CS480
Translators

Introduction to Lexical Analysis
Chap. 3
1) package lexer;           // File Lexer.java
2) import java.io.*; import java.util.*;
3) public class Lexer {
4)    public int line = 1;
5)    private char peek = ' ';
6)    private Hashtable words = new Hashtable();
7)    void reserve(Word t) { words.put(t.lexeme, t); }
8)    public Lexer() {
9)        reserve( new Word(Tag.TRUE, "true") );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)        for( ; ; peek = (char)System.in.read() ) {
14)            if( peek == ' ' || peek == '	' ) continue;
15)            else if( peek == '
' ) line = line + 1;
16)            else break;
17)        }
18)    
19)    /* 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 = ' ';
Job of a Tokenizer...

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

![Diagram of lexical analyzer and tokenized code]

```plaintext
<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]+
  – [ \t\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>$L \cup M = {s \mid s$ is in $L$ or $s$ is in $M}$</td>
</tr>
<tr>
<td>Concatenation of $L$ and $M$</td>
<td>$LM = {st \mid s$ is in $L$ and $t$ 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 $(r_1)$ $r_1r_2$ $r_1 | r_2$ $r_1^*$
- 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 $R \mid S$ denotes the language $L_R \cup L_S$, i.e, $\{x \mid x \in L_R \text{ or } x \in L_S\}$.
d) if $R$ denotes $L_R$ and $S$ denotes $L_S$ then $RS$ denotes the language $L_R L_S$, that is, $\{xy \mid x \in L_R \text{ and } y \in L_S\}$.
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 $\{x_1x_2\ldots x_i \mid x_1 \in L, \ldots, x_i \in L\}$.
f) if $R$ denotes $L_R$ then $(R)$ denotes the same language $L_R$.  

OSU Oregon State University
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 → < | > | <= | >= | = | <>

Diagram:

- **Start**: The process begins at node 0.
- **Node 1**: Decision point for '<'.
- **Node 2**: Returns if '<' is true.
- **Node 3**: Returns if '<' is false.
- **Node 4**: Returns if '=' is true.
- **Node 5**: Returns if '=' is false.
- **Node 6**: Decision point for '>'.
- **Node 7**: Returns if '>' is true.
- **Node 8**: Returns if '>' is false.
- Other conditions:
  - If 'other' conditions are met, return (relOp, value)

Nodes 1, 6, and 8 are decision points directing to appropriate return nodes.
relop → < | > | <= | >= | = | <>

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