SE 212 - Logic and Computation

Kevin James

Fall 2013

Elements of a Logic

A logic consists of **syntax** (a format for inscribing the logic), **semantics** (the definition of validity within the logic), and **proof theories** (methods of determining validity).

The syntax defines a "well-formed formula" (wff). The semantics tells us which of these wffs are true or valid. Valid formulas are given the symbol $\vdash$, and non-valid formulas are marked as $\not\vdash$. When we describe a formula as being proven by theory, we use "$\models$"; if this is a natural deduction, we use $\models_{ND}$.

Proof theories do mechanical manipulations on strings or symbols. It does not make use of the meanings of characters, but simply uses them as strings of characters. Semantics and proof theories are related by soundness and completeness.

A proof theory is **sound** if there is a way to prove it that ensures its validity, ie $\models \implies \vdash$. A proof theory is **complete** if there is only a way to prove it if it is actually valid, e.g. $\vdash \implies \models$.

Propositional Logic

Propositional Logic is also called *sentential* logic, i.e. the logic of sentences. We also refer to it as propositional calculus, sentential calculus, or boolean logic.

Syntax

A formula in propositional logic consists of

- two constant symbols: true and false
- propositional letters: any single lower- or upper-case letters
- propositional connectives: $\neg, \wedge, \vee, \implies, \impliedby$ (in that order of operations)
- brackets: these always have the highest precedence

All binary logical connectives are right associative. Example: $a \wedge b \wedge c$ means $a \wedge (b \wedge c)$.

With ands, the constants are conjuncts; with ors, they are disjuncts. Not that disjuncts are inclusive, and thus we must formulate exclusive ors by $(p \vee q) \wedge \neg(p \wedge q)$. By implication, we have the premise or antecedant or hypothesis leading to the consequent or conclusion. The contrapositive or $a \implies b$ is $\neg b \implies \neg a$.

Prime propositions are declarative sentences which can be either true or false. The primeness is based on the lack of connectives. Propositions with connectives are called **compound propositions**. Sentences that are interogative (questions) or imperative (commands) are not propositions.

Logic is concerned with the structure of an argument, particularly valid ones. We encode sentences in symbols to give us a better understanding of the structure. Avoid using t or f, since those tend to confuse us with true and false .

Our goal is to formalize all the details found in an English sentence while matching the form of the sentence as closely as possible. We do this by selecting the smallest statements without connectives about which we can can determine validity. Using propositional letters to stand for each of these sentences, we construct our statement by connecting them with connectives.

George

"It is cold but not snowing" can by entered into George with

```
#u kevin

#a 01

#q 01a

% it is cold but not snowing

#check PROP

c & !s

% where
% c means "it is cold"
5 s means "it is snowing"
```

Ambiguity

The English language is notorious for **ambiguity**. When formalizing sentences, we must be sure to reduce the ambiguity as much as possible. The use of logical connectives, particularly, is often incorrect in English. For example, "and" in English is not always commutative and must be sometimes treated temporally.

Semantics

Semantics are about giving our propositions meaning by relating words. They provide the interpretation of expressions in one world in terms of values in another world. The often act as a fucntion which converts between the two. The semantics define the limits of how a proof theory can transform a logical formula.

The syntax of our logic is the **domain** of the semantic function.

Classical logic is two-valued: true and false are the only possibilities for truth values. These are not a part of the syntax of propositional logic. The **range** of a logic is the set of its truth values.

The semantics are described using **boolean valuations** (functions in propositional logic which map from the set of formulas to the range). We can define these boolean valuations with truth tables. For example, we can map $\underline{true} \wedge \underline{false}$ to $\underline{false}$.

The truth value of a formula is uniquely determined by the truth values of the prime propositions. When describing a boolean function, we must only describe the relation of truth values.

For example: $(p \implies q) \wedge r$ where $v(p) = \underline{true}, v(q) = \underline{false}, v(r) = \underline{false}$ gives

$$\begin{aligned} v((p \implies q) \wedge r) &= v(p \implies q) \wedge v(r) \\ &= (v(p) \implies v(q)) \wedge v(r) \\ &= (\underline{true} \implies \underline{false}) \wedge \underline{false} \\ &= \underline{false} \wedge \underline{false} \\ &= \underline{false} \end{aligned}$$

We define a function as **satisfiable** if there is some combination of input values which will make the outcome $\underline{\text{true}}$. If the formula is true for all possible input values, that formula is **valid** and can be referred to as a **tautology**. A tautology is written as $\vdash p \wedge \neg p$.

