Lexical Analysis

Problem: Want to break input into meaningful units of information

Input: a string of characters
Output: a set of partitions of the input string (tokens)

Example:
```
if(x==y) {
  z=1;
} else {
  z=0;
}
```

Why Tokens?

We need to classify substrings of our source according to their role.

Since a parser takes a list of tokens as inputs, the parser relies on token distinctions:

- For example, a keyword is treated differently than an identifier

Design of a Lexer

1. Define a finite set of tokens
   - Describe all items of interest
   - Depend on language, design of parser
   
   Recall: "if(x==y) {z=1; else {z=0;}}"
   - Keyword, identifier, integer, whitespace
   - Should "==" be one token or two tokens?

2. Describe which string belongs to which token

Lexer Implementation

An implementation must do two things:
1. Recognize substrings corresponding to tokens
2. Return the value or lexeme of the token

A token is a tuple (type, lexeme):
```
if(x==y) {z=1; else {z=0;}}
```
- Identifier: (id, 'x'), (id, 'y'), (id, 'z')
- Keywords: if, else
- Integer: (int, 0), (int, 1)
- Single character of the same name: () = ;

The lexer usually discards "non-interesting" tokens that don't contribute to parsing, e.g., whitespace, comments

Lexical analysis looks easy but there are problems
Lexer Challenges

FORTRAN compilation rule: whitespace is insignificant
• Rule was motivated from the inaccuracy of card punching by operators

Consider:
• DO 5I=1,25
• DO 5I=1.25

• The first: a loop iterates from 1 to 25 with step 5
• The second: an assignment

Reading left-to-right, cannot tell if DO$5$I is a variable or DO statement until , or . is reached.

Lexer Challenges

C++ template syntax:

\[
\text{vector}\langle\text{student}\rangle
\]

C++ stream syntax:

\[
\text{cin} \gg \text{var}
\]

The problem:

\[
\text{vector}\langle\text{vector}\langle\text{student}\rangle\rangle
\]

Lexer Implementation

Two important observations:
• The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time.
• Lookahead may be required to decide where one token ends and the next one begins.

To describe tokens, we adopt a formalism based upon Regular Languages:
• Simple and useful theory
• Easy to understand
• Efficient implementations

Languages

Definition:
Let $\Sigma$ be a set of characters.
A language over $\Sigma$ is a set of strings of the characters drawn from $\Sigma$.

Examples:
Alphabet = English characters
Language = English sentences

Alphabet = ASCII
Language = C programs

Not every string on English characters is an English sentence
Not all ASCII strings are valid C programs

Notation

Languages are sets of strings.

Need some notation for specifying which set we want to designate a language.
• Regular languages are those with some special properties.
• The standard notation for regular language is using a regular expression

Regular Expressions

A single character denotes a set containing the single character itself:

\[ X = \{X\} \]

Epsilon ($\epsilon$) denotes an empty string (not the empty set):

\[ \epsilon = \{\} \]

Empty set is ($\emptyset$) = \(0\)

\[ \text{size}(\emptyset) = 0 \]
\[ \text{size}(\epsilon) = 1 \]
\[ \text{length}(\epsilon) = 0 \]
Compound REs

Alternation: if A and B are REs, then:
\[ A | B = \{ s | s \in A \text{ or } s \in B \} \]

Concatenation of sets/strings:
\[ AB = \{ ab | a \in A \text{ and } b \in B \} \]

Repetition (Kleene closure):
\[ A^* = \bigcup_{i=0}^\infty A^i \]
\[ A^* = \{ \epsilon \} + A + AA + AAA + ... \]

Convenient Abbreviations

One or more:
\[ A^+ = A + AA + AAA + ... = A A^* \]

Zero or one:
\[ A? = A | \epsilon \]

Character class:
\[ [abcd] = a | b | c | d \]

Wildcard:
\[ . \] (dot) matches any character (sometimes excluding newline)

Examples

Regular expressions to determine Java keywords:
- if | else | while | for | int | ...
- A literal string like "if" is shorthand for the concatenation of each letter

Integer literal:
- digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- digit = [0123456789]
- digit = [0-9]
- integer = digit digit*
- integer = digit+

Is this good enough?

Examples

Whitespace:
- whitespace = [ \t\n]\n
C identifiers:
- Start with a letter or underscore
- Allow letters or underscores or numbers after the first letter
- Cannot be a keyword
- id = [a-zA-Z_][a-zA-Z_0-9]*

Valid Email Addresses:
- (?:[a-z0-9!#$%&'*+/=?^_`{|}~\-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~\-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+(?:(?:(?:(?:25[0-5]|2[0-4]\d|1\d|\d)\.){3}(?:(?:(?:(?:25[0-5]|2[0-4]\d|1\d|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.|\d)\.)

Java RegEx Support

import java.util.regex.Pattern;
import java.util.regex.Matcher;

Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();

Or:
boolean b = Pattern.matches("a*b", "aaaaab");

String class:
String s = new String("aaaaab");
boolean b = s.matches("a*b");
## Predefined Patterns in Java

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[abc]</code></td>
<td>a, b, or c (simple class)</td>
</tr>
<tr>
<td><code>[^abc]</code></td>
<td>Any character except a, b, or c (negation)</td>
</tr>
<tr>
<td><code>\d</code></td>
<td>A digit: <code>[0-9]</code></td>
</tr>
<tr>
<td><code>\D</code></td>
<td>A non-digit: <code>[^0-9]</code></td>
</tr>
<tr>
<td><code>\s</code></td>
<td>A whitespace character: <code>\s</code></td>
</tr>
<tr>
<td><code>\S</code></td>
<td>A non-whitespace character: <code>[^\s]</code></td>
</tr>
<tr>
<td><code>\w</code></td>
<td>A word character: <code>[a-zA-Z_0-9]</code></td>
</tr>
<tr>
<td><code>\W</code></td>
<td>A non-word character: <code>[^\w]</code></td>
</tr>
<tr>
<td><code>^</code></td>
<td>The beginning of a line</td>
</tr>
<tr>
<td><code>$</code></td>
<td>The end of a line</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>A word boundary</td>
</tr>
<tr>
<td><code>\B</code></td>
<td>A non-word boundary</td>
</tr>
<tr>
<td><code>X{n}</code></td>
<td>X, exactly n times</td>
</tr>
<tr>
<td><code>X{n,}</code></td>
<td>X, at least n times</td>
</tr>
<tr>
<td><code>X{n,m}</code></td>
<td>X, at least n but not more than m times</td>
</tr>
</tbody>
</table>