MODULE 2

ABOUT COURSE

- logic can be used for specifying behaviour of systems & checking correctness of programs
- formal verification (FV) is based on logical reasoning
  - ↳ aka formal methods or computer-aided verification
  - ↳ check correctness of program for all possible inputs
- a logic is formal if there's only one possible interpretation of a formula in the language
  - ↳ e.g. (logical argument) If train arrives late & there's no taxis at station, then John is late for meeting. John is not late for meeting. Train arrives late. Therefore, there are taxis at station.
    - Letter | English

      p | train arrives late
      q | there's taxis at station
      r | John is late for meeting
    - if p and not q then r ; not r
- 4 main topics covered in course
  - ↳ propositional logic
  - ↳ predicate logic
    - describing relationships btwn objects & quantification over them (e.g. every course has an instructor)
  - ↳ set theory & specification
    - ways of describing what a system is required to do
  - ↳ program correctness
    - Floyd-Hoare logic : reasoning about programs

ELEMENTS OF A LOGIC

- any logic has 3 parts to it:
  1) syntax
     - ↳ parts of sentence (e.g. noun, verb)
  2) semantics
     - ↳ meaning / definition (e.g. translating from English to French)
  3) proof theory
     - ↳ aka deductive systems
     - ↳ synonyms
- syntax defines a well-formed formula (wff)
- semantics: "$\models$" for a logic describe what it means for formula / argument to be "true"
  - ↳ $\models P$ means P (wff in logic) is valid
    - i.e. P is a theorem b/c it's always true
  - ↳ $P_1, P_2, \ldots, P_n \models Q$ means from premises $P_1, P_2, \ldots, P_n$, can conclude Q
  - ↳ $\models$ isn't part of wff but meta-level symbol that says smth abt goal wrt semantics
  - ↳ define truth for a logic
- $\models P$ is goal when we don't know if it's a theorem & it's a theorem when we know it's valid (can also be called sequents / problems)
- $\models$ is said as entails, valid, or semantic entailment
- proof theories provide means of calculating things abt the logic
  - ↳ defines "$\vdash$"
    - said as proves

↳ $P_1, P_2, \ldots, P_n \vdash Q$ means from $P_1, P_2, \ldots, P_n$, can prove Q using proof theory
↳ may be multiple proofs for same logic
↳ define **proof** for a logic

- proof theories are methods that perform **mechanical manipulations on strings of symbols**
  ↳ based on **pattern matching**
- qualities of proof theory:
  ↳ **soundness**: if whenever $P_1, P_2, \ldots, P_n \vdash Q$ then $P_1, P_2, \ldots, P_n \models Q$
    - $\vdash \rightarrow \models$ (smth has proof then it's true)
  ↳ **completeness**: if whenever $P_1, P_2, \ldots, P_n \models Q$ then $P_1, P_2, \ldots, P_n \vdash Q$
    - $\models \rightarrow \vdash$ (smth is true then it has proof)

PROPOSITIONAL LOGIC SYNTAX

- aka sentential logic, propositional calculus, sentential calculus, or Boolean logic

formula in propositional logic consists of:
1) 2 constant symbols: true & false
2) prime proposition symbols (usually lower case)
3) propositional connectives ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$)
4) brackets

- **propositional connectives** (i.e. logical operators):

Symbol	Informal Meaning	george
$\neg$	negation (not)	!
$\wedge$	conjunction (and, both)	&
$\vee$	disjunction (or, at least one of)	|
$\Rightarrow$	implication (implies, conditional, if ... then)	=>
$\Leftrightarrow$	equivalent (biconditional, if & only if)	<=>

- a wff of propositional logic are obtained by following construction rules:
  ↳ proposition symbols & constants (true & false) are **atomic formulas**
  ↳ if P & Q are wffs, then these are formulas (more specifically, **compound propositions**)
    - $\neg P$
    - $P \wedge Q$
    - $P \vee Q$
    - $P \Rightarrow Q$
    - $P \Leftrightarrow Q$
  ↳ no other expressions are formulas & won't use XOR
    - XOR: $(a \vee b) \wedge \neg(a \wedge b)$
- brackets around outermost formula are usually omitted & can be omitted using rules of **precedence**:

$\neg$ highest
$\wedge$
$\vee$
$\Rightarrow$
$\Leftrightarrow$ lowest

  ↳ e.g. $((\neg p) \wedge (\neg q)) \vee r$
  ↳ e.g. $(p \Rightarrow ((q \wedge r) \vee x)) \Leftrightarrow (y \vee r)$
- all binary logical connectives are **right associative**
  ↳ e.g. $a \vee b \vee c$ means $a \vee (b \vee c)$
- terminology for propositional connectives:
  ↳ $P \wedge Q$: conjuncts
  ↳ $P \vee Q$: disjuncts

↳ $P \Rightarrow Q$: P is premise/antecedent/hypothesis & Q is consequent/conclusion

contrapositive of $P \Rightarrow Q$ is $\neg Q \Rightarrow \neg P$

↳ logically equivalent

- when formalizing natural language, proposition symbols rep declarative sentences (i.e. either true or false)
  - ↳ interrogative (questions) & imperative (commands) sentences aren't propositions
  - ↳ don't use T, F, t, or f for symbols
- prime proposition is atomic / indecomposable
  - ↳ e.g. The snow is red.
- compound proposition contains multiple primes joined by connectives
  - ↳ e.g. The snow is red and the grass is green.
- when formalizing, match form of sentence as closely as possible
  - ↳ match order of conjuncts

Connective	Possible interpretations in English
$\neg P$	not P, P does not hold, it is not the case that P, P is false
$P \wedge Q$	P and Q, P but Q, not only P but Q, P while Q, P despite Q, P yet Q, P although Q
$P \vee Q$	P or Q, P or Q or both, P "and/or" Q, P unless Q
$P \Rightarrow Q$	if P then Q, Q if P, P only if Q, Q when P, P is sufficient for Q, Q is necessary for P, P implies Q
$P \Leftrightarrow Q$	P if and only if Q (P iff Q), P is necessary and sufficient for Q, P exactly if Q, P is equivalent to Q

- e.g.
  1. It is cold but it is not snowing.
  2. It is neither snowing nor cold.
  3. It is cold if it is snowing.
  4. It is snowing only if it is cold.
  5. If a request occurs then it will be acknowledged or the process does not make progress.

↳ c means "it's cold"

↳ s means "it's snowing"

1) $c \wedge \neg s$
2) $\neg(s \vee c)$
3) $s \Rightarrow c$
4) $s \Rightarrow c$
5) $r \Rightarrow (a \vee \neg p)$
   $(r \Rightarrow a) \vee \neg p$ > ambiguity

- ambiguous sentence can have more than 1 distinct meaning
  - ↳ conjunction: $P \wedge Q = Q \wedge P$ (commutative)
    - "The driver hit the cyclist and drove on." & "The driver drove on and hit the cyclist." are diff b/c there's time element in sentence
  - ↳ disjunction: $P \vee Q$ means at least one is true (both can be true)
    - diff from XOR where exactly one disjunct is true
  - ↳ implication: false implies anything problem states that if antecedent is always false, then implication is always true

- when ambiguity occurs, can compare intuitive meaning to diff formalizations
  - ↳ e.g. it's not the case that (the student passing the assignments) is sufficient for the student to pass SE212)
    - a means "the student passes the assignments"
    - p means "the student passes SE212"

a	p	intuitive meaning	¬ a ⇒ p	¬ (a ⇒ p)
T	T	F	T	F
T	F	T	T	T
F	T	F	T	F
F	F	F	F	F

PROPOSITIONAL LOGIC SEMANTICS

- semantics define ⊨ (entails, truth, valid argument, logical implication) for propositional logic
  - ↳ provide interpretation (mapping) of values in one world to another world
    - we're relating wff of the syntax to set of truth values & their functions
- domain of semantic function is syntax for propositional logic
- range of semantic function is set of truth values Tr = {T, F}
  - ↳ classical logic is 2-valued
    - T denotes property of formula being True
    - F denotes property of formula being False
- Boolean valuation (BV) is function from set of wffs in propositional logic to set Tr
  - ↳ semantics are described using BV
  - ↳ aka model/interpretation
- given formula p∧q, [p∧q] means "meaning of formula" in certain BV
  - ↳ [ ] is function mapping syntax to its value

functions NOT, AND, OR, IMP, & IFF take on truth values (1 for NOT & 2 for rest) & returns a truth value

↳

p	NOT p
T	F
F	T

p	q	p AND q	p OR q	p IMP q	p IFF q
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

  - ↳ e.g. show truth value associated w/formula (p ⇒ q) ∧ r in BV where [p] = T, [q] = F, [r] = F

$$
\begin{aligned}
[(p \Rightarrow q) \wedge r] &= [p \Rightarrow q] \text{ AND } [r] \\
&= ([p] \text{ IMP } [q]) \text{ AND } [r] \\
&= (T \text{ IMP } F) \text{ AND } F \\
&= F \text{ AND } F \\
&= F
\end{aligned}
$$

- use truth tables to describe meaning of function in all BV
  - ↳ all tables have:
    - a row for each possible BV
    - cells that have truth values for subformulas labelled in column
  - ↳ e.g. truth table for ¬(p∧¬q)

p	q	¬q	p ∧ ¬q	¬(p ∧ ¬q)
T	T	F	F	T
T	F	T	T	F
F	T	F	F	T
F	F	T	F	T

← syntax

← BV

- ASCII soln to BV question in george:

```
#u nday
#a 01
#q 03

/*

For the Boolean valuation [p] = F, [q] = F, [r] = T

  [(p => q)  & r]
= ([p] IMP [q]) AND [r]
= (F IMP F) AND T
= T

*/
```

- ASCII soln to truth table question in george:

```
#u nday
#a 01
#q 06

/*

p | q | r ||  p => q |  (p => q) & r
-------------------------------------
F | F | F ||     T    |       F
F | F | T ||     T    |       T
F | T | F ||     T    |       F
... continue with the rest of the rows
*/
```

- formula P is satisfiable if there's a BV st [P] = T
  - ↳ i.e. truth table has at least 1 T in final column
  - ↳ e.g. is $\neg(p \Rightarrow q) \wedge \neg(\neg r \Rightarrow \neg p)$ satisfiable ?

    Yes: [p] = T, [q] = F, [r] = F

    To prove, $\neg(p \Rightarrow q) \wedge \neg(\neg r \Rightarrow \neg p)$

    = (¬(p ⇒ q)) AND (¬(¬r ⇒ ¬p))

    = NOT (p IMP q) AND NOT(NOT [r] IMP NOT [p])

    = NOT (T IMP F) AND NOT(NOT F IMP NOT T)

    = T AND NOT (T IMP F)

    = T AND T

    = T
- formula P is tautology / valid if [P] = T for all BV
  - ↳ i.e. truth table has all Ts in final column
  - ↳ when formula Q is tautology, write ⊨ Q (entails Q)
- formula P logically implies formula Q iff for all BV, if [P] = T then [Q] = T
  - ↳ P ⊨ Q
  - ↳ P ⊨ Q iff ⊨ P ⇒ Q iff P ⇒ Q is tautology
    - ⊨ P means true ⊨ P
  - ↳ set of formulas $P_1, P_2, \ldots, P_n$ logically imply formula Q iff for all BV, if

$[P_1] = T, [P_2] = T, \ldots, [P_n] = T$ then $[Q] = T$

- aka **valid argument**
- i.e. $P_1 \wedge P_2 \wedge \ldots \wedge P_n \vDash Q$
- i.e. $\vDash P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow Q$

↳ only care abt rows of truth table where it's T for every premise

- e.g. $p \Rightarrow q, q \Rightarrow p \vDash q \wedge p$?

p	q	$p \Rightarrow q$	$q \Rightarrow p$	$q \wedge p$
T	T	T	T	T
T	F	F	T	F
F	T	T	F	F
F	F	T	T	F

not match

↳ premises that are both T have conclusion that's F so this is not a valid argument

propositional formula A is **contradiction** if $[A] = F$ for all BV

↳ last column in TT is all Fs

↳ e.g. $p \wedge \neg p$ is contradiction

- **contingent** formula is neither tautology nor contradiction
  ↳ mixture of Ts & Fs in column rep formula in TT

	Last column of truth table
contradiction	all Fs
tautology ($\vDash P$)	all Ts
satisfiable	at least one T
contingent	at least one T & one F

2 formulas, P & Q, are **logically equivalent** iff for all BV, $[P] = [Q]$

↳ $P \langle\equiv\rangle Q$

↳ $P \langle\equiv\rangle Q$ iff $P \Leftrightarrow Q$

collection of formulas is **consistent** if there's a BV in which all formulas are T

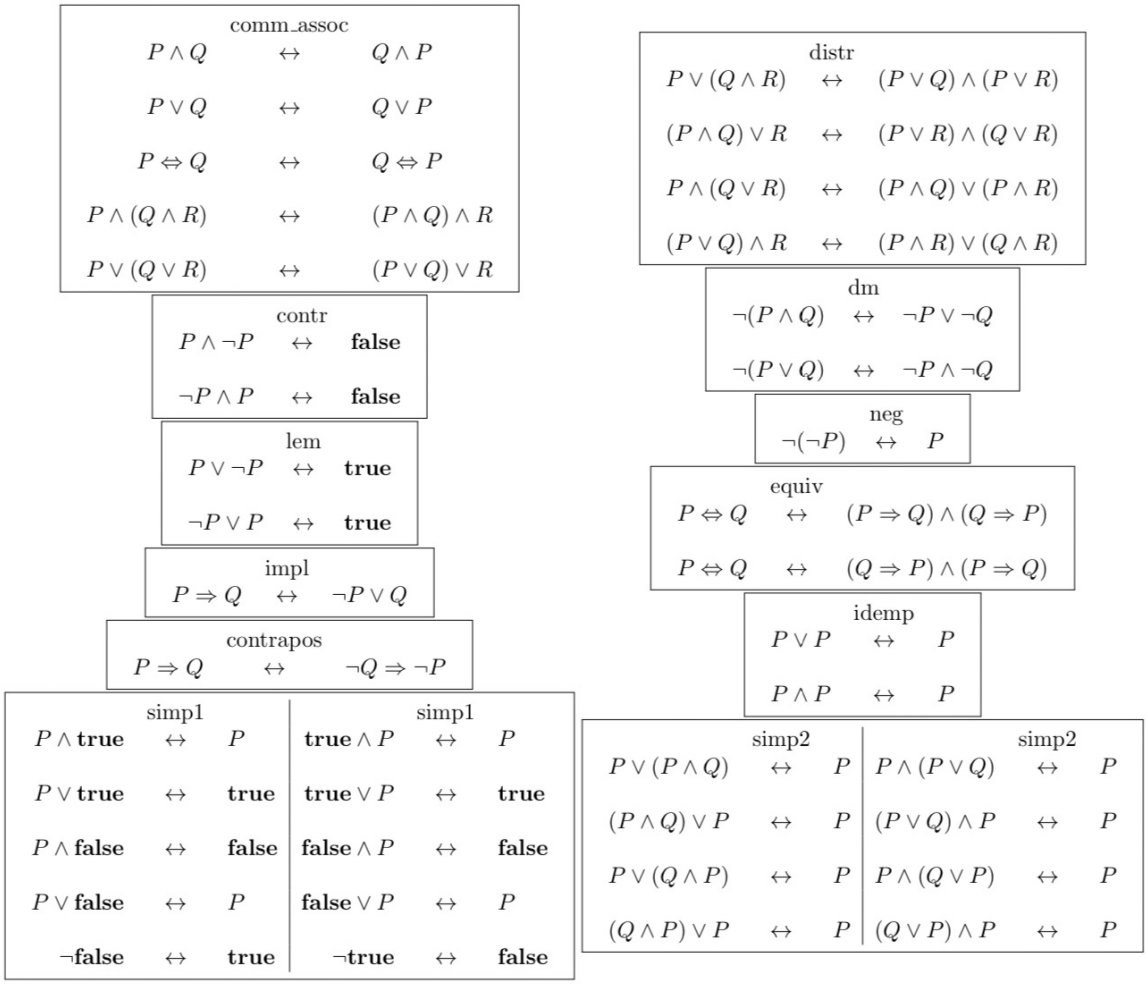
↳ to check:

- BV that maps each formula to T
- BV that maps conjunction of formulas to T (i.e. conjunction is satisfiable)
- conjunction of formulas is not contradiction

- if set of formulas in premise of argument aren't consistent, can be used to prove contradiction (consistency means at least 1 line in truth table where all formulas are T)
  ↳ i.e. $p, \neg p \vDash q \wedge \neg q \leftrightarrow p, \neg p \vDash$ false
  ↳ **false implies anything** problem: nothing is proven abt system if there's inconsistent premises
- for n proposition symbols in formula, need TT w/ 2^n rows
- **proof theory** is another way of determining whether formula is tautology
  ↳ as long as it's sound, can use it in place of TT

PROOF THEORY · TRANSFORMATIONAL PROOF

- **transformational proof ($\leftrightarrow$)**: means of determining that 2 wffs are logically equivalent by repeated exchange of subformulas that results in P being transformed into Q
  ↳ each step must follow logical law



- 2 implicit rules:
  - ↳ rule of substitution: sub equivalent formula for subformula
  - ↳ rule of transivity: if $P \leftrightarrow Q$ & $Q \leftrightarrow R$, then $P \leftrightarrow R$

rules of thumb:

1) eliminate implicity & equivalence using law of impl, equiv, & contrapos backwards
2) simplify ASAP (simp1, simp2, idemp, neg, contr, lem)
3) sometimes use various kinds of simplification bwd to prep for using distributivity

- transformational proof is sound & complete for propositional logic
  - ↳ if $P \leftrightarrow Q$ can be proved, then $P \Leftrightarrow Q$ (soundness)
  - ↳ if $P \Leftrightarrow Q$, then $P \leftrightarrow Q$ can be proved (completeness)
- e.g. $\neg(\neg p \vee \neg(r \vee s)) \leftrightarrow p \wedge r \vee p \wedge s$

  LS: $\neg(\neg p \vee \neg(r \vee s))$

  $\leftrightarrow \neg\neg p \wedge \neg\neg(r \vee s)$ dm

  $\leftrightarrow p \wedge (r \vee s)$ neg

  $\leftrightarrow p \wedge r \vee p \wedge s$ dist
- e.g. $p \vee p \wedge q \leftrightarrow p$

  LS:

  $p \vee p \wedge q$

  $\leftrightarrow p \wedge \text{true} \vee p \wedge q$

  $\leftrightarrow p \wedge (q \vee \neg q) \vee p \wedge q$

  $\leftrightarrow (p \wedge q) \vee (p \wedge \neg q) \vee (p \wedge q)$

  $\leftrightarrow (p \wedge q) \vee (p \wedge \neg q)$

  $\leftrightarrow p \wedge (q \vee \neg q)$

  $\leftrightarrow p \wedge \text{true}$

  $\leftrightarrow p$

- e.g. $\neg((p \wedge q) \Rightarrow p) \leftrightarrow$ false

  LS:

  $\neg((p \wedge q) \Rightarrow p)$

  $\leftrightarrow \neg(\neg(p \wedge q) \vee p)$ impl

  $\leftrightarrow \neg\neg(p \wedge q) \wedge \neg p$ dm

  $\leftrightarrow p \wedge q \wedge \neg p$ neg

  $\leftrightarrow p \wedge \neg p \wedge q$ comm_assoc

  $\leftrightarrow$ false $\wedge q$ contr

  $\leftrightarrow$ false simpl

- e.g. $\neg$ true $\leftrightarrow$ false

  LS: $\neg$ true

  $\leftrightarrow \neg(p \vee \neg p)$ by lem

  $\leftrightarrow \neg p \wedge \neg\neg p$ by dm

  $\leftrightarrow \neg p \wedge p$ by neg

  $\leftrightarrow$ false by contr

- e.g. $p \wedge (\neg(\neg q \wedge \neg p) \vee p) \leftrightarrow p$

  LS: $p \wedge (\neg(\neg q \wedge \neg p) \vee p)$

  $\leftrightarrow p \wedge (\neg\neg q \vee \neg\neg p \vee p)$ by dm

  $\leftrightarrow p \wedge (q \vee p \vee p)$ by neg

  $\leftrightarrow p \wedge (q \vee p)$ by idemp

  $\leftrightarrow p$ by simp2

- application is simplifying code

```
if (i OR NOT o) {
  if (NOT (o AND q) ) {
    C1
  } else {
    if (o AND NOT q)  {
      C2
    } else {
      C3
    }
  }
}
else {
  C4
}
```

i	o	q	Action
T	T	T	C3
T	T	F	C1
T	F	T	C1
T	F	F	C1
F	T	T	C4
F	T	F	C4
F	F	T	C1
F	F	F	C1

↳ C2 is dead code: $(i \vee \neg o) \wedge \neg\neg(o \wedge q) \wedge (o \wedge \neg q)$

$\leftrightarrow (i \vee \neg o) \wedge o \wedge q \wedge o \wedge \neg q$ by neg

$\leftrightarrow (i \vee \neg o) \wedge o \wedge o \wedge$ false by contr

$\leftrightarrow$ false by simpl

- set of conditions leading to C2 are inconsistent / a contradiction

↳ C4 can be simplified: $\neg(i \vee \neg o)$

$\leftrightarrow \neg i \wedge \neg\neg o$ by dm

$\leftrightarrow \neg i \wedge o$ by neg

↳ simplified code:

```
if (i AND o AND q)  {
  C3
} else {
  if (NOT(i) AND o) {
    C4
  } else {
    C1
  }
}
```