A formula p **logically implies** another formula q if and only if for all possible boolean value $v(p) \vdash v(q)$. This can be represented as a tautology with $\vdash p \implies q$. Note that we sometimes use $\Rrightarrow$

We can also generalize this over a set of functions with $p_0, p_1, ...p_n \vdash q$.

A formula is a **contradiction** or **falsehood** if it is false for all possible input values. If a formula is neither tautology nor contradiction, then it is a **contingent formula**.

Two formulas are **logically equivalent** if they have the same truth values. For example $p \vee q \leftrightarrow q \vee p$. A **material equivalence** uses the symbol $\Leftrightarrow$, which expresses the "if and only if" idea.

Though any formula can be proven with truth tables, these can sometimes be ungainly. A more concise method of determining whether a formula is a tautology is to use a **proof theory** (so long as the theory is sound).

Proof Theories

Transformational Proof

Transformational Proof is the act of using substitutions to prove a formula. For example

$$\begin{aligned} (p \wedge q) \wedge \neg q &\iff \underline{false} \\ p \wedge (q \wedge \neg q) &\iff \underline{false} \\ p \wedge \underline{false} &\iff \underline{false} \\ \underline{false} &\iff \underline{false} \end{aligned}$$

Common transformations include commutativity, associativity, distributivity, contradiction, the law of the excluded middle, double negation, simplification, DeMorgan's law, contrapositives, equivalence, implications, and idempotents ($p \wedge p \implies p$).

We implicitely use the **rule of substitution** (substituting equivalent formulas for sub-formulas) and the **rule of transistivity** (all lines are equivalent, not just the closest two).

Examples:

$$\begin{aligned}
\neg((p \land q) \implies p) &\iff \underline{false} \\
\neg(\neg(p \land q) \lor p) &\iff \underline{false} \\
\neg(\neg p \lor \neg q \lor p) &\iff \underline{false} \\
(p \land q) \land \neg p &\iff \underline{false} \\
(p \land \neg p) \land q &\iff \underline{false} \\
\underline{false} \land q &\iff \underline{false} \\
\underline{false} &\iff \underline{false}
\end{aligned}$$

$$\begin{aligned}
x &\iff x \land (y \implies x) \\
x &\iff x \land (\neg y \lor x) \\
x &\iff x
\end{aligned}$$

$$\begin{aligned}
\neg\underline{true} &\iff \underline{false} \\
\neg(p \lor \neg p) &\iff \underline{false} \\
\neg p \land p &\iff \underline{false} \\
\underline{false} &\iff \underline{false}
\end{aligned}$$

Natural Deduction

An argument is a collection of formulas beginning with a premise or multiple premises and ending with a conclusion which is justified by the remaining formulas.

If the conclusion of an argument is completely justified by the premises, it is a **deductive argument**. **Inductive arguments** conclude more general knowledge from a small number of particular facts or observations.

In this course, we will be studying deductive arguments. Later on, we will study mathematical induction.

Natural deduction is a collection of rules (inference rules), each of which allws us to infer new formulas from given ones. It is a **forward proof**, which begins from the premises and deduces new formulas which logically follow. We continue this until we have deduced the conclusion.

An **inference rule** is a primitive valid argument form. Each one enables the introduction or elimination of some logical connection.

We use a horizontal line to divide our premises from our conclusion, and write the name of the rule used on each line to the right of the preceeding line. These names may reference previous lines (including premises) by line number. The order of our premises does not matter.

Normal Forms

A literal is a proposition letter or its negation. A formula is in **conjunctive normal form** if it is a conjunction of clauses, where a clause is a disjunction of literals or a single literal. A formula is in

disjunctive normal form if it is a disjunction of clauses, where a clause is a conjunction of literals or a single literal.

