CS245 Review Notes

Preamble

Welcome to my unofficial course notes for the final exam of CS245(Summer 2014)

The course material that has influenced the contents of these course notes are as follows:

- Course Textbook.
- Course slides.
- Posts on the CS245 Piazza.
- Student perspectives.

To preserve the chronological order that the material was presented during lecture, these review notes will follow a similar structure. However, some material in these review notes may be introduced earlier or later. This will only occur if we feel as if material flows better in a separate section than it was originally introduced in.

Disclaimer

I ask that you respect all of the guidelines I set forth with respect to these review notes. If you are unable to respect any of the guidelines mentioned in the proceeding few paragraphs, please delete this file or close the hard copy of the review notes.

1. I alone reserve the right to alter and distribute these review notes. If you see a mistake or feel as if important content is missing, contact me (information provided below) so that I can make the appropriate correction(s).
2. I will be supporting the accuracy of these review notes for the remainder of the S14 semester. After such a time these review notes should be considered obsolete.
3. These review notes are provided absolutely free of charge. If you paid for a hard copy or e-copy of these review notes then you are obligated to contact us so that we can take the appropriate action(s) towards the distributor.

Keep in mind that these notes are in no way guaranteed to be accurate. There may be mistakes, outdated information or tangents that will not be directly related to some of the material presented in the course. These notes have been developed by me, a student during my undergraduate studies taking CS245. No professor, TA or anyone involved in the administration at the University of Waterloo has endorsed these notes.

If you feel a need to contact me at any time for clarification, issues with the notes whatever, you can reach me at my contact information below:
Jacob Willemsma, author (**wjwillem@uwaterloo.ca**).

Acknowledgements

Some further acknowledgments to the CS245 material and other influencing figures.

- Design inspired by ”Linear Algebra Course Notes” by Dan Wolczuk (with permission).
- CS136 Course Notes by Jacob Pollack and Jacob Willemsma

Table of Contents

1 Propositional logic

1.1 Introduction/Background

Welcome to CS245 - Logic and Computation.

To begin, we will review a couple topics covered in MATH135 (a prerequisite course) to warm up and then jump straight into proposition logic.

I will not be covering the brief introduction to **set theory** at the beginning of this course because at this point, most of you should have a strong sense of that material. If however you do not feel confident with this, please review the slides 1-12 on propositional logic.

DEFINITION
Induction

Induction is typically a scary word, however the logic behind it is fairly simple. All **induction** serves to accomplish is to *prove* a theorem is true by showing that if it is true for any *particular* case and it is also true for the next case in a series, then it must be true for all particular cases.

EXAMPLE 1

We can use an inductive process to define how to find *natural numbers*.

- $0 \in$ N.
- For any n, if $n \in$ N, then n' in N, where n' is the successor of n.
- $n \in$ N only if n has been generated by step 1 and step 2.

There are plenty of examples of induction online. I will let you look those up as well.

1.2 Logic

DEFINITION
Logic

Logic is the study of the principles of valid reasoning and inference. Computer science and math are not the only two major schools that study logic, in fact philosophy is obsessed with it. The act of being able to properly reason is at the foundation for debate and life itself.

EXAMPLE 1

Is the following argument valid? Which is to say, does it have a sound bases in logic?

If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. Therefore, there were taxis in the station.

To be able to show that this sentence is valid, it is much easier to break it down into the form of symbols (atoms) and relations (connectives).

For instance, the example sentence can be broken down into the following line of reasoning:

EXAMPLE 2

If p and not q, then r. Not r. p. Therefore q.

In order to obtain this string of logic, we have broken down the statement as follows.

- p means that the train arrives late.
- q means that there are taxis at the station. *n.b. we negate this in the string of logic.*
- r means John is late for his meeting.

DEFINITION
Proposition

We begin our quest to understand logic by defining a formal language where we can express sentences and logical structures. we will call this a **proposition**. A proposition is a declarative sentence that is either *true* or *false*.