- **literal**: proposition symbol or negation of one
  conjunctive normal form (CNF): conjunction of 1+ clauses
  ↳ clause is disjunction of literals or single one
- **disjunctive normal form (DNF)**: disjunction of 1+ clauses
  ↳ clause is conjunction of literals or single one
- every formula can be converted to equivalent in CNF or DNF
  ↳ normal form of formula is not unique
  ↳ formulas consisting only of true & false are both CNF & DNF
    - true & false cannot be clause in conjunction / disjunction w/ others
- e.g. is following CNF or DNF?
  ↳ $(p \wedge q \wedge r) \vee (\neg q \wedge \neg r)$ DNF
  ↳ $\neg (p \Rightarrow q) \wedge (r \Leftrightarrow q)$ neither
  ↳ $(p \wedge q) \vee \neg q \vee \neg r$ DNF
  ↳ $(p \vee q) \wedge \neg r$ CNF
  ↳ $p \vee q$ both
  ↳ $\neg (p \wedge q) \vee r$ neither
  ↳ $\neg p$ both
  ↳ $p \wedge r \vee p \wedge (q \vee \neg r)$ neither
- e.g. $\neg a \vee \neg (b \wedge \neg a) \leftrightarrow$ true

  LS:

  $\neg a \vee \neg (b \wedge \neg a)$

  $\leftrightarrow \neg a \vee \neg b \vee \neg \neg a$ dm

  $\leftrightarrow \neg a \vee \neg b \vee a$ neg

  $\leftrightarrow$ true $\vee \neg b$ lem

  $\leftrightarrow$ true simpl

PROOF THEORY: NATURAL DEDUCTION

- **argument**: collection of formulas where one conclusion is justified by other premises
  ↳ **valid** if all BV where premises have value T, conclusion has T too
    - $P_1, P_2, \ldots, P_n \models C$ means premises logically imply conclusion
- $\models P$ means P is tautology
  in $P_1, P_2, \ldots, P_n \models C$, C can be weaker than conjunction of premises since $\models \neq \Leftrightarrow$
  - **weaker** means C has T in more BV
- premises are **inconsistent** if there's no BV where all of them are T (still valid argument)
- argument is invalid iff there's at least 1 BV in which premises are T but conclusion is F
  ↳ to show, find **counterexample**
    - i.e. BV where premises are T & conclusion is F
- in general, can't use proof theory to show argument is invalid
  ↳ if argument is contradiction, can use proof theory that its negation is tautology

- **natural deduction**: collection of inference rules that allows us to infer new formulas from given ones
  ↳ form of fwd proof
  ↳ $P_1, P_2, \ldots, P_n \vdash Q$ means there's a proof in natural deduction
- **inference rule**: primitive valid argument form & enables elimination or intro of logical connective

and-introduction

$$\frac{P \quad Q}{P \wedge Q}\ \text{and\_i}$$

and-elimination

$$\frac{P \wedge Q}{P}\ \text{and\_e} \qquad \frac{P \wedge Q}{Q}\ \text{and\_e}$$

- formulas above line are premises
- formula below line is conclusion
- these are mini arguments: $P, Q \vdash P \wedge Q$

present proofs in linear format using line labels

- first list premises of argument w/ "premise" beside each line
- to use inference rule, formulas matching premises of rule must appear on existing lines of proof
- premise of rule must match whole formula on line ≠ not subpart

	INTRODUCTION	ELIMINATION
$\wedge$	$\dfrac{P \quad Q}{P \wedge Q}\ \text{and\_i}$	$\dfrac{P \wedge Q}{P}\ \text{and\_e}$ $\quad$ $\dfrac{P \wedge Q}{Q}\ \text{and\_e}$

$\vee$	$\dfrac{P}{P \vee Q}\ \text{or\_i}$ $\quad$ $\dfrac{Q}{P \vee Q}\ \text{or\_i}$
	
	$\dfrac{}{P \vee \neg P}\ \text{lem}$
		Case Analysis
		$P \vee R$
		case P {
		$\vdots$
		Q
		}
		case R {
		$\vdots$
		Q
		}
		$\dfrac{}{Q}\ \text{cases}$
		
		Disjunctive Syllogism
		$\dfrac{P \vee Q \quad \neg P}{Q}\ \text{or\_e}$
		
		$\dfrac{P \vee Q \quad \neg Q}{P}\ \text{or\_e}$

$\Rightarrow$	Conditional Proof
	assume R {
	$\dots$
	Q
	}
	$\dfrac{}{R \Rightarrow Q}\ \text{imp\_i}$
		Modus Ponens
		$\dfrac{P \Rightarrow Q \quad P}{Q}\ \text{imp\_e}$
		
		Modus Tollens
		$\dfrac{P \Rightarrow Q \quad \neg Q}{\neg P}\ \text{imp\_e}$
		
		$\dfrac{P \Rightarrow \neg Q \quad Q}{\neg P}\ \text{imp\_e}$

$\neg$	Indirect Proof
	disprove R {
	$\dots$
	false
	}
	$\dfrac{}{\neg R}\ \text{raa}$
	
	disprove $\neg R$ {
	$\dots$
	false
	}
	$\dfrac{}{R}\ \text{raa}$
		$\dfrac{P \quad \neg P}{Q}\ \text{not\_e}$

$\neg\neg$	$\dfrac{P}{\neg\neg P}\ \text{not\_not\_i}$	$\dfrac{\neg\neg P}{P}\ \text{not\_not\_e}$
$\Leftrightarrow$	$\dfrac{P \Rightarrow Q \quad Q \Rightarrow P}{P \Leftrightarrow Q}\ \text{iff\_i}$ $\dfrac{P \Leftrightarrow Q \quad Q \Leftrightarrow R}{P \Leftrightarrow R}\ \text{trans}$	$\dfrac{P \Leftrightarrow Q}{P \Rightarrow Q}\ \text{iff\_e}$ $\quad$ $\dfrac{P \Leftrightarrow Q}{Q \Rightarrow P}\ \text{iff\_e}$ $\dfrac{P \Leftrightarrow Q \quad P}{Q}\ \text{iff\_mp}$ $\quad$ $\dfrac{P \Leftrightarrow Q \quad Q}{P}\ \text{iff\_mp}$
	$\dfrac{}{\textbf{true}}\ \textbf{true}$	

- e.g. $a \wedge (a \Rightarrow c)$, $c \vdash (a \Rightarrow c) \wedge c$

1)	$a \wedge (a \Rightarrow c)$	premise
2)	c	premise
3)	$a \Rightarrow c$	and_e on 1
4)	$(a \Rightarrow c) \wedge c$	and_i on 2, 3

- e.g. $a, \neg\neg(b \wedge c) \vdash \neg\neg a \wedge c$

1)	a	premise
2)	$\neg\neg(b \wedge c)$	premise
3)	$\neg\neg a$	not_not_i on 1
4)	$b \wedge c$	not_not_e on 2
5)	c	and_e on 4
6)	$\neg\neg a \wedge c$	and_i on 3, 5

- e.g. $a, a \Rightarrow b, b \Leftrightarrow c \vdash b \wedge c$

1)	a	premise
2)	$a \Rightarrow b$	premise
3)	$b \Leftrightarrow c$	premise
4)	b	imp_e on 1, 2
5)	$b \Rightarrow c$	iff_e on 3
6)	c	imp-e on 4, 5
7)	$b \wedge c$	and_i on 4, 6

skip $b \Leftrightarrow c \rightarrow c$ iff_mp on 3, 4

- e.g. $a \wedge b \vdash a \vee c$

1)	$a \wedge b$	premise
2)	a	and_e on 1
3)	$a \vee c$	or_i on 2

- e.g. $\neg\neg a \Leftrightarrow b \wedge c, a \vdash b \vee s$

1)	$\neg\neg a \Leftrightarrow b \wedge c$	premise
2)	a	premise
3)	$\neg\neg a$	not_not_i on 2
4)	$b \wedge c$	iff_mp on 1, 3
5)	b	and_e on 4
6)	$b \vee s$	or_i on 5

- e.g. $\neg a \Leftrightarrow \neg b, \neg b \wedge a \vdash c$

1)	$\neg a \Leftrightarrow \neg b$	premise
2)	$\neg b \wedge a$	premise
3)	$\neg b$	and-e on 2
4)	a	and_e on 2
5)	$\neg a$	iff-mp on 1, 3
6)	c	not_e on 4, 5

- some natural deduction rules use subordinate proofs (aka subproofs)
  - ↳ start by choosing formula that we assume is true then see what we can prove based on that assumption + any prev deduced formulas
- 3 proof rules that use subproofs:
  - ↳ conditional (imp_i)
  - ↳ indirect proof / proof by contradiction (raa)
  - ↳ case analysis (cases)
- general form of subproofs:

x) sub_proof_opening R {
⋮
x+y) Q by rule B
}
x+y+1) P by rule C on x – x+y

↳ can be nested subproofs
↳ rule C must be one of inference rules: imp_i, raa, or cases
↳ once indented part is complete, subproof is closed
• lines x to x+y can't be used w/inference rules after line x+y+1

· active formulas in a subproof are those that don't occur in a closed subproof
↳ can only use active formulas to derive new ones
· assumptions are temp & can't use after subproof is closed
· for conditional proof, assume R & then prove Q:

assume R {
...
Q
}
———— imp_i
$R \Rightarrow Q$

↳ lines within subproof implicitly depend on assumption
↳ conclusion is last line of subproof
• outside scope of subproof b/c it explicitly states dependency on assumption

· e.g. $b \Rightarrow c \vdash \neg c \Rightarrow \neg b$

1) $b \Rightarrow c$ premise
2) assume $\neg c$ {
3) $\neg b$ imp_e on 1, 2
}
4) $\neg c \Rightarrow \neg b$ imp_i on 2,3

e.g. $a \Rightarrow (b \Rightarrow c), b \vdash a \Rightarrow c$

1) $a \Rightarrow (b \Rightarrow c)$ premise
2) b premise
3) assume a {
4) $b \Rightarrow c$ by imp_e on 1,3
5) c by imp_e on 2,4
}
6) $a \Rightarrow c$ by imp_i on 3-5

· e.g.

```
1  #check ND
2
3  |- (c => d) =>((!c => !b) =>(b => d))
4
5  1) assume c=>d {
6    2) assume !c =>!b {
7      3) assume b {
8        4) !!c by imp_e on 2,3
9        5) c by not_not_e on 4
10       6) d by imp_e on 1, 5
11     }
12     7) b => d by imp_i on 3–6
13   }
14   8) (!c => !b) =>(b => d) by imp_i on 2–7
15 }
16 9) (c =>d) =>((!c =>!b) =>(b =>d)) by imp_i on 1–8
```

- for indirect proof (aka not_i, proof by contradiction, reductio ad absurdum), assume smth is true & get false as final line in subproof so negation of assumption can be proven true
  - ↳ format:

disprove R {	disprove $\neg R$ {
...	...
false	**false**
}	}
——— raa	——— raa
$\neg R$	R

  - often use not_e rule to get false

- e.g. $b \Rightarrow c$,
  $b \Rightarrow \neg c$
  $\vdash$
  $\neg b$

  1) $b \Rightarrow c$ premise
  2) $b \Rightarrow \neg c$ premise
  3) disprove b {
     4) c by imp_e on 1,3
     5) $\neg c$ by imp_e on 2,3
     6) **false** by not_e on 4,5
  }
  7) $\neg b$ by raa on 3-6

  - ↳ using TP to prove: $(b \Rightarrow c) \wedge (b \Rightarrow \neg c) \Rightarrow \neg b \leftrightarrow$ true

- e.g. $p \wedge \neg q \Rightarrow r$,
  $\neg r$,
  p
  $\vdash$
  q

  1) $p \wedge \neg q \Rightarrow r$ premise
  2) $\neg r$ premise
  3) p premise
  4) $\neg(p \wedge \neg q)$ by imp_e on 1,2
  5) disprove $\neg q$ {
     6) $p \wedge \neg q$ by and_i on 3,5
     7) **false** by not_e on 4,6
  }
  8) q by raa on 5-7

- e.g. $b \vdash a \Rightarrow b$ (how to copy premise into body of subproof)
  1) b premise
  2) assume a {
     3) disprove $\neg$b {
        4) false by not_e on 1,3
     }
     5) b by raa on 3-4
  }
  6) $a \Rightarrow b$ by imp_i on 2-5

- if there's disjunction, can use **case analysis**
  - ↳ assume each of disjuncts & conclude same formula from every case
  - ↳ case analysis format (i.e. or_e):

$P \vee R$
case P {
⋮
Q
}
case R {
⋮
Q
}
_____ cases
Q

    - disjunction should be listed as 1st case

- e.g. $a, \neg b \Leftrightarrow a, c \vee b, c \Rightarrow b \vdash$ false

1) a premise
2) ¬b⇔a premise
3) c∨b premise
4) c⇒b premise
5) ¬b by iff_mp on 1,2
6) case c {
   7) ¬c by imp_e on 4,5
   8) false by not_e on 6,7
   }
9) case b {
   10) false by not_e on 5,9
   }
11) false by cases on 3, 6-8, 9-10

OR

1) a premise
2) ¬b⇔a premise
3) c∨b premise
4) c⇒b premise
5) ¬b by iff_mp on 1,2
6) c by or_e on 3,5
7) b by imp_e on 4,6
8) false by not_e on 5,7

- one derived rule is **disjunctive syllogism** (i.e. or_e): $p \vee q, \neg p \vdash q$ OR $p \vee q, \neg q \vdash p$
  - ↳ to prove using ND:

1) p ∨ q premise
2) ¬p premise
3) case p {
   4) q by not_e on 2,3
   }
5) case q {
   }
6) q by cases on 1, 3-4, 5-5

- another derived rule is **law of excluded middle** (i.e. lem): $\vdash p \vee \neg p$
  - ↳ to prove using ND:

1) disprove ¬(p∨¬p) {
   2) disprove p {
      3) p ∨ ¬p by or_i on 2
      4) false by not_e on 1,2
      }
   5) ¬p by raa on 2-4

6) p ∨ ¬p by or_i on 5
7) false by not_e on 1, 6
}
8) p ∨ ¬p by raa on 1-7

- when doing fwd proof, set subgoals to reach conclusion
  ↳

To prove	Possible subgoals
$A \wedge B$	Both A and B
$A \vee B$	Either A or B
$A \Rightarrow B$	assume A then prove B (conditional proof)
A	$A \wedge B$ for some B
A	$\neg A$ and derive a contradiction (indirect proof)

- when faced w/negated premise, try to generate contradiction w/it or use lem to set up cases analysis
- e.g. ¬a ∨ ¬b ⊢ ¬(a ∧ b) (proving dm law)

1) ¬a ∨ ¬b premise
2) disprove a ∧ b {
3) a by and_e on 2
4) b by and_e on 2
5) ¬¬a by not_not_i on 3
6) ¬b by or_e on 1, 5
7) false by not_e on 4, 6
}
8) ¬(a ∧ b) by raa 2-7

- e.g. ¬(a ∨ b) ⊢ ¬a ∧ ¬b (another dm law)

1) ¬(a ∨ b) premise
2) disprove a {
3) a ∨ b by or_i on 2
4) false by not_e on 1, 3
}
5) ¬a by raa on 2-4
6) disprove b {
7) a ∨ b by or_i on 6
8) false by not_e on 1, 7
}
9) ¬b by raa on 6-8
10) ¬a ∧ ¬b by and_i on 5, 9

- e.g. b ⇒ c ⊢ ¬b ∨ c

1) b ⇒ c premise
2) b ∨ ¬b by lem
3) case b {
4) c by imp_e on 1, 3
5) ¬b ∨ c by or_i on 4
}
6) case ¬b {
7) ¬b ∨ c by or_i on 6
}
8) ¬b ∨ c by cases on 2, 3-5, 6-7

- natural deduction is sound & complete for propositional logic
  - ↳ if $P_1, P_2, \ldots, P_n \vdash Q$ then $P_1, P_2, \ldots, P_n \models Q$ (soundness)
    - ND proves only valid arguments
  - ↳ if $P_1, P_2, \ldots, P_n \models Q$ then $P_1, P_2, \ldots, P_n \vdash Q$ (completeness)
    - ND can be used to prove all valid arguments

PROOF THEORY: SEMANTIC TABLEAUX

- semantic tableaux is tree rep all ways conjunction of formulas at root of tree can be true
  - ↳ general form:

```
1) formula
2) formula
by rule on #
4) smaller formula
by rule on #
{
   (possibly more { ... })
   5) smaller formula
   closed on #,#
} {
   (possibly more { ... })
   6) smaller formula
   closed on #,#
}
```

  - ↳ {...} show branches of tree
  - ↳ every formula on path from root to leaf must be true in a BV that makes conjunction of formulas at root true
- in each step of ST proof, we either:
  - ↳ use ST rule to decompose 1 compound formula & add new formula(s) to branch
    - pattern must match to entire formula
  - ↳ close branch b/c it contains contradictory formulas
    - means there's no BV that makes all formulas on path from root to leaf T

formulas at root of are **inconsistent** if every branch is closed

- ↳ i.e. no BV in which conjunction of formulas at root is true

```
1) P ∧ Q
by and_nb on 1
2) P
3) Q
```

non-branch

```
1) ¬(P ∧ Q)
by not_and_br on 1
{
   2) ¬P
}
{
   3) ¬Q
}
```

branch

- ↳ **branch** rep disjunction
  - 2 ways to make conjunction of formulas at root true
- ↳ **non-branch** rep conjunction
  - both formulas needed to make conjunction of formulas at root true
- branch is **closed** if P & ¬P both appear on same branch
  - ↳ i.e contradiction on branch
  - ↳ conjunction of formulas on closed branch is contradiction
  - ↳ line labels of formulas listed w/ "closed" line contradict each other

· tableaux expansion rules:

Summary of Semantic Tableaux
for Propositional Logic

	POSITIVE	NEGATIVE

$\wedge$	1) $P \wedge Q$
	by and_nb on 1
	2) P
	3) Q
		1) $\neg(P \wedge Q)$
		by not_and_br on 1
		{
		2) $\neg P$
		}
		{
		3) $\neg Q$
		}

$\vee$	1) $P \vee Q$
	by or_br on 1
	{
	2) P
	}
	{
	3) Q
	}
		1) $\neg(P \vee Q)$
		by not_or_nb on 1
		2) $\neg P$
		3) $\neg Q$

$\Rightarrow$	1) $P \Rightarrow Q$
	by imp_br on 1
	{
	2) $\neg P$
	}
	{
	3) Q
	}
		1) $\neg(P \Rightarrow Q)$
		by not_imp_nb on 1
		2) P
		3) $\neg Q$

$\neg$		1) $\neg\neg P$ by not_not_nb on 1 2) P

$\Leftrightarrow$	1) $P \Leftrightarrow Q$
	by iff_br on 1
	{
	2) $P \wedge Q$
	}
	{
	3) $\neg P \wedge \neg Q$
	}
		1) $\neg(P \Leftrightarrow Q)$
		by not_iff_br on 1
		{
		2) $P \wedge \neg Q$
		}
		{
		3) $\neg P \wedge Q$
		}

· e.g. $b \wedge c$, d , $\neg(c \wedge d)$ is inconsistent → $b \wedge c$, d , $\neg(c \wedge d) \vdash$ false

1) $b \wedge c$
2) d
3) $\neg(c \wedge d)$
by and_nb on 1
4) b
5) c
by not_and_br on 3
{
6) $\neg c$
closed on 5, 6
}
{
7) $\neg d$
closed on 2, 7
}

↳ tree form:

1) $b \wedge c$
2) d
3) $\neg(c \wedge d)$

4) $\neg c$ | 5) $\neg d$ — closed on 2, 5
6) b
7) c
— closed on 4, 7

apply nb rules first b/c it usually results in shorter proofs

· ST is often used to prove set of formulas is inconsistent by putting false in conclusion

of argument

- to prove argument is valid using ST, show $P_1, P_2, \ldots, P_n, \neg Q$ is inconsistent set of formulas
  - ↳ i.e. put premises ∧ negation of conclusion at root of tableau
  - ↳ by closing all branches, can prove argument is valid
    - conclude $P_1, P_2, \ldots, P_n \vdash Q$
- e.g. $b \vee c \vdash c \vee b$

  1) $b \vee c$

  2) $\neg(c \vee b)$

  by not_or_nb on 2

  3) $\neg c$

  4) $\neg b$

  by or_br on 1

  {

  5) b

  closed on 4, 5

  }

  {

  6) c

  closed 3, 6

  }

- e g Excluded middle

  $\vdash$

  $b \vee \neg b$

  1) $\neg(b \vee \neg b)$

  by not_or_nb on 1

  2) $\neg b$

  3) $\neg\neg b$

  closed on 2, 3

- e.g. $p, p \Leftrightarrow q \vdash q$

  1) p

  2) $p \Leftrightarrow q$

  3) $\neg q$

  by iff_br on 2

  {

  4) $p \wedge q$

  by and_nb on 4

  6) p

  7) q

  closed on 3, 7

  }

  {

  5) $\neg p \wedge \neg q$

  by and_nb on 5

  8) $\neg p$

  9) $\neg q$

  closed on 1, 8

  }

some proof form rules:

- ↳ ST rule only applies to 1 formula/line of tree & must apply to outermost propositional connectives of that formula
- ↳ closing tableau requires 2 contradicting formulas on same branch
- ↳ branching is disjunction so there's multiple ways to make formula true
  - must close all branches to prove formula false
- ↳ rules for $\wedge$ & $\vee$ can be applied to formula w/ 2+ conjuncts/disjuncts in single step
- ↳ can expand branches in any order
- ↳ left & right branches can be swapped

- to show an argument is invalid, provide BV in which premises are T & conclusion is F
- to show set of formulas is consistent, provide BV in which all formulas are T
- semantic tableaux is sound & complete for propositional logic
  - ↳ if $P_1, P_2, \ldots, P_n \vdash Q$ then $P_1, P_2, \ldots, P_n \models Q$ (soundness)
    - ST proves only valid arguments
  - ↳ if $P_1, P_2, \ldots, P_n \models Q$ then $P_1, P_2, \ldots, P_n \vdash Q$ (completeness)
    - ST can be used to prove all valid arguments

- e.g. For two aircraft A and B, the vertical separation minimum shall be:

  - 1000 feet if flight A is below FL290;
  - 2000 feet if flight A is at or above FL290 except if flight A is above FL450 and flight B is supersonic where 4000 feet shall be used.

