CS480
Translators
More Lexical Analysis
Chap. 3
package lexer;       // File Lexer.java
import java.io.*; import java.util.*;
public class Lexer {
    public int line = 1;
    private char peek = ' ';
    private Hashtable words = new Hashtable();
    void reserve(Word t) { words.put(t.lexeme, t); }
    public Lexer() {
        reserve( new Word(Tag.TRUE, "true") );
        reserve( new Word(Tag.FALSE, "false") );
    }
    public Token scan() throws IOException {
        for( ; ; peek = (char)System.in.read() ) {
            if( peek == ' ' || peek == '
' ) continue;
            else if( peek == '\n' ) line = line + 1;
            else break;
        }
        /* continues in Fig. 2.35 */
    }

    Figure 2.34: Code for a lexical analyzer, part 1 of 2
if( Character.isDigit(peek) ) {
    int v = 0;
    do {
        v = 10*v + Character.digit(peek, 10);
        peek = (char)System.in.read();
    } while( Character.isDigit(peek) );
    return new Num(v);
}

if( Character.isLetter(peek) ) {
    StringBuffer b = new StringBuffer();
    do {
        b.append(peek);
        peek = (char)System.in.read();
    } while( Character.isLetterOrDigit(peek) );
    String s = b.toString();
    Word w = (Word)words.get(s);
    if( w != null ) return w;
    w = new Word(Tag.ID, s);
    words.put(s, w);
    return w;
}

    Token t = new Token(peek);
    peek = ",
    return t;
}

Figure 2.35: Code for a lexical analyzer, part 2 of 2
Job of a Tokenizer...

```c
if( peek == '\n' ) line = line + 1;
```

Lexical Analyzer

```
(if () (id, "peek") (eq) (const, '\n') ())
(id, "line") (assign) (id, "line") (+) (num, 1) (;)
```
Patterns/Regular Expressions

- Defining tokens
  - if while assign
  - ( ) = >
  - \[a-zA-Z][a-zA-Z0-9]*
  - [0-9]+
  - [ \\
  \n]+
### Languages/Regular Expressions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition and Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of ( L ) and ( M )</td>
<td>( L \cup M = { s \mid s \text{ is in } L \text{ or } s \text{ is in } M } )</td>
</tr>
<tr>
<td>Concatenation of ( L ) and ( M )</td>
<td>( LM = { st \mid s \text{ is in } L \text{ and } t \text{ is in } M } )</td>
</tr>
<tr>
<td>Kleene closure of ( L )</td>
<td>( L^* = \bigcup_{i=0}^{\infty} L^i )</td>
</tr>
<tr>
<td>Positive closure of ( L )</td>
<td>( L^+ = \bigcup_{i=1}^{\infty} L^i )</td>
</tr>
</tbody>
</table>

- Every symbol of \( \Sigma \) is a regular expression
- \( \varepsilon \) is a regular expression
- if \( r_1 \) and \( r_2 \) are regular expressions, so are
  - \((r_1) \ r_1r_2 \ r_1 \mid r_2 \ r_1^*\)
- Nothing else is a regular expression.
Languages/Regular Expressions

- **Given an alphabet** $\Sigma$, **the regular expressions over** $\Sigma$ **and their corresponding regular languages are**
  a) $\varepsilon$, the empty string, denotes the language \{\varepsilon\}.
  b) for each $a$ in $\Sigma$, $a$ denotes \{a\} --- a language with one string.
  c) if $R$ denotes $L_R$ and $S$ denotes $L_S$ then $R | S$ denotes the language $L_R \cup L_S$, i.e, \{ $x$ | $x \in L_R$ or $x \in L_S$ \}.
  d) if $R$ denotes $L_R$ and $S$ denotes $L_S$ then $RS$ denotes the language $L_RL_S$, that is, \{ $xy$ | $x \in L_R$ and $y \in L_S$ \}.
  e) if $R$ denotes $L_R$ then $R^*$ denotes the language $L_R^*$ where $L^*$ is the union of all $L^i (i=0,\ldots,\infty)$ and $L^i$ is just \{$x_1x_2\ldots x_i | x_1 \in L, \ldots, x_i \in L$\}.
  f) if $R$ denotes $L_R$ then $(R)$ denotes the same language $L_R$. 
Implementing Regular Expressions

• Build finite automata for all patterns
• Connect start states for NDFA
• Simplify to make DFA
• Algorithm:
  – When asked for token, start in combined state
  – Read characters, advancing state, until cannot advance further
  – If needed, push back last character(s)
  Return token associate with last state
Finite State Automata

• A recognizer for a language is a program that takes a string $x$ as an input and answers "yes" if $x$ is a sentence of the language and "no" otherwise.
• One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton.
• A finite automaton can be non-deterministic or deterministic.
• Both automata are capable of recognizing regular expressions.
Transition Diagram

• Flowchart with **states** and **edges**; each edge is labeled with characters; certain subset of states are marked as “**final states**”

• Transition from state to state proceeds along edges according to the next **input character**

• Every string that ends up at a **final state** is accepted

• If get “stuck”, there is no transition for a given character, it is an **error**
\[ relop \rightarrow < \mid > \mid \leq \mid \geq \mid = \mid <> \]
\textit{relop} → < | > | <= | >= | = | <>

```c
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
             or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                if ( c == '<' ) state = 1;
                 else if ( c == '=' ) state = 5;
                 else if ( c == '>' ) state = 6;
                 else fail(); /* lexeme is not a relop */
                break;
            case 1: ...
            ...
            case 8: retract();
                retToken.attribute = GT;
                return(retToken);
        }
    }
}
```
Collection of Keywords

OSU Oregon State University
Quiz #3

• **Tokenize** the following C statement:

```c
float limitedSquare(x) float x; {
    /*returns x-squared, but never more than 100*/
    return (x<=-10.0 || x>=10.0)?100:x*x;
}
```

• Given Σ = {a,b}, provide regular expressions for languages below:
  – all strings beginning and ending in a
  – all strings of a’s and b’s of even length
  – all strings with an odd number of a’s
  – string of zero or more a’s followed by same number of b’s