Every proposition can only ever have the result true, or the result false. It can also not simultaneously be both true and false.

We will focus on building **symbolic** logics, therefore we will use strings of symbols to state propositions and build arguments using connectives.

For instance, the train argumentation can be summed up into the following:

EXAMPLE 3

$((\mathrm{p} \land \lnot\ \mathrm{q}) \to \mathrm{r})$

A logic is formalized by the following:

- Syntax.
- Semantics.
- Proof procedures.

1.3 Syntax of propositional logic

As briefly stated before, in propositional logic, we will refer to the simple propositions that are the building blocks to create **compound propositions** as simply **atoms**.

Right then, let's get to defining the propositional language.

DEFINITION
Propositional Language

The **propositional language** (denoted L^{p}) consists of the:

- **proposition symbols** (denoted with small Latin letters like p, q). This set is know as *Atom(L^p)* and is the basic building blocks of proposition language.
- 2 **punctuation** symbols "(" and ")".
- 5 **connectives**:
  - $\neg$ (not or negation).
  - $\wedge$ (and).
  - $\vee$ (or).
  - $\rightarrow$ (implies).
  - $\leftrightarrow$ (equivalence).

DEFINITION
Expression

An **expression** is a finite string of symbols where the length of the expression is the number of symbols in the expression.

At this time, the course notes and slides go into a great deal about terminology for expressions in L^{p}. I don't particularly find these useful to go into detail about, so I will quickly bullet point them:

- The **empty expression** is denoted: $\emptyset$.
- Two expressions are **equal** if and only if they are the same length and every symbol is the same at each index [1 ... n].
- UV is the **concatenation** of two expressions U and V. It has length $U + V$.
- If U is part of a expression V, then it is a **segment** of V.
- If U is part of a expression V and $U \neq V$, then it is a **proper segment** of V.
- We may use $*$ to refer to an arbitrary **binary connective**.

DEFINITION
Formation Rules

The set of **formulas** (denoted *Form(L^p)*) is inductively defined as follows:

- *Atom(L^p)* $\subseteq$ *Form(L^p)*.
- If A $\in$ *Form(L^p)*, then ($\neg$A) $\in$ *Form(L^p)*.
- If A, B $\in$ *Form(L^p)*, then (A $*$ B) $\in$ *Form(L^p)*.

DEFINITION
Form(L^p)

Form(L^p) is the smallest class of expressions of L^p which is **closed** under the formation rules of L^p. We say that it is a **well formed formula**.

How then do we test if a expression is a well formed formula? There are two methods, an algorithm or drawing a parse tree.

DEFINITION
Algorithm for testing WFF

To test whether or not an expression U is a well formed formula in *Form(L^p)* we will follow this algorithm:

1. If the formula is empty, it is **not** a well formed formula.
2. If $U \in$ *Atom(L^p)*, then it **is** a well formed formula. If U is any other single symbol, then it is **not** a well formed formula.
3. If U contains more than one symbol and it does not start with "(", then it is **not** a well formed formula.
4. If the second symbol is $\neg$, U must be $(\neg V)$ where V is an expression. Otherwise it is **not** a well formed formula. Apply the same algorithm to V.
5. If U begins with "(" but the second symbol is not $\neg$, scan from left to right until a V segment is found where V is a proper sub-expression. U must be of the form $(V * W)$ where W is also a proper sub-expression. Otherwise it is **not** a well formed formula. Apply the same algorithm recursively to V and W.

Feel free to look up how to draw a parse tree in the course slides, it is not very difficult.

Let's now introduce a new type of induction which will come in handy for proving theorems related to formulas in *Form(L^p)*.

DEFINITION
Structural Induction

In order to prove properties of propositional formulas, we apply induction on the **height** of the parse tree. This proof technique is called **structural induction**.

I will not provide a worked out example for this here, but the general idea for proving this is quite simple, and once you understand you understand.

1. The first step is to prove a base case. Generally a small one.
2. The next step is to assume it works for a tree of height n.
3. Finally in the inductive step, we prove that it works for a tree of height $n + 1$. To do this, we will show that if the $n + 1$ connective is a unary or binary, the inductive step will still hold.