There are no repeated literals in a clause and no clause contains $\underline{\text{true}}$ or $\underline{\text{false}}$. No two clauses should contain the same literals. Note that $\underline{\text{true}}$ and $\underline{\text{false}}$ by themselves are in CNF and DNF.

Every formula has an equivalent formula in CNF and DNF.

We can convert our formula to CNF with the following methodology:

1. Remove all $\rightarrow$ and $\implies$ using implication and equivalence laws
2. Use the negation law or DeMorgan's theorem to remove any negated compound subformulas
3. Use distributivity to reduce the scope
4. Simplify so there are no repeated literals in a clause and no clause contains $\underline{\text{true}}$ or $\underline{\text{false}}$

Subproofs

Some natural deduction rules use **subordinate proofs**, which are enclosed by { and }, are indented, and involve and assumption and the conclusion it leads to. Once we've proven a subproof, we refer to it as **closed**. We must close all subproofs to complete the proof.

The **active** formulas at any stage of a proof are those which do NOT occur in a subproof.

Predicate Logic

Predicate logic is a method of formalizing an English sentence. A sentence expressed in predicate logic looks like the following:

"Rich actors collect some valuables" becomes $\forall x \cdot rich(x) \wedge actor(x) \implies \exists y \cdot collects(x, y) \wedge valuable(y)$. "Not everyone in Louisiana speaks French, but everyone in Louisiana knows someone in Louisiana who speaks French" becomes $\neg(\forall x \cdot inL(x) \implies speaks(x, French)) \wedge (\forall c \cdot inL(x) \implies \exists y \cdot knows(x, y) \wedge inL(speaks(y, French)))$.

Types

A type is a set of objects describing the possible values of a variable. We assume types to be non-empty. Any unary predicates are interchangeable with the type of the object they describe.

Quantifiers

For a **universally** qualified variable, the formula must be true for all sbstituions of an object int he domain for the quantified variable. For an **existentially** qualified variable, the formula must be true for some substitution.

We can eliminate $\forall$ by simply selecting a possiblity, i.e. $\forall x \implies x_g$. The opposite of this would be to introduce an $\exists$, i.e. $x_g \implies \exists x$.

We can also introduce $\forall$ terms; informally: if a formula is true for an arbitrary value, it is true for all values. For example:

```
for every g {
  ...
  ...
  P[g/x]
}
A x * P
```

We can similarly eliminate $\exists$ terms with

```
E x * P
for some u P[u/x] {
  ...
  q
}
q
```

Interpretations

Interpretations are sets of rules our equation must follow. An interpretation must contain a **domain** ($xin\{0, 1, 2 \cdots\}$) and a

The **environment** is the mapping between variables and objects within the domain.

For example: $\forall dinD : I[x => \text{something}(x, y)], e[x \rightsquigarrow d]$.

Terms

A predicate formula **satisfies** a logic if it is true in that interpretation and environment. If there exists any interpretation and environment for which it is true, that formula is **satisfiable**. It is a **tautology** (valid) if every environment and interpretation satisfies the formula. It is a **contradiction** if it is not satisfiable in any interpretation or environment.

A logic is **closed** if it contains no free occurances of a variable. In this case, environments have no effect (e.g. it is satisfiable dependant only on the interpretatation, not also the environment).

Predicate logic is **valid** if there exists some interpretation with both valid premises and conclusions. It is **invalid** if and only if there exists some interpretation such that the premises are true but a conclusion is false.

Types of Variables

A **genuine variable** is a free variable such that the universal quantification of it yields a true formula. For example: $\forall x \cdot x + x = 2x$. We tend to represent these with x_g, y_g, etc.

A **universal variable** requires an existential quanitification to be rendered true; it represents a specific but unknown value. Example: $\exists x \cdot x + 1 = 2$. We represent these as x_u, y_u, etc.

Induction

There are two branches in the philosophical study of logic:

- **Deduction** is showing a conclusion follows from the stated premises using rules of inference.

- **Philosophical induction** is the process of deriving general principles from particular observations.
- **Mathematical induction** deals with generalizations, but does not allow exceptions since we are dealing with an ordered domain.