  FL290 = 29000 ft, etc.

  These are the rules that air traffic controllers use. CAATs is going to turn that into a computer program. We know computers take things exactly literally, so we have to make sure the rules are clearly spelled out and there aren't any mistakes in the specification, so we're going to start with some analysis.

  This is a simplified and asymmetric version of the rules.

  1. Under what conditions are each of the amounts of vertical separation required?
  2. Is there an amount of vertical separation required for all possible input conditions?
  3. Is there only one amount of vertical separation required for all possible input conditions? (i.e., is the specification unambiguous?)

1) 1000 a290
   2000 $\neg$ a290 $\wedge$ $\neg$(a450 $\wedge$ bsup)
   4000 a450 $\wedge$ bsup
   a290 means flight A is below FL290
   a450 means flight A is above FL450
   bsup means flight B is supersonic

2) $\models \underbrace{(a290) \vee (\neg a290 \wedge \neg(a450 \wedge bsup)) \vee (a450 \wedge bsup)}_{\text{conclusion}}$

↓

$\vdash$

Using ND:

1) $(a450 \wedge bsup) \vee \neg(a450 \wedge bsup)$ by lem

2) case $a450 \wedge bsup$ {

3) conclusion by or_i on 3

}

4) case $\neg(a450 \wedge bsup)$ {

5) $a290 \vee \neg a290$ by lem

6) case $a290$ {

7) conclusion by or_i on 6

}

8) case $\neg a290$ {

9) $\neg a290 \wedge \neg(a450 \wedge bsup)$ by and_i on 4, 8

10) conclusion by or_i on 9

}

11) conclusion by cases on 5, 6-7, 8-10

}

12) conclusion by cases on 1, 2-3, 4-11

Using TP: $(a290) \vee (\neg a290 \wedge \neg(a450 \wedge bsup)) \vee (a450 \wedge bsup) \leftrightarrow$ true

$(a290) \vee (\neg a290 \wedge \neg(a450 \wedge bsup)) \vee (a450 \wedge bsup)$

$\leftrightarrow (a290) \vee \neg(a290 \vee (a450 \wedge bsup)) \vee (a450 \wedge bsup)$ by dm

$\leftrightarrow (a290 \vee (a450 \wedge bsup)) \vee \neg(a290 \vee (a450 \wedge bsup))$ by comm_assoc

$\leftrightarrow$ true by lem

3) $\neg(a290 \wedge a450) \vdash \neg(\underbrace{a290}_{1000} \wedge \underbrace{a450 \wedge bsup}_{4000})$

↑ environmental constraint

Using ST:

1) $\neg(a290 \wedge a450)$

2) $\neg\neg(a290 \wedge a450 \wedge bsup)$

by not_not_nb on 2

3) $a290 \wedge a450 \wedge bsup$

by and_nb on 3

4) $a290 \wedge a450$

5) $bsup$

closed on 1, 4

MODULE 3

PREDICATE LOGIC

- syllogisms are oldest system of logic
  - ↳ e.g. all P are Q, X is P, therefore X is Q
- predicate: symbol denoting meaning of an attribute/property of value or meaning of relationship btwn 2+ values
  - ↳ unary predicate takes single value as argument
  - ↳ binary predicate takes 2 values as arguments
  - ↳ n-ary predicate takes n values as arguments
  - ↳ true or false when applied to values
- constant: symbol denoting particular value
- e.g. The Earth is round. → round(Earth)
  - ↳ "Earth" is value & "round" is predicate
  - ↳ round(y) means y is round
- must state what predicates mean b/c order matters but constants don't have to be explicitly stated if their meaning is obvious
- rule of thumb for formalization:
  - ↳ values are usually nouns
  - ↳ attributes are adjectives, adverbs, & verbs
- e.g. 10 is greater than 5. → 10 > 5
  - ↳ "10" & "5" are values
  - ↳ > is infix predicate
    - ∘ don't have to state what this means
- predicate applied to arguments has truth value & can be used anywhere a prime proposition is used in propositional logic
  - ↳ all logical connectives of propositional logic are also part of syntax of predicate logic
  - ↳ e.g.

1. John is happy if John visits Vancouver.
   $visits(John, Vancouver) \Rightarrow happy(John)$
   where
   $visit(x, y)$ means x visits y
   $happy(x)$ means x is happy
2. Rima is a dancer and Rima is a student.
   $dancer(Rima) \wedge student(Rima)$
   where
   $dancer(x)$ means x is a dancer
   $student(x)$ means x is a student

- variable rep place where value can be substituted
  - ↳ e.g. between(x,y,z) means city rep by x is btwn cities rep by y & z
- quantifier is always followed by variable (e.g. $\forall x \cdot$)
  - ↳ e.g.

1. Everything likes Fridays.
   $\forall x \bullet likes(x, Fridays)$
2. Something likes Fridays.
   $\exists y \bullet likes(y, Fridays)$

↳ "·" is "such that"

↳

Name	Symbol	george	Description
Universal quantification	$\forall$	`forall`	formula holds for all values of the variable ("every", "all", "for all")
Existential quantification	$\exists$	`exists`	formula holds for some value of the variable (at least one) ("some", "there exists")

· e.g.

1. Something is rotten in the state of Denmark.
2. Everyone is a tall adult.
3. Someone is either happy or hungry or both.
4. Every child likes Mickey Mouse.

1) $\exists x \cdot rotten(x) \wedge in(x, Denmark)$
2) $\forall z \cdot tall(z) \wedge adult(z)$
   ↳ every adult is tall
   $\forall z \cdot adult(z) \Rightarrow tall(z)$
3) $\exists x \cdot happy(x) \vee hungry(x)$
4) $\forall y \cdot child(y) \Rightarrow likes(y, Mickey\ Mouse)$

↳ always write meaning of predicates
- e.g. include predicate like person when formalizing "everyone"

· e.g. formalize "Mary's age is less than 20."
↳ $\exists y \cdot age(y, Mary) \wedge y < 20$
- i.e. there's some value which is Mary's age ∴ it's less than 20
- Mary could have multiple ages ∴ above wff would be true

↳ use function to capture idea that there's only 1 value in age relationship w/Mary
- usually shortens predicate logic formula ∴ captures more info
- as a function: $age(Mary) < 20$

· e.g.

1. Eunsuk was born north of Toronto.

northof(birthplace(Eunsuk), Toronto)
↳ northof(x, y) means x is north of y
↳ birthplace(x) is birthplace of x

· e.g.

2. For all x, if "input" has the value x then "output" has the value x squared.

$\forall x \ value(input) = x \Rightarrow value(output) = x^2$

SYNTAX

· syntax for untyped predicate logic:

↳ alphabet consists of:
- constants (e.g. c, John, true, false, proposition symbols)
- variables (e.g. x, y)
- function symbols (e.g. g, h, father)
- predicate symbols (e.g. P, <, is_west_of)
- logical connectives: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$
- quantifiers: $\forall$, $\exists$
- punctuation: , .
- brackets: ( )

↳ note that capitals will usually refer to places where any formula can be sub & lowercase refer to specific predicates/ functions used in examples

↳ terms are defined as:
- every constant & variable is a term
- if $t_1, t_2, \ldots, t_n$ are terms & g is n-ary function symbol, then $g(t_1, t_2, \ldots, t_n)$ is term

↳ terms describe values & functions can return Boolean values

↳ well-formed formulas (wff) are defined as:
- proposition symbols, true, & false are wff
  → atomic formulas
- if t_is are terms & P is n-ary predicate symbol, then $P(t_1, t_2, \ldots, t_n)$ is wff
  → atomic formulas
- if P & Q are wffs, then so are these:
  → $\neg P$
  → $P \wedge Q$
  → $P \vee Q$
  → $P \Rightarrow Q$
  → $P \Leftrightarrow Q$
- if P is wff & x is variable then $\forall x \cdot P$ & $\exists x \cdot P$ are wffs

↳ same precedence & associativity rules as propositional logic for logic connectives
- quantifiers are least binding (lowest precedence)

↳ wffs have truth values

· e g

1. Everything doesn't like something.
   $\forall x \bullet \exists y \bullet \neg likes(x, y)$

2. Nothing likes everything.
   $\neg(\exists x \bullet \forall y \bullet likes(x, y))$

· logical laws:

↳ $\neg(\forall x \cdot P(x)) \leftrightarrow \exists x \cdot \neg P(x)$

↳ $\neg(\exists x \cdot P(x)) \leftrightarrow \forall x \cdot \neg P(x)$

· e.g. b(x) means x is bicycle
g(x) means x is in garage

↳ All bicycles are in the garage.
$\forall x \cdot b(x) \Rightarrow g(x)$
- i.e. for all things, if thing is bicycle, then it's in the garage

↳ Some bicycles are in the garage.
$\exists x \cdot b(x) \wedge g(x)$

- i.e. something is a bicycle and is in the garage
- $\exists x \cdot b(x) \Rightarrow g(x)$ is **weak**

  $\leftrightarrow \exists x \cdot \neg b(x) \vee g(x)$

  → as long as smth is not a bicycle, it can be in garage (F implies T)

↳ All things are bicycles in the garage.

$\forall x \cdot b(x) \wedge g(x)$

↳ Everything is a bicycle or is in the garage.

$\forall x \cdot b(x) \vee g(x)$

↳ Something is a bicycle or is in the garage.

$\exists x \cdot b(x) \vee g(x)$

- usually don't use $\Rightarrow$ w/existential quantifier

- **scope** of quantifier is subformula over which quantifier applies in given formula
  ↳ w/o brackets, assume scope extends to right end of formula
- variable is **bound** to closest quantifier to left of its name whose scope it's within
  ↳ all variable occurrences bound to same quantifier rep same value
- variable is **free** if it doesn't fall within scope of quantifier for that variable
- wff is **closed** if it contains no free variables
- e.g. $\forall x \cdot \exists y \cdot p(x,y) \wedge q(y)$

- e.g. $\forall x \ (\exists y \cdot p(x,y)) \wedge (\exists y \cdot q(y))$

- e.g. $\forall x \cdot (\exists y \cdot p(x,y) \wedge \exists y \cdot q(y)) \wedge r(x,y)$

↳ y is free var is above wff is open

e.g. $\forall x \cdot \exists y \ p(x,y) \wedge (\exists x \cdot p(x,y))$

- can change variable names (i.e. **alpha renaming**) b/c they're just placeholders

- when formalizing sentence, **order** to look for elements:
  1) logical connectives
  2) quantifiers
  3) constants
  4) functions ↾ what values they apply to
  5) predicates ↾ what values they apply to
- when formalizing sentence by itself, formalize as much of its structure as possible
- when formalizing set of sentences in valid argument, formalize as much structure as needed to show argument is valid

- e.g.
  1. All students who like software engineering also like logic.
  2. All rich actors collect some valuables.
  3. Not everyone who lives in Louisiana speaks French, but everyone who lives in Louisiana knows someone who lives in Louisiana who speaks French.

1) $\forall x \cdot student(x) \wedge likes(x, SE) \Rightarrow likes(x, logic)$
2) $\forall x \cdot rich(x) \wedge actor(x) \Rightarrow \exists y \cdot collects(x, y) \wedge valuable(y)$
3) $\neg(\forall x \cdot lives(x, Louisiana) \Rightarrow speaks(x, French)) \wedge (\forall x \; lives(x, Louisiana) \Rightarrow \exists y \cdot knows(x, y) \wedge lives(y, Louisiana) \wedge speaks(y, French))$

- George understands following predicates & functions: +, -, *, /, ^, =, !=, <, >, >=, <=
  - ↳ also recognizes numeric constants
  - ↳ e.g.

```
1 #check PRED
2
3 !(forall x . lives(x,Louisiana)
4                  => speaks(x,French)) &
5  (forall x . lives(x,Louisiana) =>
6     exists y . knows(x, y)
7                & lives(y,Louisiana)
8                 & speaks(y, French))
```

- type is set of possible values of variable
  - ↳ must be non-empty b/c if not, $\forall x \; P(x) \Rightarrow \exists x \cdot P(x)$ would not be tautology
  - ↳ e.g. IceCreamFlavours = { MapleWalnut, MintChocolate, ... }
- e.g. adding types to formulas:
  - $\forall x \bullet \exists y \bullet x \geq y$
    $\forall x : nat \bullet \exists y : nat \bullet x \geq y$
  - $\forall p \bullet likes(p, ice\_cream)$
    $\forall p : Person \bullet likes(p, ice\_cream)$
  - $\exists b \bullet flightlevel(b) > FL290$
    $\exists b : Aircraft \bullet flightlevel(b) > FL290$

  - ↳ basic types are like Aircraft
- special type called bool
- each constant has type
  - ↳ e.g. true: bool, c: T1
- each n-ary function has type
  - ↳ g: T1 x T2 x .. x Tn → RetType
    - i.e. g takes arguments a1: T1, .., an: Tn & returns value of type RetType
- each n-ary predicate has type
  - ↳ P: T1 x T2 x ... x Tn → bool

- i.e. predicates are functions that return values of type bool

- in typed predicate logic, don't syntactically distinguish btwn functions & predicates
- syntax for typed predicate logic:
  - ↳ **alphabet** is same as untyped w/ addition of ":" & names for types
  - ↳ **terms** are defined as:
    - every constant & variable is term
      - → may be followed by $:T1$
    - if $a1 : T1, \ldots, an : Tn$ are terms & g is n-ary function symbol of type $T1 \times T2 \times \ldots \times Tn \to RetType$, then $g(a1, a2, \ldots, an)$ is term of type RetType
    - if P & Q are terms of type bool, then so are $\neg P$, $P \wedge Q$, $P \vee Q$, $P \Rightarrow Q$, & $P \Leftrightarrow Q$
    - if P is term of type bool, x is variable of type $T1$, & all free occurrences of x in P are of type $T1$, then $\forall x \cdot P$ & $\exists x \cdot P$ are terms of type bool
  - ↳ formulas are terms of type bool

- e.g. $c : T1$
  $f : T1 \to T2$
  $g : T2 \to T3$
  $p : T1 \to bool$
  $q : T3 \times T1 \to bool$
  Variables: $x : T1$, $y : T3$

  Are the following well-typed?

  - $q(g(f(c)), c) \wedge p(c)$ Y
  - $p(f(c)) \Rightarrow q(c, f(c))$ N
  - $\forall x \bullet \exists y \bullet q(f(x), y)$ N
  - $\exists x \bullet p(f(x)) \Rightarrow \forall y \bullet q(y, g(f(c)))$ N

a) $\dfrac{q(\overset{\overset{T1}{\overset{T2}{T3}}}{g(f(c))}, \underset{T1}{c}) \wedge \overset{T1}{p(c)}}{bool}$

b) $p(\overset{T1}{f(c)}) \Rightarrow q(\underset{T1}{c}, \overset{T1}{f(c)})$ (T2, T1)

c) $\forall x \cdot \exists y \; q(\overset{T1}{f(x)}, \underset{T3}{y})$ (T1, T2)

d) $\exists x \cdot p(f(x)) \Rightarrow \forall y \cdot q(\underset{T3}{y}, \overset{\overset{T1}{T2}}{g(f(c))})$ (T3, T1)

- **well-typed** formula satisfies rules of typed predicate logic
- **type inference** is process of figuring out types for functions, constants, & variables that make formula satisfy rules
- **most general typing** of formula means using max # of diff types st formula satisfies rules

- e.g. Are there types that make the following each be well-typed?
  - $\exists x, y \bullet p(x, y) \wedge p(f(x), y)$
  - $\forall x, y \bullet q(f(x, y)) \Rightarrow q(y, f(x))$
  - $\exists x, y . q(f(y)) \wedge q(g(x))$

a) x: T1
y: T2
p: T1 x T2 → bool
f: T1 → T1

b) x: _
y: _
q: _ → bool
f: _ x _ → _

q & f are inconsistent w/ # arguments

c) x: T1
y: T2
f: T2 → T3
g: T1 → T3
q T3 → bool

- values may belong to only 1 type
- regarding type vs attribute / unary predicate on variable:
  - $\forall x : Q \bullet P(x)$ is the same as $\forall x \bullet Q(x) \Rightarrow P(x)$
  - $\exists x : Q \bullet P(x)$ is the same as $\exists x \bullet Q(x) \wedge P(x)$

  ↳ when adding type to variable, there's not always unary predicate available to turn into type
- for 2+ variables w/ same quantifier & same type, can write quantifier & type once while separating variables w/ commas
  ↳ e.g. ∀x: Person · ∀y: Person → ∀x,y: Person
- for 2+ variables w/ same quantifier & diff type, can write quantifier once while separating variables w/ commas
  ↳ e.g. ∃x: Person · ∃y: Rainbow → ∃x: Person, y: Rainbow

george checks syntax & ignores types

```
#check PRED

forall y:Rich . actor(y) =>
     exists z:Valuable . collects(y,z)
```

SEMANTICS

- semantics define a valid argument (⊨) & logical equivalence of 2 formulas (⟺)
- to describe meaning of formula in predicate logic, need domain of discourse (D) in addition to set of truth values
  ↳ terms map to values in domain
  ↳ wffs map to truth values
- describe PL formula in particular interpretation that includes:
  ↳ non-empty domain
  ↳ for every constant, function, & predicate in formula:

Syntax	Meaning
constants	each maps to a value of the domain
functions	each maps to a total function of matching arity that takes arguments from the domain, and returns values in the domain
predicates	each maps to a total predicate of matching arity that takes arguments from the domain, and returns values of **Tr**

diff constant symbols can be associated w/ same value in domain

- diff predicates or functions can map to same predicates or functions on values
- for universally quantified variable, formula must be T for all sub of value in domain for variable
  - ↳ i.e. conjunction of meaning of formula for each value in domain
- for existentially quantified variable, formula must be T for some sub of value in domain for variable
  - ↳ i.e. disjunction of meaning of formula for each value in domain
- e.g. $[p(g(c), d) \wedge q(c, d)]$
  - $= [p(g(c), d)]$ AND $[q(c, d)]$
  - $= P(G(0), 1)$ AND $Q(0, 1)$
  - $= G(0) < 1$ AND $0 = 1$
  - $= (0+1) \bmod 3 < 1$ AND $0 = 1$
  - $= 1 < 1$ AND $0 = 1$
  - $=$ F AND F
  - $=$ F

Interpretation:

1) $D = \{0, 1, 2\}$
2) Mapping

Syntax	Meaning
c	0
d	1
$g(\cdot)$	$G(x) := (x+1) \bmod 3$
$p(\cdot, \cdot)$	$P(x, y) := x < y$
$q(\cdot, \cdot)$	$Q(x, y) := x = y$

- to state meaning of quantification, must expand formula inside quantifier w/ sub for each of domain values
  - ↳ e.g. $D = \{a, b\}$

$$
\begin{aligned}
& [\forall x \bullet \exists y \bullet P(x, y)] \\
= \; & [\exists y \bullet P(x, y)]\{x \mapsto a\} \text{ AND } [\exists y \bullet P(x, y)]\{x \mapsto b\} \\
= \; & ([P(x, y)]\{x \mapsto a, y \mapsto a\} \text{ OR } [P(x, y)]\{x \mapsto a, y \mapsto b\}) \text{ AND} \\
& ([P(x, y)]\{x \mapsto b, y \mapsto a\} \text{ OR } [P(x, y)]\{x \mapsto b, y \mapsto b\} \\
= \; & \textit{etc}
\end{aligned}
$$

  - $\{x \mapsto a, y \mapsto b\}$ mappings are called an environment
- as shortcut, ^d means variable has value d
  - ↳ domain elements aren't part of syntax
  - ↳ don't use ^c for constant b/c c is part of syntax
- george doesn't check semantic problems
  - ↳ e.g. Provide an interpretation in which the following formula has the truth value T, and demonstrate that the formula has this result in your interpretation (Note: c is a constant):

$$(\exists x \bullet g(x) \wedge b(k(x))) \wedge \neg b(c)$$

```
(exists x . g(x) & b(k(x))) & !b(c)

An interpretation in which this formula is T:

1) Domain = {d1,d2}

2) Syntax   | Meaning
   ---------------------
   c        | d1
   b(.)     | B(d1) := F
            | B(d2) := T
   g(.)     | G(d1) := T
            | G(d2) := T
   k(.)     | K(d1) := d1
            | K(d2) := d2
```

```
1 Demonstration:
2
3   [ (exists x . g(x) & b(k(x))) & !b(c) ]
4 = [ exists x. g(x) & b(k(x)) ] AND NOT [b(c)]
5 = ([ g(^d1) & b(k(^d1)) ] OR [ g(^d2) & b(k(^d2)) ] ) AND
      NOT [b(c)]
6 = ((G(d1) AND B(K(d1))) OR (G(d2) AND B(K(d2)))) AND NOT B(
      d1)
7 = ((G(d1) AND B(d1)) OR (G(d2) AND B(d2))) AND NOT B(d1)
8 = ((T AND F) OR (T AND T)) AND NOT F
9 = T
```

- e.g. nested quantifiers

  Example: Find an interpretation in which the following formula is T:

$$\forall x \bullet \exists y \bullet p(x) \Leftrightarrow \neg p(y)$$

  Interpretation:

  1. $D = \{d_1, d_2\}$
  2. Mapping

Syntax	Meaning
$p(\cdot)$	$P(d_1) := \mathsf{T}$
	$P(d_2) := \mathsf{F}$

$$\begin{aligned}
& [\forall x \bullet \exists y \bullet p(x) \Leftrightarrow \neg p(y)] \\
= \; & [\exists y \bullet p(\hat{}d_1) \Leftrightarrow \neg p(y)] \text{ AND } [\exists y \bullet p(\hat{}d_2) \Leftrightarrow \neg p(y)] \\
= \; & ([p(\hat{}d_1) \Leftrightarrow \neg p(\hat{}d_1)] \text{ OR } [p(\hat{}d_1) \Leftrightarrow \neg p(\hat{}d_2)]) \text{ AND} \\
& ([p(\hat{}d_2) \Leftrightarrow \neg p(\hat{}d_1)] \text{ OR } [p(\hat{}d_2) \Leftrightarrow \neg p(\hat{}d_2)]) \\
= \; & ((P(d_1) \text{ IFF NOT } P(d_1)) \text{ OR } (P(d_1) \text{ IFF NOT } P(d_2))) \text{ AND} \\
& ((P(d_2) \text{ IFF NOT } P(d_1)) \text{ OR } P(d_2) \text{ IFF NOT } P(d_2))) \\
= \; & ((\mathsf{T} \text{ IFF NOT } \mathsf{T}) \text{ OR } (\mathsf{T} \text{ IFF NOT } \mathsf{F})) \text{ AND} \\
& ((\mathsf{F} \text{ IFF NOT } \mathsf{T}) \text{ OR } (\mathsf{F} \text{ IFF NOT } \mathsf{F})) \\
= \; & \mathsf{T}
\end{aligned}$$

- interpretation consists of:
  - ↳ domain: list of its elements or equate to existing standard domain (e.g. nat)
  - ↳ mapping from predicates, functions, & constants of formula to those over elements of domain
    - if domain is infinite, use "..." in def if extension is obvious
- [...syntax...] = ... semantics ...
  - ↳ exception is sub domain value into syntax using ^
- only show enough of conjunction/disjunction to justify reduction to T/F
- if there's infinite domain, use .../for all/for some in solution
- only show enough of function/predicate def to justify result
- if value doesn't matter, still choose one
- if formula has truth value in B for # interpretations in A, we says it's C

A # of Interpretations	B Truth Value	C Terminology
some	T	Satisfiable
all	T	Tautology
all	F	Contradiction

↳ set of closed PL formulas is inconsistent iff there's no interpretation in which all formulas are satisfied

e.g. Show that $(\forall x \bullet p(x) \vee q(x)) \models ((\forall x \bullet p(x)) \vee (\forall x \bullet q(x)))$ is not valid and demonstrate that your answer is correct.