DEFINITION
Scopes

We have two types of scopes, one for unary and one for binary connectives.

- If $(\neg A)$ is a segment of C, then A is called the **scope** in C of the $\neg$ on the left of A.
- If $(A * B)$ is a segment of C, then A and B are called the left and right **scopes** in C of the $*$ between A and B.

1.4 Semantics of propositional logic

DEFINITION
Semantics

The semantics of a logic describe how to interpret the well formed formulas of the logic. Semantics help give meaning to the atoms, connectives and formulas.

2 First Order Predicate Logic

2.1 Introduction/Motivation

In propositional logic, only the logical forms of the propositions are analyzed. We never actually *talk* about the objects we are working with. This is important because we need some way to properly address what kind of qualities or properties are attributed to an object, some objects or all objects, without enumerating all the objects in a set.

EXAMPLE
Motivating Example

Consider the statement:
Every student is younger than some instructor.

This statement is certainly true, but how can we show this? Certainly not with propositional logic, we need some way to assign being *younger* than someone, some way to differentiate instructor from student, and some way to show that all of something is younger than some of another thing.

DEFINITION
First-Order Logic

First-order logic (FOL) gives us a language to express statements about objects.

FOL is expressed with the following ingredients:

- Domain of objects (What kind of object they are)
- Variables
- Designated individuals (E.g. constants)
- Functions (E.g. $+$)
- Relations (E.g. $=$)
- Quantifiers
- Propositional connectives

For example, we could write $S(andy)$ to denote that Andy is a student and $I(paul)$ to denote that Paul is an instructor. Likewise, $Y(andy, paul)$ could mean that Andy is younger than Paul. In oder to make predicates like these more useful and easier to write, we use *variables* or *placeholders*. These assigned relations to variables are called **predicates**.

In the previous case would would have something as follows:

- $S(x)$: x is a student.
- $I(x)$: x is an instructor.

- $Y(x, y)$: x is younger than y.

Quantifiers are the next big change brought into FOL. Quantifiers are used to make the difference between *every* and *some*. We will use what we learned in MATH135 to describe these:

- $\forall$: For all (every).
- $\exists$: There exists (some).

For all is called the *universal quantifier* and there exists is called the *existential quantifier*.

Just with these two additions and our knowledge of propositional logic, we can right the previous statement (although a little paraphrased) completely in FOL.

EXAMPLE First FOL Statement

The Statement *Every student is younger than some instructor.* can be translated into FOL as the following:

$\forall$x.(S(x)$\rightarrow$($\exists$y.(I(y) $\wedge$ Y(x,y))))

Now, let's add a little more functionality to the language with *functions*.

Functions give us a way to express statements more concisely then just simple predicates. Predicates can lead to ambiguity. For instance, if we wanted to show that every student is younger than *their* professor, our simple professor predicate would not cut it.

EXAMPLE Removing Ambiguity

For example, here is how to integrate functions to show that *Every student is younger than their instructor.*

$\forall$x.(S(x)$\rightarrow$(Y(x,I(x))))

In this statement, we've changed I(x) to return the professor of x. This way we know that a student x has a professor and that for all x, x is younger than its professor.

2.2 Syntax of Predicate Logic

The syntax of Predicate Logic contains many similarities to propositional logic, though initially there is a lot of new content. Rest assured, it will all make sense soon.

Predicate logic (L) contains 7 different types of symbols:

- c, d, c_1, c_2: Constants
- f, g, h, f_1: Functions
- x, y, z, x_1: Variables
- P, Q, P_1, P_2: Predicate Symbols
- $\rightarrow$, $\leftrightarrow$, $\neg$, $\wedge$, $\vee$: Logical Connectives
- $\exists$, $\forall$: Quantifiers
- ”(”, ”)”, ”.”, ”,”: Punctuation

To make this a little easier to swallow, the majority of the syntax breaks down into two categories:

DEFINITION
Terms

The set *Term(L)* of **terms** of L is define dusing the following rules:

- All constants are terms.
- All variables are terms.
- If $t_1,\ldots,t_n \in$ *Term(L)* and f is an n-ary function symbol, then $f(t_1,\ldots,t_n) \in$ *Term(L)*.

