Lexical Analysis
Part I

Chapter 3: Regular Expressions

Slides adapted from:
© Robert van Engelen, Florida State University
Input Stream

• Example:

```java
if (i == j)
    z = 0;
else
    z = 1;
```

• The input is just a string of characters:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```
Preliminary Operations

- Remove comments by matching special markers
- Remove white spaces (separators)
- Trace line numbers (for error messages)
Interaction with Parser

Source Program

Lexical Analyzer

Parser

Symbol Table

Get next token

error
The Reason Why Lexical Analysis is a Separate Phase

• Simplifies the design of the compiler
  – LL(1) or LR(1) parsing with 1 token lookahead would not be difficult

• Provides efficient implementation
  – Systematic techniques to implement lexical analyzers by hand or automatically from specifications

• Portability
  – encapsulate language peculiarities into scanner
Bad Language Choices

• Whitespace is insignificant in Fortran

- \textbf{DO 5 I = 1,25}

- \textbf{DO 5 I = 1.25}

“run code between here and line 5 a total of 25 times, setting ‘I’ to values from 1 to 25”

“set the variable named ‘DO5I’ to have the value 1.25”
Attributes of Tokens

\[ y = 31 + 28 \times x \]

- **Lexical analyzer**
- **Parser**

(token, name, attribute)
Tokens, Patterns, and Lexemes

• **Tokens** are pairs consisting of a *token name* (a classification label) and an optional *attribute*
  – Example: ⟨id, "x"⟩, ⟨num, 28⟩

• **Lexemes** are the specific character strings that make up a token
  – Example: abc, 123

• **Patterns** are rules describing the set of lexemes belonging to a token name
  – Example: “*letter followed by letters and digits*” and “*non-empty sequence of digits*”
Most Common Token Names

- One token name for each keyword
- Token names for individual operators or classes of operators (e.g. `comparison`)
- One token name for all identifiers (id)
- One or more token names for constants (number, literal, …)
- Token names for punctuation symbols
## Examples

<table>
<thead>
<tr>
<th>Token</th>
<th>Pattern</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>characters i, f</td>
<td>if</td>
</tr>
<tr>
<td>else</td>
<td>characters e, l, s, e</td>
<td>else</td>
</tr>
<tr>
<td>comparison</td>
<td>&lt;, &gt;, &lt;=, ==, …</td>
<td>&lt;,&lt;=,…</td>
</tr>
<tr>
<td>id</td>
<td>Letter followed by letters and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric constant</td>
<td>3.14, 0,…</td>
</tr>
<tr>
<td>literal</td>
<td>Anything but “surrounded by “</td>
<td>&quot;core&quot;</td>
</tr>
</tbody>
</table>
Exercise

• For the code fragment

\[
\begin{align*}
x &= 0; \\
\text{while} & \quad (x < 10) \quad \{ \\
\text{tx} &= +; \\
\}
\end{align*}
\]

count the number of tokens in each class:
whitespace, keyword, identifier, number, other (\{ \} \( \) < ++ ; =)
Lexical Errors

• Related to assumptions in the design of the programming language

• Example:

  \[
  \text{if (x === 10) ...}
  \]

  – If white spaces are optional, should be scanned as \langle\text{relOp, ==}, \langle\text{Op, =}\rangle (if we always take the longest match, as explained later)

  – If white spaces are mandatory, should be scanned as an error, since \text{===} does not exist in the language
Lexical Errors

• Hard to detect some lexical errors without aid of other components

• Example:

  \[
  \text{fi } (x \equiv 10) \ldots
  \]

  need syntactic analysis to detect keyword error
Lexical Errors

- When scanner is unable to proceed, use error-recovery actions
  - Delete successive characters (panic mode)
  - Insert missing character
  - Replace character
  - Switch two characters
Input Buffering

- Involves two buffers that are alternatively reloaded
  - Pointer `lexemeBegin` marks the beginning of the current lexeme
  - Pointer `forward` scans ahead until a pattern match is found, and is retracted afterward
Language Definitions

• An **alphabet** $\Sigma$ is a finite set of symbols (characters)

• A **string** $s$ is a finite sequence of symbols from $\Sigma$
  – $|s|$ denotes the length of string $s$
  – $\varepsilon$ denotes the empty string, thus $|\varepsilon| = 0$

• A **language** is a specific set of strings over some fixed alphabet $\Sigma$
String Operations

