CS 3100 – Models of Computation – Fall 2011 – Notes for L3

August 30, 2011

1 Roadmap for this Course

This course will examine the following foundational principles

- Languages, Grammars, and Machines
  - Languages and why study them?
  - Regular, context-free and recursively enumerable languages
  - Other languages: Recursive, NP-complete, etc.
  - Machines to describe languages
    - Machines with and without non-determinism
  - Grammars to describe languages
- *Midterm 1*
- Mathematical Logic
  - Propositional Logic
    - Building propositional solvers using DFA: the BDD approach
    - Solving puzzles using BDDs
  - First-order Logic
    - A taste of first-order logic: Solving Ken-ken
- Schröder-Bernstein Theorem, knowing how to put sets into various cardinalities
- Diagonalization
- Turing Machines and Undecidability; how to show the undecidability of the Halting Problem
- *Midterm 2*
- Launch Projects
- Complexity: NP-completeness
- Showing simple problems NP-complete
- *Finals*

2 Formal Languages, Grammars, Machines, Non-determinism

Formal Languages ("languages") have many roles, some important ones being these:

- Describe desired strings (to help filter out undesirable ones out). Some uses:
  - In spreadsheets
  - Compilers, GUIs, other text-oriented information processing systems
  - Network malware detection
  - Biology, Identifying Genomes
- Help encode algorithms (problems to be solved) and help analyze the complexity (time and space) of machines deployed as *recognizers* for these languages.

Machines help specify languages:

- Gives us an *operational* flavor (plug and chug along a string)
- Machines can be programmed directly to give us *algorithms* to recognize languages (filter bad strings out)

Grammars help specify languages:

- Sometimes machines are tedious to draw out (are huge)
- Machines are not "that machine-readable"—so *grammars* serve the purpose of a *machine-readable machine notation*

Equivalence, the power of alternative notations, non-determinism

- Humans are fallible!
- Humans *can't be trusted* to program correctly or specify anything correctly
- Humans are poor at checking other humans' work!
- So we need *machines to keep us honest*
- If you specify a machine and a grammar, you'll be thinking differently in arriving at each, and so by comparing the machine and the grammar and showing that they denote the same language, you increase your confidence in having specified the desired language.
- *Compactness, and notational economy, is a virtue*
  - Compare Java and purely functional notations (such as Python) for expressing intricate ideas compactly
  - The less the lines of "code" the less the maintenance headache
  - More natural notations are needed often!
  - Sometimes grammars are the most compact/natural to use
  - Sometimes machines are the most compact/natural
  - Ideally, you must try doing it both ways and comparing for agreement

  - But in general, for some languages we prefer grammars, for some we prefer machines, etc.
    * Knowing what to use comes with experience!
    * Sometimes you can have "rules of the thumb"; example: for a language of strings over 0 and 1 where the pattern of interest is "the third from the end is a 1, *don't even try to build a DFA*!

- The reason we are using Python: sometimes, math is too non-obvious; by programming something, you understand it much better!
- The idea of "non-determinism" is a *fundamental* invention of computer science. It makes many descriptions far more compact and more natural!
- It is almost like the invention of *complex number*: it makes many tasks easy (and finally we know how to "take the real part" and descend into reality)
- The same way, we use non-determinism for a "mental high" that allows us to get near the "final answer" quickly; once we do, we can always "descend to reality" by "boiling away" non-determinism
- Oops, but sometimes, when you boil away non-determinism, you pay a big price: things blow up into exponential space/time on your face! You should learn when to expect this to happen!
- In some machines, adding non-determinism fundamentally alters their power; in some machines, it only makes machines smaller / easier to understand (it does not fundamentally alter them). Know when this happens.
- After all, non-determinism is very close to the lazy approach:
  - Instead of solving a problem the hard way, guess an answer; then check it
  - Instead of trying to be smart about guessing, invent the notion of a *free guess*
  - Then all we need to do is *check*!
  - **But the real world is cruel**. Even if you assume free guesses, some problems are too hard to solve.
- The structure of strings packs so much "power" in it that by encoding problems well, you can use machines for tasks they were not intended to do! Example:
  - By encoding the truth of Boolean formulae as languages in a certain way, we can "teach DFA" to become Boolean solvers!
  - This truly illustrates the CS approach: **Computer scientists cannot depend on nature to give them the electrons or the Benzene ring, or anything; they have to pretty much do it all using their own imagination.**
- NFA, PDA, and TM represent three ways of programming in a language with "if-then" and "goto", and using zero stacks, one stack, or two stacks respectively.
- Another CS approach (Garey and Johnson's cartoon):
  - If you don't know how to solve something, at least find enough things that look similar, then group them and tell your boss "look I can't solve X, but at least I've shown that X belongs to a large class that all these other smart people have failed to solve." If anyone solves one member of this class some day, the whole class will have been solved.

