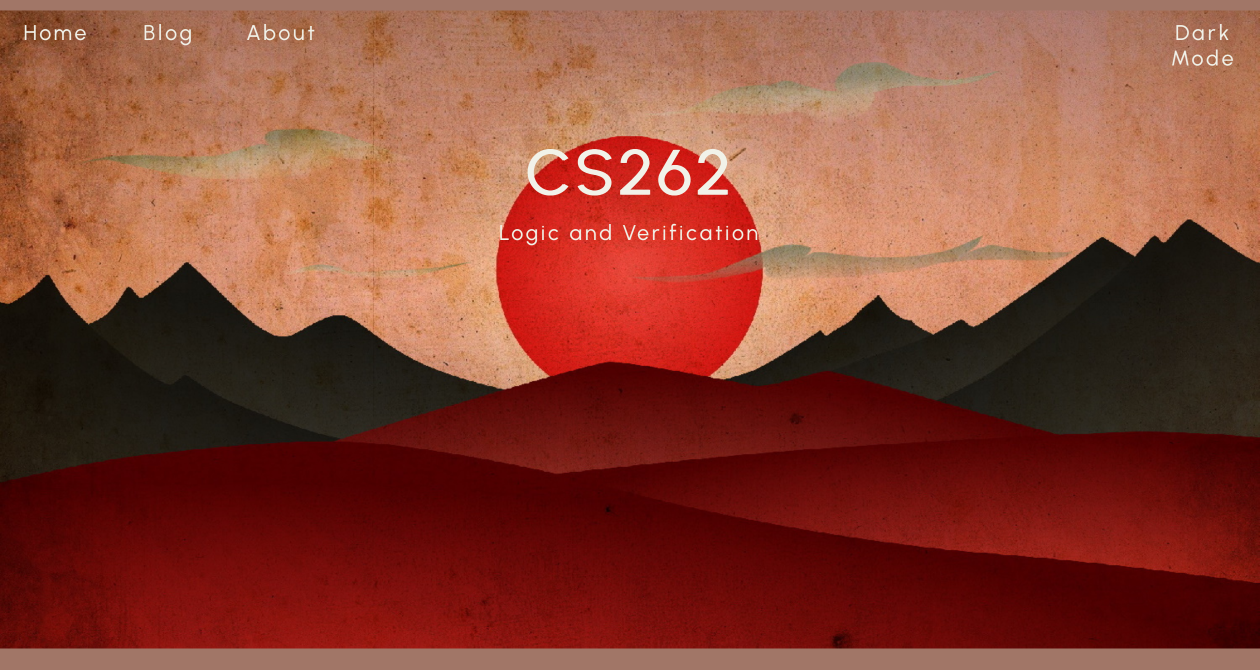

Introduction

There's not much time, so I'll just be writing down brief notes.

1. Propositional Logic
2. Natural Forms
3. Logical Proofs
4. First Order Logic
5. Verification

Propositional Logic

The notation for **NAND** and **NOR** can be thought of as $\wedge, \vee$ with vertical bars through the middle of them - i.e. $\uparrow, \downarrow$.

The notation for **XNOR** and **XOR** respectively are $\equiv$ and $\not\equiv$ (think about the truth tables.)

I use $\implies$ with the double arrow, the exam uses $\rightarrow$ the single arrow. Please bear in mind.

Top $\top$ and bottom $\bot$ are True and False respectively.

The **degree** of a parse tree is the number of *inner* nodes.

A **valuation** is "a mapping of a formula to true or false" i.e. what it resolves to. Always true: **tautology**. Always false: **contradiction**. Sometimes true: **satisfiable**.

X is a **consequence** of S (set) $S \models X$ if every formula in S is true $\implies$ X is true. A tautology is a consequence of nothing $\models X$.

Normal Forms

Write a conjunction $X_1 \wedge X_2 \wedge \cdots$ as an angle list $\langle X_1, X_2, \cdots \rangle$ and a disjunction $Y_1 \vee Y_2 \vee \cdots$ as a square list $([])Y_1, Y_2, \cdots]$. X can be a complex formula, or a *literal* $(\top, \bot, x, \neg x)$. A **dis**junction of literals is a **clause** and it's **con**junction counterpart is a **dual clause**.

