Lecture 12: Regex/Grammars

- I hope you guys are feeling CREATIVE today!
- Today we're going to spend some time talking about some specific applications of computing: PARSING and GRAMMARS
    - These are used in many fields, like "natural language processing" (analyzing language)
    - Parsing
        - Definition: Breaking up some text into "tokens" - individual pieces, e.g. words
        - We've used Scanners to do this parsing
        - We'll use a different technique today
- Regular expressions
    - A way we describe text patterns
        - Can describe a specific string (e.g. "hello")
        - Or can describe a family of strings (for example, all strings that consist of 1 or more spaces)
        - Not all families of strings can be described by regexes, but many can
    - They come up in computer science and it's good to know what they are/the basics of how to use them
    - There are hundreds of books about regular expressions
        - Regexes are very powerful, so we'll just skim the surface
    - We will use regular expressions in order to do parsing - breaking text into tokens
    - We saw using a Scanner, but now we'll use the string's "split(regex)" method
    - This is what you'll use on the next homework (out today)
- Examples:
    - Try "four score and seven years ago"
        - Split on a space
        - Split on 's'
        - Split on 'e'
    - Try "four   score   and   seven   years" (with extra spaces)
        - Split on a space
        - Split on two spaces
        - We want to split on "1 or more spaces" - more than one char per delimiter
        - Split on ' +'
    - Try "four   score   and   seven years   ago" (with tabs)
        - Split on ' +'
        - Split on tab
        - (we can also use \t)
    - Try "four--score   and --seven-----   years -ago" (mult types of delimiters)
        - Split on spaces
        - Split on dashes
        - But we want to split on BOTH
        - Split on '[-]+'
        - Brackets indicate "OR"
    - Try "four   score   and" - mult spaces and tabs
        - Split on spaces
        - Split on tabs
        - Split on '[\t]+' (you'll use this on your homework

- This&&^^$$- isn't!!!,,,going;;;to be<><>easy!
    - We want to identify words (without the punctuation) (opposite of what we did)
    - How can we identify words with our regular expressions?
    - [abcdefghijklmnopqrstuvwxyz]+
    - But there's a simpler way to identify them
    - [a-z]+
    - But the T is left behind - we want capital letters too!
    - [a-zA-Z]+
    - But this is the opposite of what we want - we want to keep the words!
    - [^a-zA-Z]+
    - But we want "isn't" to stay together
    - [^a-zA-Z']+
- You can also use regex delimiters in a scanner
    - Instead of using whitespace, tell the scanner to use something else
    - input.useDelimiter("[^a-zA-Z']+")

- Grammars
    - A grammar is a set of rules governing a language
    - Computational linguistics is a field of computer science, and we write a lot of languages too (aka Java)
    - We are going to study a system for producing new sentences based on a set of rules
    - The system is called BNF (Backus-Naur Form)
        - BNF defines a set of rules governing the language
        - Distinguishes between TERMINALS (words like "run", "hippo", "Jane")
        - And NONTERMINALS (abstract concepts like sentence, noun, verb....)
    - Example
        - <s>::= <np> <vp>
        - We use "::=" to separate the nonterminal and the rule that goes with it
        - Read this as "a sentence is composed of a noun-phrase followed by a verb phrase"
        - <s>, <np>, <vp> are nonterminals of the grammar
        - Nonterminals can be defined in terms of other nonterminals
        - We don't expect these nonterminals to appear in the actual sentences
    - (It turns out you can draw a diagram of how sentences are derived from a grammar)
        - (show image from Wikipedia)
        - This is called a "parse tree"
- Let's drill down a bit more into the components of the grammar
    - We defined a sentence, but let's define the components of a sentence
    - The simplest version of a noun phrase is probably just a proper noun:
      <np>::= <pn>
    - But then we also need to define the proper noun
      <pn>::= ... | ... | ... | ...
    - We use the pipe character "|" to distinguish between possible rules - read the pipe as OR
    - Values on the right-hand side here are TERMINALS - they will appear in the final sentence
    - All the rules are tokenized by white space (a la regex)
        - So a terminal with two words is two separate terminals, one followed by the other - this is ok

    - (RUN GRAMMARMAIN)
        - generate <pn> x 5 (randomly chooses)
        - generate <s> x 5
        - Notice that the sentence always has <vp> in it - why?
        - Because we didn't define a rule for a <vp>
            - So the program assumes that it's a terminal
            - The triangular brackets DO NOT necessarily indicate a non-terminal - only things that appear on the left-hand side of a ::=
            - Brackets are just a convention
- Expand the grammar: other noun phrases
    - We could have "the" or "a" followed by a noun
        - What are those called? determiner!
        - So we can add another rule to the <np> description

          <np>::= <pn> | <det> <n>
        - Notice how we separate the rules with the vertical bar
    - Now we can add the rule for the determiner

      <det>::= the | a | this | some
    - And finally, let's add some nouns

      <n>::= ... | ... | ... | ...
    - (RUN GRAMMARMAIN)
        - generate <np> x 5 (randomly chooses between pn and det-noun rules)
        - generate <s> x 5 (we see the different types of nouns, but still no vp
- Ok, let's add verbs
    - Does anyone know what the different types of verbs are?
        - Transitive, intransitive
    - So we can make a rule that chooses between transitive and intransitive

      <vp>::= <tv> <np> | <iv>
    - Then we need to define our <tv>

      <tv>::= hit | ... | ... | ...
    - And our <iv>

      <iv>::= laughed | ... | ... | ...
    - (note that we're not dealing with tenses right now)
    - (RUN GRAMMARMAIN)
        - generate <s> x 10
        - But these are kind of boring - how can we spice these up?
- Adding adjectives
    - Let's add a rule

      <adj>::= foxy | ... | ... | ...
    - And where does that <adj> get used?

      <np>::= <pn> | <det> <n> | <det> <adj> <n>
    - We keep our old det-n but add the adj possibility
    - But what if we want to be REALLY descriptive?
        - We could use more than one adjective
    - This is getting complicated, so let's add an <adjp>

      <np>::= <pn> | <det> <n> | <det> <adjp> <n>
    - What does the <adjp> rule look like? How many adjectives could we have?

      <adjp>::= <adj> | <adj> <adj> | <adj> <adj> <adj>
    - Could be arbitrarily many - what can we do?

    - Use recursion!

      `<adjp>::= <adj> | <adj> <adjp>`
    - (base case = one adjective, recursive case = an adjective + an adjp)
    - (RUN GRAMMARMAIN)
        - generate <np> x 5
        - generate <s> x 10
        - More interesting!
- Adding adverbs
    - Let's add a rule

      `<adv>::= frankly | quickly | verily | recursively`
    - And add it into the grammar

      `<vp>::= <tv> <np> | <iv> | <adv> <vp>`
    - And then generate more sentences
- Let's just go over again how we generate a sentence
    - (show parse tree diagram from the spec)
    - This assignment will use RECURSION and MAPS
    - non-terminals are the map keys, rules are the map values