Lecture 2: Lexical Analysis

- Register yourself electronically using one of the "Account Administration" link on the class home page.
- You can also start adding yourself to a team (2-4 people) on this page.
- Please also add yourself to the Piazza newsgroup (link on home page).

Review: Front End Compiler Structure

- A sequence of translations that each:
  - Filter out errors
  - Remove or put aside extraneous information
  - Make data more conveniently accessible.
- Strategy: find tools that partially automate this procedure.
- For lexical analysis: convert description that uses patterns (extended regular expressions) into program.

Tokens

- Token consists of **syntactic category** (like "noun" or "adjective") plus **semantic information** (like a particular name).
- Parsing (the "customer") only needs syntactic category:
  - "Joe went to the store" and "Harry went to the beach" have same grammatical structure.
- For programming, semantic information might be text of identifier or numeral.
- Example from Notes:
  
  ```
  if(i== j)  
  z = 0; /* No work needed */
  else
  z = 1;
  
  ⇒ IF, LPAR, ID("i"), EQUALS, ID("j"), RPAR, ID("z"), ASSIGN, INTLIT("0"), SEMI, ELSE, ID("z"), ASSIGN, INTLIT("1"), SEMI
  ```

Classical Regular Expressions

- Regular expressions denote formal languages, which are sets of strings (of symbols from some alphabet).
- Appropriate since internal structure not all that complex yet.
- Expression R denotes language L(R):
  - L(ε) = L(""), L(R) = \{ε\}.
  - If c is a character, L(c) = \{c\}.
  - If R₁, R₂ are r.e.s, L(R₁R₂) = \{x₁x₂ | x₁ ∈ L(R₁), x₂ ∈ L(R₂)\}.
  - L(R⁺) = L(ε) ∪ L(R) ∪ L(R) ∪ ... .
  - L((R)) = L(R).
- Precedence is * (highest), concatenation, union (lowest). Parentheses also provide grouping.
Abbreviations

• Character lists, such as [abcf-mxy] in Java, Perl, or Python.
• Negative character lists, such as [^aeiou].
• Character classes such as . (dot), \d, \s in Java, Perl, Python.
• $L(R^+) = L(RR^*)$.
• $L(R?) = L(\varepsilon|R)$.

Extensions

• "Capture" parenthesized expressions:
  - After $m = \text{re.match}(r'\s*(\d+)\s*,\s*(\d+)\s*', '12,34')$, have $m\text{.group(1)} = '12'$, $m\text{.group(2)} = '34'$.

• Lazy vs. greedy quantifiers:
  - $\text{re.match}(r'(\d+).*', '1234ab')$ makes $m\text{.group(1)}$ match '1234'.
  - $\text{re.match}(r'(\d+?).*', '1234ab')$ makes $m\text{.group(1)}$ match '1'.

• Boundaries:
  - $\text{re.search}(r'(\^abc|qef)', L)$ matches abc only at beginning of string, and qef anywhere.
  - $\text{re.search}(r'(\?m)(\^abc|qef)', L)$ matches abc only at beginning of string or of any line.
  - $\text{re.search}(r'rowr(?=baz)', L)$ matches an instance of 'rowr', but only if 'baz' follows (does not match baz).
  - $\text{re.search}(r'(?=rowr)baz', L)$ matches an instance of 'baz', but only if immediately preceded by 'rowr' (does not match rowr).

• Non-linear patterns: $\text{re.search}(r'(\S+),\1', L)$ matches a word followed by the same word after a comma.

An Example

SL/1 "language":

```
+ - * / = ; , ( ) < >
>= <= -->
if def else fi while
identifiers
decimal numerals
```

Comments start with # and go to end of line.

(Review of programs in Chapter 2 of Course Notes.)

Problems

• Decimal numerals in C, Java.
• All numerals in C, Java.
• Floating-point numerals.
• Identifiers in C, Java.
• Identifiers in Ada.
• Comments in C++, Java.
• XHTML markups.
• Python bracketing.
Some Problem Solutions

- Decimal numerals in C, Java: 0 | [1-9] [0-9]*
- All numerals in C, Java: [1-9] [0-9]+ | 0 [xX] [0-9a-fA-F]+ | 0 [0-7]*
- Floating-point numerals: (d+\.d* | d+.d+)([eE][-+]d+)? | 0 [0-9]+ [eE][-.]
- Identifiers in C, Java. (ASCII only, no dollar signs):
  [a-zA-Z\_] [a-zA-Z0-9]*
- Identifiers in Ada: [a-zA-Z] ([a-zA-Z0-9] | [a-zA-Z0-9]* [a-zA-Z0-9])*
- Comments in C++, Java: //.*|\*(\^[\^]*|[^\*])\*\*/
  or, using some extended features: //.*|\*(\^[\^]*|[^\*])\*\*/
- Python bracketing: Nothing much you can do here, except to note
  blanks at the beginnings of lines and to do some programming in the
  actions.