Regular Expression R
RE → NFA
NFA → DFA
Minimize DFA
Input String w
DFA Simulation
Yes, if w ∈ L(R)
No, if w ∉ L(R)

Lexical Analyzers vs Acceptors

- Lexical analyzers use the same mechanism, but they:
  - Have multiple RE descriptions for multiple tokens
  - Have a character stream at the input
  - Return a sequence of matching tokens at the output (or an error)
  - Always return the longest matching token
  - For multiple longest matching tokens use rule priorities

Lexical Analyzers

REs for Tokens R₁ ... Rᵣ
Character Stream program
DFA Simulation
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
**Issue**
- JLex tries to find the longest matching sequence
- **Problem:** what if the lexer goes past a final state of a shorter token, but then doesn’t find any other longer matching token later?
- Consider R = 00 | 10 | 0011 and input: 0010

![Diagram of automaton](image)

- We reach state 3 with no transition on input 0!
- **Solution:** record the last accepting state

**Automating Lexical Analysis**
- All of the lexical analysis process can be automated!
  - RE → NFA → DFA → Minimized DFA
  - Minimized DFA → 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**

**Lexical Specification File**
- JLex = 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 Parser;
Import Error, LexicalError;

%%
digits = 0(1-9)(0-9)*
letter = [A-Za-z]
identifier = (letter)((letter)|(0-9,)]*)*
whitespace = [\n\r]+
%%
(digits) ( return new Token(INT, Integer.valueOf(yptext())); )
*if*     ( return new Token(IF, null); )
*while*  ( return new Token(WHILE, null); )
(identifier) ( return new Token(ID, yptext()); )
.*      ( ErrorMsg.err("Illegal character"); )
```

**Start States**
- Mechanism which specifies in which state to start the execution of the DFA
- **Define states in the second section**
  - %state 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 pre-defined initial state: YYINITIAL
Example

```
%state STRING
%%
<YYINITIAL> "\f"  { return new Token(IF, null); }
<YYINITIAL> "\v"  { yybegin(STRING); }
<STRING>  "\v"  { yybegin(YYINITIAL); }
<STRING>  .    { } 
```

Start States and REs

- The use of states allow the lexer to recognize more than regular expressions (or DFAs)
  - 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
  - The global variable essentially models an infinite number of states!

Conclusion

- Regular expressions: concise way of specifying tokens
- Can convert RE to NFA, then to DFA, then to minimized DFA
- Use the minimized DFA to recognize tokens in the input stream
- Automate the process using lexical analyzer generators
  - Write regular expression descriptions of tokens
  - Automatically get a lexical analyzer program which identifies tokens from an input stream of characters