CMPSCI 250: Introduction to Computation

Lecture #24: Review For Final Exam
David Mix Barrington
30 April 2013

Review For Final Exam

- Exam Format and New Question Type
- Exam Content and Coverage
- Part I of the Course: Propositional Logic, Sets, Number Theory
- Part II of the Course: Predicate Logic, Induction, and Graphs
- Part III of the Course: Searching, Regular Expressions, and FSM's
- Review of Spring 2012 Final Exam

Exam Format

- You have two hours for 120 or 125 total points, with an A level around 110 and a C level around 70.
- Questions involve a combination of carrying out constructions and actually proving things.
- The format is much like that of previous final exams, with one exception on the next slide.
- Again, you'll get a list of definitions, some of which are vocabulary from the course and some of which are new constructs to be used in problems. I think it's important that you be able to apply the principles of the course in a brand new situation.

A New Question Type

- I’ve used **vocabulary questions** in CMPSCI 187 but not yet in CMPSCI 250.
- I’ll give you five two-point **identification questions**, where you get a vocabulary term like “Kleene’s Theorem” or “bijection” and have to tell me enough to convince me that you are familiar with the term.
- I’ll also give five two-point **compare and contrast questions**, where you get two terms and have to tell me the difference between them in a way that convinces me you are familiar with both. Examples would be “divisor and multiple”, “prime and relatively prime”, or “directed graph and undirected graph”.

Exam Content and Coverage

- 25% of the exam will cover the first third of the course, before the first test.
- 25% of the exam will cover the middle third of the course, up to the second test.
- 50% of the exam will cover the last third of the course, since the second test.
- I will try to devise some questions that combine concepts from different sections of the course, such as having you prove concept about a DFA by induction.

Part I of Course: Propositional Logic

- The key concepts here are **propositions**, **logical operators** such as AND and OR, and **propositional proofs**.
- There will be a few **translations**, so look again at the **implication sign** which most of you messed up on the second midterm.
- To get a conclusion from a number of propositional statements, you might use **truth tables** or **propositional inference rules**.
- The names of propositional rules are less important than the validity of the ones you use.
- Case analysis and contradiction are often useful in propositional proofs.

Part I of Course: Sets and Cardinality

- Sets and set operations occur throughout the course, particularly in the formal language theory of the last section.
- We can compare the **cardinality** of either finite or infinite sets, using the rule that a **bijection** is possible only between sets of the same cardinality.
- An **injection** (one-to-one function) from X to Y is possible only if $|X| \leq |Y|$.
- A **surjection** (onto function) from X to Y is possible only if $|X| \geq |Y|$.
- We proved the **Cantor-Schroeder-Bernstein Theorem**, which says that if there are injections from X to Y and from Y to X, then $|X| = |Y|$.

Part I of Course: Number Theory

- We learned about **prime numbers** and **divisibility** -- every positive integer may be factored into primes.
- The **Euclidean Algorithm** lets us compute the **greatest common divisor** of two positive integers, thus testing whether they are **relatively prime**. An extension of the Euclidean Algorithm lets us find a **linear combination** of two relatively prime integers that equals 1.
- We worked with **congruences** and **modular arithmetic**. The **Chinese Remainder Theorem** lets us take two or more congruences where the moduli are pairwise relatively prime, and combine them into a single congruence. For example, if $x \equiv 2 \pmod 7$, $x \equiv 7 \pmod 8$, and $x \equiv 7 \pmod 9$, we can eventually compute that $x \equiv 79 \pmod{504}$.

Part II of Course: Predicate Logic

- **Predicates** and **quantifiers** give us the basic language for a huge variety of mathematical statements. We first learn to translate from English to statements in the **predicate calculus**, and vice versa.

- There are four basic **quantifier proof rules**, two each for **existential** and for **universal** quantifiers. The **instantiation** rules derive an unquantified statement from a quantified one, such as $P(a)$ from $\exists x:P(x)$. The **generalization** rules derive a quantified statement from an unquantified one, such as $\exists x:P(x)$ from $P(a)$.

- The most complicated rule to use is **Universal Generalization**, where we “let x be arbitrary”, prove the statement $P(x)$, then conclude “since x was arbitrary”, $\forall x:P(x)$. But this is the most powerful rule as it gets a universal statement as its conclusion.

Part II of Course: Induction

- The basic **Law of Mathematical Induction** lets us prove a statement of the form ∀x: P(x), where the type of x is "natural number" or "positive integer". We first prove a **base case** of P(0) for naturals or P(1) for positives. We then prove the **inductive step** P(x) → P(x+1). In the course of this, P(x) is called the **inductive hypothesis**.

- In **Strong Induction**, we use an inductive hypothesis that says not only that P(x) is true, but also P(y) for any y with 0 ≤ y ≤ x (for naturals) or 1 ≤ y ≤ x (for positives). With this stronger hypothesis, we still need only prove P(x+1). We use strong induction when the truth of P(x+1) follows not from P(x) but from one or more of the other statements P(y).

