CS480
Translators

Introduction to Lexical Analysis
Chap. 3
The Scanner

```
for ( ; ; peek = next input character ) {
    if ( peek is a blank or a tab ) do nothing;
    else if ( peek is a newline ) line = line+1;
    else break;
}
```

Figure 2.29: Skipping white space

- What is the purpose of line?
- What is the purpose of peek?
Reading Ahead

• Read the next char, it is an “i”
• Could be int, if, or an identifier, so read next char, “f”
• Could be if, could still be an identifier, so read next char, “(”
• Oops, we’ve gone too far, push back “(”
Buffers

• Why is this important?
• Ways to implement:
  – Two pointers into buffer (start_char, look_ahead)
  – Push back buffer (peek)
The Lexical Analyzer

```c
if ( peek holds a digit ) {
    v = 0;
    do {
        v = v * 10 + integer value of digit peek;
        peek = next input character;
    } while ( peek holds a digit );
    return token ⟨num, v⟩;
}
```

Figure 2.30: Grouping digits into integers
Keywords vs. Identifiers

• count = count + increment;

  <id, “count”> <=> <id, “count”> <+> <id, “increment”> <;>

• How do we know count is an id vs. keyword?
• Why use a hash table?
• What is in the hash table?
How to distinguish words?

```java
if ( peek holds a letter ) {
    collect letters or digits into a buffer b;
    s = string formed from the characters in b;
    w = token returned by words.get(s);
    if ( w is not null ) return w;
    else {
        Enter the key-value pair (s, ⟨id, s⟩) into words
        return token ⟨id, s⟩;
    }
}
```

Figure 2.31: Distinguishing keywords from identifiers
public class Token {
    public final int tag;
    public Token(int t) { tag = t; }
}

Figure 2.32: Class Token and subclasses Num and Word
Numbers vs. Words

1) package lexer; // File Num.java
2) public class Num extends Token {
3)     public final int value;
4)     public Num(int v) { super(Tag.NUM); value = v; }
5) }

1) package lexer; // File Word.java
2) public class Word extends Token {
3)     public final String lexeme;
4)     public Word(int t, String s) {
5)         super(t); lexeme = new String(s);
6)     }
7) }

Figure 2.33: Subclasses Num and Word of Token
Token Data Structures

```c
struct token_t {
    int tag;
    union {
        char *lexeme;
        int value;
    } val;
};
```

```c
struct token_t {
    int tag;
    OR
    int tag;
    void *val;
};
```
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;
}
Job of a Tokenizer...

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

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

Lexical Analyzer
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 \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 $R \mid S$ 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 $L_RL_S$, 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 $L_R$. 

16