The **conjunctive normal form** is a conjunction of disjunctions $\langle [] \rangle$ of literals. And is associated with **alpha** α.

The **disjunctive normal form** is a disjunction of conjunctions $[\langle\rangle]$ of literals. Or is associated with **beta** β.

Every binary formula under the sun *(except xor and xnor)* can be reduced to $A \wedge B$ or $A \vee B$, where A, B can be literals or other formulas, via a combination of simplifying formulas like $\implies$ and **de-morgan's** law.

Note $A \not\Rightarrow A \equiv \neg(A \Rightarrow A) \equiv \neg(\neg A \vee A) \equiv A \wedge \neg A$.

If a formula simplifies to $A \wedge B$, it is an **alpha formula**. If it simplifies to $A \vee B$, it is a **beta formula**.

Reduction to CNF

From a truth table, look at rows with **False** as output. Write the "inverse or", i.e. if a row is $x = T, y = T, z = F \longrightarrow F$, then the clause for that row is $[\neg x, \neg y, z]$. Put all of these clauses together in a conjunction.

For a formula X, start by writing $\langle [X] \rangle$, then reduce via algorithm.

In short, the algorithm is represented as:

$\neg\top$		$\neg\bot$		$\neg\neg Z$		β		α	
$\bot$		$\top$		Z		β_1		α_1	α_2
						β_2			

Where β is a beta formula $(\beta_1 \vee \beta_2)$, and α is an alpha formula $(\alpha_1 \wedge \alpha_2)$.
In long:

- If beta: $\langle \cdots [\beta_1 \vee \beta_2, \cdots] \cdots \rangle$, replace the $\vee$ with a comma.
- If alpha: $\langle \cdots [\alpha_1 \wedge \alpha_2, \cdots] \cdots \rangle$, *copy* the disjunction it's in, and in the place of $\alpha_1 \wedge \alpha_2$ put α_1 in one and α_2 in the other.
- Otherwise replace in place as shown.

Repeat until all items are **literals**.

Reduction to DNF

From a truth table, look at the rows with **True** as output. Write the and of the literals, i.e. if a row is $x = T, y = F, z = T \longrightarrow T$, write the dual clause $\langle x, \neg y, x \rangle$. Put all of these together in a disjunction.

For a formula start with $[\langle X \rangle]$. The algorithm is identical, except the role of the α and β is swapped.

$\neg\top$		$\neg\bot$		$\neg\neg Z$		β			α
$\bot$		$\top$		Z		β_1	β_2		α_1
									α_2

Logical Proofs

Two methods, **semantic tableau** (tree) ~ DNF, and **resolution** ~ CNF.

Both prove if a formula is a **tautology** by contradiction.

Semantic Tableau

Tree form. Branches (from root to tip) are conjunctions (ands), and the whole tree's "value" is a disjunction (ors) of branches. We prove by expanding branches. A node *can* be expanded multiple times, but in a **strict** tableau expand only once, but this is still sufficient.

To prove a formula X, start with the antiformula $\neg X$ as the root of the tree. Now look at the DNF table.

1. If there is an alpha formula, add two nodes to the end of any branch it is on with those α_1, α_2 values (like in table).
2. If there is a beta formula, add a **split** onto the end of that branch, and make two new branches from it with the β_1, β_2.
3. **Close** a branch if that branch, somewhere along it, has a formula X, and its antiformula $\neg X$. This is an atomic closure if X, ¬X are atomic.

When all branches are closed, X is proved to be a tautology. Denote a tableau proof as $\vdash_t X$.

Resolution

The dual to the tableau, this one is just written as a numbered list of disjunctions. Similarly it proves X is a tautology by contradiction.

Make line 1 $[\neg X]$. Then look at the CNF table

1. If there is a beta forumla, replace that formula with β_1, β_2 (comma). (Write the updated thing on a new line)
2. If there is an alpha formula, copy that line onto two new lines, one with the α_1, one with the α_2.
3. If one line has a formula X, and the other line has a formula ¬X, perform the **resolution rule**: make a new line with all the formulas from the prior two *except* any Xs and ¬Xs. *Do not try do this over multiple different formulas in one go*.

