CS412/413

Introduction to Compilers
Tim Teitelbaum

Lecture 4: Lexical Analyzers
29 Jan 07

Outline

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

Finite Automata

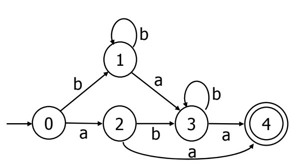
- Finite automata:
  - States, transitions between states
  - Initial state, set of final states
- DFA: Deterministic Finite Automaton
  - Each transition consumes an input character
  - Each transition is uniquely determined by the input character
- NFA: Non-deterministic Finite Automaton
  - ε-transitions, which do not consume input characters
  - Multiple transitions from the same state on the same input character

From RE to DFA

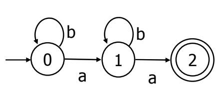
- Two steps:
  - Convert the regular expression to an NFA
  - Convert the resulting NFA to a DFA
- The generated DFAs may have a large number of states
- State Minimization is an optimization that converts a DFA to another DFA that recognizes the same language and has a minimum number of states

State Minimization

- Example:
  - DFA1:



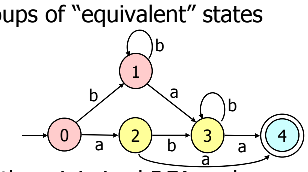
  - DFA2:



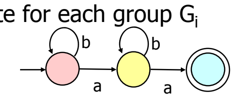
  - Both DFAs accept: b*ab*a

State Minimization

- Step1. Partition states of original DFA into maximal-sized groups of "equivalent" states $S = \{G_1, \ldots, G_n\}$

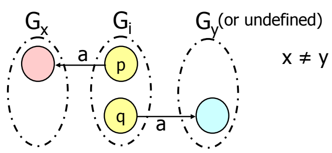


- Step 2. Construct the minimized DFA such that there is a state for each group G_i



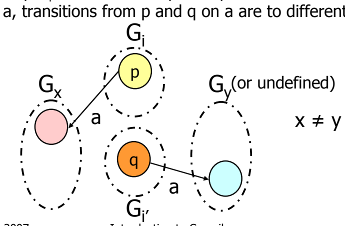
DFA Minimization

- Step1. Partition states of original DFA into maximal-sized groups of "equivalent" states
  - Step 1a. Discard states not reachable from start state
  - Step 1b. Initial partition is S = {Final, Non-final}
  - Step 1c. Repeatedly refine the partition $\{G_1,\ldots,G_n\}$ while some group G_i contains states p and q such that for some symbol a, transitions from p and q on a are to different groups

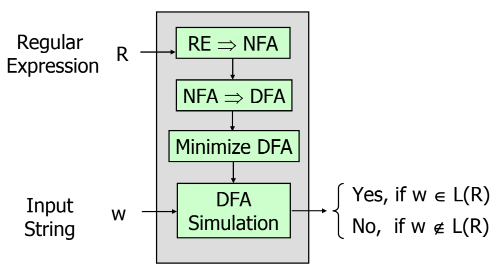


DFA Minimization

- Step1. Partition states of original DFA into maximal-sized groups of "equivalent" states
  - Step 1a. Discard states not reachable from start state
  - Step 1b. Initial partition is S = {Final, Non-final}
  - Step 1c. Repeatedly refine the partition $\{G_1,\ldots,G_n\}$ while some group G_i contains states p and q such that for some symbol a, transitions from p and q on a are to different groups



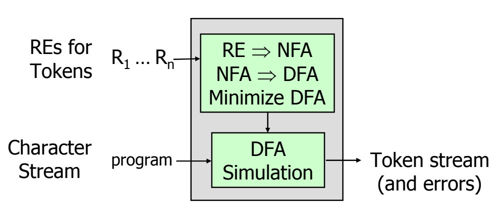
Optimized Acceptor



Lexical Analyzers vs Acceptors

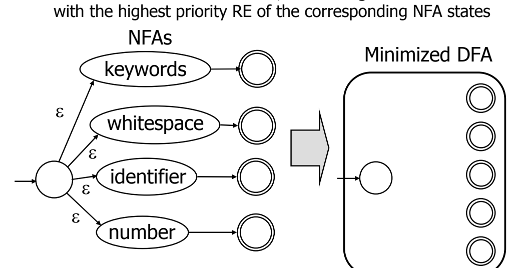
- Lexical analyzers use the same mechanism, but they:
  - Have multiple RE descriptions for multiple tokens
  - Output a sequence of matching tokens (or an error)
  - Always return the longest matching token
  - For multiple longest matching tokens, use rule priorities

Lexical Analyzers



Handling Multiple REs

- Construct one NFA for each RE
- Associate the final state of each NFA with the given RE
- Combine NFAs for all REs into one NFA
- Convert NFA to minimized DFA, associating each final DFA state with the highest priority RE of the corresponding NFA states

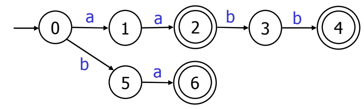


Scanning Algorithm

- Scan input and simulate DFA until no further transition is possible keeping track of most recently visited final state F
- Roll input back to position at the time F was entered
- Emit token associated with F
- For each successive token, scan remaining input and simulate DFA from the start state, i.e., scanner is "stateless" (NB. this is to be changed below.)

Example of Roll Back

Consider R = aa | ba | aabb and input: aaba

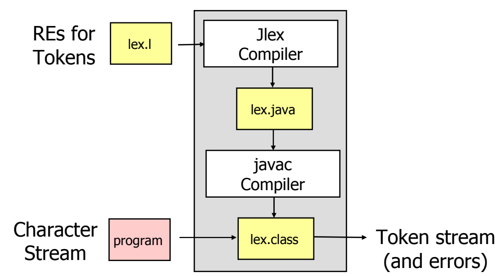


- Reach state 3 with no transition on next character a
- Roll input back to position on entering state 2 (i.e., having read aa)
- Emit token for aa

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



Jlex 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 recognized)

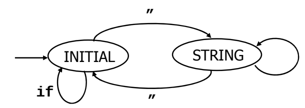
Example Specification File

```
Package Parse;
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()); }
.             { ErrorMsg.error("illegal character"); }
```

Start States

- Mechanism that specifies state in which to start the execution of the DFA
- Declare 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> "if"    { return new Token(IF, null); }
<YYINITIAL> "\""    { yybegin(STRING); … }
<STRING>    "\""    { yybegin(YYINITIAL); … }
<STRING>     .      { … }
```


Start States and REs

- The use of start states allows 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