- We can use **structural induction** to prove ∀x:P(x) when the data type of x is any set that has a recursive definition, such as strings or trees.

Part II of Course: Graphs, Paths, and Trees

- A **graph** is a collection of **nodes** (or **vertices**) connected by **edges** (or **arcs**). In an **undirected graph**, each edge is between two nodes. In a **directed graph**, each arc goes from one node to another node (or is a **loop** from a node to itself).

- We define **paths**, sequences of edges or arcs where the end of each edge is the beginning of the next. **Search algorithms** are used to test whether a path exists from one node to another. An undirected graph is **connected** if every node has a path to every other -- otherwise it is divided into **connected components**. A directed graph is **strongly connected** if there are paths from any node to any other.

- A **tree** is a connected undirected graph that has no **cycles**, that is, no simple paths from any vertex to itself. These **graph-theoretic trees** can be related to other classes of trees defined recursively.

Part III of Course: Searching

- We described a **general search algorithm**, where we begin with a single **start node** in a data structure called an **open list**. The basic move of the search is to remove a node from the open list and place all of its unseen neighbors on the open list. This continues until we find a **goal node** on the open list, or empty the open list entirely.

- If the open list is a stack, our search is a **depth-first search**. If it is a queue, we have a **breadth-first search**. If it is a priority queue based on distance from the start node (in a **labeled graph**), we have a **uniform-cost search** that will find us a shortest path from the start node to a goal if any such path exists. Finally, **A* search** is a variant of uniform-cost search that finds the same shortest path but may find it more quickly by making use of a **heuristic**.

Part III of Course: Regular Expressions

- A **formal language** is any set of strings over an **alphabet** Σ. Some formal languages are denoted by regular expressions.

- **Regular expressions** are defined recursively, with **base cases** of "∅" and "a", where a is any letter in Σ. These denote the languages ∅ and {a} respectively. We can combine existing regular expressions R and S by the **union operator** to get R + S, or the **concatenation operator** to get RS, or the **star operator** to get R*. The strings in R* are all concatenations of zero or more strings, each of which is in R.

Part III of Course: Non-Regular Languages

- **Deterministic finite automata** or **DFA**'s are machines that read strings and decide whether they are in a particular language. A DFA has a **state set**, a **start state**, a set of **final states**, and a **transition function**. Given a string, we begin at the start state and follow the transition from the current state for each letter, in turn, until we have read the whole string. At that point we accept the string if we are in a final state and otherwise reject it.

- For a given language L, two strings u and v are **L-inequivalent** or **L-distinguishable** if there exists a string w such that exactly one of the strings uw and vw is in L. The relation of L-equivalence divides Σ^* into equivalence classes, and the **Myhill-Nerode Theorem** says that the number of such classes is the size of the smallest possible DFA for L. If there is an infinite set of pairwise L-distinguishable strings, there is no DFA for L at all.

- Given a correct DFA for L, we have an algorithm to get the **minimal DFA** for L.

Part III of Course: Kleene's Theorem

- **Kleene's Theorem** says that a language has a DFA if and only if it is denoted by some regular expression. The proof is constructive, telling us how to build a DFA from a regular expression or vice versa. Part of the proof introduces a new type of finite machine called a **nondeterministic finite automaton** or **NFA**.
- Given a regular expression, a **recursive construction** can be used to build an equivalent **λ-NFA.** The **Killing λ-Moves Construction** then gets us an **ordinary NFA** with the same number of states. Finally the **Subset Construction** produces a DFA, possibly with many more states than the NFA.
- Given a DFA, we can add a new start state and new final state if necessary to get an **r.e.-NFA**, which must satisfy a set of four rules. From an r.e.-NFA, we can **eliminate states** to finally get a two-state r.e.-NFA, from which we can determine a single regular expression for the r.e.-NFA's language.

Spring 2012 Final Exam: Question 1

Question 1 (20): This boolean logic problem deals with the atomic propositions and the Statements 1, 2, and 3 defined above.

- (a, 5) Translate statements 1 and 2 into symbols, and translate Statement 2 into English.
- (b, 15) From Statements 1, 2, and 3, prove that Duncan is barking. You may use either truth tables or propositional proof rules. (Hint: The similar "murder mystery" made heavy use of Proof by Cases. You might also try Proof by Contradiction.)

- This exam's boolean logic problem involves four atomic propositions and three compound propositions:
    - DB means "Duncan is barking".
    - HF means "Duncan heard a fox".
    - HO means "Duncan heard an owl".
    - IN means "Duncan imagined a noise".
    - Statement 1 is "If Duncan is not barking, then he heard an owl if and only if he imagined a noise."
    - Statement 2 is "¬(¬HF ∨ IN) → DB".
    - Statement 3 is "If Duncan heard an owl, then he is barking, and if he is not barking, then either he heard a fox or he imagined a noise or both."

- Statement 1: ¬DB → (HO ↔ IN), Statement 3: (HO → DB) ∧ (¬DB → (HF ∨ IN))

- ¬DB implies (HO ↔ IN), (HF ∨ IN), (¬HF ∨ IN), ¬HO

Spring 2012 Final Exam: Question 2

- Here are two properties that a formal language X might or might not have, expressed in predicate logic where the type of the variables is "string". Since I can't think of good descriptive names for these properties, I have named them after my dogs.