We have Peano's axioms of **arithmetic**:

1. $\forall n \cdot \neg(suc(n) = 0)$
2. $\forall x, y \cdot (suc(x) == suc(y)) \implies (x == y)$
3. $\forall m \cdot m + 0 == m$, $\forall m \cdot 0 + m == m$
4. $\forall m, n \cdot m + suc(n) == suc(m + n)$
5. $\forall n \cdot n \times 0 == 0$
6. $\forall m, n \cdot m \times suc(n) == m \times n + m$
7. $(P(0) \wedge (\forall k \cdot P(k) \implies P(suc(k)))) \implies \forall n \cdot P(n)$

Note that the seventh axiom is the basis for mathematical induction.

Set Theory

A set is a collection of elements or numbers. They have no order and may or may not be infinite. All elements are distinct, i.e. no set contains more than one of the same element.

We define x as being a member of S with $x \in S$ and its converse with $xotinS$. Note that $xotinS \leftrightarrow \neg(x \in S)$.

In addition to defining exactly which members are in a set, we can use set construction notation as follows

$$S = \{2x \cdot x : \mathbb{N} \mid (5 < x) \wedge (x < 10)\}$$

for the set of all even numbers between five and ten.

We can refer to the size of set A as $\#A$ or $|A|$, where the size is equal to the number of elements in A. A **singleton set** is any set with a size of one.

Set Comprehension

We have two axioms for dealing with set comprehension:

- $x \in \{y : S \mid P(y)\} \iff x \in S \wedge P(x)$
- $x \in \{f(y) \cdot y : S \mid P(y)\} \iff \exists y \cdot y \in S \wedge P(y) \wedge x == f(y)$

Special Sets

The **empty set**, which contains no elements, is written as $\emptyset$. This can be defined as $\forall x \cdot \neg(x \in \emptyset)$.

The **universal set** contains all objects of concern (e.g. all countries in the world, all people...). This set is denoted U.

The **power set** of A is the set of all sets such that $\mathbb{P}A = \{B \mid B \subseteq A\}$. For any finite set, $\#\mathbb{P}A = 2^{\#A}$. We have the following axiom for power sets: $S \in \mathbb{P}Q \iff (\forall x \cdot x \in S \implies x \in Q)$.

Set Comparisons

Sets are equal if and only if they contain exactly the same elements. The corresponding axiom is

$$A == B \iff (\forall x \cdot x \in A \iff x \in B)$$

We define A to be a subset of B if all elements of A are contained in B. Note that this includes the case where $A == B$. The subset axiom is $A \subseteq B \iff (\forall x \cdot x \in A \implies x \in B)$.

If A is a subset of B and the two sets are not equal, we refer to A as a **proper subset** of B. $A \subset B \iff A \subseteq B \wedge \neg(A == B)$.

Derived Laws

We can derive a few laws in set theory:

- The empty set is a subset of every set

  $\vdash \emptyset \subseteq A$

- Every set is a subset of itself

  $\vdash A \subseteq A$

- Subsets are transitive

  $\vdash A \subseteq B \wedge B \subseteq C \implies A \subseteq C$

- Equal sets can be shown as subsets

  $\vdash A == B \iff A \subseteq B \wedge B \subseteq A$

Set Functions

We can perform **unions** and **intersections** of sets as follows:

- Set Union: $A \cup B = \{x \mid x \in A \vee x \in B\}$
- Set Intersection: $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

Two sets A and B are **disjoint** if their intersection is empty (i.e. $A \cap B = \emptyset$).

Note: $\#(A \cup B) = \#A + \#B - \#(A \cap B)$

We can also take the **absolute complement** of a set with $A' = \{x \mid x \in U \wedge \neg(x \in A)\}$ and the **relative complement**, or set difference, with $A - B = \{x \mid x \in A \wedge xotinB\}$.

