NFA to DFA Conversion

- Construct the subsets of states the NFA could be in simultaneously
- Construct new transition function / table
- [optional but important in practice] Minimize resulting DFA

Algorithm

- Subset construction algorithm (aka. Büchi’s algorithm)

**Büchi’s Algorithm (aka subset construction)**

- Special operations: (T set of NFA states)
  - $\text{\varepsilon}$-closure(s) = set of NFA states reachable from NFA state $s$ on \(\varepsilon\)-transitions alone
  - $\text{\varepsilon}$-closure(T) = set of NFA states reachable from some NFA state $s$ in T on \(\varepsilon\)-transitions alone
  - $\text{move}(T,a)$ = set of NFA states to which there is a transition on input symbol $a$ from some NFA state $s$ in T
Büchi’s Algorithm

Initially, ε-closure(s₀) only state in Dstate, unmarked
while unmarked state T in Dstates do
    mark T;
    for each input symbol a do
        if ε-closure(move(T,a)) is not in Dstates then
            Dstates := Dstates ∪ ε-closure(move(T,a));
            Dtran[T,a] := U
    end
end

Büchi’s Algorithm - Example

NFA for (a|b)*abb

Dtran transition table

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lexical Analysis in Practice

Implementation alternatives:

- Generate NFA, convert to DFA and implement with transition tables
- Lazy DFA construction used in pattern matching tools (e.g., egrep)
- Simulate NFA directly
  - On-the-fly subset generation (algorithm 3.4 in Dragon book)
- Generate DFA directly
  - Cf. Dragon book
Tradeoffs
- \( r \) regular expression, \( x \) input string
- NFA
  - \( O(|r|) \) space
  - \( O(|r| \cdot |x|) \) time
- DFA
  - \( O(2^{|r|}) \) space, usually much less in practice
  - \( O(|x|) \) time

What actual tools do
- GNU flex
  - Builds NFA
  - Converts to DFA
  - Generates transition tables and driver (similar to figure 3.16 in Dragon Book)
- Important optimizations
  - State minimization ("equivalence classes")
  - Table compression

DFA State Minimization
- Idea
  - Merge indistinguishable states
  - \( s \) distinguished from \( t \) by string \( w \) \(<\to \)
    starting from \( s \ w \) leads to accepting state
    but \( t \ w \) to non-accepting state (or vice versa)
CS 1622 Lecture 5

**Jlex: a scanner generator**

- **Jlex specification**
  - `xxx.jlex`
  - `xxx.jlex.java`
- **Generated scanner**
  - `JLex.Main.java`
- **Output of P2.main**
  - `Yylex.class`
- **JLex**
  - `javac P2.main.java`
  - `Yylex.class`

---

**P2.java: or how to create & call the scanner**

```java
public class P2 {
    public static void main(String[] args) {
        FileReader inFile = new FileReader(args[0]);
        Yylex scanner = new Yylex(inFile);
        Symbol token = scanner.next_token();
        while (token.sym != sym.EOF) {
            switch (token.sym) {
                case sym.INTLITERAL:
                    System.out.println("INTLITERAL (" + ((IntLitTokenVal)token.value).intVal + ")");
                    break;
                ...
            }
            token = scanner.next_token;
        }
    }
}
```

---

**Jlex Specification file**

*`xxx.jlex`*

- **User code**
  - copied to `xxx.jlex.java`
  - use it to define auxiliary classes and methods.
- **%**
- **JLex directives**
  - macro definitions
  - use to specify what letters, digits, whitespace are.
- **%**
- **Regular expression rules**
  - specify how to divide up input into tokens.
  - regular expressions are followed by actions
    - print error messages, return token codes
    - no need to put characters back to input (done by Jlex)
**Regular expression rules**

```
regular-expression { action }
```

- **pattern to be matched**
- **code to be executed**

When `next_token()` method is called, it repeats:

- Find the longest sequence of characters in the input (starting with the current character) that matches a pattern.
- Perform the associated action (plus "consume the matched lexeme").
- Until a return in an action is executed.