When [] is found then the resolution is considered proved. Denote as $\vdash_r X$. A line can be "expanded" multiple times, and we also have the idea of "strictness" in resolution.

Proving Consequence

To prove $S \models X$ (for a set S of formulas), add an **S-introduction** rule to both methods.

- **Tableau:** Add any formula from S onto the end of any branch at any time.
- **Resolution:** Add any formula from S as a new line at any time.

Denote these proofs as $S \vdash_t X$ and $S \vdash_r X$.

Natural Deduction

Note: all boxes here are done horizontally for easy formatting. They should be vertical in reality.

Two techniques to prove a formula: start with the antiformula, or work backwards (e.g. for $x \implies y$ we must assume x and logically conclude y).

In proving only active formulas not in closed boxes can be used.

Assumptions and Premises

Assumptions **start** a box: $[x \cdots$. Premises are formulas S-introduced when proving consequence, and **do not** start a box.

Standard Rules

A double arrow is used for implies, a single arrow is used for "produces"

- Constant rule: $\bot \longrightarrow X$ (from false can introduce any X)
- Constant rule: $\longrightarrow \top$ (anthing can introduce true)
- Negation rule: $X, \neg X \longrightarrow \bot$
- Negation rule: $[X \cdots \bot] \neg X$ or $[\neg X \cdots \bot] X$
- Alpha elim: $\alpha \longrightarrow \alpha_1$ or $\alpha \longrightarrow \alpha_2$
- Alpha introduction: $\alpha_1, \alpha_2 \longrightarrow \alpha$
- Beta elim: $\neg\beta_1, \beta \longrightarrow \beta_2$ or $\neg\beta_2, \beta \longrightarrow \beta_1$
- Beta introduction: $[\neg\beta_1 \cdots \beta]\beta_2$ or $[\neg\beta_2 \cdots \beta]\beta_1$

Derived rules

- Double negation: $\neg\neg X \longrightarrow X$ or $X \longrightarrow \neg\neg X$
- Copy: $X \longrightarrow X$
- Implication: $[X \cdots Y](X \implies Y)$
- Excluded middle: $\top \longrightarrow X \vee \neg X$
- Modus ponens: $X, X \implies Y \longrightarrow Y$
- Modus tollens: $\neg Y, X \implies Y \longrightarrow \neg X$

Write as $S \vdash_d X$ if S entails X has a deduction proof.

First Order Logic

Definitions

First Order Logic contains the existing propositional logic connectives, as well as

- **Quantifiers:** $\forall, \exists$
- **Variables** which don't have to be boolean
- **Relation Symbols:** $<, \leq, =, \geq, >$ (which are like boolean functions)
- **Functions:** $p(x), Succ(x), +(x, y)$, etc. (functions can be written infix $x + y$)
- **Constants** like numbers

Define it as $L(R, F, C)$ where **R** is a finite/countable set of relation/predicate symbols, **F** a set of functions, and **C** a set of constant symbols.

Define **Terms:**

- A **variable**, or a constant is a term.
- A **function** with n arguments: $f(t_1, t_2, \ldots, t_n)$ where each t is a term, is also a term.

Define **Atmoic formulas** as any **relation** $R(t_1, \ldots, t_n)$ where each t is a term; or $\bot, \top$.

Thus define **Formulas:**

- An atomic formula A or its negation $\neg A$
- A binary formula $A \circ B$ with $\circ$ as a binary connective
- A first order formula $\forall x, A$ and $\exists x : A$ where A is a formula

A **bound variable** is a variable that appears with a quantifier, whilst a **free variable** does not.

Semantics

A **model** for $L(R, F, C)$ is a pair $M = (D, I)$ where

- D is the **domain** $(\neq \varnothing)$
- I is the mappiong **"interpretation"** that assigns every constant symbol a value, and defines what every function does.
  If we have a constant c, then $c^I = 4$ means "c is set to 4 in the interpretation I". Similarly, $f^I : D^n \longrightarrow D$ means f is a function with n parameters in interpretation I

An **assignment** under model $M = (D, I)$ maps **variables** to domain values. x^A assigns a value to the variable x.