Derived Laws

These laws can be derived from our axioms:

- $A \cap B = B \cap A$
- $A \cup B = B \cup A$
- $A \cap (B \cap C) = (A \cap B) \cap C$
- $A \cup (B \cup C) = (A \cup B) \cup C$

- $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
- $(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$
- $(A \cap B)' = A' \cup B'$
- $(A \cup B)' = A' \cap B'$
- $A \cap \emptyset = \emptyset$
- $A \cup \emptyset = \emptyset$
- $A - \emptyset = A$
- $\emptyset - A = \emptyset$
- $\emptyset' = U$
- $A \cap U = A$
- $A \cup U = U$
- $A - U = \emptyset$
- $U - A = A'$
- $U' = \emptyset$
- $A \cap B \subseteq A$
- $A \subseteq A \cup B$

Tuples

A **tuple** is an object containing two or more components. For example, $(Ferrari, \mathit{Red}, 1962)$ is a tuple containing a make, color, and year of a car. We user the terms pair and triple as shorthand for 2- and 3-item tuples. Note that the order of element in a tuple is important.

We can also use **maplets** to describe pairs, i.e. $(a, b) \implies a \mapsto b$.

Cartesian Product

The **Cartesian Product** of two sets B and C is written as $B \times C$. It is the set of all pairs of elements such that the first element in the pair belongs to B and the second belongs to C.

$$B \times C = \{(b, c) \mid b \in B \wedge c \in C\}$$

Relations

Relations are subsets of the Cartesian Products of two or more sets such that it is a set of tuples, or $R \subseteq A \times B$. We can also state that R is a relation from set A to B as $R : A \leftrightarrow B$. A set of pairs is a **binary relation**.

For example, for some set $\mathit{People} = \{p_1, p_2, \dots p_n\}$ and $Cars = \{c_1, c_2, \dots c_n\}$, we may have $own : \mathit{People} \times Cars = \{(p_1, c_2), (p_4, c_1), (p_8, c_5)\}$ where the relation *own* shows which people own which cars.

For a binary relation, the first item is the **domain** and the second is the **range**. For relation $R : D \leftrightarrow B$, we have $dom\ R = \{x : D \mid \exists y : B \cdot (x, y) \in R\}$ and $ran\ R = \{y : B \mid \exists x : D \cdot (x, y) \in R\}$.

We define the **inverse** of R as $R^\sim = \{(b, a) \mid (a, b) \in R)\}$ and the **id** as $id\ D = \{(a, a) \mid a \in D\}$. The **relative complement** of two relations R and S is $S \cdot R == R; S == \{(a, c) \mid \exists b \cdot (a, b) \in R \wedge (b, c) \in S\}$.

We define **iteration** over a relation as $R^n = R; R^{n-1}$, where $R^0 = id\ A$. Iterations obey the following rules: $R^{m+n} == R^m; R^n$ and $R^{mn} == (R^m)^n$.

The **relative image** of S through R is $R(|S|) == \{y : B \mid \exists x : D \cdot (x, y) \in R \wedge x \in S\}$. Basically, the relative image is the set of elements that are the second element of any pair in R who's first element is in S.

We can use the following rules when dealing with relative images: $R(|D \cup B|) == R(|D|) \cup R(|B|)$, $R(|D \cap B|) \subseteq R(|D|) \cap R(|B|)$, and $R(|dom\ R|) == ran\ R$.

Derived Laws

For two relations $S : D \leftrightarrow B$ and $T : D \leftrightarrow B$, we have

- $dom\ S \subseteq D$
- $dom\ (S \cup T) == (dom\ S) \cup (dom\ T)$
- $dom\ (S \cap T) \subseteq (dom\ S) \cap (dom\ T)$
- $ran\ S \subseteq B$
- $ran\ (S \cup T) == (ran\ S) \cup (ran\ T)$
- $ran\ (S \cap T) \subseteq (ran\ S) \cap (ran\ T)$

For $R : D \leftrightarrow B, S : B \leftrightarrow C, T : C \leftrightarrow D$, we can derive

- $R; (S; T) == (R : S); T$
- $(R; S)^\sim == S^\sim; R^\sim$
- $id\ (dom\ R) \subseteq R; R^\sim$
- $id\ (ran\ R) \subseteq R^\sim; R$
- $id\ A; R == R$
- $R; id\ B == R$

Restrictions

We can use the following functions to interact between our relations and our tuples:

- Domain restriction: $S \lhd R == \{(a, b) \cdot a : D, b : B \mid (a, b) \in R \wedge a \in S\}$
- Domain co-restriction / subtraction: $S ⩤ R == \{(a, b) \cdot a : D, b : B \mid (a, b) \in R \wedge \neg(a \in S)\}$
- Range restriction: $R \rhd S == \{(a, b) \cdot a : D, b : B \mid (a, b) \in R \wedge b \in S\}$
- Range co-restriction / subtraction: $R ⩥ S == \{(a, b) \cdot a : D, b : B \mid (a, b) \in R \wedge \neg(b \in S)\}$

where we retain only the pairs whos first/second element is/is not part of a set.