Interpretation:

1) D = {d1, d2}

2) Mapping

Syntax	Meaning
p(·)	P(d1) := T
	P(d2) := F
q(·)	Q(d1) := F
	Q(d2) := T

Premise:

[∀x · p(x) v q(x)]

= (P(d1) OR Q(d1)) AND (P(d2) OR Q(d2))

= (T OR F) AND (F OR T)

= T OR T

= T

Conclusion:

[(∀x · p(x)) v (∀x · q(x))]

= (P(d1) AND P(d2)) OR (Q(d1) AND Q(d2))

= (T AND F) OR (F AND T)

= F OR F

= F

e.g. Show that $(\forall x \bullet p(g(x))) \Leftrightarrow (\forall x \bullet p(x))$ is not a tautology and demonstrate that your solution is correct.

Interpretation

1. $Domain = \{d1, d2\}$
2. Mapping

Syntax	Meaning
$g(\cdot)$	$G(d1) := d1$ $G(d2) := d1$
$p(\cdot)$	$P(d1) := \mathsf{T}$ $P(d2) := \mathsf{F}$

Demonstration:

$$
\begin{aligned}
& [(\forall x \bullet p(g(x)) \Leftrightarrow (\forall x \bullet p(x))] \\
= \; & [(\forall x \bullet p(g(x)))] \text{ IFF } [\forall x \bullet p(x)] \\
= \; & [p(g(\hat{}d1))] \text{ AND } [p(g(\hat{}d2))] \text{ IFF } [p(\hat{}d1)] \text{ AND } [p(\hat{}d2)] \\
= \; & (P(G(d1)) \text{ AND } P(G(d2))) \text{ IFF } (P(d1) \text{ AND } P(d2)) \\
= \; & (P(d1) \text{ AND } P(d1)) \text{ IFF } (P(d1) \text{ AND } P(d2)) \\
= \; & (\mathsf{T} \text{ AND } \mathsf{T}) \text{ IFF } (\mathsf{T} \text{ AND } \mathsf{F}) \\
= \; & \mathsf{F}
\end{aligned}
$$

- if PL formula has types, when we provide interpretation, have set of domains (one for each type) instead of just one
- e.g. Find an interpretation in which
  $q(c, d) \wedge \exists z : T_1 \bullet \neg(z = c) \wedge q(z, d)$
  is T and demonstrate that your answer is correct.
  Note: $c : T_1$, $d : T_2$

  Interpretation:

  1. Domain:
     $T_1 = \{v_1, v_2\}$
     $T_2 = \{w_1\}$
  2. Mapping:

Syntax	Mapping
c	v_1
d	w_1
$q(\cdot, \cdot)$	$Q(v_1, w_1) := \mathsf{T}$
	$Q(v_2, w_1) := \mathsf{T}$

$$
\begin{aligned}
& [q(c, d) \wedge \exists z : T_1 \bullet \neg(z = c) \wedge q(z, d)] \\
= \; & [q(c, d)] \text{ AND} \\
& ([\neg(\hat{}v_1 = c) \wedge q(\hat{}v_1, d)] \text{ OR} \\
& [\neg(\hat{}v_2 = c) \wedge q(\hat{}v_2, d)]) \\
= \; & Q(v_1, w_1) \text{ AND} \\
& (\text{NOT } (v_1 = v_1) \text{ AND } Q(v_1, w_1) \text{ OR} \\
& \text{NOT } (v_2 = v_1) \text{ AND } Q(v_2, w_1)) \\
= \; & \mathsf{T} \text{ AND } (\text{NOT } (\mathsf{T}) \text{ AND } \mathsf{T} \text{ OR } \text{ NOT } (\mathsf{F}) \text{ AND } \mathsf{T}) \\
= \; & \mathsf{T}
\end{aligned}
$$

- finite model finding: if domain is finite, can try all possible interpretations to show argument is valid
- when checking infinite domains, use proof theory
- TP for PL is uncommon b/c it's more typical to prove valid arguments w/ premises stronger than conclusion
  - ↳ in PL, usually go from general to specific (e.g. $\forall x \cdot p(x) \rightarrow p(a)$)
    - $\forall x \cdot p(x)$ is stronger than $p(a)$ (i.e. in any interpretation where $\forall x \cdot p(x)$, so is $p(a)$ but not vice versa

NATURAL DEDUCTION

- add in pair of rules for each kind of quantification: one to intro & one to eliminate
- forall elimination: if formula is true for all values, then it's true for any particular value
  - ↳ $\frac{\forall x \cdot P}{P[t/x]}$, forall-e
    - t can be any term

given variable x, term t, & formula P, define $P[t/x]$ to be obtained by replacing every free occurrence of variable x in P w/ t

↳ substitution occurs after quantifier has been removed in forall-e

↳ e.g. P is $q(x)$
$P[y/x]$ is $q(y)$

↳ e.g. P is $\forall x \cdot q(y, x)$
$P[4/y]$ is $\forall x \cdot q(4, x)$

↳ e.g. P is $\forall x \cdot q(x,y) \wedge \forall y \cdot q(x,y)$
$P[a+b/y]$ is $\forall x \cdot q(x, a+b) \wedge \forall y \cdot q(x,y)$

↳ e.g. P is $\forall x \cdot \exists y \cdot q(x,y)$
$P[f(y)/x]$ is $\forall x \cdot \exists y \cdot q(x, y)$

- variable capture: location that had free var in it becomes bound or bound var becomes bound to diff quantifier after sub
  - ↳ e.g. P is $\forall x \cdot q(y, x)$
    $P[x/y]$ isn't allowed b/c it leads to $\forall x \cdot q(x, x)$
  - ↳ e.g. P is $\forall y \cdot a(x) \wedge b(y)$
    $P[g(f(y))/x]$ isn't allowed
  - ↳ e.g. P is $(\forall y \cdot \exists x \cdot q(x, y, z)) \wedge r(y)$
    $P[x+1/y]$ is $(\forall y \cdot \exists x \cdot q(x,y,z)) \wedge r(x+1)$
    $P[x+1/z]$ is not allowed
- substitution: given term t, var z, & formula P, t is free for z in P if in P, no free z s occur within scope of $\forall w$ or $\exists w$ for any free var w occuring in t
  - ↳ if t is not free for z in P, subbing t for z in P would result in var capture
  - ↳ every var in t is free b/c it's a term & has no bound var
- when using forall_e, in sub of t for x, no free var in t should be captured

exists introduction: if P is true for some value, then there exists a value for which P is true

↳ $\frac{P[t/x]}{\exists x \cdot P}$, exists_i
- since $P[t/x]$ is premise, choose term t in formula (alr on line of proof) & conclude (i.e. add to proof) formula w/ t replaced by x & $\exists x \cdot$ in front
- not all occurrences of t need to be replaced
- t is witness to truth of $\exists x \cdot P$

↳ when choosing t:
- term must have no bound var in it to satisfy rules abt var sub
- var x that's introduced shouldn't be var that's free outside of t b/c it might get bound to smth else

- e g.

$\forall x \bullet a(x) \wedge b(x)$
$\vdash$
$\exists x \bullet a(x)$

1) $\forall x \bullet a(x) \wedge b(x)$ premise
2) $a(c) \wedge b(c)$ by forall_e on 1
3) $a(c)$ by and_e on 2
4) $\exists x \bullet a(x)$ by exists_i on 3

- e.g. for $\forall y \cdot p(g(y), x) \wedge q(y)$, can't choose g(y) as term & conclude $\exists z \cdot \forall y \cdot p(z, x) \wedge q(y)$ b/c g(y) contains var that's bound in formula
- unknown var means not all values would satisfy it
  - ↳ e.g. $x+1 = 2$ is acc $\exists x \cdot x+1 = 2$
  - ↳ free var st existential quantification of it yields formula that's true
  - ↳ rep specific & unknown value
  - ↳ can write as x_u

- genuine var means all values would satisfy it
  - ↳ e.g. $x + x = 2x$ is acc $\forall x \cdot x + x = 2x$
  - ↳ free var st universal quantification of it yields formula that's true
  - ↳ rep any value
  - ↳ can write as x_g

forall introduction: if formula is true for unconstrained value, then it's true for all values
- ↳ aka generalization
- ↳

for every x_g {
  ...
  $P[x_g/x]$
}
——— forall_i
$\forall x \bullet P$

  - ° x_g must be genuine var not free on any prev active line of proof
    - → x_g can't appear outside subproof
  - ° sub rule: x_g must be free for x in P

exists elimination: if formula is true for some value & using unknown var x_u, derive that a formula Q & doesn't contain x_u, conclude Q holds
- ↳

$\exists x \bullet P$
for some x_u assume $P[x_u/x]$ {
  ...
  Q
}
——— exists_e
Q

  - ° x_u can't appear outside subproof
  - ° sub rule: x_u must be free for x in P

- if formula has typed var, can mostly ignore types in proofs & they don't need to be included
  - ↳ when doing sub, respect types
    - ° e.g. don't sub term of type T1 into predicate of type T3 → bool
- e.g. $\vdash (\forall x, y \cdot p(x, y)) \Rightarrow (\forall y, x \cdot p(x, y))$
  1) assume $\forall x, y \cdot p(x, y)$ {
    2) for every y_g, x_g {
      3) $p(x_g, y_g)$ by forall_e on 2
    }
    4) $\forall y, x \cdot p(x, y)$ by forall_i on 2-3
  }
  5) $(\forall x, y \cdot p(x, y)) \Rightarrow (\forall y, x \cdot p(x, y))$ by imp_i on 1-4
- e.g. $\forall x \cdot p(x, x) \vdash \forall x \cdot \exists y \cdot p(x, y)$
  1) $\forall x \cdot p(x, x)$ premise
  2) for every x_g {
    3) $p(x_g, x_g)$ by forall_e on 1

4) $\exists y \cdot p(xg, y)$ by exists_i on 3

}

5) $\forall x \cdot \exists y \cdot p(x, y)$ by forall_i on 2-4

- e.g. $\exists x \cdot p(x) \vdash \neg(\forall x \cdot \neg p(x))$

1) $\exists x \cdot p(x)$ premise

2) disprove $\forall x \cdot \neg p(x)$ {

3) for some xu assume $p(xu)$ {

4) $\neg p(xu)$ by forall_e on 2

5) false by not_e on 3,4

}

6) false by exists_e on 1, 3-5

}

7) $\neg(\forall x \cdot \neg p(x))$ by raa on 2-6

- e.g. $\neg(\exists x \cdot p(x)) \vdash \forall x \cdot \neg p(x)$ (dm law)

1) $\neg(\exists x \cdot p(x))$ premise

2) for every xg {

3) disprove $p(xg)$ {

4) $\exists x \cdot p(x)$ by exists_i on 3

5) false by not_e on 1,4

}

6) $\neg p(xg)$ by raa on 3-5

}

7) $\forall x \cdot \neg p(x)$ by forall_i on 2-6

- e.g. $\forall x \ \exists y \cdot \neg wins(x, y) \vdash \neg(\exists x \ \forall y \cdot wins(x, y))$

1) $\forall x \cdot \exists y \cdot \neg wins(x, y)$ premise

2) disprove $\exists x \cdot \forall y \cdot wins(x, y)$ {

3) for some xu assume $\forall y \cdot wins(xu, y)$ {

4) $\exists y \cdot \neg wins(xu, y)$ by forall_e on 1

5) for some yu assume $\neg wins(xu, yu)$ {

6) $wins(xu, yu)$ by forall_e on 3

7) false by not_e on 5,6

}

8) false by exists_e on 4, 5-7

}

9) false by exists_e on 2, 3-8

}

10) $\neg(\exists x \cdot \forall y \cdot wins(x, y))$ by raa on 2-9

- natural deduction for predicate logic is sound & complete
- summary of ND rules:

These rules are in addition to the rules already given for propositional logic.

	INTRODUCTION	ELIMINATION

$\forall$	for every x_g {
	$\quad\vdots$
	$\quad P[x_g/x]$
	}
	$\dfrac{}{\forall x \bullet P}$ forall_i
		$\dfrac{\forall x \bullet P}{P[t/x]}$ forall_e

$\exists$	$\dfrac{P[t/x]}{\exists x \bullet P}$ exists_i	$\exists x \bullet P$
		for some x_u assume $P[x_u/x]$ {
		$\quad\vdots$
		$\quad Q$
		}
		$\dfrac{}{Q}$ exists_e
		x_u does not appear in Q.

Theory of Equality

Reflexivity

$$\frac{}{t = t}\ \text{eq\_i}$$

Substitution

$$\frac{t_1 = t_2 \quad P[t_2/x]}{P[t_1/x]}\ \text{eq\_e}$$

where t_1 and t_2 are free for x in P.

Substitution

$$\frac{t_1 = t_2 \quad P[t_1/x]}{P[t_2/x]}\ \text{eq\_e}$$

where t_1 and t_2 are free for x in P.

Theory of arithmetic

bc) $P(0)$ ...
ih) inductionstep $P(k_g)$ {
$\quad\vdots$
$\quad$ ih+x) $P(k_g + 1)$ by ...
}
x+1) $\forall n \bullet P(n)$ by induction on bc, ih-ih+x

SEMANTIC TABLEAUX

- universal quantification:
  - 1) $\forall x \bullet P(x)$
    by forall_nb on 1
    2) $P(t)$
  - ↳ aka universal instantiation
  - ↳ t is term ? it's free for x in P
    - t may be used in universal instantiation of another formula
  - ↳ may be useful to apply this rule to same line lt times w/diff instantiations
- existential quantification:
  - 1) $\exists x \bullet P(x)$
    by exists_nb on 1
    2) $P(y)$
  - ↳ y is unknown var that's not free var in tableau so far

- when doing ST, apply non-branching rules first
- use existential quantification & negative universal rules before universal & negative existential rules
- e.g. $\exists x \cdot p(x) \vdash \neg(\forall x \cdot \neg p(x))$

  1) $\exists x \cdot p(x)$

  2) $\neg\neg(\forall x \cdot \neg p(x))$

  by not_not_nb on 2

  3) $\forall x \cdot \neg p(x)$

  by exists_nb on 1

  4) $p(a)$

  by forall_nb on 3

  5) $\neg p(a)$

  closed on 4, 5
- e.g. $\exists x \cdot \neg p(x) \vdash \exists x \cdot p(x) \Rightarrow q(x)$

  1) $\exists x \cdot \neg p(x)$

  2) $\neg(\exists x \cdot p(x) \Rightarrow q(x))$

  by exists_nb on 1

  3) $\neg p(xu)$

  by not_exists_nb on 2

  4) $\forall x \cdot \neg(p(x) \Rightarrow q(x))$

  by forall_nb on 4

  5) $\neg(p(xu) \Rightarrow q(xu))$

  by not_imp_nb on 5

  6) $p(xu)$

  7) $\neg q(xu)$

  closed on 3, 6

- ST for predicate logic is sound & complete
- summary of ST rules:

All the rules for propositional logic can also be used.

	POSITIVE	NEGATIVE
$\forall$	1) $\forall x \bullet P(x)$ by forall_nb on 1 2) $P(t)$ where t is a term that is a legal substitution.	1) $\neg(\forall x \bullet P(x))$ by not_forall_nb on 1 2) $\exists x \bullet \neg P(x)$
$\exists$	1) $\exists x \bullet P(x)$ by exists_nb on 1 2) $P(y)$ where y is a variable that has not been used in the tableau so far.	1) $\neg(\exists x \bullet P(x))$ by not_exists_nb on 1 2) $\forall x \bullet \neg P(x)$

ROLE OF ABSTRACTION

- if asked to formalize single sentence by itself, formalize all details (all logical connectives, quantifiers, predicates, functions, & constants)

- if asked to formalize sentences of an argument & then prove it's valid, choose how many details we formalize based on what we want to prove
- rules of thumb:
  - ↳ for all sentences:
    - identify all possible logical connectives
    - identify what might be quantified
      - → e.g. implicit quantifiers, constants
    - identify possible predicates & functions
      - → quantified vars must be args to predicates
    - look for similar phrases to determine how much detail you need to provide
      - → i.e. how much info can 1 predicate/function capture
    - think of outline on how proof will proceed
- types serve 2 purposes:
  - ↳ limit values for quantified var (useful for counterexamples)
  - ↳ limit possible uses of pred / func
- $\forall x : T \cdot P(x)$ is same as $\forall x \cdot T(x) \Rightarrow P(x)$
- $\exists x : T \cdot P(x)$ is same as $\exists x \cdot T(x) \wedge P(x)$
- if unary predicate is used in all premises & conclusion as part of antecedant of universally quantified formula or one conjunct of existentially quantified formula, good candidate to make type

- e.g.

Premises:

1. Two parallel lines that are not identical do not intersect.
2. If two lines are parallel, then they have no points in common or all points in common.
3. Two lines that have all points in common intersect.

Conclusion:

- Two parallel lines that are not identical have no points in common.

Common phrases:
↳ two lines
↳ parallel
↳ intersect
↳ not identical
↳ no pts in common
↳ all pts in common

Formalization:
$p(x)$ means x is parallel
$d(x)$ means x is not identical
$n(x)$ means x has no points in common
$a(x)$ means x has all points in common
$i(x)$ means x intersects
$\forall x \cdot TwoLines \cdot p(x) \wedge d(x) \Rightarrow \neg i(x)$,
$\forall x : TwoLines \cdot p(x) \Rightarrow n(x) \mid a(x)$,
$\forall x : TwoLines \cdot a(x) \Rightarrow i(x)$
$\vdash$
$\forall x : TwoLines \cdot p(x) \wedge d(x) \Rightarrow n(x)$

1) $\forall x : TwoLines \cdot p(x) \wedge d(x) \Rightarrow \neg i(x)$ premise

2) $\forall x : TwoLines \cdot p(x) \Rightarrow n(x) \mid a(x)$ premise

3) $\forall x : TwoLines \cdot a(x) \Rightarrow i(x)$ premise

4) for every xg {

  5) assume $p(xg) \wedge d(xg)$ {

    6) $p(xg) \wedge d(xg) \Rightarrow \neg i(xg)$ by forall_e on 1

    7) $\neg i(xg)$ by imp_e on 5, 6

    8) $p(xg)$ by and_e on 5

    9) $p(xg) \Rightarrow n(xg) \mid a(xg)$ by forall_e on 2

    10) $n(xg) \mid a(xg)$ by imp_e on 8, 9

    11) $a(xg) \Rightarrow i(xg)$ by forall_e on 3

    12) $\neg a(xg)$ by imp_e on 7, 11

    13) $n(xg)$ by or_e on 10, 12

  }

  14) $p(xg) \wedge d(xg) \Rightarrow n(xg)$ by imp_i on 5-13

}

15) $\forall x \cdot p(x) \wedge d(x) \Rightarrow n(x)$ by forall_i on 4-14

module 4

HIDDEN PREMISES

- for single argument, unwritten info is **hidden premise**
- a **theory** is when we have multiple arguments relating to same info & we collect knowledge that relates constants, predicates, & functions to each other
- **enthymeme** is argument that contains hidden premise
  - ↳ i.e. contains unstated premises that are obv true
- e.g.