---

**Matching rules**

- If several patterns that match the same sequence of characters, then the **longest** pattern is considered to be matched.
- If several patterns that match the same (longest) sequence of characters, then the **first such pattern** is considered to be matched.
- So the order of the patterns can be important!
- If an input character is not matched in any pattern, the scanner throws an **exception**
- Make sure that there can be no unmatched characters, otherwise the scanner will "crash" on bad input.

---

**Regular expressions**

- Similar to those discussed in class.
- Most characters match themselves:
  - `abc`
  - `==`
  - `while`
- Characters in quotes, including special characters, except `\`, match themselves:
  - `"a\b"` matches `a\b` not `a` or `b`
  - `"a\\\\\"tb"` matches `a"\"tb` not `a\\\\\"tb`
Regular-expression operators

- the traditional ones, plus the ? operator

<table>
<thead>
<tr>
<th></th>
<th>means &quot;or&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>means zero or more instances of</td>
</tr>
<tr>
<td>+</td>
<td>means one or more instances of</td>
</tr>
<tr>
<td>?</td>
<td>means zero or one instance of</td>
</tr>
<tr>
<td>()</td>
<td>are used for grouping</td>
</tr>
</tbody>
</table>

Backslash is special escape character:

- \n newline
- \t tab
- " double quote

To match a backslash character, put it in quotes

More operators

- ^ matches beginning of line
  ^main matches string "main" only when it appears at the beginning of line.

- $ matches end of line
  main$ matches string "main" only when it appears at the end of line.
  . (dot) matches any character except newline – usually used in the last rule of specification to match all "bad" characters
Character classes

- \[[abc]\] matches one character (either a or b or c)
- \[[a-z]\] matches any character between a and z, inclusive
- \[[^abc]\] matches any character except a, b, or c
- ^ has special meaning only at 1st position in [...]  
- \[[\t\n] matches tab or \n
[a bc] is equivalent to a|" \|b|c: white-space in char class and strings matches itself

JLex directives

- specified in the second part of xxx.jlex
- can also specify (see the manual for details)
  - the value to be returned on end-of-file
  - that line counting should be turned on, and
  - that the scanner will be used with the parser generator java cup.
- directives includes macro definitions (very useful):
  - name = regular-expression
  - name is any valid Java identifier
  - DIGIT= [0-9]
  - LETTER= [a-zA-Z]
  - WHITESPACE= [ \t\n]
  - To use a macro, use its name inside curly braces.
    - {LETTER}{LETTER}{DIGIT}*

Comments

- You can include comments in the first and second parts of your JLex specification,
  - in the third part, JLex would think your comments are part of a pattern.
  - use Java comments // ...
A Small Example

```
\% DIGIT= [0-9]
\% LETTER= [a-zA-Z]
\% WHITESPACE= \s+ // space, tab, newline
\% for compatibility with Java CUP
\%implements java_cup.runtime.Scanner
\%function next_token
\%type java_cup.runtime.Symbol
// Turn on line counting
\%
```

Continued

```
\% (LETTER)|(LETTER)|(DIGIT)]* 
  {System.out.println(yyline+1 
  + ": ID = " + yytext());}
\% DIGIT= {System.out.println(yyline+1 + ";
    INT");}
\% "=" {System.out.println(yyline+1 + ";
     ASSIGN");}
\% "==" {System.out.println(yyline+1 + ";
     EQUALS");}
\% WHITESPACE+ { System.out.println(yyline+1 + ": bad
    char");}
```

Another example ( from simple.jlex)

```
\%DIGIT= { int val = (new Integer(yytext())).intValue();
    Symbol S = new Symbol(sym.INTLITERAL,
    new IntLitTokenVal(yyline+1, CharNum.num, 
    val));
    CharNum.num += yytext().length();
    return S;
}
\%WHITESPACE+ {CharNum.num += 
    yytext().length();}
```