We can also override the values in our relation with $R \oplus S == ((dom\ S) ⩤ R) \cup S$. This will replace any pairs in R with any pairs in S of the same domain. Note that $R \oplus R == R$ and $R \oplus (S \oplus T) == (R \oplus S) \oplus T$.

We can derive the following laws for domain restrictions:

- $D \lhd (B \lhd R) == (D \cap B) \lhd R$
- $D ⩤ (B ⩤ R) == (D \cup B) ⩤ R$
- $(D \lhd R) \cup (D ⩤ R) == R$

Functions

Elements of the system can also be elements of a compound type, denoted by arrows with various lines to explain the function: a standard arrow is drawn. With a tail ($\rightarrowtail$), this becomes injective. With a second head ($\twoheadrightarrow$), it is surjective. With both of these (i.e. a tail and a second head $⤖$), it becomes bijective. With a vertical strike-through ($⇸$), it becomes non-total. Of course, any of these may be used in concert.

A function must have a single range value for every domain value (i.e. a one-to-one mapping). Any elements in the origin set which do not have a mapping are said to be undefined (for that element).

A set of functions is called a **function space**.

The set of all **partial functions** (all functions are partial functions) between A and B is $A ⇸ B == \{f : A \to B \mid \forall x : A \cdot \forall y, z : B \cdot (x, y) \in f \wedge (x, z) \in f \implies y == z\}$.

A **total function** from sets A to B is defined for all elements in A. This is a subset of partial functions from A to B. $A \to B == \{A ⇸ B \mid dom\ f == A\}$.

A function f is **surjective** if every element in B is matched with some element in A. We have $A \twoheadrightarrow B \iff \forall b : B \cdot \exists a : A \cdot f(a) == b$. Note that for a function ro be surjective, there must be at least as many elements in A as in B.

We have the **partial surjective** $A ⤀ B == \{f : A ⇸ B \mid ran\ f == B\}$ and the **total surjective** $A \twoheadrightarrow B == \{f : A ⤀ B \mid dom\ f == A\} == \{f : A \to B \mid ran\ f == B\}$.

A function is **injective** if its inverse is a function such that every element of B appears at most once as the second component of any ordered pair in f. If f is injective, so is $f^{\sim}$.

We have the **partial injective** $A ⤔ B == \{f : A ⇸ B \mid f^{\sim} \in B ⇸ A\}$ and the **total injective** $A \rightarrowtail B == \{f : A ⤔ B \mid dom\ f == A\} == \{f : A \to B \mid f^{\sim} \in B ⇸ A\}$.

A function is **bijective** if it is both surjective and injective such that every element from B appears exactly once as the second componenet of an ordered pair in f.

We have the **partial bijective**

$$\begin{aligned} A ⤗ B &== \{f : A ⤔ B \mid ran\ f == B\} \\ &== \{f : A ⤀ B \mid f^{\sim} \in B ⇸ A\} \\ &== (A ⤔ B) \cap (A ⤀ B) \end{aligned}$$

and the **total bijective**

$$\begin{aligned} A ⤖ B &== \{f : A \rightarrowtail B \mid ran\ f == B\} \\ &== \{f : A \twoheadrightarrow B \mid f^{\sim} \in B ⇸ A\} \\ &== (A \rightarrowtail B) \cap (A \twoheadrightarrow B) \end{aligned}$$

When the domain of a function is infinite, we can't write out all the pairs to describe the function, so we use set comprehension. Example: $double == \{(x, 2x) \cdot x \in \mathbb{N}\}$.

