CS321 Languages and Compiler Design I
Fall 2010
Lecture 4

**Lexical Analysis**

- Convert source file characters into **token stream**.
- Remove content-free characters (comments, whitespace, ...)
- Detect lexical errors (badly-formed literals, illegal characters, ...)
- Output of lexical analysis is input to syntax analysis.
- Could just do lexical analysis as part of syntax analysis.
- But choose to handle separately for better modularity and portability, and to allow make syntax analysis easier.

Idea: Look for **patterns** in input character sequence, convert to **tokens** with **attributes**, and pass them to parser in **stream**.

---

### Lexical Analysis Example

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Token</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>PRINT</td>
<td></td>
</tr>
<tr>
<td>then</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>ASSIGN</td>
<td></td>
</tr>
<tr>
<td>letter followed by letters or digits</td>
<td>RELOP</td>
<td>enum</td>
</tr>
<tr>
<td>digits</td>
<td>ID</td>
<td>symbol</td>
</tr>
<tr>
<td>chars between double quotes</td>
<td>NUM</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>STRING</td>
<td>string</td>
</tr>
</tbody>
</table>

**Source code:**

```plaintext
if x > 17 then count := 2
else (* oops !*) print "bad!"
```

**Lexeme** | **Token** | **Attribute** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>ID</td>
<td>&quot;x&quot;</td>
</tr>
<tr>
<td>&gt;</td>
<td>RELOP</td>
<td>GT</td>
</tr>
<tr>
<td>17</td>
<td>NUM</td>
<td>17</td>
</tr>
<tr>
<td>then</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>ID</td>
<td>&quot;count&quot;</td>
</tr>
<tr>
<td>:=</td>
<td>ASSIGN</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>NUM</td>
<td>2</td>
</tr>
<tr>
<td>else</td>
<td>ELSE</td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>PRINT</td>
<td></td>
</tr>
<tr>
<td>&quot;bad!&quot;</td>
<td>STRING</td>
<td>&quot;bad!&quot;</td>
</tr>
</tbody>
</table>
A **token** describes a class of character strings with some distinguished meaning in language.  
- May describe **unique** string (e.g., IF, ASSIGN)  
- or set of possible strings, in which case an **attribute** is needed to indicate which.  

(Tokens are typically represented as elements of an **enumeration**.)  

A **lexeme** is the string in the input that actually matched the pattern for some token.  

**Attributes** represent lexemes converted to a more useful form, e.g.,:  
- strings  
- symbols (like strings, but perhaps handled separately)  
- numbers (integers, reals, ...)  
- enumerations  

**Whitespace** (spaces, tabs, new lines, ...) and **comments** usually just disappear (unless they affect program meaning).

---

**Hand-coded Scanner (in Pseudo-Java)**

```java
Token getToken() {
    while (true) {
        char c = read();
        if (c is whitespace)
            ignore it;
        else if (c is digit) {
            int n = 0;
            do {n = n * 10 + (c- '0');
                c = read();} 
            until (c not a digit);
            unread(c);
            return new Token(NUM,n);
        } else if (c is alpha) {
            String s = "";
            do {s = s + c;
                c = read();} 
            until (c is not an alphanumeric);
            unread(c);
            return new Token(ID,S);
        } else ... }
}
```

---

**Pros and Cons of Hand-coded Scanners**

Efficient!  
But easy to get wrong!  
Note intermixed code for input, output, patterns, conversion.  
Hard to specify! (esp. **patterns**).
FORMALIZING PATTERN DESCRIPTIONS

Ex.：“An identifier is a letter followed by any number of letters or digits.”