The crime was committed by someone at the General Store at 6pm.
$\exists x \bullet cc(x, GS, 6)$

Billy the Kid was in the Jail at 6pm.
$locn(B, J, 6)$

Therefore, Billy the Kid did not commit the crime at the General Store at 6pm.
$\neg cc(B, GS, 6)$

where

- $cc(x, y, z)$ means x commited the crime at y at time z
- $locn(x, y, z)$ means x was at location y at time z
- B is the constant Billy
- J is the constant Jail
- GS is the constant General Store

↳ hidden premises:

1) If smth commits crime at location at a time, they're at that location at that time (relates cc & locn in all interpretations)
   → $\forall x, y, z \cdot cc(x, y, z) \Rightarrow locn(x, y, z)$
2) Smth can't be at 2 locations at same time (limits interpretations of locn)
   → $\forall x, y, z, w \cdot locn(x, y, w) \wedge locn(x, z, w) \Rightarrow (y = z)$
3) General store & Jail are not same
   → $\neg(J = GS)$

↳

hp1) $\forall x, y, z \bullet cc(x, y, z) \Rightarrow locn(x, y, z)$ premise
hp2) $\forall x, y, z, w \bullet locn(x, y, w) \wedge locn(x, z, w) \Rightarrow y = z$ premise
1) $\exists x \bullet cc(x, GS, 6)$ premise
2) $locn(B, J, 6)$ premise
hp3) $\neg(J = GS)$ premise
4) disprove $cc(B, GS, 6)$ {
  5) $\forall y, z \bullet cc(B, y, z) \Rightarrow locn(B, y, z)$ by forall_e on hp1
  6) $\forall z \bullet cc(B, GS, z) \Rightarrow locn(B, GS, z)$ by forall_e on 5
  7) $cc(B, GS, 6) \Rightarrow locn(B, GS, 6)$ by forall_e on 6
  8) $locn(B, GS, 6)$ by imp_e on 4,7
  91) $\forall y, z, w \bullet locn(B, y, w) \wedge locn(B, z, w) \Rightarrow y = z$ by forall_e on hp2
  92) $\forall z, w \bullet locn(B, J, w) \wedge locn(B, z, w) \Rightarrow J = z$ by forall_e on 91
  93) $\forall w \bullet locn(B, J, w) \wedge locn(B, GS, w) \Rightarrow J = GS$ by forall_e on 92
  9) $locn(B, J, 6) \wedge locn(B, GS, 6) \Rightarrow J = GS$ by forall_e on 93
  10) $locn(B, J, 6) \wedge locn(B, GS, 6)$ by and_i on 2,8
  11) $J = GS$ by imp_e on 9,10
  12) **false** by not_e on hp3,11
}
13) $\neg cc(B, GS, 6)$ by raa on 4-12

- formalizing arguments force us to discover hidden premises b/c we can't prove argument w/o them
  another term for hp is environmental assumption
- hp must be valid in every interpretation for argument ∴ often universally quantified formulas

THEORIES

- theory: set of axioms (i.e. facts) abt specific constants, functions, & predicate symbols
  - ↳ built on logic
  - ↳ we will use pred logic as underlying logic
- e.g.

> Let's build a theory of family relationships that allows us to prove the following argument:
>
> Havi is John's brother. John is Bo's brother.
>
> Therefore, Havi is Bo's brother.
>
> where:
>
> - $br(x, y)$ means x is y's brother
> - H is for Havi
> - J is for John
> - B is for Bo
>
> $br(H, J), br(J, B) \models br(H, B)$

  - ↳ axiom is br pred is transitive: $\forall x, y, z \cdot br(x,y) \wedge br(y,z) \Rightarrow br(x,z)$
    - becomes premise
  - ↳

```
1  br(H,J),
2  br(J,B),
3  forall x, y, z . br(x,y) & br(y,z) => br(x, z)
4  |-
5  br(H,B)
6
7  p1) br(H, J)  premise
8  p2) br(J, B)  premise
9  a1) forall x, y, z . br(x,y) & br(y,z) => br(x, z)
       premise
10
11 1) forall y, z . br(H,y) & br(y,z) => br(H, z)  by
       forall_e on a1
12 2) forall z . br(H,J) & br(J,z) => br(H, z)  by
       forall_e on 1
13 3) br(H,J) & br(J,B) => br(H,B) by forall_e on 2
14 4) br(H,J) & br(J,B) by and_i on p1,p2
15 5) br(H,B)  by  imp_e on 3,4
```

- interpretation in which all axioms of theory are true is a model of the theory
  - ↳ when constructing theory, we usually have standard/normal interpretation of theory in mind
- theories come in 2 flavours:
  - ↳ general ∴ mathematical: well-known theories developed by others

- e.g. arithmetic, sets, ? sequences
- axioms are added to set of rules used for pred logic

↳ theories for particular systems / environments: constructed to describe real world
- usually particular to arguments we're trying to show are valid
- add axioms of these theories to premises

· check axioms are consistent or else, every wff would be in theory (including false)

PREDICATE LOGIC WITH EQUALITY

· in all interpretations, syntactic symbol = will mean equality of values in domain

· = is used for equality predicate

↳ binary infix predicate ? its values must be terms

↳ lower precedence than other functions

·

Syntax	Semantics	Proof Theory
$\Leftrightarrow$, iff	$A \Leftrightarrow B$ logical equiv	$\leftrightarrow$ TP
	IFF	
=	=	

↳ $\Leftrightarrow$ is for truth value equality

· can't write $a=b=c$

↳ instead, $(a=b) \wedge (b=c)$

· e.g.

1. There are at least two solutions.

$$\exists x, y \cdot soln(x) \wedge soln(y) \wedge x \neq y$$

2. There are not two or more solutions.

$$\neg(\exists x, y \cdot soln(x) \wedge soln(y) \wedge x \neq y)$$

3. There are exactly two solutions.

$$\exists x, y \cdot soln(x) \wedge soln(y) \wedge x \neq y \wedge \underbrace{\forall z \cdot soln(z) \Rightarrow z = x \vee z = y}_{\updownarrow}$$

$$\neg(\exists z \cdot soln(z) \wedge z \neq x \wedge z \neq y)$$

↳ counting # of solutions ($num(soln) = 1$) doesn't work well b/c soln doesn't make sense as constant

· e.g.

1. Alice likes only bubblegum ice cream.

$$\forall y \cdot likes(Alice, y) \wedge likes(Alice, bgic) \Rightarrow y = bgic$$

↓ shorter

$$\forall y \cdot likes(Alice, y) \Leftrightarrow y = bgic$$

2. Only Alice likes bubblegum ice cream.

$$\forall x \cdot likes(x, bgic) \Leftrightarrow x = Alice$$

3. The only kind of ice cream Alice likes is bubble gum.

$$\forall x \cdot iceCream(x) \wedge likes(Alice, x) \Leftrightarrow x = bgic$$

· natural deduction rules for equality:

↳ Reflexivity

$$\frac{}{t = t}\ \text{eq\_i}$$

↳ Substitution

$$\frac{t_1 = t_2 \quad P[t_2/x]}{P[t_1/x]}\ \text{eq\_e} \qquad \frac{t_1 = t_2 \quad P[t_1/x]}{P[t_2/x]}\ \text{eq\_e}$$

where t_1 and t_2 are free for x in P.

- 2 other properties can be derived:
  - ↳ symmetry: ⊢ ∀x,y · (x = y) ⇒ (y = x)
    - proof:
      1) for every xg {
      2) for every yg {
      3) assume xg = yg {
      4) xg = xg by eq_i
      5) yg = xg by eq_e on 3,4
      }
      6) xg = yg ⇒ yg = xg by imp_i on 3-5
      }
      7) ∀y · xg = y ⇒ y = xg by forall_i on 2-6
      }
      8) ∀x,y · (x = y) ⇒ (y = x) by forall_i on 1-7
  - ↳ transivity: ⊢ ∀x,y,z · (x = y) ∧ (y = z) ⇒ (x = z)
- Leibniz's Law: if $t_1 = t_2$ is a theorem, then so is $P[t_1/x] \Leftrightarrow P[t_2/x]$
  - ↳ i.e. ability to sub equals for equals
- in semantics, standard interpretations are where = is interpreted as equality on values of domain
- e g.

Show the following argument is not valid:
$\exists x \bullet p(x) \wedge q(x), p(A), A = B \models q(B)$
where A, B are constants

Interpretation:
Domain = {d_1, d_2}
Mapping:

Syntax	Mapping
A	d_1
B	d_1
p(.)	P(d_1) = T
	P(d_2) = T
q(.)	Q(d_1) = F
	Q(d_2) = T

P1: [∃x · p(x) ∧ q(x)]
= P(d1) AND Q(d1) OR P(d2) AND Q(d2)
= T AND F OR T AND T
= F OR T
= T

P(2): [p(A)]
= P(dl)
= T

P3: [A = B]
= $d_1 = d_1$
= T

Conclusion. [q(B)]
= Q(dl)
= F

e.g.

Danger lies before you, while safety lies behind,
Two of us will help you, whichever you would find,
P1 One among us seven will let you move ahead,
P2 Another will transport the drinker back instead,
P3 Two among our number hold only nettle wine,
P4 Three of us are killers, waiting hidden in line.
Choose, unless you wish to stay here forevermore,
To help you in your choice, we give you these clues four:
P5 First, however slyly the poison tries to hide
You will always find some on nettle wine's left side;
P6 Second, different are those who stand at either end,
P7 But if you would move onwards, neither is your friend;
P8 Third, as you see clearly, all are different size,
Neither dwarf nor giant holds death in their insides;
P9 Fourth, the second left and the second on the right
Are twins once you taste them, though different at first sight.

This puzzle has multiple solutions. As pointed out on the web the author forgot (or the publishers edited away!) information on the sizes of the bottles standing in the row. Here are the sizes of the bottles at each position, where 1 is the smallest bottle and 7 is the largest bottle:

P W A P P W B

Position	1	2	3	4	5	6	7
Size	3	4	1	2	6	7	5

With this information there is only one solution to the puzzle.

P1: $\exists x \cdot b(x) = A \wedge \forall k \cdot b(k) = A \Rightarrow k = x$

P2: $\exists x \cdot b(x) = B \wedge \forall k \cdot b(k) = B \Rightarrow k = x$

P3: $\exists x, y \cdot b(x) = W \wedge b(y) = W \wedge x \neq y \wedge \forall z \cdot b(z) = W \Rightarrow z = x \vee z = y$

P4: $\exists x, y, z \cdot b(x) = P \wedge b(y) = P \wedge b(z) = P \wedge x \neq y \wedge y \neq z \wedge x \neq z \wedge \forall w \cdot b(w) = P \Rightarrow w = x \vee w = y \vee w = z$

P5: $\forall i \cdot b(i+1) = W \Rightarrow b(i) = P$

P6: $\neg(b(1) = b(7))$

P7: $b(1) \neq A \wedge b(7) \neq A$

P8: $\neg(b(3) = P \vee b(6) = P)$

P9: $b(2) = b(6)$

HP1. $\forall x, y \cdot b(x) = A \wedge b(y) = B \Rightarrow x \neq y$
HP2: $\forall x, y \cdot b(x) = A \wedge b(y) = W \Rightarrow x \neq y$
HP3. $\forall x, y \cdot b(x) = A \wedge b(y) = P \Rightarrow x \neq y$
HP4: $\forall x, y \cdot b(x) = B \wedge b(y) = W \Rightarrow x \neq y$
HP5: $\forall x, y \cdot b(x) = B \wedge b(y) = P \Rightarrow x \neq y$
HP6: $\forall x \cdot y \cdot b(x) = P \wedge b(y) = W \Rightarrow x \neq y$

P9 1) $b(2) = b(6)$ premise
P1 2) $\exists i \cdot b(i) = A \wedge (\forall k \cdot b(k) = A \Rightarrow k = i)$ premise
⊢
$\neg(b(2) = A)$
3) disprove $b(2) = A$ {
4) $b(6) = A$ by eq_e on 1, 3
5) for some iu assume $b(iu) = A \wedge (\forall k \cdot b(k) = A \Rightarrow k = iu)$ {
6) $b(iu) = A$ by and_e on 5
7) $\forall k \cdot b(k) = A \Rightarrow k = iu$ by and_e on 5
8) $b(2) = A \Rightarrow 2 = iu$ by forall_e on 7
9) $2 = iu$ by imp_e on 3, 8
10) $b(6) = A \Rightarrow 6 = iu$ by forall_e on 7
11) $6 = iu$ by imp_e on 4, 10
12) $2 = 6$ by eq_e on 9, 11
13) false by arith on 12
}
14) false by exists_e on 2, 5-13
}
15) $\neg(b(2) = A)$ by raa on 3-14

P7 1) $b(1) \neq A \wedge b(7) \neq A$ premise
2) $b(1) \neq A$ by and_e on 1

P7 1) $b(1) \neq A \wedge b(7) \neq A$ premise
2) $b(7) \neq A$ by and_e on 1

P9 1) $b(2) = b(6)$ premise
2) $\exists x \cdot b(x) = A \wedge \forall k \cdot b(k) = A \Rightarrow k = x$ premise
3) disprove $b(6) = A$ {
4) for some xu assume $b(xu) = A \wedge (\forall k \cdot b(k) = A \Rightarrow k = xu)$ {
5) $b(xu) = A$ by and_e on 4
6) $\forall k \cdot b(k) = A \Rightarrow k = xu$ by and_e on 4
7) $b(2) = A \Rightarrow 2 = xu$ by forall_e on 6
8) $b(2) = A$ by eq_e on 1, 3
9) $2 = xu$ by imp_e on 7, 8
10) $b(6) = A \Rightarrow 6 = xu$ by forall_e on 6
11) $6 = xu$ by imp_e on 3, 10
12) $2 = 6$ by eq_e on 9, 11
13) false by arith on 12
}
14) false by exists_e on 2, 4-13

}

15) $\neg(b(G) = A)$ by raa on 3-14

THEORY OF ARITHMETIC

- natural #s are integers ≥ 0
  - ↳ nat = {0, 1, 2, 3, ...}
- theory of arithmetic contains:
  - ↳ constants: 0
  - ↳ functions:
    - suc (plus 1)
    - + (addition)
    - x (multiplication)
  - ↳ predicates:
    - < (less than)
- suc(0) = 1, suc(suc(0)) = 2, etc. is finite rep of infinite set
- Peano's axioms for arithmetic:
  1) 0 is not successor : $\forall n \cdot \neg(suc(n) = 0)$
  2) successors are distinct : $\forall x \cdot \forall y \cdot (suc(x) = suc(y)) \Rightarrow (x = y)$
     - contrapositive: $\forall x \cdot \forall y \cdot \neg(x = y) \Rightarrow \neg(suc(x) = suc(y))$
  3) 0 is identity of addition:
     - $\forall m \cdot m + 0 = m$
     - $\forall m \cdot 0 + m = m$
  4) addition: $\forall m, n \cdot m + suc(n) = suc(m+n)$
     - replace suc(n) w/ n+1 so axiom becomes $\forall m, n \cdot m + (n+1) = (m+n) + 1$
  5) multiplication by 0: $\forall n \cdot n \times 0 = 0$
  6) multiplication: $\forall m, n \cdot m \times suc(n) = m \times n + m$
     - axiom turns multiplication into repeated addition
  7) mathematical induction: used to prove all natural #s have certain property
     - i.e. $\forall n : nat \cdot P(n)$
     - for all P: $(P(0) \wedge (\forall k \cdot P(k) \Rightarrow P(suc(k)))) \Rightarrow \forall n \cdot P(n)$
       - → P(0) is base case
       - → P(k) is induction hypothesis
       - → $\forall k \cdot P(k) \Rightarrow P(suc(k))$ is induction proof / inductive step
- standard model for theory of arithmetic:
  1. *Domain* = *nat* (natural numbers)
  2. Mapping

Syntax	Meaning
0	0
$suc(\cdot)$	$Suc(x) := x + 1$
$\cdot + \cdot$	$Plus(x, y) := x + y$
$\cdot \times \cdot$	$Times(x, y) := x \times y$
$\cdot < \cdot$	$LessThan(x, y) := x < y$

- within ND, parts that are underlined are common to all induction proofs:

...
bc) $P(0)$ // base case
...
ih) for every k_g {
ih+1) assume $P(k_g)$ { // induction hypothesis
⋮
ih+k) $P(k_g + 1)$ by ...
}
ih+k+1) $P(k_g) \Rightarrow P(k_g + 1)$ by imp_i on ih+1 - ih+k
}
ih+k+2) $\forall k \bullet P(k) \Rightarrow P(k + 1)$ by forall_i ih - ih+k+1
ih+k+3) $\forall n \bullet P(n)$ by induction on bc, ih+k+2

↳ shortcut on george.

bc) $P(0)$ ...
ih) inductionstep $P(k_g)$ {
⋮
ih+x) $P(k_g + 1)$ by ...
}
x+1) $\forall n \bullet P(n)$ by induction on bc, ih-ih+x

- practical approach on using theory of arithmetic:
  - ↳ use #s as constants (0, 1, 2, ...)
  - ↳ use x+1 instead of suc(x)
  - ↳ use mathematical functions
    - e.g. +, −, ×, ÷, Σ, π, √, etc.
  - ↳ use pred w/ well-known meanings
    - e.g. even, odd, <, ≤, >, ≥, etc.
  - ↳ in george: +, -, *, /, ^, <, >, <=, >=
    - arithmetic operators are left associative except for ^
  - ↳ rule "by arith" includes all regular arithmetic properties we know as laws
  - ↳ axioms that all #s are distinct is included
    - e.g. ¬(1 = 2), ¬(2 = 7), etc. are all theorems
- e.g. $\forall n \cdot n \geq 5 \Rightarrow P(n)$

```
1) assume 0 ≥ 5 {
   2) ¬(0 ≥ 5)   by arith
   3) P(0)       by not_e on 1, 2
  }
bc) 0 ≥ 5 ⇒ P(0)  by imp_i on 1-3
ih) induction ng ≥ 5 ⇒ P(ng) {
    10) assume ng+1 ≥ 5 {
        11) ng = 4 v ng ≥ 5   by arith on 10
        12) case ng = 4 {
            ⋮ // prove P(5)
          17) P(4+1)   by ...
          18) P(ng+1)  by eq_e on 12,17
```

}

19) case $ng \geq 5$ {

20) P(ng) by imp_e on ih, 19

⋮

25) P(ng+1) by . .

}

26) P(ng+1) by ...