- We can combine these into $c^{I,A} = c^I; x^{I,A} = x^A; [f(t_1, \ldots)]^{I,A} = f^I(t_1^{I,A}, \ldots)$

The **truth value** of a formula Φ is written $\Phi^{I,A}$, whilst it has a lot of formal definition, literally just how you work out truth things normally.

A formula is **valid**, if it is true for *every possible assignment of variables*, and formulas are **satisfiable** if there is at least one true assignment.

Proving

For FoL, we introduce **gamma** and **delta formulas** for $\forall$ and $\exists$ respectively

γ	$\gamma(t)$
$\forall x, \Phi$	$\Phi\{x/t\}$
$\neg\exists x : \neg\Phi$	$\neg\Phi\{x/t\}$

δ	$\delta(t)$
$\exists x : \Phi$	$\Phi\{x/t\}$
$\neg\forall x, \neg\Phi$	$\neg\Phi\{x/t\}$

Where x/t means **free occurences** of the variable x is **replaced** by a term t (may be represented by a generic placeholder name).

Tableau

Define **par** as a set of *parameters* disjoint from C in $L(R, F, C)$, and denote $L(R, F, C \cup par)$ by L^{par}

The tableau rules are

γ
$\gamma(t)$

δ
$\delta(p)$

Where t is any **closed term** (no variables), and p is any **new parameter**.

For example, $\forall x(R(x) \implies Q(x, y))$ is gamma-replaced by $R(t) \implies Q(t, y)$.

NOTE: **parameters can be used as closed terms *after* they are defined**.

Also, gamma and delta expansions **may need to be used more than once**. It may be absolutely necessary that gamma and delta needs to be repeated more than once, with different replacements.

Resolution

The rules are identical for resolution.

Note that for both of these, S-introduction also remains the same.

Natural Deduction

The rule is now called $\gamma\delta$**-elimination**, and is... pretty much the same.

- $\gamma \longrightarrow \gamma(t)$ or $\forall x F(x) \longrightarrow F(t)\ [x/t]$
- $\delta \longrightarrow \delta(p)$ or $\exists x F(x) \longrightarrow F(p)\ \ (\text{new } p)$

There is also $\gamma\delta$**-introduction**:

- $F(t) \longrightarrow \forall x F(x)$ provided t does not appear in any **non-closed** statement as a **free** variable.
- $F(p) \longrightarrow \exists x F(x)$ provided p doesn't clash with any bound terms in F.

Verification

We want to verify properties about programs. Let p be a program, and φ be a property. If we can prove this property, we write $p \vdash \varphi$.

Programs and Hoare Triples

Syntax falls under domains (i.e. types): E for integer expressions, B for boolean expressions, and C for command expressions. "Expressions" mean you can combine stuff, its not atomic.

Our given programming langauge is defined with the following syntax:

- Assigment: `x = E` (for an E expression)
- Sequential Composition: `C1; C2`
- Decision: `if B then { C1 }` (`else { C2 }`)
- Loop: `while B { C }`

Undeclared variables are allowed, and assumed to have some constant / unknown value.

To formally prove certain conditions hold about programs, we define

- The **Postcondition** being a property that holds **after** the program executes,
- The **Precondition** being properties that hold **before** the program executes, which can be $\top$

Hoare triples are triples containing a **precondition**, a **program**, and a **postcondition**. They are written

$$\llparenthesis Pre \rrparenthesis\ Program\ \llparenthesis Post \rrparenthesis$$

Hoare triples can be valid: `⦇y = 1⦈ x = y + 1 ⦇x = 2⦈` or invalid: `⦇y = 1⦈ x = y + 1 ⦇x < 1⦈`

The process of proving is basically finding intermediate conditions between lines of the program, which makes that specific line of a given program into a valid hoare triple.

Say we have `⦇?⦈ x + 3 ⦇x > 3⦈`, we can find many different preconditions that will satisfy this triple, but for the best effect, we want to mind the most general condition – the weakest preconditon.

Weakest Preconditions

The **Weakest Precondition** is a valid precondition that is implied by *all other* valid preconditions. Given program P and postcondition $Post$, we denote the weakest precondition as $wp(P, Post)$.

