314 Exam II topics

Haskell

* Recursive functions without mutable variables
* Basic syntax
* Data types and pattern matching
* Essential data types: lists, tuples, integers, Maybe
  + Review Maybe
* Essential functions: arithmetic, equality checking and comparison
  + Need these functions

Regular Expressions

* Languages
  + For computers, we want precise specifications
    - Humans and computers should agree what is a valid program
    - Humans and computers should agree what a program means
      * An alphabet is a finite set of symbols
      * A string is a finite sequence of symbols belonging to an alphabet
        + Every symbol in the string comes from the same alphabet
  + A language is a (possibly infinite) set of strings in some alphabet
  + Uses?
    - Programming languages, Email addresses, JPEG files etc.
  + Recognition: if a string belongs to a language
  + Parsing: extracting information from a string in a language
  + Recognizer function allows us to answer more questions about a language that we may not have been able to before with just a recognizer
* Syntax
  + a only matches “a”
  + E (Greek E) only *matches* “”, the empty string (different from an empty *language*)
  + a|b only matches “a” and “b” (| == Union) 🡪 R|S is the union of language R & S
  + ab matches “ab”
  + (a|b)(c|d) matches “ac”, “ad”, “bc”, “bd”
    - AND has precedence over OR
  + a\* matches “”, “a”, “aa”, “aaa”…
  + (ab)\* matches “”, “ab”, “abab”, “ababab” …
  + R\* = ε | R | RR | RRR | RRRR | ...
  + R+ = R | RR | RRR | RRRR | ...
  + R+ == the concatenations of 1 or more of the strings while R\* is 0 or more
  + a+ only matches “a”, “aa”, “aaa”, … not the empty case
* Does a particular string match a regular expression?
  + What is a match?
    - a string and a regular expression match if the string is in the language for the regular expression
  + How to match:
    - First thing we try 🡪 direct recursion
      * Problem: how can you handle matching RS?
      * Solution: Try every way to divide the string in two, attempt the first and second parts with R and S
        + Problems: terrible performance for R\* and infinite loop for R\*\*
    - So.. what do we do??
      * Idea: Improve performance when testing RS by only testing string divisions where the first part matches R
        + Instead of dividing the string and trying every combination, see what suffixes remain after matching R to the start of the string
        + Let R determine how the string is divided:

(a|aa)b 🡪 R matches in more than 1 way, so to handle, return a list of remainders and match S with all of them

* + - Matching a **prefix**:
      * + Improves performance, but what if there are infinitely ways to match a prefix?

Ex: (a|E)\* = E | (a|E) | (a|E)(a|E) …

Matching ‘a’ works, so we stop once we find it, matching ‘b’ however, *never terminates*

* + - How do we avoid these new problems?
      * Keep a list of prefixes in a set, but with this you can be repeatedly comparing strings and it’s hard to keep track of what is already produced
    - To avoid having infinitely many matches, we disallow matching empty matches for R when matching R\*
      * How 🡪 we modify R to make R1 we modify matchPre to return the suffix *and* whether it consumed any input
* Equivalence between regular expressions
  + Two Regular expressions are equal if they match the same language
    - That is, R == S if and only if Ls == Lr
  + Equivalent expressions
    - (ε|R)\* = R\*
    - (ε|R)+ = R\*
    - (Q|R)(S|T) = QS|QT|RS|RT
    - R\*R\* = R\*; R\*\* = R\*
    - (R\*SR\*)+ = R\*(SR\*)+
  + If we already have a recognizer, we can enumerate by filtering the list of all possible strings…. But…problem:
    - We can’t distinguish finite and infinite languages
    - The solution: Laziness
  + Laziness allows us to work with languages that are infinite, as long as we don’t try to consume the entire list
    - But we do not have the property that every string in the language will appear at some point in the list
  + Concatenation can work for matching large strings, but performance is not good with large input strings
    - How can we avoid trying every possible way to divide the input string?
      * Solution: instead of looking to see if the string matches, see if the prefix matches, and then what is left over?
      * This way, matching is a *special* case when nothing is left over