Function composition is the sequential application of multiple functions to a value. We write this as $(f; g)(x) == g(f(x))$ provided that $x \in \mathit{dom}\ f \wedge \mathit{ran}\ f \in \mathit{dom}\ g$.

- If f is a function (x, y), $y = f(x)$
- If f is an injective function (x, y), $x = f^{\sim}(y)$
- If R is a relation $X \leftrightarrow Y$, $y = R(|x|)$
- If R is a relation $X \leftrightarrow Y$, $x = R^{\sim}(|y|) = \mathrm{dom}(R \rhd y)$

Formal Specification

Formal Specification means writing the system to built in a language such that each formula is unambiguous (i.e. has a single distinct meaning). We describe what a system should do, for all possible input cases, but don't worry about how it is to be implemented.

Since systems can be so large and complex so as to make set theory and predicate logic alone unable to provide enough structure to define every aspect of them, we need a formal language which can be used to specify changes over time. The **Z Formal Specification Language** is one such tool.

In Z, we think of the execution of a system as a progression of system states, where each state is a mapping between variables / state elements and values. In a Z system specification, we must define the following (in order):

1. Types of data manipulated by the system
2. Constants and their types
3. System elements (states), their types, and invariants about the relationship between said elements
4. Initial system state
5. System operations
   - System operations which change the system state or elements (Δ)
   - System operations which report the system state (Ξ)

We use **Z Schemas** for the latter three, sometimes four, of these.

A schema is a way of grouping information. So, if we have

```
A = 1
B = 2
C = 3
-----
A < B
B < C
```

we could say that the declarations (the **signature**) above the line is a schema where we introduce elements and the **predicate** part below the line contains well-formed formulas which are implicitely joined together.

In the system state schema, the predicate part lists invariants, which must be true in all states.

We can reference schema by name and include them later simply by mentioning them. i.e.

```
-- Declare ----
- X is a number
- Y is a number
---------------
- X < Y
---------------

-- Initial ----
- Declare
---------------
- X == 0
- Y == 5
---------------
```

To describe how a schema changes, we must refer to both the present and future values of an element. We do so with X and X'. We can refer to the value of the next state with $CurrentState'$. We can refer to both cases with **delta schemas**, e.g. we can replace

```
Schema2
Schema2'
```

with Δ `Schema2`

We also use deltas in the system state reference in the signature for any operation schemas which change the system state, e.g.

```
-- MySystem ----
- X is a number
- Y is a number
----------------
- X < Y
----------------

-- Inc ----------
- Delta MySystem
-----------------
- Xprime == X + 1
- Yprime == Y + 1
-----------------
```

We use ? and ! for input and output, e.g. p? is an input and q! is an output.

Operations tend to have both pre- and post-conditions. So a non-negative subtraction schema may look like

```
-- Subtract ------
- Delta MySystem
- i? is a number
------------------
- i? <= X
- Xprime == X - i?
- Yprime == Y
------------------
```

where we have the pre-condition `i? <= X` and the other two predicate lines are post-conditions.

We can refer to several types in Z:

- Built-in types ($\mathbb{Z}, \mathbb{R}$)
- Generic types (*Flight*, *Location*)

  Note: we do not need to know the implementation of the object, simply that it is a universal set.
- Free (enumerated) types (`Colors ::== Red | Blue | Green`)
- Compound types

For example, if we have type *Books* of which there is some pre-defined parameter which serves as the maximum number, we may define

```
[Books]
MaxBooks : N
---
#Books <= MaxBooks
```

Exception handlers use Ξ specifications. For example, to ensure one is at least 18 years old, we may have

```
-- TooYoung --
Xi Vars
person? : People
error : Errors, Errors ::|| TooYoung
--
Age(person?) < 18
error! = TooYoung
--
```

Program Correctness

The goal of program correctness is to show that a program has the correct behaviour. A specification describes the desired output for admissible inputs: correctness, then, is relative to this specification.

We have proof rules which can be used to prove that a program satisfies its specification.

Program testing can be used to show the presence of bugs, but *never to show their absence.* At best, verification helps reduce bugs in code and reduces debugging and maintenance costs. Remember, though, that we are only checking that the program obeys the specification, not that the specification is accurate.