We calculate it with the following rules:

- ASSIGNMENT: $wp($`x = E`$, Post) = Post[x/E]$ (using the same "replaces" notation as with FoL).
- COMPOSITION: $wp($`P; Q`$, Post) = wp(P, wp(Q, Post))$
- CONDITIONAL: $wp($`if B then { C1 } else { C2 }`$, Post) =$

$$= (B \implies wp(C_1, Post)) \wedge (\neg B \implies wp(C_2, Post))$$
$$\text{or } (B \wedge wp(C_1, Post)) \vee (\neg B \wedge wp(C_2, Post))$$

Hoare Logic

A program is a series of instructions: $C_1, C_2, \ldots, C_n$.

Proofs are laid out (vertically) as

$$\llparenthesis Pre \rrparenthesis\ C_1\ \llparenthesis \varphi_2 \rrparenthesis\ C_2\ \llparenthesis \varphi_3 \rrparenthesis \cdots \llparenthesis \varphi_{n-1} \rrparenthesis\ C_n\ \llparenthesis Post \rrparenthesis$$

The idea is to *work backwards* from the end, filling in the most appropriate weakest preconditions, until a full chain of valid hoare tuples can be built from the start to the end, via these following rules:

ASSIGMENT RULE. $\llparenthesis Post[x/E] \rrparenthesis$ `x=E` $\llparenthesis Post \rrparenthesis$

Ex. `⦇x + y = 10⦈ x = x + y ⦇x = 10⦈`

Assignment finds the weakest precondition. We can also always **strengthen the precondition** with zero problems, and also **weaken the postcondition** with no problems.

IMPLICATION RULE. $Pre \implies P \wedge \llparenthesis P \rrparenthesis$ `Program` $\llparenthesis Q \rrparenthesis \wedge Q \implies Post$

$\rightarrow \llparenthesis Pre \rrparenthesis$ `Program` $\llparenthesis Post \rrparenthesis$

COMPOSITION RULE. $\llparenthesis Pre \rrparenthesis$ `C1` $\llparenthesis Mid \rrparenthesis \wedge \llparenthesis Mid \rrparenthesis$ `C2` $\llparenthesis Post \rrparenthesis$

$\rightarrow \llparenthesis Pre \rrparenthesis$ `C1; C2` $\llparenthesis Post \rrparenthesis$

CONDITIONAL RULE. $\llparenthesis Pre \wedge B \rrparenthesis$ `C1` $\llparenthesis Post \rrparenthesis \wedge \llparenthesis Post \wedge \neg B \rrparenthesis$ `C2` $\llparenthesis Post \rrparenthesis$

$\rightarrow \llparenthesis Pre \rrparenthesis$ `if B then { C1 } else { C2 }` $\llparenthesis Post \rrparenthesis$

Ex.

```
        ⦇⊤⦈
if (x < y) then {
        ⦇⊤ ∧ x < y⦈
        ⦇x ≤ x ∧ x ≤ y⦈
    z = x
        ⦇z ≤ x ∧ z ≤ y⦈
} else {
        ⦇⊤ ∧ ¬(x < y)⦈
        ⦇y ≤ y ∧ y ≤ x⦈
    z = y
        ⦇z ≤ x ∧ zleqy⦈
}
        ⦇z ≤ x ∧ x ≤ y⦈
```

LOOP RULE. $B \wedge L \rrparenthesis$ `C` $\llparenthesis L \rrparenthesis$

$\rightarrow \llparenthesis L \rrparenthesis$ `while B { C }` $\llparenthesis \neg B \wedge L \rrparenthesis$

Where L is a first order logic **loop invariant** statement, which is something that holds true throughout the loop. Ex.

```
        ⦇x ≥ 0⦈
while (x > 0) {
        ⦇x ≥ 0 ∧ x > 0⦈
        ⦇x − 1 ≥ 0⦈
    x = x - 1
        ⦇x ≥ 0⦈
}
        ⦇¬(x > 0) ∧ x ≥ 0⦈
        ⦇x = 0⦈
```

The loop invariant used is $x \geq 0$. When picking a loop invariant, pick something "useful", whatever that means.