3 Regular Languages

3.1 Why are Regular Languages Crucially Important?

Regular languages capture the simplest and most prelavent forms of patterns. You should not ask for any form of infinite memory to recognize these patterns.
Here are examples of languages (some identified to be regular/non-regular; ohers left to you; let $\#_1(x)$ mean "number of $1s$ in x"; further, let YAP stand for "Your Answer Please"): The way you have to identify whether a language is regular or not: **try to write a conditional and goto-based program without using a stack**. By the way, such programs are called DFAs. In Assignment 2, you are being told how we are going to mathematically represent DFA as a Python `dict` object.

- Regular: $L_{odd1s} = \{x \mid x \in \{0,1\}^* \; and\ has\ odd\ number\ of\ 1s\}$
- Regular: $L_{1sMinus0sOdd} = \{x \mid x \in \{0,1\}^* \; and \mid (\#_1(x) - \#_0(x)) \mid \; is\ odd\}$ Here $| \ldots |$ denotes the absolute value
- Non-Regular: $L_{eq1s0s} = \{x \mid x \in \{0,1\}^* \; and\ has\ equal\ number\ of\ 0s\ and\ 1s\}$
- Non-Regular: $L_{0n1n} = \{0^n 1^n \mid n \geq 0\}$

- ? (YAP!): $L_{thirdIs1} = \{x \mid x \in \{0,1\}^* \; and\ the\ third\ char\ is\ a\ 1\}$
- ? (YAP!): $L_{thirdLastIs1} = \{x \mid x \in \{0,1\}^* \; and\ the\ third\ from\ the\ end\ is\ a\ 1\}$
- ? (YAP!): $L_{endsWith0101} = \{x \mid x \in \{0,1\}^* \; and\ ends\ in\ 0101\}$
- ? (YAP!): $L_{nonObvious} = \{x \mid x \in \{0,1\}^* \; and\ has\ odd\ 1s\ or\ even\ 0s\ and\ fifth\ from\ right\ is\ 0\}$
- ? (YAP!): $L_{0n1n2n} = \{0^n 1^n 2^n \mid n \geq 0\}$

3.1.1 Application 1: Scanners in Various Applications

Think of parsing Python and separate identifiers, keywords, numbers, strings (especially those nice triply quoted strings with embedded double-qoutes and back-slaskes). This is clearly a non-trivial task – one that calls for the *full power of regular expressions*. You had better not miss a keyword, confuse an identifier for a number, a comment for a string, etc.!

3.1.2 Application 2: Malware Detection in Networks

Read the article "Use multicore flow processing to boost network router/security appliance throughput. A copy has been kept on the class webpage in the list of references for Lecture 3 as `network-flow-regexp-article.pdf`.

4 Three Ways to Specify Regular Languages

> **Regular languages are those for which there exists a Deterministic Finite Automaton (DFA)** (of course, you can't say so and walk away; you have to draw or program one!)

But we will also allow you to take other approaches to do so.

- **Describe a DFA Grammatically!** The "grammar" that we use to write down a DFA is variously called:
  - Regular expression (RE)
  - Right-linear grammar
  - Left-linear grammar

  If you take this approach, you will be asked to write down a regular expression, and demonstrate that you can write a program to parse a regular expression and build a DFA out of it.
- **Describe a DFA through a non-deterministic finite automaton (NFA)!** If you take this approach, you must demonstrate that you can convert NFA to DFA by writing a Python program (*i.e.*, algorithmically).
- For added reassurance, if you take any of these alternative approaches, you will be told to further show that you can
  - turn an RE into an NFA
  - NFA into a DFA
  - DFA into an RE
  - RE into gold (n.b. This part is optional.)

4.1 Which Regular Languages are Easily Described, and Which Are Not?

The answer will be apparent once you answer these questions:

- Why are these "easy" to describe using a DFA: L_{odd1s}, $L_{thirdIs1}$, $L_{endsWith0101}$.
- Why are these not so easy (!!) to describe using a DFA (watch out for possible leg-pull from me): $L_{1sMinus0sOdd}$, $L_{thirdLastIs1}$, L_{eq1s0s}, L_{0n1n}, $L_{nonObvious}$.
- In particular, why is $L_{thirdLastIs1}$ harder than $L_{endsWith0101}$?
- Is L_{0n1n2n} in the same language category as L_{0n1n}? Explain!

Once you understand these issues, you have gotten half-way to somewhat understanding DFA (!) — naah, you are well off to understand them!

4.2 How to Describe Difficult (Regular) Languages?

First, you have to know if they are even regular (they may not be!). If you are sure that they are regular, then you must pick the best method (using RE or NFA). If they are not even regular, but nevertheless you are expected to describe them, then you should know how to classify the language.

4.3 How to Debug DFAs?

You will be using several methods:

- Write a Python program to take a DFA and help you test it (feed strings to it, and your program tells whether the strings are accepted or not)
- Write a Python program to convert the DFA into a regular expression (you'll pick up all you need to know about REs in a jiffy)
- Write an NFA, compare against your DFA
- Write an RE, compare against your DFA