}

27) $ng+1 \geq 5 \Rightarrow P(ng+1)$ by imp_i on 10-26

}

28) $\forall n \cdot n \geq 5 \Rightarrow P(n)$ by induction on bc, ih-27

- to write def of function on ordered domain like nat, use recursive function
  - ↳ define function by giving:
    - its value for 0 (or another terminating clause)
    - its value for n+1 in terms of n (or its value for n in terms of n-1)
  - ↳ e.g. $\sum_{i=0}^{n} i = 0+1+2+3+\ldots+n$ as recursive def:
    - $\sum_{i=0}^{0} i = 0$
    - $\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$
- e.g prove every 3rd fib # is even

f(0) = 0

f(1) = 1

f(n) = f(n-1) + f(n-2) for $n \geq 2$

$\vdash \forall n : nat \cdot \exists y \cdot f(3n) = 2y$

1) f(3×0) = f(0) by arith

2) f(3×0) = 0 by arith on 1 // defn of f

3) f(3×0) = 2×0 by arith on 2

bc) $\exists y \cdot f(3 \times 0) = 2y$ by exists_i on 3

ih) inductionstep $\exists y \cdot f(3ng) = 2y$ {

4) for some yu assume $f(3ng) = 2yu$ {

5) $3ng+3 \geq 2$ by arith // ng is natural #

6) $3ng+2 \geq 2$ by arith

7) f(3(ng+1)) = f(3ng+3) by arith

8) f(3(ng+1)) = f(3ng+2) + f(3ng+1) by arith on 5,7 // defn of f

9) f(3(ng+1)) = f(3ng+1) + f(3ng) + f(3ng+1) by arith on 6, 8 // defn of f

10) f(3(ng+1)) = 2f(3ng+1) + f(3ng) by arith on 9

11) f(3(ng+1)) = 2f(3ng+1) + 2yu by eq_e on 4,10

12) f(3(ng+1)) = 2(f(3ng+1) + yu) by arith on 11

13) $\exists y \cdot f(3(ng+1)) = 2y$ by exists_i on 12

}

14) $\exists y \cdot f(3(ng+1)) = 2y$ by exists_e on ih, 4-13

}

15) $\forall n \ \exists y \cdot f(3n) = 2y$ by induction on bc, ih-14

- e.g

```
1  #check ND
2
3  |- forall n. n >= 1 => sum(i,1,n, (2 * i) - 1) = n^2
4
5  1) assume 0 >= 1 {
6       2) !(0 >= 1) by arith
7       3) sum(i,1,0, (2 * i) - 1) = 0^2 by not_e on 1,2
8  }
9  bc) 0 >= 1 => sum(i,1,0, (2 * i) - 1) = 0^2
10    by imp_i on 1-3
```

Continued.

```
ih) inductionstep ng >= 1 => sum(i,1,ng, (2 * i) - 1)=ng^2 {
    4) assume ng + 1 >= 1 {
        5) ng = 0 | ng >= 1 by arith on 4
        6) case ng = 0 {
            7) sum(i,1,0+1, (2 * i) - 1) = 2 * 1 - 1
               by arith // defn of sum
            9) sum(i,1,0+1, (2 * i) - 1) = (0+1)^2
               by arith on 7
            11) sum(i,1,ng+1, (2*i) -1) = (ng+1)^2
                by eq_e on 6,9
        }
```

```
        12) case ng >= 1 {
            13) sum(i,1,ng, (2 * i) - 1) = ng^2
                by imp_e on ih,12
            14) sum(i,1,ng+1, (2 * i) - 1) = 2*(ng+1) - 1
                   + sum(i,1,ng, (2 * i) -1)
                by arith // defn of sum
            16) sum(i,1,ng+1, (2 * i) - 1) = 2*(ng+1) - 1
                   + (ng^2)
                by eq_e on 13,14
            17) sum(i,1,ng+1, (2 * i) - 1) = ng^2
                   + 2*ng +1
                by arith on 16
            18) sum(i,1,ng+1, (2 * i) - 1) = (ng + 1)^2
                by arith on 17
        }
        20) sum(i,1,ng+1,(2* i) - 1) = (ng+1)^2
            by cases on 5, 6-11,12-18
    }
    21) ng +1 >= 1 => sum(i,1,ng+1,(2 * i) - 1) = (ng+1)^2
        by imp_i on 4-20
}
22) forall n . n >= 1 => sum(i,1,n, (2 * i) - 1) = n^2
    by induction on bc, ih-21
```

module 5

INTRO TO SETS

- **set**: collection of elements / members
  - ↳ e.g. set of primes less than 20 is SmallPrimes = {2, 3, 5, 7, 11, 13, 17, 19}
  - ↳ may be ∞
  - ↳ no order among elements
  - ↳ do not contain multiple copies of same element (i.e. elements are all distinct)
- ∈ is pred in infix notation meaning "belongs to / is in"
  - ↳ $x \in S$ means x is in set S
  - ↳ $x \notin S \iff \neg(x \in S)$

2 ways to formalize def of elements in set:

- ↳ **set enumeration**: list items
  - e.g. {apple, orange, grapes}
- ↳ **set comprehension**: define set using wff
  - {⟨term⟩ | ⟨wff⟩}
  - e.g. $\{x \mid x \in nat \wedge x \leq 6\}$
    - → x is free var in wff
    - → elements in set are values of x in domain for which wff evaluates to T
    - → wff is a **characteristic predicate**

- using x before | in set comp binds the var
- e.g.
  1. Natural numbers that are divisors of 10.
     Set enumeration: $\{1, 2, 5, 10\}$
     Set comprehension: $\{x \mid x \in nat \wedge \exists y \bullet y \in nat \wedge x * y = 10\}$
  2. Students from BC attending lecture.
     Set enumeration: $\{Joe, Jianwei, Jose, \ldots\}$
     Set comprehension:
     $\{x \mid student(x) \wedge from(x, BC) \wedge atlecture(x)\}$ or
     $\{x \mid x \in student \wedge from(x, BC) \wedge atlecture(x)\}$

- axiom: **types as sets**
  - ↳ $(\forall x : \mathbb{B} \cdot P(x)) \iff (\forall x \cdot x \in \mathbb{B} \Rightarrow P(x))$
  - ↳ $(\exists x : \mathbb{B} \cdot P(x)) \iff (\exists x \cdot x \in \mathbb{B} \wedge P(x))$
- **Z notation** for set comp: {⟨term⟩ · ⟨signature⟩ | ⟨wff⟩}
  - ↳ ⟨term⟩ is term in pred logic
    - i.e. exp using vars / functions to return value
    - can be omitted if it's just a var & we have a ⟨signature⟩
  - ↳ if using types, ⟨signature⟩ lists vars used in ⟨term⟩ & their types
    - if not using types, ⟨signature⟩ can be omitted but need ⟨term⟩ instead
  - ↳ ⟨wff⟩ is wff in pred logic w/ vars used in ⟨term⟩ as free vars
    - if no pred needed (i.e. formula is true), can omit ⟨wff⟩
    - only free vars in wff should be those in ⟨term⟩

· e.g.

1. $\{x \mid x \in nat \land x \leq 6\}$
   can be written as
   $\{x \bullet x : nat \mid x \leq 6\}$
   which can be shortened to
   $\{x : nat \mid x \leq 6\}$

2. $\{x + y \bullet x, y : nat \mid x \leq 2 \land y \leq 1\}$
   is the set $\{0, 1, 2, 3\}$

· e.g.

1. People who live in Waterloo

   $\{x : People \mid lives(x, Waterloo)\}$

2. The set of numbers each of which is the double of some natural number between 5 and 10 exclusively

   $\{2 * y, y : nat \mid 5 < y \land y < 10\}$

3. Ages of students who attend UW

   $\{age(y) \cdot y : Student \mid attend(y, UW)\}$

· axiom: set comprehension
  ↳ $x \in \{y : S \mid P(y)\} \iff x \in S \land P(x)$
  ↳ $x \in \{f(a,b,..) \cdot a : S, b : R, ... \mid P(a,b,...)\} \iff$
  $\exists a, b, ... \cdot a \in S \land b \in R \land ... \land x = f(a,b,..) \land P(a,b,...)$
  • only x is free var on RHS
  ↳ e.g. $z \in \{x+y \cdot x : nat, y : nat \mid x \leq 2 \land y \leq 1\} \iff$
  $\exists x, y \cdot x \in nat \land y \in nat \land z = x+y \land x \leq 2 \land y \leq 1$

· axiom: empty set
  ↳ $\forall x \cdot \neg (x \in \emptyset)$

· axiom: universal set
  ↳ $\forall x \cdot x \in U$
  ↳ consists of all values of concern in domain
  ↳ w/o types, denoted as U (univ in george)
  ↳ w/ types, can be multiple universal sets
  • e.g. $\{x \cdot nat \mid x \leq 6\}$ is subset of univ set nat

· axiom: set equality
  ↳ $D = B \iff (\forall x \cdot x \in D \iff x \in B)$
  ↳ 2 sets are equal iff they have exactly same elements

axiom: subset
  ↳ $D \subseteq B \iff (\forall x \cdot x \in D \Rightarrow x \in B)$
  ↳ set D is subset of set B iff elements of D are also in B

· axiom: proper subset
  ↳ $D \subset B \iff D \subseteq B \land \neg(D = B)$
  ↳ set D is proper subset of set B iff elements of D are also in B but $D \neq B$

· e.g. Using only $\in, \subseteq, \subset, =, \neg$, formalize:

1. Esmerelda is a duck.

   $Esmerelda \in Ducks$

2. All CS students are students at Waterloo.

   $CS\,Students \subseteq Waterloo\,Students$

3. Not all days in November are holidays.

   $\neg (Days\,Of\,Nov \subseteq Holidays)$

· derived law for relationship btwn set equality & subset pred:
$(D = B) \iff D \subseteq B \wedge B \subseteq D$
for finite sets, size / cardinality of set is # elements in set
↳ for set D, $\#D$ is size (can also use $|D|$)
↳ singleton set consists of 1 elem
· axiom: power set
↳ $\mathbb{P}(D) = \{B \mid B \subseteq D\}$
↳ power set of set is set of all its subsets
↳ for finite sets, $\#(\mathbb{P}(A)) = 2^{\#A}$
set theory summary:

Axioms

[Types as sets] (if $\exists x \bullet x \in B$)
$(\forall x : B \bullet P(x)) \quad \Leftrightarrow \quad (\forall x \bullet x \in B \Rightarrow P(x))$
$(\exists x : B \bullet P(x)) \quad \Leftrightarrow \quad (\exists x \bullet x \in B \wedge P(x))$
[Set Comprehension]
$x \in \{y \bullet y : S \mid P(y)\} \quad \Leftrightarrow \quad x \in S \wedge P(x)$
$x \in \{f(y) \bullet y : S \mid P(y)\} \quad \Leftrightarrow \quad \exists y \bullet y \in S \wedge P(y) \wedge (x = f(y))$

[Empty Set] $\forall x \bullet \neg(x \in \emptyset)$

[Set Equality] $D = B \quad \Leftrightarrow \quad (\forall x \bullet x \in D \Leftrightarrow x \in B)$

[Subset] $D \subseteq B \quad \Leftrightarrow \quad (\forall x \bullet x \in D \Rightarrow x \in B)$
[Proper Subset] $D \subseteq B \quad \Leftrightarrow \quad D \subseteq B \wedge \neg(D = B)$

[Power Set] $\mathbb{P}(D) \quad = \quad \{B \mid B \subseteq D\}$

- e g. prove $\vdash \emptyset \subseteq B$

```
1) for every xg {
    2) assume xg ∈ ∅ {
        3) ¬(xg ∈ ∅)        by set // empty set
        4) xg ∈ B
    }
    5) xg ∈ ∅ ⇒ xg ∈ B      by imp_i on 2-4
}
6) ∀x · x ∈ ∅ ⇒ x ∈ B       by forall_i on 1-5
7) ∅ ⊆ B                    by set on 6 // subset
```

- e.g. prove $(D = B) \Leftrightarrow D \subseteq B \wedge B \subseteq D$

One dir:

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

```
1) assume A = B {
    2) ∀x · x ∈ A ⇔ x ∈ B       by set //set equality
    3) for every xg {
        4) xg ∈ A ⇔ xg ∈ B      by forall_e on 2
        5) xg ∈ A ⇒ xg ∈ B      by iff_e on 4
    }
    6) ∀x · x ∈ A ⇒ x ∈ B       by forall_i on 3-5
    7) A ⊆ B                    by set on 6 // subset
    8) for every xg {
        9) xg ∈ A ⇔ xg ∈ B      by forall_e on 2
        10) xg ∈ B ⇒ xg ∈ A     by iff_e on 9
    }
    11) ∀x · x ∈ B ⇒ x ∈ B      by forall_i on 8-10
    12) B ⊆ A                   by set on 11 // subset
    13) A ⊆ B ∧ B ⊆ A           by and_i on 7, 12
}
14) A = B ⇒ A ⊆ B ∧ B ⊆ A       by imp_i on 1-13
```

- e.g. prove $S \in \mathbb{P}(Q) \Rightarrow (\forall x \cdot x \in S \Rightarrow x \in Q)$

$\vdash S \in \mathbb{P}(Q) \Rightarrow (\forall x \cdot x \in S \Rightarrow x \in Q)$

```
1) assume S ∈ ℙ(Q) {
    2) S ∈ {B | B ⊆ Q}      by set on 1 // power set
    3) S ⊆ Q                by set on 2 // set comp
    4) ∀x · x ∈ S ⇒ x ∈ Q   by set on 3 // subset
}
5) S ∈ ℙ(Q) ⇒ (∀x · x ∈ S ⇒ x ∈ Q)   by imp_i on 1-4
```

- george syntax:

Concept	Symbol	George
Set Membership	$\in$	in
Empty Set	$\emptyset$	empty
Universal Set	U	univ
Set Equality	$=$	=
Subset	$\subseteq$	sube
Proper Subset	$\subset$	sub
Size of a Set	$\#(Q)$	card(Q)
Power Set	$\mathbb{P}(Q)$	pow(Q)
Set Product	$*$	prod (in formulas) * (in typechecking error msgs)

- george does typechecking as such:

  Example: A sube B & A in B

  results in the following error message:

```
1 -- Error: line 3: Type error: for `A in B`,
2  cannot match use of `in`
3          as having type: `(pow(TV0) * pow(TV0)) -> TV1`,
4  with its expected type: `(pow(TV0) * pow(pow(TV0))) -> bool
       `.
5  Mistmatch at `TV0` vs `pow(TV0)`
```

  ↳ type of set A, which contains elem of T1, is pow(T1)
  ↳ A sube (A, B)
  sube(A, B)
  sube : ℙ(TV0) * P(TV0) → bool
  A : ℙ(TV0)
  B : ℙ(TV0)
  A in B
  in(A, B)
  in : TV1 * ℙ(TV1) → bool
  A : TV1
  B : ℙ(TV1) = ℙ(ℙ(TV0))

  mismatched

SET FUNCTIONS AND PROOFS

- axiom: set union
  ↳ $A \cup B = \{x \mid x \in A \vee x \in B\}$
- axiom: set intersection
  ↳ $A \cap B = \{x \mid x \in A \wedge x \in B\}$
- 2 sets are disjoint if intersection is empty
  ↳ i.e. $A \cap B = \emptyset$
- axiom: absolute complement
  ↳ $compl(A) = \{x \mid x \in U \wedge \neg(x \in A)\}$
  ↳ $compl(A) = \{x \mid \neg(x \in A)\}$
- axiom: set diff / relative complement
  ↳ $A - B = \{x \mid x \in A \wedge \neg(x \in B)\}$
  ↳ order of operands matter
- e.g.

For the sets: $U = \{a, b, c, d, e\}, X = \{a, b, c\}, Y = \{a, c, e\}$

1. $X \cup Y$ is $\{a, b, c, e\}$
2. $X \cap Y$ is $\{a, c\}$
3. Is $X \subseteq Y$? No.
4. $X - Y$ is $\{b\}$
5. $Y - X$ is $\{e\}$
6. $compl(X)$ is $\{d, e\}$
7. $compl(Y)$ is $\{b, d\}$

- george syntax:

Concept	Symbol	George
Union	$\cup$	union
Intersection	$\cap$	inter
Complement	$compl(B)$	compl(B)
Difference	$-$	diff
Generalized Union	$\bigcup$	gen_U
Generalized Intersection	$\bigcap$	gen_I
Product	$*$	prod (in formulas)

↳ diff & prod is used to differentiate − & * used on #s

e.g. Formalize the following without using set comprehension.

The following are wff:

1. No one from Sask. has a favourite colour of blue.
   FromSask ∩ FavColBlue = ∅ OR
   FavColBlue ⊆ compl(FromSask)
2. All those who like hockey or basketball and whose favourite colour isn't red are from PEI.
   (LikeHockey ∪ Like Bball) − FavColRed ⊆ fromPEI OR
   (LikeHockey ∪ Like Bball) ∩ compl(FavColRed) ⊆ fromPEI

- union of multiple sets is written ∪J
  ↳ J is set of sets
  ↳ e.g. ∪{s} = s
  ↳ e.g. ∪({s} ∪ t) = s ∪ (∪t)
    - s is set & t is set of sets
- intersection of multiple sets is written ∩J
  ↳ e.g. ∩{s} = s
  ↳ e.g. ∩({s} ∪ t) = s ∩ (∩t)
- e.g.

$$J = \{\{Mon, Tue\}, \{Tue, Wed\}, \{Fri, Tue\}\}$$

$$\bigcup J = \{Mon, Tue, Wed, Fri\}$$

$$\bigcap J = \{Tue\}$$

- 3 styles of TP for set theory (to prove ⇔ or =)
  ↳ style 1: to prove A ⇔ B, prove A ↔ B
    - A & B are wff
    - use axioms & derived laws that are ⇔ as logical laws
    - e.g. ⊢ B = C ⇔ B ⊆ C ∧ C ⊆ B

      B = C ↔ B ⊆ C ∧ C ⊆ B

      B = C

      ↔ ∀x · x∈B ⇔ x∈C by set // set equality

      ↔ ∀x · (x∈B ⇒ x∈C) ∧ (x∈C ⇒ x∈B) by equiv

      ↔ (∀x · x∈B ⇒ x∈C) ∧ (∀x · x∈C ⇒ x∈B) by forall_over_and

      ↔ B⊆C ∧ C⊆B by set // subset * 2

  ↳ style 2: to prove ⊢ A=B, prove A = B ↔ true where A & B are terms rep sets

- transform $A=B$ into $\forall x \cdot x \in A \Leftrightarrow x \in B$ (by set // set equality) & proceed
- another method is to directly transform subterms of A & B until they're syntactically same term
  → i.e. matches axiom $A=A$
- e.g. $\vdash U-(B \cap U) = \text{compl}(B)$

  $U-(B \cap U) = \text{compl}(B) \leftrightarrow$ true

  $U-(B \cap U) = \text{compl}(B)$

  $\leftrightarrow U-(B \cap U) = U-B$ by set // univ set identities

  $\leftrightarrow U-(B \cap U) = U-(B \cap U)$ by set // univ set identities

  $\leftrightarrow$ true by set // set equality derived law

↳ style 3: to prove $A=B$, prove $x \in A \leftrightarrow x \in B$ where A & B are terms rep sets

- shortcut for style 2
- e.g. $\vdash \text{compl}(B \cap C) = \text{compl}(B) \cup \text{compl}(C)$

  $x \in \text{compl}(B \cap C) \leftrightarrow x \in (\text{compl}(B) \cup \text{compl}(C))$

  $x \in \text{compl}(B \cap C)$

  $\leftrightarrow \neg(x \in B \cap C)$ by set // abs compl

  $\leftrightarrow \neg(x \in B \wedge x \in C)$ by set // set intersection

  $\leftrightarrow \neg(x \in B) \vee \neg(x \in C)$ by dm

  $\leftrightarrow x \in \text{compl}(B) \vee x \in \text{compl}(C)$ by set // abs compl x2

  $\leftrightarrow x \in \text{compl}(B) \cup \text{compl}(C)$ by set // union

· in interpretations of set theory, set operations map to their intended meaning on elements in domain

↳ sets are formed from objects in domain

↳ e.g. Provide a counterexample for $\models G \cup (G \cup B) = G$

1. $\textit{Domain} = \{c, d\}$
2. Mapping

Syntax	Meaning
G	$\{c\}$
B	$\{d\}$

$$
\begin{aligned}
& [G \cup (G \cup B) = G] \\
= \; & \{c\} \cup (\{c\} \cup \{d\}) = \{c\} \\
= \; & \{c, d\} = \{c\} \\
= \; & \mathsf{F}
\end{aligned}
$$

· paradox: valid argument leading to contradiction

↳ e.g. consider set whose elements are sets that don't contain themselves: $S = \{X \mid \neg(X \in X)\}$; is $S \in S$?

```
1
2  1) (S in S) | !(S in S) by lem
3  2) case S in S {
4       3) S in {X if !(X in X)} by set on 2 // defn of S
5       4) !(S in S) by set on 3 // set comp
6       5) false by not_e on 2,4
7  }
8  6) case !(S in S) {
9       7) !(S in {X if !(X in X)}) by set on 6 //defn of S
10      8) !!(S in S) by set on 7 // set comp
11      9) false by not_e on 6,8
12 }
13 10) false by cases on 1, 2-5, 6-9
```

- ⊢ false shouldn't be possible unless there's inconsistent premises
- this proof doesn't typecheck in george so S can't exist

· to counteract paradoxes like above, arrange sets in a **hierarchy** where sets at a level can only contain sets at a lower level
  ↳ level 1: individual elements
    - i.e. scalars
    - e.g. 1, 2, ...
  ↳ level 2: sets of individual elements
    - e.g. {1, 2}, {2, 3}, ...
  ↳ level 3: sets of sets of individual elements
    - e.g. {{1, 2}, {2, 3}}, {{1, 2}, {4, 5}}, ...

RELATIONS AND PROOFS

· **tuple**: value containing 2+ components
  ↳ e.g. (Rima, 4) is tuple containing name & age
  ↳ pair has 2 items
  ↳ triple has 3 items
  ↳ order of elements in tuple matters

· **Cartesian product** of 2 sets, C & B, is written as $C * B$; set of all pairs of elements st 1st element belongs to C & 2nd belongs to B
  ↳ $C * B = \{(c, b) \mid c \in C \land b \in B\}$
  ↳ can extend over multiple sets

· a **relation**, R, is subset of Cartesian prod of 2+ sets
  ↳ $R \subseteq C * B$ or $R \in \mathbb{P}(C * B)$
  ↳ can state R is relation from C to B as type
    - $R : C \leftrightarrow B$ or $R : \mathbb{P}(C * B)$
  ↳ binary relation is set of pairs

· relations & multiple arity preds are diff ways of expressing same info
  ↳ e.g. charges (Zellers, CD, 15) is same as (Zellers, CD, 15) ∈ Charges

· all pred/func on sets can be applied to relations
  ↳ e.g.

1. The relation from people to cars in which the people fix cars of a model they own

2. The relation from people to cars in which the people fix cars that they do not own.

3. Everyone who fixes a car owns one of that car.

own : People ↔ Cars
fix : People ↔ Cars
Terms
1) own ∩ fix
2) fix − own
Formula
3) fix ⊆ own

- in binary relation, domain is all 1st elements & range is all 2nd elements in all pairs
- for $R: A \leftrightarrow B$
  - ↳ axiom : domain
    - $dom(R) = \{x: A \mid \exists y: B \cdot (x,y) \in R\}$
  - ↳ axiom: range
    - $ran(R) = \{y: B \mid \exists x: A \ (x,y) \in R\}$
- e.g.
  1. People who own cars.
  2. Cars of people who both own and fix the car.
  3. Everyone who fixes a car fixes at least one car that they don't own.