• The **concatenation** of two strings $x$ and $y$ is denoted by $xy$

• The null element for concatenation is $\varepsilon$
  
  – We have $s\varepsilon = \varepsilon s = s$

• The **exponentiation** of a string $s$ is defined by

  $s^0 = \varepsilon$

  $s^i = s^{i-1}s$ for $i > 0$
Language Operations

• Union
  \[ L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \]

• Concatenation
  \[ LM = \{ xy \mid x \in L, y \in M \} \]

• Exponentiation
  \[ L^0 = \{ \varepsilon \}; \quad L^i = L^{i-1}L, \ i > 0 \]

• Kleene closure
  \[ L^* = \bigcup_{i \geq 0} L^i \]

• Positive closure
  \[ L^+ = \bigcup_{i \geq 1} L^i \]
Regular Expressions

• Basis symbols:
  – $\varepsilon$ is a regular expression denoting language $\{\varepsilon\}$
  – $a \in \Sigma$ is a regular expression denoting $\{a\}$

• If $r$ and $s$ are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  – $r \mid s$ is a regular expression denoting $L(r) \cup M(s)$
  – $rs$ is a regular expression denoting $L(r)M(s)$
  – $r^*$ is a regular expression denoting $L(r)^*$
  – $(r)$ is a regular expression denoting $L(r)$

• A language defined by a regular expression is called a regular set
Regular Expressions

- Example: \( r = ab | c, \ L(r) = ?? \)

\[
\begin{align*}
a & \quad L(a) = \{a\} \\
b & \quad L(b) = \{b\} \\
ab & \quad L(ab) = L(a)L(b) = \{ab\} \\
c & \quad L(c) = \{c\} \\
ab | c & \quad L(ab | c) = L(ab) \cup L(c) \\
& = \{ab\} \cup \{c\} = \{ab, c\}
\end{align*}
\]
Regular Expressions

- Example: \( r = a(b | c) \), \( L(r) = \ ?? \)

<table>
<thead>
<tr>
<th>Character</th>
<th>( L() )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( b )</td>
<td>( {b} )</td>
</tr>
<tr>
<td>( c )</td>
<td>( {c} )</td>
</tr>
<tr>
<td>( b</td>
<td>c )</td>
</tr>
<tr>
<td>( (b</td>
<td>c) )</td>
</tr>
<tr>
<td>( a )</td>
<td>( {a} )</td>
</tr>
<tr>
<td>( a(b</td>
<td>c) )</td>
</tr>
<tr>
<td></td>
<td>( = {a} \ {b, c} = {ab, ac} )</td>
</tr>
</tbody>
</table>
Regular Expressions

ab,
aεb,
εaεb,
εaεb,
εaεεb

All the above regular expressions are equivalent, i.e., they generate the same language
Regular Definitions

- **Regular definitions** introduce a naming convention:
  
  \[d_1 \rightarrow r_1\]
  \[d_2 \rightarrow r_2\]
  \[\ldots\]
  \[d_n \rightarrow r_n\]

  where each \(r_i\) is a regular expression over
  \(\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}\)

- Any \(d_j\) in \(r_i\) can be textually substituted in \(r_i\) to obtain an equivalent set of definitions
Regular Definitions

• Example:

\[
\begin{align*}
\text{letter} & \rightarrow A \mid B \mid \cdots \mid Z \mid a \mid b \mid \cdots \mid z \\
\text{digit} & \rightarrow 0 \mid 1 \mid \cdots \mid 9 \\
\text{id} & \rightarrow \text{letter} ( \text{letter} \mid \text{digit} )^* \\
\end{align*}
\]

• Regular definitions are not recursive:

\[
\begin{align*}
\text{digits} & \rightarrow \text{digit} \ \text{digits} \mid \text{digit} \quad \text{wrong!}
\end{align*}
\]
Notational Shorthand

• The following short-hands are often used:
  \[ r^+ = rr^* \]
  \[ r? = r \mid \varepsilon \]
  \[ [r_1 \ r_2 \ \ldots \ r_q] = r_1 \mid r_2 \mid \ldots \mid r_q \]
  \[ [a-z] = a \mid b \mid c \mid \ldots \mid z \]

• Example:
  \[ \text{digit} \rightarrow [0-9] \]
  \[ \text{num} \rightarrow (+ \mid -)? \ \text{digit}^+ (\ . \ \text{digit}^+)? (\ E \ (+ \mid -)? \ \text{digit}^+ )? \]
Example: Toy Grammar