DEFINITION
Atoms

Let P be a predicate (n-ary relation). An expression of L is an **atom** in *Atoms(l)* if and only if it is of the form $P(t_1,\ldots,t_n)$ where $t_1,\ldots,t_n$ are terms in *Term(L)*

Just as in propositional logic with *Form*(L^p), *Form(L)* is defined inductively.

DEFINITION
Form(L)

The set ***Form(L)*** of first-order logic formulas as follows:

1. *Atom(L)* $\subseteq$ *Form(L)*
2. If $\alpha \in$ *Form(L)*, then $(\neg\alpha) \in$ *Form(L)*
3. If $\alpha,\beta \in$ *Form(L)*, then $(\alpha * \beta) \in$ *Form(L)*, where $* \in \{\rightarrow, \leftrightarrow, \wedge, \vee\}$.
4. If $\alpha \in$ *Form(L)* and if x is a variable, then $(\forall x.\alpha) \in$ *Form(L)* and $(\exists x.\alpha) \in$ *Form(L)*.

Similar again to propositional logic, you can easily trace out a parse tree for FOL to determine whether or not a formula is in *Form(L)*.

I will not be drawing up examples in the notes, however I will detail the differences between FOL parse trees and propositional logic parse trees.

- Quantifiers $\forall x$ and $\exists y$ form nodes like negation. (i.e., only one sub-tree)
- Predicate $P(t_1,...,t_n)$ has a node labelled P with a sub-tree for each of the terms $t_1,...,t_n$.

DEFINITION
Bound and Free

We say that an occurrence of x is **free** in first-order formula α, if in the parse tree of α, there is no upwards path from x to a node labeled by $\forall x$ or $\exists x$. An occurrence that is not free is called **bound**.

Once again, in another similarity, predicate logic also contains the idea of a **scope**.

DEFINITION
Scope

If $\forall x.\mathrm{A}(x)$ or $\exists x.\mathrm{A}(x)$ is a segment of B, $\mathrm{A}(x)$ is called the **scope** in B of the $\forall x$ or $\exists x$ on the left of $\mathrm{A}(x)$.

In FOL, a formulate $\mathrm{A} \in$ *Form(L)* is **closed** if the set of free variables is empty. That is, there are no free variables in the formula.

However, if there are free variables, we may consider using a very important aspect of predicate logic, that of **substitution**.

DEFINITION
Substitution

Given a variable x, a term t and a formula, α. We define $\alpha[x/t]$ to be the formula obtained by replacing each free occurrence of variable x in α with t.

Note that with this definition, if there are no free occurrences of a variable (a **closed** formula) then we have that: $\alpha[x/t] = \alpha$.

Should we be able to make the substitution with term t, we say that it is **free for** a variable x. You **cannot** substitute in a variable which will become bound, which basically means, don't use the same letter as something else already in the formula... since they are not the same.

2.3 Semantics of Predicate Logic

3 Program verification

3.1 Introduction

DEFINITION
Program correctness

The act of checking if a given program satisfies its specification, i.e. does it do what it should?

There are a few ways to check for program correctness.

- inspection.
- testing (white box and black box).
- formal verification.

You guessed it, for this class, we'll do it the formal way.

DEFINITION
Program Specification

Rules by which the program must obey. Preconditions and Postconditions.

DEFINITION
Formal verification

Formal verification tests a programs correctness by using formal program states and by proving the a program satisfies the specification for **all** valid inputs.

We're obviously not going to do this with a whole complicated language like C++, instead we'll focus on a smaller subset of a language like C/C++ or Java. Our *Core Programming Language* contains some familiar concepts:

- Integer and Boolean Expressions.
- Assignment.
- Sequence.
- If - Then - Else.
- While/For loops.
- Arrays.
- Functions and Procedures.