1) dom(own)
2) ran(fix ∩ own)
3) dom(fix) ⊆ dom(fix - own)

- e.g. ⊢ $dom(S \cup T) = dom(S) \cup dom(T)$

$x \in dom(S \cup T) \leftrightarrow x \in dom(S) \cup dom(T)$

$x \in dom(S \cup T)$
$\leftrightarrow \exists y \cdot (x,y) \in S \cup T$ by set // domain
$\leftrightarrow \exists y \ (x,y) \in S \vee (x,y) \in T$ by set // union
$\leftrightarrow (\exists y \cdot (x,y) \in S) \vee (\exists y \cdot (x,y) \in T)$ by exists_over_or
$\leftrightarrow x \in dom(S) \vee x \in dom(T)$ by set // domain x2
$\leftrightarrow x \in dom(S) \cup dom(T)$ by set // union

- axiom: inverse
  - ↳ $R^{\sim} = \{(b,a) \ a: A, b: B \mid (a,b) \in R\}$
  - ↳ inverse rltn is created by order of elements in all pairs of $R: A \leftrightarrow B$
- axiom: identity
  - ↳ $id(B) = \{(a,a) \mid a \in B\}$
  - ↳ pairing of every elem of set B w/itself
- axiom: relational composition
  - ↳ $R;S = \{(a,c) \ a: A, c: C \mid \exists b \cdot (a,b) \in R \wedge (b,c) \in S\}$
    - $R: A \leftrightarrow B$
    - $S: B \leftrightarrow C$
  - ↳ i.e. R followed by S
  - ↳ can also write as $S \circ R$, which corresponds to function notation
- axiom: relational image
  - ↳ $R(|A1|) = \{y: B \mid \exists x: A \ (x,y) \in R \wedge x \in A1\}$
    - $R: A \leftrightarrow B$
    - $A1 \subseteq A$
  - ↳ e.g. $own = \{(Rima, Honda), (Joe, Honda), (Sarah, BMW)\}$

$own(|\{Rima\}|) = \{Honda\}$

$own(|\{Joe, Sarah\}|) = \{Honda, BMW\}$

· e.g. Show $\mathrm{id}(\mathrm{dom}(R)) = R; R^{\sim}$ where $R : C \leftrightarrow B$ is not valid.

A counterexample is

1. $C = \{c_1, c_2\}$
   $B = \{b\}$
2. Mapping

Syntax	Meaning
$R : C \leftrightarrow B$	$\{(c_1, b), (c_2, b)\}$

$$
\begin{array}{ll}
 & [\mathrm{id}(\mathrm{dom}(R)) = R; R^{\sim}] \\
= & \mathrm{id}(\mathrm{dom}(\{(c_1, b), (c_2, b)\})) = \{(c_1, b), (c_2, b)\}; \{(c_1, b), (c_2, b)\}^{\sim} \\
= & \mathrm{id}(\{c_1, c_2\}) = \{(c_1, b), (c_2, b)\}; \{(c_1, b), (c_2, b)\}^{\sim} \\
= & \mathrm{id}(\{c_1, c_2\}) = \{(c_1, b), (c_2, b)\}; \{(b, c_1), (b, c_2)\} \\
= & \{(c_1, c_1), (c_2, c_2)\} = \{(c_1, c_1), (c_1, c_2), (c_2, c_1), (c_2, c_2)\} \\
= & \mathrm{F}
\end{array}
$$

· axiom: domain restriction
  ↳ $A1 \lhd R = \{(a,b) \cdot a : A, b : B \mid (a,b) \in R \wedge a \in A1\}$
  ↳ retains only pairs whose 1st elems are part of set $A1 \subseteq A$

· axiom: domain subtraction
  ↳ $A1 \lhd R = \{(a,b) \cdot a : A, b : B \mid (a,b) \in R \wedge \neg (a \in A1)\}$
  ↳ retains pairs whose 1st elems are not part of set $A1 \subseteq A$

· axiom: range restriction
  ↳ $R \rhd B1 = \{(a,b) \; a : A, b : B \mid (a,b) \in R \wedge b \in B1\}$
  ↳ retains only pairs whose 2nd elems are part of set $B1 \subseteq B$

· axiom: range subtraction
  ↳ $R \rhd B1 = \{(a,b) \cdot a : A, b : B \mid (a,b) \in R \wedge \neg (b \in B1)\}$
  ↳ retains pairs whose 2nd elems are not part of set $B1 \subseteq B$

· e.g.

$$own = \{(Rima, Honda), (Joe, Honda), (Sarah, BMW)\}$$

$$\{Rima, Joe\} \lhd own = \{(Rima, Honda), (Joe, Honda)\}$$

$$\{Rima\} \lhd own = \{(Joe, Honda), (Sarah, BMW)\}$$

$$own \rhd \{Honda\} = \{(Rima, Honda), (Joe, Honda)\}$$

$$own \rhd \{Honda\} = \{(Sarah, BMW)\}$$

· e.g. $A \lhd (B \lhd R) = (A \cap B) \lhd R$

$(x,y) \in A \lhd (B \lhd R) \leftrightarrow (x,y) \in (A \cap B) \lhd R$

$(x,y) \in A \lhd (B \lhd R)$

$\leftrightarrow (x,y) \in B \lhd R \wedge x \in A$ by set // dom res

$\leftrightarrow (x,y) \in R \wedge x \in B \wedge x \in A$ by set // dom res

$\leftrightarrow (x,y) \in R \wedge x \in A \cap B$ by set // intersection

$\leftrightarrow (x,y) \in (A \cap B) \lhd R$ by set // dom res

· axiom: relational overriding
  ↳ $R \oplus S = ((\mathrm{dom}(S) \lhd R) \cup S$
  $= \{(a,b) \mid (a,b) \in R \wedge a \notin \mathrm{dom}(S) \vee (a,b) \in S\}$
  - $R : A \leftrightarrow B$
  - $S : A \leftrightarrow B$
  ↳ update R by adding in pairs of S to R & any pairs in R w/same 1st elem as pairs in S are eliminated

↳ e.g. $own =$
$\{(Rima, Honda), (Joe, Honda), (Sarah, BMW)\}$

$S = \{(Rima, BMW)\}$

$own \oplus S = \{(Rima, BMW), (Joe, Honda), (Sarah, BMW)\}$

· e.g. Using the following sets,
owns: People $\leftrightarrow$ Dwellings
rents: People $\leftrightarrow$ Dwellings
Students: $\mathbb{P}(People)$
Houses: $\mathbb{P}(Dwellings)$
Formalize these sentences:

1. All houses that are rented are owned.
2. Not every student owns a house.
3. Students who rent houses do not own any dwellings.
4. Everyone who owns a house also rents that house.

Students $\subseteq$ People
Students : $\mathbb{P}$ (People)

1) $\underbrace{ran(rents \rhd Houses)}_{\text{houses that are rented}} \subseteq \underbrace{ran(owns)}_{\substack{\text{dwellings that are owned (don't have to limit to Houses} \\ \text{b/c air limited on LS)}}}$

2) $\neg (Students \subseteq \underbrace{dom(owns \rhd Houses)}_{\text{people who own houses}})$

3) $\underbrace{(dom(Students \lhd rents \rhd Houses))}_{\text{students who rent houses}} \lhd owns = \emptyset$ OR
$(Students \lhd rents \rhd Houses)^{\sim} ; owns = \emptyset$

4) $owns \rhd Houses \subseteq rents$
↳ comparing a pair

· e.g. $\underbrace{R}_{y} (| \underbrace{dom(S)}_{x} |) = dom(\underbrace{R^{\sim}}_{\substack{y \leftrightarrow x \\ y \leftrightarrow z \\ y}} ; S)$

$R : x \leftrightarrow y$, $S : x \leftrightarrow z$
$y \in R (| dom(S) |) \leftrightarrow y \in dom(R^{\sim}; S)$
$y \in R (| dom(S) |)$
$\leftrightarrow \exists x \cdot (x, y) \in R \wedge x \in dom(S)$ by set // rel image
$\leftrightarrow \exists x \cdot (x, y) \in R \wedge (\exists z \cdot (x, z) \in S)$ by set // domain
$\leftrightarrow \exists x \cdot (y, x) \in R^{\sim} \wedge (\exists z \cdot (x, z) \in S)$ by set // inverse
$\leftrightarrow \exists x \cdot \exists z \cdot (y, x) \in R^{\sim} \wedge (x, z) \in S$ by move_exists
$\leftrightarrow \exists z \cdot (\exists x \cdot (y, x) \in R^{\sim} \wedge (x, z) \in S)$ by swap_vars
$\leftrightarrow \exists z \quad (y, z) \in R^{\sim} ; S$ by set // rel comp
$\leftrightarrow y \in dom(R^{\sim} ; S)$ by set // domain

MODULE 6

INTRODUCTION TO Z

- **formal specification**: writing description of system to be built in a language (e.g. logic) where each formula has unique meaning (i.e. unambiguous)
- in system specification, describe what system does & not how it does it
- **Z formal specification language** provides ways to organize info of a specification & shorthands to reuse text
- in Z, execution of system is treated as seq of states
  - ↳ i.e. $state_0 \rightarrow state_1 \rightarrow \ldots$
  - ↳ each state is mapping from variables (i.e. state elems) to values
- in **Z system spec**, we describe:
  - ↳ generic & free types for data manipulated by system
    - optional
  - ↳ constants (i.e. elems not changed by system) & their types
    - optional
  - ↳ state space: system elems, their types, & invariants abt rltnship btwn system elems
    - i.e. set of possible states
  - ↳ initial state
    - optional
  - ↳ operations & how they change elems of system
- **schema** is way of grouping tgt & later reusing related info
  - ↳ e.g.

Example

$$A : nat$$
$$B : nat$$
$$C : nat$$

$$A < B \;;;$$
$$C = A + B \;;;$$

    - above middle line is signature, where elems & their types are introduced
    - below middle line is predicate that contains wffs implicitly conjoined tgt
  - ↳ order of lines in each part of schema doesn't matter
- **state space** schema describes possible states
  - ↳ a state is set of vars w/ their types
  - ↳ vars must change value during some op of system to be listed in schema
  - ↳ pred part lists **invariants**, which are formulas stating what must be true in every state of execution
- can include 1 schema in another by listing its name in signature part of 2nd schema (i.e. **schema inclusion**)
  - ↳ adds signature & preds from included schema
  - ↳ pred part must still be consistent set of formulas
- to describe how a state changes, refer to current value of var in state by its name & value of var in next state by adding prime (') to its name
- to describe operation on state, use **delta schemas**
  - ↳ ΔName Of Schema is equiv to Name Of Schema
    Name Of Schema'

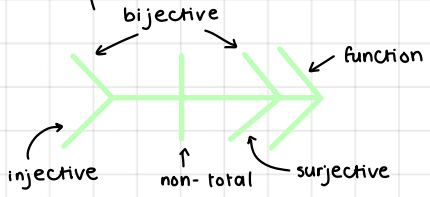
- invariants are implicitly part of postcondition so listed postconditions in op must stay consistent w/them
- p? means p? is input
- q! means q! is output
- local variable is one that appears in signature of op w/o ? or !
  - ↳ equiv to using existentially quantified var in constraints of schema
  - ↳ useful if there's repeated term used in pred part
  - ↳ unlike inputs, outputs, & system vars, local vars are not observable to user
- to describe op, usually have preconditions & postconditions
  - ↳ in pred part of schema
- if var is to remain unchanged in op, must be included in postconditions
  - ↳ e.g. $x' = x$

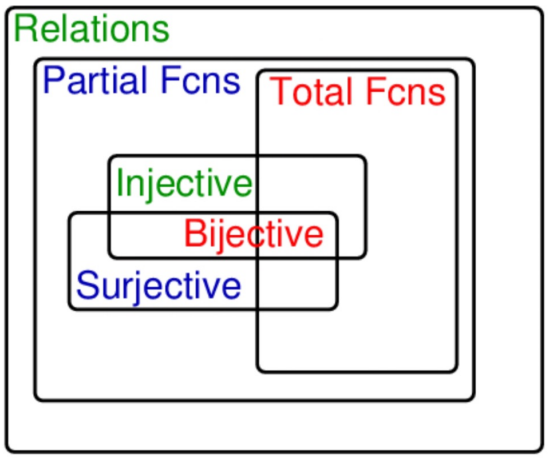
in Z, constants are declared in axiom schema w/no name but in george, we use:

```
1 schema Constants begin
2   MaxBooks:nat
3 pred
4   100 <= MaxBooks;;
5 end
```

  - ↳ constants schema is implicitly included in all other schemas
  - ↳ doesn't need formulas limiting values of constraints
- types in Z:
  - ↳ built-in: nat, int, etc.
  - ↳ generic: types that exist w/o explicitly saying what elems of types are
    - e.g [Flight, Location]
    - aka uninterpreted types b/c they don't have standard interpretation
  - ↳ free (enumerated): new type where we describe its possible values
    - e.g. Colours ::= Red | Blue | Green
      - → values are distinct
  - ↳ compound
- type names can only be within type expressions & can't be used within formulas
- compound types are a relation / function type:

Name	Notation
Relation	$R : \mathbb{P}(A \times B)$ or $R : A \leftrightarrow B$
Non-total Fcn	$f : A \pfun B$
Total Fcn	$f : A \fun B$
Non-total Surjective (onto) Fcn	$f : A \psurj B$
Total Surjective (onto) Fcn	$f : A \surj B$
Non-total Injective (1-to-1) Fcn	$f : A \pinj B$
Total Injective (1-to-1) Fcn	$f : A \inj B$
Non-total Bijective Fcn	$f : A \pbij B$
Total Bijective Fcn	$f : A \bij B$

  - ↳ implicit invariants associated w/ these types
- 



- function is rltn in which each domain elem is associated w/ unique range elem
  - ↳ rltn $f: A \leftrightarrow B$ (i.e. $f \in \mathbb{P}(A * B)$ or $f \subseteq A * B$) is function iff $\forall x: A \cdot \forall y, z: B \cdot (x,y) \in f \wedge (x,z) \in f \Rightarrow y = z$
- partial functions are most general kind of function
  - ↳ for function f from set A to set B, domain of f may include any # of elems in A
  - ↳ if elem of A is not in domain, f is undefined for that elem
- function space is set of functions
- set of non-total functions is $A \nrightarrow B = \{f \mid A \leftrightarrow B \mid \forall x: A; \forall y, z: B \cdot (x,y) \in f \wedge (x,z) \in f \Rightarrow y = z\}$
- total function from A to B is defined for all elems in A
  - ↳ $A \rightarrow B = \{f: A \nrightarrow \mid dom(f) = A\}$
- surjective function from A to B means every elem of B appears at least once as 2nd component of ordered pair in f (i.e. onto)
  - ↳ $\forall b: B \cdot \exists a: A \cdot f(a) = b$
  - ↳ domain must have at least as many elems as range
- injective function from A to B means its inverse is a function (i.e. one-to-one)
  - ↳ an elem of B appears at most once as 2nd component of ordered pair in f
  - ↳ $\forall x, y: A \cdot f(x) = f(y) \Rightarrow x = y$
  - ↳ if f is injective, $f^{\sim}$ is injective function
- bijective function from A & B means it's both surjective & injective
  - ↳ every elem of B appears exactly once as 2nd component of ordered pair in f
- to determine type of arrow:
  - ↳ function?
  - ↳ total?
  - ↳ one-to-one?
  - ↳ onto?
- e.g. $f : x \rightarrowtail\!\!\!\!\!\rightarrow y$
  $f^{\sim} : y \rightarrowtail x$
- e.g. 1) IP Addresses $\leftrightarrow$ Computer Names (relation)
  2) Moons $\rightarrow$ Planets
  3) Students $\leftrightarrow$ Courses (relation)
  4) Country Capital Cities $\twoheadrightarrow$ Countries
  5) Cars $\rightarrowtail$ License Number
- some op on state variables don't change state so we use $\equiv$ schemas instead of Δ schemas
  - ↳ may want to query system or handle exceptions

· e.g. Create a Z specification of the game of musical chairs:

> *The game involves a collection of chairs and players. There is always one less chair than players. In each round the players walk around while music plays and when the music goes off, the players must immediately sit on a chair. The person who doesn't get a chair is eliminated. The winner is the person who gets a chair in the round where there are only two players left.*

[Player, Chair]

Input ::= MusicStarts | MusicStops | EliminatePlayer | DeclareWinner

↳ no constants

MusicalChairs

$pl : \mathbb{P}(Player)$

$ch : \mathbb{P}(Chair)$

$occupied : Chair \rightarrowtail Player$

$\#(ch) = \#(pl) - 1$;;

$dom(occupied) \subseteq ch$;;

$ran(occupied) \subseteq pl$;;

InitialMusicalChairs

MusicalChairs

$\#(pl) > 1$;;

Operations:

MusicStarts

Δ MusicalChairs

in? : Inputs

/* preconditions */

in? = MusicStarts ;;

$\#(pl) > 1$;;

/* postconditions */

$pl' = pl$;;

$occupied' = \emptyset$;;

MusicStops

Δ MusicalChairs

in? : Inputs

/* pre */

in? = MusicStops ;;

$\#(pl) > 1$;;

$occupied = \emptyset$;;

/* post */

$pl' = pl$;;

$dom(occupied') = ch'$;;

EliminatePlayer

Δ MusicalChairs
in? : Inputs
player_out! : Player

/* pre */
in? = EliminatePlayer ;;
#(pl) > 1 ;;
dom(occupied) = ch ;;
/* post */
player_out! ∈ pl - ran(occupied) ;;
pl' = ran(occupied) ;;
/* alt : pl' = pl diff player_out! ;; */
#(ch') = #(ch) - 1 ;;

DeclareWinner

Ξ MusicalChairs
in? : Inputs
winner! : Player

/* pre */
in? = DeclareWinner ;;
#(pl) = 1 ;;
/* post */
winner! ∈ pl ;;

- some ops on state vars don't change state (e.g. query system or handle exceptions) so we use Ξ (xi) schema inclusion
  - ↳ e.g.

Query

Ξ MySystem
out! : *nat*

---

out! = *X* ;;
...

- e.g Create a Z specification for the voting system for the Canadian federal House of Commons. Create operations for:

  1. Add candidates for ridings. The candidate must reside in the riding and be Canadian and at least 18 years of age.
  2. Register voters. Voters must be at least 18 years old, Canadian, and live in the riding. Ridings are determined by the voter's postal code.
  3. Voting. A voter must be registered and can only vote for a candidate in the riding where they reside. A voter can make at most one vote.
  4. Determining the winner(s) for a riding.

Types : [People, CdnPCs, Ridings]
Errors ::= TooYoung | NotCanadian

Constants

$Cdns : \mathbb{P}(People)$

$Age : People \to nat$

$Riding : CdnPCS \twoheadrightarrow Ridings$ /* all ridings have PCs in them */

$PC : People \nrightarrow CdnPCs$ /* not all ppl have PCs & some PCs can have no one */

$MinAge : nat$

$MinAge = 18$;;

VotingSystem

$CandidateToRiding : People \nrightarrow Ridings$

$VotesForCandidate : People \nrightarrow nat$

$RegisteredVoters : Ridings \to \mathbb{P}(People)$

$Voted : \mathbb{P}(People)$

/* candidates have to be in both relations */

$dom(CandidateToRiding) = dom(VotesForCandidate)$;;

/* anyone who's voted has to be registered voter */

$Voted \subseteq \bigcup(ran(RegisteredVoters))$;;

/* candidates have to be Canadian */

$dom(CandidateToRiding) \subseteq Cdns$;;

/* registered voters are Cdn */

$\bigcup(ran(RegisteredVoters)) \subseteq Cdns$;;

/* all voters have to be above minAge */

$\forall x \cdot x \in Voted \Rightarrow Age(x) \geq MinAge$;;

/* all candidates have to be above MinAge */

$\forall x \cdot x \in dom(CandidateToRiding) \Rightarrow Age(x) \geq MinAge$;;

/* candidates must be candidate for riding they live in */

$CandidateToRiding \subseteq (PC ; Riding)$;;

/* voters must vote in riding they live in */

$\{(p,r) \cdot p : People, r : Ridings \mid p \in \bigcup(RegisteredVoters(|\{r\}|))\} \subseteq (PC ; Riding)$;;

/* voter can't be registered in 2 ridings */

$\forall r1, r2 : Ridings \cdot r1 \neq r2 \Rightarrow \bigcup(RegisteredVoters(|\{r1\}|)) \cap \bigcup(RegisteredVoters(|\{r2\}|)) = \emptyset$

↳ simplify RegisteredVoters to be $Ridings \leftrightarrow People$ or $People \nrightarrow Ridings$

InitialVotingSystem

VotingSystem

$CandidateToRiding = \emptyset$;;

$VotesForCandidate = \emptyset$;;

/* all ridings have empty set of registered voters */

$ran(RegisteredVoters) = \{\emptyset\}$;;

$Voted = \emptyset$;;

AddCandidate

ΔVotingSystem
candidate? : People
riding? : Ridings

/* pre */
candidate? $\in$ Cdns ;;
Age(candidate?) $\geq$ MinAge ;;
(candidate?, riding?) $\in$ (PC ; Riding) ;;
$\neg$ (candidate? $\in$ dom (CandidateToRiding)) ;;
/* post */
CandidateToRiding' = CandidateToRiding $\cup$ {(candidate?, riding?)} ;;
VotesForCandidate' = VotesForCandidate $\cup$ {(candidate?, 0)} ;;
RegisteredVoters' = RegisteredVoters ;;
Voted' = Voted ;;

RegisterVoter

ΔVotingSystem
voter? : People
/* riding is local var */
riding : Ridings
currentridingvoters : $\mathbb{P}$(People)