Grammar

\[ stmt \rightarrow \text{if } expr \text{ then } stmt \]
\[ \quad \rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \]
\[ \quad \rightarrow \varepsilon \]

\[ expr \rightarrow term \text{ relop } term \]
\[ \quad \rightarrow term \]

\[ term \rightarrow \text{id} \]
\[ \quad \rightarrow \text{number} \]
Example: Recognition of Tokens

Regular definitions

\[
\begin{align*}
digit & \rightarrow [0-9] \\
digits & \rightarrow digit^+ \\
number & \rightarrow digits (. digits)? (E [+-]? digits )? \\
letter & \rightarrow [A-Za-z] \\
id & \rightarrow letter ( letter | digit )* 
\end{align*}
\]
Example: Recognition of Tokens

Regular definitions (cont’d)

if → if
then → then
else → else
relop → < | <= | <> | > | >= | =
blank → ’ ’
tab → \t
newline → \n
ws → (blank | tab | newline)^+
Exercise

• Choose the regular languages that are equivalent to $(0 \mid 1)^*1(0 \mid 1)^*$

  – $(01 \mid 11)^*(0 \mid 1)^*$
  – $(0 \mid 1)^*(10 \mid 11 \mid 1)(0 \mid 1)^*$
  – $(1 \mid 0)^*1(1 \mid 0)^*$
  – $(0 \mid 1)^*(0 \mid 1)(0 \mid 1)^*$
Exercise

• Twelve-hour times of the form “04:13PM”. Minutes should always be a two digit number, but hours *may* be a single digit.
Exercise (cont’d)

• Correct or wrong?
  – (0 | 1)?[0-9]:[0-5][0-9](AM | PM)
  – ((0 | ε)[0-9] | 1[0-2]):[0-5][0-9](AM | PM)
  – (0*[0-9] | 1[0-2]):[0-5][0-9](AM | PM)
  – (0?[0-9] | 1(0 | 1 | 2):[0-5][0-9](A | P)M
Transition Diagrams

• Transition diagrams represent regular expressions/definitions by means of:
  – *States*: summarizing all we need to know about characters in between `lexemeBegin` pointer and `forward` pointer
  – *Edges*: labeled transitions from state to state
  – *Initial* state and set of *final* states
  – Operator * to retract `forward` pointer
Transition Diagrams

\[ \text{relop} \rightarrow < \mid \leq \mid \neq \mid > \mid \geq \mid = \]
Transition Diagrams

number → digits ( . digits)? (E [+-]? digits )?
Reserved Words

• **id** recognizes also language **reserved words**: if, else, …

• We can distinguish reserved words from **id** by introducing matching precedence

• Two possible solutions
Reserved Words

• Solution 1
  – Install all reserved words in symbol table initially
  – Use return to implement precedence

\[ \text{id} \rightarrow \text{letter} \ (\ \text{letter} \ | \ \text{digit})^* \]
Reserved Words

• Solution 2
  – Create transition diagrams for each reserved word
  – Must prioritize tokens
  – In case of multiple matches for the same lexeme, use priority
Architecture of Scanner Based on Transition-Diagrams

- Transition diagrams for each token to be tried sequentially
  - Exit at the first match
  - `fail()` moves to next diagram
- Run transition diagrams in parallel
  - Take longest match
  - Use priority in case of tie
Architecture (cont’d)

• Combine transition diagrams into one
  – Same strategy as for parallel
  – Construction to be specified later
Exercise (difficult)

- Scan comments
  - Wrapped by /* * /
  - Allow strings wrapped by pairs of "
  - Ignore comment markers inside string

\[
\begin{align*}
\text{letter} & \rightarrow [A-Za-z] \\
\text{digit} & \rightarrow [0-9] \\
\ldots \\
\text{string} & \rightarrow "(\text{letter} \mid \text{digit} \mid \ldots)^*" \\
\ldots
\end{align*}
\]
Coding Transition Diagrams

```java
TOKEN getRelOp() {
    TOKEN retToken = new(RELOP)
    // state is a global variable
    while (1) {
        switch (state) {
            case 0:
                c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail();
                break;
            case 1:
                ...
            case 8:
                retract();
                retToken.attribute = GT;
                return(retToken);
                ...
        }
    }
    return(null);
}
```
Coding Transition Diagrams

• Function `fail()`
  – Resets `forward` pointer to `lexemeBegin`
  – Changes value of `state` to another transition diagram for another token
  – If no other transition diagram, initiate error-correction phase