RexEx and Parsing

3-28-2012
Any questions about the quiz?

Minute essay comments:

- Command prompts in games/god mode.
Examples of RegEx

- Let's run through some different examples of using regular expressions.
  - Decimal numbers
  - Points in 2-D or 3-D
  - Dates
  - Polynomials
There are times when you might want to include elements in your programs that go beyond regular grammars.

An example of this would be an internal DSL (Domain Specific Language). This is like a little language that is understood in your program.

Mathematical formulas count as these, but so would simple commands that have some structure to them.
Here is a CF grammar for math expressions:

- `expr ::= term { “+” term | “-” term }
- `term ::= factor { “*” factor | “/” factor }
- `factor ::= floatingPointNumber | “(“ expr “)”

Use {} for 0 or more and [] for 0 or 1.

Lots of languages here:
- http://www.antlr.org/grammar/list
Scala Parsers

- import scala.util.parsing.combinator._
- class Arith extends JavaTokenParsers {
  - def expr:Parser[Any] = term~rep("+"~term | "-"~term)
  - def term:Parser[Any] = factor~rep("*"~factor | "/"~factor)
  - def factor:Parser[Any] = floatingPointNumber | "("~expr~")"
- }
Conversion Rules

- Put in a class that extends one of the Parsers.
  - Productions become methods.
  - Results are Parsers. Next class we'll see how to make it more specific than Any.
  - Consecutive symbols are adjoined with ~.
  - The {...} is replaced with rep(...).
  - The [...] is replaced with opt(...).
Using the Parser

- Call parseAll or parse on your class.
- Takes two arguments:
  - First argument is the parser to use.
  - Second argument is the string to parse.
- Let's code this all up and see it in action.
- Strings match themselves.
- RegEx and tokens give strings.
- \( P \sim Q \) gives back \( \sim(p,q) \), where \( p \) and \( q \) are the matches of \( P \) and \( Q \).
- \( P \mid Q \) gives either \( p \) or \( q \).
- \( \text{rep}(P) \) or \( \text{repsep}(P, \text{seperator}) \) give a list of \( p \) values.
- \( \text{opt}(P) \) gives an Option, either \( \text{Some}(p) \) or \( \text{None} \).
Specifying Output

- You can override the default of \( P \) by using \( P \uparrow \uparrow f \). The \( f \) is a function (or partial function) that takes the normal output of \( P \).
- The output you get is \( f(p) \).
- Example uses:
  - `floatingPointNumber \uparrow \uparrow (\_\.toDouble)`
  - “true” \( \uparrow \uparrow (x=>true) \)
  - “(“~ident~”,”~ident~”)” \( \uparrow \uparrow \{ \text{ case } \string{ “(“~i1~”,”~i2~”)” } => (i1,i2) \} \)
In something like the last example shown, there are strings that are part of the parse that really don't impact the result. When you have this type of situation you can use ~> or <~ instead of just ~. The parse result will only include what the arrow points to.

```
"(~>ident~,~ident<~)" ^^ { case i1~,~i2 => (i1,i2) }
```
Let's work on putting this type of functionality in our formula code.

We want to parse to a tree similar to what we produced with the recursive parser we wrote ourselves.

With that we can make this alternate code functional.
Questions? Can you think of anyplace you might use this in your project?