

REGULAR EXPRESSIONS

Prof. Chris Jermaine
cmj4@cs.rice.edu

Prof. Scott Rixner
rixner@cs.rice.edu

Parsing

- Parsing is a fundamental problem in CS
 - The task of performing a syntactic analysis of a doc (or a computer program)
 - Usually a pre-cursor to interpreting or trying to understand the document

Lexing

- Pre-cursor to parsing is “lexing”
 - Where you go through a document and “tokenize” it
 - That is, you remove white-space, extraneous characters, gibberish, etc.
 - And only return a sequence of “tokens”

Lexing

- Pre-cursor to parsing is “lexing”
 - Where you go through a document and “tokenize” it
 - That is, you remove white-space, extraneous characters, gibberish, etc.
 - And only return a sequence of “tokens”

- Example

```
Date: Thu, 8 Jan 2009 14:06:27 -0600 [01/08/2009 02:06:27 PM CST]
From: Michael Lightfoot <milight@rice.edu>
To: cmj4@rice.edu, cjermain@cise.ufl.edu
Subject: CS account created
Hi Chris,
```

```
I've created your CS account, which basically gives you access to the departmental compute
nodes, home directory, web server, etc. The username and password are the same as your
NetID, cmj4.
```

```
Thanks,
```

```
Michael
```

- Tokens might be “Date”, “Thu”, “Jan”, “From”, “Michael”, etc.

How To Recognize Valid Tokens?

- This is where one of the most fundamental CS ideas comes in
 - The notion of a “regular expression”
- A regex is a mechanism for specifying a set of strings
 - In practice, this is the set of strings you want your lexer to return
 - All other strings are ignored
- Think of a regex as a little machine
 - You feed it a string
 - And it either accepts it or rejects it

How To Write a RegEx?

- In the simplest case, any string is a regex
- Ex: myRegEx
 - This “accepts” exactly the set of strings {“myRegEx”}

Or

- Can put an “or” in your regex
- Ex: `myRegex|yourRegex`
 - This “accepts” exactly the set of strings {“myRegex”, “yourRegex”}

Or

- Can put an “or” in your regex
- Ex: `myRegex|yourRegex`
 - This “accepts” exactly the set of strings {“myRegex”, “yourRegex”}
- Also standard to list single-char alternatives in square brackets
- Ex: `[tT]`
 - This “accepts” exactly the set of strings {“t”, “F”}

Grouping

- Can group parts of a regex using parens
- The following are equivalent:
 - `myRegEx|yourRegEx`
 - `(my|your)RegEx`
 - Both “accept” exactly the set of strings {“myRegEx”, “yourRegEx”}

Quantification

- Most powerful capability of regular expressions!
- Can say how many times a pattern can occur
 - ? means zero or one times
 - * means zero or more times
 - + means one or more times

Quantification

- Most powerful capability of regular expressions!
- Can say how many times a pattern can occur
 - ? means zero or one times
 - * means zero or more times
 - + means one or more times
- Ex: floating point number, no exponent
 - `[+-]?[0-9]*(\.[0-9]+|[0-9]+)`
 - Note the “\.”... this is ‘cause a “.” is standard short-hand for “any character”
 - Also note `[0-9]`... short-hand for `(0|1|2...|8|9)`
- What if we want to add an exponent?

Quantification

- Most powerful capability of regular expressions!
- Can say how many times a pattern can occur
 - ? means zero or one times
 - * means zero or more times
 - + means one or more times
- Ex: floating point number, no exponent
 - `[+-]?[0-9]*(\.[0-9]+|[0-9]+)`
 - Note the “\.”... this is ‘cause a “.” is standard short-hand for “any character”
 - Also note `[0-9]`... short-hand for `(0|1|2...|8|9)`
- What if we want to add an exponent?
 - Add this to the end: `([eE][+-]?[0-9]+)?`

Lexing Using Regular Expressions

- Now that we've seen what they are...
 - Let's take a look at how they are used

Regular Expressions In Java

- They are a bit of a mess
 - Many classes, many ways to do the same thing
- If you are worried about efficiency
 - You should be, since lexing is quite expensive
 - Then your lexing code will resemble the following:

```
public int countChris (String inMe) {  
    Pattern patern = Pattern.compile ("Chris(topher)?")  
    Matcher matcher = patern.matcher (inMe);  
    int returnVal = 0;  
    for (; matcher.find (); returnVal++) {  
        System.out.println (matcher.group ());  
    }  
    return returnVal;  
}
```

- Question: why does Java have separate “Pattern” and “Matcher” classes?

Regular Expressions In Java

- One other thing to note
- There's a bit of a disconnect between idea of a regex
 - And the task of lexing
- Why?
 - A regex is really just a spec for a set of strings
 - But when you lex, you want a sequence of strings from that set
 - Often several ways to match a given regex
 - Example: `Chris(topher)?` and the input string "Christopher"
 - Do we match "Chris" or "Christopher"?
- Default in Java is the longest string
 - But can make quantifiers "reluctant" by adding a `?`
 - Ex: `Chris(topher)??` will find "Chris" in the string "Christoper"

Finally...

- Could go on for a long time talking about details of regex's in Java
- Look over the JavaDoc for the “Pattern” class
 - It will have all of the gory details

Questions?