Formal Verification describes the program specification in some logic, then uses a proof procedure to prove that the program satisfies its specification. Formal verification allows us to check programs for all possible inputs.

We can check **partial correctness** by verifying that the program *does not terminate with invalid post-conditions.* Note that this does not mean the program must terminate.

To check **total correctness**, however, we must ensure termination occurs as well.

We may sometimes have the need for logical variables, i.e. variables which do not exist in the program itself. For example:

```
x = x0; x > 0
y^2 < x0
```

Proving Correctness

A specification for a program consists of a pre-condition and a post-condition. Both are well-formed formulas which describe the system state. For example:

```
  @ x == x0 && x > 0 @
double(x);
  @ x == 2 * x0 @
```

Note that it is possible for us to have no pre-conditions, in which case we simply annotate our code with `@ true @`.

We prove programs by including assertions between virtually every command such that the assertions form a logical link between the pre-conditions and the post-conditions. For example:

```
  @ y == y0 & x == x0 @
R := x;
  @ y == y0 & R == x0 @ Asn
x := y;
  @ x == y0 & R == x0 @ Asn
y = R;
  @ x == y0 & y == x0 @ Asn
```

The notation `Asn` is used to refer to an annotation which is simply the previous proof line given a new assignment for some variable. We can use any natural deduction law in the same way, though any proof requiring more than a single application of a simple rule must be proven separately. For example:

```
  @ x >= 5 & y >= 0 @
  @ x + y >= 5 @ Implied(VC 1)
z := x;
  @ z + y >= 5 @ Asn
z := z + y;
  @ z >= 5 @ Asn
v := z;
  @ v >= 5 @ Asn
  @ v >= 3 @ arith(common)

VC 1
x >= 5 & y >= 0 |= x + y >= 5
1) x >= 5 & y >= 0 premise
2) y >= 0 and_e on 1
3) x >= 5 and_e on 1
4) x + y >= 5 + 0 by arith(common) on 2,3
5) x + y >= 5 by arith(common) on 4
```

We can **strengthen** or **weaken** our proof by either assuming more than we need (pre-condition strengthening) or concluding less than is true (post-condition weakening).

Conditionals

When we come across a conditional, we must prove the conclusion for every branch. For example:

```
  @ true @
if (max < A) then {
    @ max < A @ If-Then
    @ A >= A @ Implied(arith)
  max := A;
    @ max >= A @ Asn
}
  @ max >= A @ If-Then (VC 1)

VC 1: !(max < A) |= max >= A
```

Note that we must prove that we conclude `max >= A` both inside and outside of the conditional (i.e. by negating the if-test).

This is shown more explicitely with an if-then-else statement:

```
  @ P @
if B then {
    @ P & B @ If-Then-Else
  C1;
    @ Q @
} else {
    @ P & !B @ If-Then-Else
  C2;
    @ Q @
}
  @ Q @ If-Then-Else
```

Loops

A loop (such as a `while` statement) has an **invariant** and a **guard**. The invariant is some w.f.f. which is true both at the beginning and end of each loop iteration, and the guard is the looping condition. By partial `while` analysis, we can see that at the end of the loop the invariant and the opposite of the guard is true. For example:

```
  @ I @
while G do {
    @ I & G @ Invariant & Guard
  C;
    @ I @
}
  @ I & !G @ Partial While
```

Arrays

Arrays are a mapping of indices to values. We can access the ith member of array A with $A[i]$. When an array appears on the right-hand-side of an equation, we can simply treat it as we have other variables. **Array assignment**, though, must be handled differently.

We use relational overriding to show this relationship:

```
  @ (B (+) {k,14})[4] == c @
B[k] = 14;
  @ B[4] == c @
```

or more expansively:

```
  @ ((A (+) {j, c+1}) (+) {k, b-1})[5] == 6 @
A[j] = c+1;
  @ (A (+) {k, b-1})[5] == 6 @
A[k] = b-1;
  @ A[5] == 6 @
```

Proving Incorrectness

To prove incorrectness, we must provide a counter-example. Simply trace the program to determine a set of initial and final values for each variable, then prove that though the inital variables satisfy the pre-condition, the final variables do not satisfy the post-condition.