DEFINITION
State

The *state* of a program is the values of the variable at a **particular time** in the execution of the program. The reason for a state being only circumstance of a program at a particular time is that expressions often evaluate relative to the current state of the program (i.e. we can go inside the program and see what's going on). Commands change the state of the program.

Say we had the following function (calculate the factorial).

```
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
```

A few program states would then be:

- Initial state s0: z=0, y=1.
- State s1: z=1, y=1.
- State s2: z=2, y=2.
- State s3: z=3, y=6.
- State s4: z=4, y=24.
- ...

In order to verify program correctness, we need to create a formal of which to present our data.

DEFINITION
Hoare triple

A specification of a program C is a Hoare triple with C as the second component: $\{P\}$ C $\{Q\}$
In this format, P represents the preconditions, C the program code and Q the post-conditions. P and Q collectively make up the program specifications.

EXAMPLE 1

If the input x is a positive integer, compute a number whose square is less than x.

```
{x > 0} C { y * y < x}
```

3.2 Correctness

Given that we now have the notion of a Hoare triple, we want to develop a notion of a proof which will allow us to prove that a program C satisfies the given specification P and Q.

DEFINITION
Partial correctness

A triple $\{P\}$ C $\{Q\}$ is **satisfied under partial correctness** if and only if while executing the program on every possible **state** the resulting state, *s'*, if the program were to terminate, *s'* would also satisfy $\{Q\}$.
We write $\models_{par}$ $\{P\}$ C $\{Q\}$.

Important to note triples that are partial satisfied do *not* necessary terminate. All that matters is that if it were to terminate, the current state would satisfy the postcondition $\{Q\}$.

EXAMPLE 2

In fact, here's a example of a partially correct triple that does **not** terminate.

```
{true}
while true {x = 0;}
{x = 0;}
```

Partial correctness seems like an awfully weak way to classify programs as correct. Any function that does not terminate would actually be partially correct under this definition (provided the current state satisfies $\{Q\}$).

What if we want to make sure the program works on all inputs **and** terminates?

DEFINITION
Total correctness

A triple $\{P\}$ C $\{Q\}$ is satisfied under total correctness if and only if while executing the program on every possible **state** the resulting state, *s'* also satisfies $\{Q\}$. That is to say, it is partially correct **and** terminates!
We write $\models_{tot}$ $\{P\}$ C $\{Q\}$.

EXAMPLE 3

The following is an example of total correctness.

```
{x >= 0;}
y = 1;
z = 0;
while (z != x) {
    z = z + 1;
    y = y * z;
}
{y = x!;}
```

EXAMPLE 4

The following is an example of neither, as the input is consumed.

```
{x >= 0;}
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
{y = x!;}
```

In this example, we would continuously, independently from the precondition be comparing y = 0!, since we have no notion that x in was the same as the x in $\{P\}$.

To prove correctness, we are interested in being satisfied by *total correctness*. However, in order to find out if a triple is *satisfied under total correctness* we will first go about proving that it is *satisfied under partial correctness* and then prove termination separately.

- Proving partial correctness will be done by introducing a *sound* and *complete* set of **inference rules**. Similar to what we did in natural deduction.
- Proving termination is a separate idea, but is often pretty trivia.

Occasionally in our specifications we will need additional variable that do not appear in the program.

DEFINITION
Logical variables

Logicals serve as placeholders to help differentiate variables used in code from the ones in the specifications. They do not appear in the code and are themselves not quantified (though they can follow rules for the variables in the specification.)

EXAMPLE 2

For example, in the example where x was previously consumed.

```
{x_0 >= 0;}
y = 1;
while (x != 0) {
    y = y * x;
    x = x - 1;
}
{y = x_0!;}
```

Similar to the way we did substitutions back in the glory days of CS136. Oh yeah... you remember. You wish you didn't, but you do.

3.3 Inference Rules