  - X **has the Cardie property** if ∃u:∀v: uv ∈ X.
  - X **has the Duncan property** if ∀u:∃v: uv ∈ X.

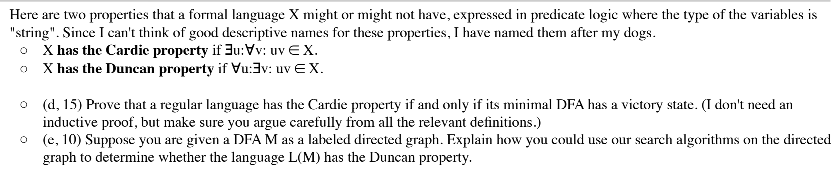
  - (a, 5) Give English descriptions of the Cardie property and the Duncan property.
  - (b, 10) Prove, using quantifier proof rules, that the language a^*b^* does not have the Duncan property.
  - (c, 10) A given language might or might not have either of those properties. Give four regular expressions, representing:
    - A language with neither property
    - A language with the Cardie property but not the Duncan property
    - A language with the Duncan property but not the Cardie property
    - A language with both properties

Here we have two new properties you have not seen before, defined in predicate calculus. The Cardie property says that there is a string whose extensions are all in X, and the Duncan property says that any string can be extended to be in X.

If v = ba, uv cannot be in a*b*, so a*b* is not Duncan.

The empty language has neither property, and Σ* has both. aΣ* is Cardie but not Duncan, and Σ*a is Duncan but not Cardie.

Spring 2012 Final Exam: More Question 2



Here are two properties that a formal language X might or might not have, expressed in predicate logic where the type of the variables is "string". Since I can't think of good descriptive names for these properties, I have named them after my dogs.

- X **has the Cardie property** if ∃u:∀v: uv ∈ X.
- X **has the Duncan property** if ∀u:∃v: uv ∈ X.

- (d, 15) Prove that a regular language has the Cardie property if and only if its minimal DFA has a victory state. (I don't need an inductive proof, but make sure you argue carefully from all the relevant definitions.)
- (e, 10) Suppose you are given a DFA M as a labeled directed graph. Explain how you could use our search algorithms on the directed graph to determine whether the language L(M) has the Duncan property.

- Here we also use the new definitions of “victory state” and “death state”.

- If u is the string from the Cardie Property, all strings in uΣ* are equivalent and thus u goes to a victory state. And any string going to a victory state can be used as u.

- For the Duncan property we need that every state that is reachable from the start state must have a path to a final state. We can check this with one DFS from the start node, followed by a new DFS from each reachable node.

Spring 2012 Final Exam: Question 3

The function g from naturals to naturals is defined by the rules g(0) = 1 and g(n + 1) = 2g(n) + 2. So g(1) = 4, g(2) = 10, and g(3) = 22.

- The function h from naturals to naturals is defined by the following recursive pseudo-Java code:

```
public natural h (natural n) {
   if (n == 0) return 0;
   if ((n % 2) == 1) return 1;
   return 2 * h(n/2);}
```

- **Question 3 (25):** These problems refer to the functions g and h defined above.
  - (a, 10) Prove, by ordinary induction on naturals, that for any natural n, g(n) = $3(2^n) - 2$.
  - (b, 10) Prove, by strong induction on naturals, that for any *positive* natural n, the number h(n) divides n.
  - (c, 5) Describe the output of the function h(n) in terms of the prime factorization of the input n.

- The ordinary induction (a) has the key step: if g(n) = $3(2^n) - 2$, then 2g(n) + 2 = $3(2^{n+1}) - 2$.

- The strong induction (b) has the key step: if h(n/2) divides n/2, then 2h(n/2) divides n.

- For (c), we try examples to see that h(n) is the largest power of 2 dividing n.

Spring 2012 Final Exam: Question 4

The λ-NFA M has state set {1, 2, 3, 4}, start state 1, final state set {4}, and transitions (1, a, 2), (1, b, 3), (2, b, 2), (2, λ, 4), and (3, a, 4). It looks like this:

```
        /---\ b
        \   /
         \ V
      /-->(2)--\
   a /          \ lambda
    /            V
>(1)           ((4))
    \            ^
   b \          / a
      \-->(3)--/
```

- (a, 5) Using the construction from lecture, build a λ-NFA from the regular expression $ab^* + ba$. (This λ-NFA is equivalent to, but different from, the λ-NFA M given above. We will use M for the rest of the problem, to make your life easier.)
- (b, 10) Build an ordinary NFA N that is equivalent to the given λ-NFA M, so that L(N) = L(M).
- (c, 10) Using the Subset Construction, build a DFA D such that L(D) = L(N).
- (d, 5) Show the steps of the State Elimination algorithm to get the regular expression $ab^* + ba$ from D.

- I'll work these out on the projector.