• Exactly what is a letter?
  
  \[ \text{LETTER} \rightarrow \text{a} | \text{b} | \text{c} | \text{d} | \text{e} | \text{f} | \text{g} | \text{h} | \text{i} | \text{j} | \text{k} | \text{l} | \text{m} | \text{n} | \text{o} | \text{p} | \text{q} | \text{r} | \text{s} | \text{t} | \text{u} | \text{v} | \text{w} | \text{x} | \text{y} | \text{z} \]

• Exactly what is a digit?
  
  \[ \text{DIGIT} \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

• How can we express “letters or digits”?
  
  \[ \text{LORDS} \rightarrow \text{LETTER} \mid \text{DIGIT} \]

• How can we express “any number of”?
  
  \[ \text{LORDS} \rightarrow \text{LORDS}^* \]

• How can we express “followed by”?
  
  \[ \text{IDENTIFIER} \rightarrow \text{LETTER} \text{LORDS} \]

LANGUAGES: SOME PRELIMINARY DEFINITIONS

• An alphabet is a set of symbols (e.g., the ASCII character set).

• A language over an alphabet is a set of strings of symbols from that alphabet.

• We write \( \emptyset \) for the empty string (containing zero characters); some authors use \( \lambda \) instead.

• If \( x \) and \( y \) are strings, then the concatenation \( xy \) is the string consisting of the the characters of \( x \) followed by the characters of \( y \).

• If \( L \) and \( M \) are languages, then their concatenation \( LM = \{ xy \mid x \in L, y \in M \} \).

• The exponentiation of a language \( L \) is defined thus: \( L^0 = \{ \epsilon \} \), the language containing just the empty string, and \( L^i = L^{i-1}L \) for \( i > 0 \).

REGULAR EXPRESSIONS

A regular expression (R.E.) is a concise formal characterization of a regular language (or regular set).

Example: The regular language containing all IDENTs is described by the regular expression

\[ \text{LETTER} (\text{LETTER} \mid \text{DIGIT} )^* \]

where “|” means “or” and “\( \ast \)" means “zero or more copies of \( \epsilon \).”

Regular languages are one particular kind of formal languages.

REGULAR EXPRESSIONS AND LANGUAGES

Each R.E. over an alphabet \( \Sigma \) denotes a regular language over \( \Sigma \), according to the following inductive definition:

Base rules:

• The R.E. \( \epsilon \) denotes \( \{ \epsilon \} \).

• For each \( a \in \Sigma \), the R.E. \( a \) denotes \( \{ a \} \), the language containing the single string containing just \( a \).

Inductive rules: If the R.E. \( R \) denotes \( L_R \) and the R.E. \( S \) denotes \( L_S \), then

• \( R \mid S \) denotes \( L_R \cup L_S \).

• \( R \cdot S \) (or just \( RS \)) denotes \( L_R L_S \).

• \( R^\ast \) denotes \( L_R^\ast = \bigcup_{i=0}^{\infty} L_i \), the “Kleene closure” (the concatenation of zero or more strings from \( L_R \)).

Also: \( (R) \) denotes \( L_R \).

Precedence rules: \( () \) before \( \ast \) before \( \cdot \) before \( | \).
**Examples** (over alphabet \{a, b\})

- \(a^*\): zero or more a’s
- \((a | b)^*\): all strings of a’s and b’s of length \(\geq 0\)
- \((a^*b^*)^*\): ditto
- \((aa | ab | ba | bb)^*\): all strings of a’s and b’s of even length

**Counterexamples** (Not every language is regular!)

- \{a^n b^n | n \geq 0\}
- Set of strings over \{(, )\} such that parentheses are properly matched.

Implication: regular languages can’t be used to describe arithmetic expressions.

**R.E.'s are everywhere in command-line programming tools**

grep, Perl, shell commands, etc.

---

**Specify Lexical Analyzers**

Can define lexical analyzer via list of pairs:

- **regular expression, action**

where regular expression describes token pattern (maybe using auxiliary regular definitions),

and action is a piece of code, parameterized by the matching lexeme, that returns a (token,attribute) pair.

**Example**

```plaintext
(digit^+, {return new Token(NUM,parseInt(lexeme));})
(alpha(alpha|digit)^*, {return new Token(ID,lexeme);})
(space|tab|newline, {})
(,,)
(,,)
(,,)
```

So R.E.'s can help us specify scanners.

But can they help us generate running code that does pattern matching?