/* pre */
Age(voter?) $\geq$ MinAge ;;
voter? $\in$ Cdns ;;
voter? $\in$ dom (PC) ;;
$\neg$ (voter? $\in$ $\bigcup$ ran (RegisteredVoters)) ;;
(voter?, riding) $\in$ (PC ; Riding) ;;
(riding, currentridingvoters) $\in$ RegisteredVoters ;;
/* post */
RegisteredVoters' = (RegisteredVoters $\oplus$ {(riding, currentridingvoters $\cup$ {voter?})}) ;;
CandidateToRiding' = CandidateToRiding ;;
VotesForCandidate' = VotesForCandidate ;;
Voted' = Voted ;;

↳ RegisteredVoters is non-total so must treat it as relation ? can't write RegisteredVoters(riding)

Vote

Δ VotingSystem
voter? : People
candidate? : People
/* local vars */
riding : Ridings
votesforcandidate : nat

/* pre */
$\neg$ (voter? $\in$ Voted) ;;

voter? ∈ dom(PC);;
voter? ∈ ∪(ran(RegisteredVoters));;
/* candidate is in riding of voter */
∃r : Ridings · (candidate?, r) ∈ CandidateToRiding ∧ voter? ∈
∪(RegisteredVoters(|{r}|));;
(candidate?, riding) ∈ CandidateTo Riding;;
/* voter must have address for pair to exist */
(voter?, riding) ∈ (PC ; Riding);;
(candidate?, votesforcandidate) ∈ VotesForCandidate;;
/* post */
VotesForCandidate' = (VotesForCandidate ⊕ {(candidate?,
votesforcandidate + 1)});;
Voted' = Voted ∪ {(voter?)};;
CandidateToRiding' = CandidateToRiding;;
RegisteredVoters' = RegisteredVoters;;

DetermineWinner

≡ VotingSystem
riding? : Ridings
winners! : ℙ(People)
/* local vars */
candidates : ℙ(People)
max : nat

/* could be tie so could be set of winners */
candidates = dom(CandidateToRiding ▷ {riding ?});;
candidates ≠ ∅;;
/* post */
∀c : People · ∀v : nat · c ∈ candidates ∧ (c, v) ∈ VotesForCandidate ⇒
max ≥ v;;
max ∈ ran(candidates ◁ VotesForCandidate);;
winners! = dom(VotesForCandidates ▷ {max});;

Example exceptions:

RegisterVoterTooYoung

≡ VotingSystem
voter? : People
error! : Errors

/* pre */
Age(voter?) < MinAge;;
/* post */
error! = TooYoung;;

RegisterVoterNotCanadian

$\equiv$ VotingSystem
voter? : People
error! : Errors

/* pre */
$\neg$ (voter? $\in$ Cdns) ;;
/* post */
error! = NotCanadian ;;

- if there's pre-condition for op, describe both primary op (i.e. desired state change when pre-condition is satisfied) & exception handling ops (i.e. system's behaviour when pre-condition isn't met)
- for exception handling, use $\equiv$ schemas b/c don't want system elems to change & instead output error msg
  - ↳ often use free type to describe error msgs instead of char strings
    - e.g. Report ::= NumTooHigh
- if there's multiple exceptions w/ same error msg, group into 1 exception handler
  - ↳ disjunction of negations of all pre-conds must be explicitly included in pre-cond of handler
- to access individual parts of binary rltn:
  - ↳ if tuple is part of total function f, get 2nd elem associated w/ x by f(x)
  - ↳ if tuple is part of rltn R, get set of possible 2nd elems associated w/ x by R(| {x} |)
    - from set, extract arbitrary elem using $y \in R(|\{x\}|)$
  - ↳ if tuple is part of rltn R, get set of possible 1st elems associated w/ y by dom(R $\rhd$ {y}) or $R^{\sim}(|\{y\}|)$
  - ↳ if f is non-total function, might result in empty set

MODULE 7

INTRO TO PROGRAM CORRECTNESS

- in **formal verification**, formally describe in logic the spec of what code is supposed to do & then prove program is correct for its spec using a proof theory
  - ↳ checks program for all possible inputs
- we'll verify imperative, sequential, & transformational programs
  - ↳ imperative program consists of sequence of commands that modify values of vars
  - ↳ sequential program has no concurrency
  - ↳ transformational programs are given inputs, compute outputs, & terminate
- **assignment** command is $V := E$
  - ↳ next state of program is prev state changed by assigning var V the value of expression E evaluated in state before command is executed
  - ↳ e.g. $V_2 := V_1 + V_3$
- **imperative programs** manipulate state of the store (i.e. mem)
  - ↳ state of program is value of vars at particular time in execution of program
  - ↳ e.g.

```
    State: X == ?, Y == ?
X := 2;
    State: X == 2, Y == ?
Y := 1;
    State: X == 2, Y == 1
X := X + 1;
    State: X == 3, Y == 1
Y := Y + X;
    State: X == 3, Y == 4
```

- spec for a program consists of a precondition & postcondition
  - ↳ **precondition** is wff constraining values of program vars in state before program has started
  - ↳ **postcondition** is wff constraining values of program vars in state after program has completed
- in program correctness, we prove arguments of a form called a **triple**:
  - ↳ assert(P);
    C;
    assert(Q);
    - P is precondition
    - C is program
    - Q is postcondition
  - ↳ e.g compute a # stored in var Y, whose square is less than input X
    assert($X > 0$)
    C;
    assert($Y^2 < X$)
- **partial correctness** is written as $\models_{par}$ assert P; C; assert(Q);
  - ↳ i.e. for all program executions of C that start in state satisfying P, if execution of C terminates, then at termination Q is satisfied
  - ↳ if program doesn't terminate, any spec is true
- **total correctness** is written as $\models_{tot}$ assert(P), C; assert(Q);

↳ i.e. for all program states that satisfy P, C is guaranteed to terminate & resulting state satisfies Q

↳ total correctness = partial correctness + termination

- a triple will only refer to partial correctness
- multiple programs can satisfy same spec
  - ↳ e.g.

```
assert(x > 0);
y := 0;
assert(y² < x);
```

```
assert(x > 0);
y := 0;
while (y * y < x) do {
   y := y + 1;
};
y := y - 1;
assert(y² < x);
```

- logical vars don't appear in program
  - ↳ e.g.

```
assert(x = x₀ ∧ x > 0 );
C;
assert(y² < x₀);
```

  - in final state of program, y^2 must be less than value of x in beginning

- e.g.

```
assert(x ≥ 0);
y := 1;
z := 0;
while !(z == x) do {
   z := z + 1;
   y := y * z;
}
assert(y = fact(x));
```

```
assert(x = x₀ ∧ x ≥ 0);
y := 1;
while !(x == 0) do {
   y := y * x;
   x := x - 1;
}
assert(y = fact(x₀));
```

↳ although both programs compute factorial of x, right one consumes x so we need logical var

- we'll use Floyd-Hoare Logic proof theory for program correctness
  - ↳ usually sound but often incomplete
  - ↳ in proofs, we'll use ND & some set theory

ASN AND IMPLIED

- proof rule for assignment command is axiom
  - ↳ no premises
  - ↳ format:

$$\frac{}{\texttt{assert}(P\ [E/Var]);\ Var := E;\ \texttt{assert}(P);}\ \text{Asn}$$

  - Var is variable
  - E is expression
    - → i.e. terms in pred logic
  - P[E/Var] is substitution & must obey sub rules
- in program correctness, often work backward from postcondition
  - ↳ for assignment precondition, replace all LHS occurrences w/ RHS in asn command

- e.g. assert (z = y)
  x := 2,
  assert (x = y) by asn;
- e.g. assert (2 > 0)
  x := 2;
  assert (x > 0) by asn;
- e.g. assert (x+1 = 2);
  x := x + 1,
  assert (x = 2) by asn;
- e.g. assert (0 < x+1 ∧ x+1 < 10)
  x := x+1;
  assert (0 < x ∧ x < 10) by asn;
- e.g. assert (z = x+y+1-4);
  x := x + y + 1,
  assert (z = x - 4) by asn;
- composition proof rule allows us to chain tgt steps in a proof
  - ↳ format:

$$\frac{\begin{array}{ll}\textbf{assert}(P); & \textbf{assert}(Q);\\ C1\ ; & C2\ ;\\ \textbf{assert}(Q); & \textbf{assert}(R);\end{array}}{\begin{array}{l}\textbf{assert}(P);\\ C1;\\ C2\ ;\\ \textbf{assert}(R);\end{array}}\ \text{Composition}$$

  - ↳ proofs use it implicitly so don't have to keep repeating same lines
  - ↳ e.g. assert (x ≥ 5 ∧ y ≥ 0);
    assert (x + y ≥ 5) by implied on VC1;
    z := x;
    assert (z + y ≥ 5) by asn;
    z := z + y;
    assert (z ≥ 5) by asn;
    u := z;
    assert (u ≥ 5) by asn;
    VC1: x ≥ 5 ∧ y ≥ 0 ⊢ x + y = 5
    * ND proof *

    (proof step: assert (x + y ≥ 5) by implied on VC1; z := x;
    proof step: assert (z + y ≥ 5) by asn; z := z + y;
    proof step: assert (z ≥ 5) by asn; u := z; assert (u ≥ 5) by asn;
    composition: the three proof steps above)
- annotated program has a precondition, postcondition, ? assertions (wffs in pred logic) btwn almost all commands of program
  - ↳ every assertion except precondition requires justification
    - no line #s needed
- implied proof rule allows us to have 2 assertions at same point in program
  - ↳ format: $P' \vdash P$

$$\frac{\begin{array}{l}\textbf{assert}(P);\\ C;\\ \textbf{assert}(Q);\\ \\ Q \vdash Q'\end{array}}{\begin{array}{l}\textbf{assert}(P');\\ C;\\ \textbf{assert}(Q');\end{array}}\ \text{Implied}$$

- $P' \vdash P$ & $Q \vdash Q'$ are verification conditions (VCs) that must be proven separately

↳ e.g.

```
assert(X ≥ 5 ∧ Y ≥ 0);
assert(X + Y ≥ 5) by implied on VC1;
Z := X;
assert(Z + Y ≥ 5) by asn;
Z := Z + Y;
assert(Z ≥ 5) by asn;
U := Z
assert(U ≥ 5) by asn;
```

↓ can change to "using arith" & skip VC b/c it's only a 2-line proof

VC 1:
$X \geq 5 \wedge Y \geq 0 \vdash X + Y \geq 5$
1) $X \geq 5 \wedge Y \geq 0$ premise
2) $X + Y \geq 5$ by arith on 1

↳ 1st assertion must imply 2nd one

↳ 2 common forms of implied proof rule:

$$\frac{P' \vdash P \qquad \texttt{assert(P); C; assert(Q);}}{\texttt{assert(P'); C; assert(Q);}}\ \text{Implied} \qquad \frac{Q \vdash Q' \qquad \texttt{assert(P); C; assert(Q);}}{\texttt{assert(P); C; assert(Q');}}\ \text{Implied}$$

- program correctness proofs consists of 2 parts:
  - ↳ annotated program
  - ↳ proofs of VCs
- in $A \vdash B$ or $\vdash A \Rightarrow B$, A is stronger than B
- precondition strengthening is when we know $P' \vdash P$ & assert(P); C; assert(Q), & conclude assert(P'); C; assert(Q)
  - ↳ assuming more in precondition than we need to prove program satisfies postcondition
    - i.e. set of states where P' is true is smaller than set for P
- postcondition weakening is when we know $Q \vdash Q'$ & assert(P); C; assert(Q), & conclude assert(P); C; assert(Q')
  - ↳ concluding less in postcondition than what's acc true at end of program
    - i.e. set of states where Q is true is smaller than set for Q'
- notes when using implied proof rule:
  - ↳ sometimes 2 assertions are logically equiv but only need 1 dir of implication
  - ↳ if VC only requires 2-line proof, can write "by implied using nd_proof_rule"
- for asn rule subs, can skip writing $\vdash$ & double negations

- e.g.

```
assert(true)
assert(2*(X+2)-2*X=4) by implied using arith;
K := X;
assert(2*(X+2)-2*K=4) by asn;
X := X+2;
assert(2*X-2*K=4) by asn;
```

↑ trivial proof so can just say "by arith"

```
X := 2 * X;
assert (X - 2 * K = 4) by asn;
Y := X - 2 * K
assert (Y = 4)  by asn;
```

· e.g.

Recall defn of fib:

$fib(0) = 0$
$fib(1) = 1$
$fib(i+1) = fib(i) + fib(i-1)$ for $i \geq 1$

The following code would be part of loop to calculate the kth Fibonacci number.

proc main() {
  assert$(f = fib(k) \wedge g = fib(k-1) \wedge k \geq 1)$;
  $t := g$;
  $g := f$;
  $f := f + t$;
  $k := k + 1$;
  assert$(f = fib(k) \wedge g = fib(k-1))$ by asn ;
}

```
proc main() {
  assert (f = fib(k) ∧ g = fib(k-1) ∧ k ≥ 1) ;
  assert (f+g = fib(k+1) ∧ f = fib(k)) by implied on VC1;
  t := g;
  assert (f+t = fib(k+1) ∧ f = fib(k)) by asn;
  g := f;
  assert (f + t = fib(k+1) ∧ g = fib(k)) by asn;
  f := f+t;
  assert (f = fib(k+1) ∧ g = fib(k)) by asn;
  k := k + 1;
  assert (f = fib(k) ∧ g = fib(k-1)) by asn;
}
```

VC1: $f = fib(k) \wedge g = fib(k-1) \wedge k \geq 1 \vdash f+g = fib(k+1) \wedge f = fib(k)$

1) $f = fib(k) \wedge g = fib(k-1) \wedge k \geq 1$ premise
2) $f = fib(k)$ by and_e on 1
3) $g = fib(k-1)$ by and_e on 1
4) $k \geq 1$ by and_e on 1
5) $fib(k) + fib(k-1) = fib(k+1)$ by arith on 4 /* defn of fib */
6) $f + fib(k-1) = fib(k+1)$ by eq_e on 2, 5
7) $f+g = fib(k+1)$ by eq_e on 3, 6
8) $f+g = fib(k+1) \wedge f = fib(k)$ by and_i on 2, 7

IF-THEN-ELSE

if-then proof rule (i.e. one-armed conditional) format:

$$\frac{\textbf{assert}(P \wedge B);\ C;\ \textbf{assert}(Q)\ ; \quad P \wedge \neg B \vdash Q}{\textbf{assert}(P);\quad \text{if B then } \{\ C;\ \};\quad \textbf{assert}(Q)\ ;}\ \text{if\_then}$$

↳ generates VC $P \wedge \neg B \vdash Q$

- format for if-then in annotated program:

```
assert(P);
if B then {
   assert(P ∧ B)    by if_then;
   C;
   assert(Q)    by Proof Rule;
};
assert(Q)    by if_then on VCi;
```

VC i: $P \wedge \neg B \vdash Q$

e.g.
```
assert(true)        ← guard
if (max < B) then {
   assert (max < B)   by if_then;
   assert (B ≥ B)     by implied using arith
   max := B;
   assert (max ≥ B)   by asn;
};
assert(max ≥ B)   by if_then on VC1;
```

VC1 : ¬(max < B) ⊢ max ≥ B by arith

↳ need to see VC for if_then rule written out b/c statement of VC isn't obvious

- in if-then rule, if precondition & guard are inconsistent, then program nvr enters conditional block
  ↳ false implies anything so proof can still be done
- format for if-then-else (i.e. two-armed conditional):

```
assert(P)
if B then {
   assert(P ∧ B)    by if_then_else;
   C1;
   assert(Q)    by Proof_Rule;
} else {
   assert(P ∧ ¬B)    by if_then_else;
   C2;
   assert(Q)    by Proof_Rule;
};
assert(Q)    by if_then_else;
```

↳ no VCs required

- e.g.
```
assert(true);
assert (??[x+1/a])
a := x + 1;
assert (??)    by asn;
if ( a − 1 = 0 ) then {
   assert(?? ∧ (a-1=0))   by if_then_else;
   assert (1 = x+1)        by implied on VC1;
   y := 1;
   assert (y = x+1)    by asn;
} else {
   assert(?? ∧ ¬(a-1=0))  by if_then_else;
   assert (a = x+1)
   y := a;
   assert (y = x+1)    by asn,
};
assert(y = x + 1)   by if_then_else;
```

assert(true);
assert (x+1 = x+1) by implied using eq_i;
a := x + 1;
assert (a = x+1) by asn;
if (a − 1 = 0) then {
  assert((a = x+1) ∧ (a − 1 = 0)) by if_then_else;
  assert (1 = x+1) by implied on VC1;
  y := 1;
  assert (y = x+1) by asn;
} else {
  assert((a = x+1) ∧ ¬(a−1 = 0)) by if_then_else;
  assert (a = x+1) by implied using and_e;
  y := a;
  assert (y = x+1) by asn,
};
assert(y = x + 1) by if_then_else;

VC1 : (a = x+1) ∧ (a−1 = 0) ⊢ 1 = x+1

1) (a = x+1) ∧ (a−1 = 0) premise
2) a = x+1 by and_e on 1
3) a − 1 = 0 by and_e on 1
4) a = 1 by arith on 3
5) 1 = x+1 by eq_e on 2, 4

· derived if-then-else format allows us to work backwards when annotating program:

assert((B ⇒ P1) ∧ (¬B ⇒ P2));
if B then {
  assert(P1) by if_then_else;
  C1;
  assert(Q) by Proof_Rule;
} else {
  assert(P2) by if_then_else;
  C2;
  assert(Q) by Proof_Rule;
};
assert(Q) by if_then_else;

e.g.
assert(X ≥ 5);
assert (X − 2 < 4 ⇒ X ≥ 5) by implied on VC1;
Y := X;
assert(X − 2 < 4 ⇒ Y ≥ 5) by asn;
X := X − 2;
assert(X < 4 ⇒ Y ≥ 5) by asn;
assert((X ≥ 4 ⇒ X ≥ 1) ∧ (¬(X ≥ 4) ⇒ Y ≥ 5)) by implied on VC2;

if (X ≥ 4) then {
  assert (X ≥ 1) by if_then_else;
  assert (X+1 ≥ 2) by implied using arith;
  Y := 1;
  assert (X+Y ≥ 2) by asn;
} else {
  assert (Y ≥ 5) by if_then_else;
  assert (0 − 3 + Y ≥ 2) by implied using arith;
  X := 0 − 3;
  assert (X+Y ≥ 2) by asn;
};
assert(X + Y ≥ 2) by if_then_else;

VC1: $X \geq 5 \vdash X - 2 < 4 \Rightarrow X \geq 5$

VC2: $X < 4 \Rightarrow Y \geq 5 \vdash ((X \geq 4 \Rightarrow X \geq 1) \wedge (\neg(X \geq 4) \Rightarrow Y \geq 5))$

WHILE

- while proof rule format:

```
assert(Inv);
while B do {
   assert(Inv ∧ B)    by partial_while;
   C;
   assert(Inv)    by Proof Rule;
};
assert(Inv ∧ ¬B)    by partial_while;
```

  - ↳ called partial_while b/c it doesn't require termination
  - ↳ Inv is loop invariant
    - wff / assertion that's true before & after each execution of loop's body C
    - if C is sequence of commands, loop invariant doesn't have to be true continuously thru execution of C
    - there isn't only 1 invariant but try to find smallest formula that works
      - → e.g. true is inv of every loop
- have to find Inv st:

```
assert(P);
assert(Inv)    by implied on VCi;
while B do {
   assert(Inv ∧ B)    by partial_while;
   C;
   assert(Inv)    Proof Rule;
};
assert(Inv ∧ ¬B)    by partial_while;
assert(Q)    by implied on VCj;
```

Loop Invariant: *Inv*

VC i: $P \vdash Inv$

VC j: $Inv \wedge \neg B \vdash Q$

  - ↳ Inv ∧ ¬ B ⊢ Q
    - inv is strong enough
  - ↳ assert (Inv ∧ B); C; assert (Inv);
    - it's acc an inv
  - ↳ P ⊢ Inv
    - inv isn't stronger than P
- Inv is usually assertion that intended computation of loop is correct up to current step in execution
  - ↳ usually expresses rltnship btwn vars manipulated by body of loop
- method for choosing loop inv:
  1) determine candidate
     - trace behaviour of loop for particular var values
     - look at postcondition for hints
     - for nested loops, start w/ outer

- inv may include info that value in program doesn't change throughout loop

2) annotate program

3) try to prove VCs
   - if can't prove, strengthen loop inv (i.e. add to it)

- e.g.

```
assert(true);

y := 1;

z := 0;
assert(??)
while ¬(z = x) do {
    assert(?? ∧ ¬(z = x)) by partial_while;

    z := z + 1;

    y := z * y;
    assert(??)
};
assert(?? ∧ z = x) by partial_while;
assert(y = fact(x))
```

↳ trace behaviour of loop:

For X = 4 (X doesn't change):

z	y	¬(z = x)
0	1	true
1	1	true
2	2 · 1	true
3	3 · 2 · 1	true
4	4 · 3 · 2 · 1	false

Inv: $y = fact(z)$

```
assert(true);
assert (1 = fact(0)) by implied using arith;
y := 1;
assert (y = fact(0)) by asn;
z := 0;
assert (y = fact(z)) by asn;
while ¬(z = x) do {
    assert (y = fact(z) ∧ ¬(z = x)) by partial_while;
    assert ((z+1) * y = fact(z+1)) by implied on VC1;
    z := z + 1;
    assert (z * y = fact(z)) by asn;
    y := z * y;
    assert(y = fact(z)) by asn;
};
assert(y = fact(z) ∧ z = x) by partial_while;
assert(y = fact(x)) by implied on VC2;
```

Loop inv: $y = fact(z)$

VC2: $y = fact(z) \wedge z = x \vdash y = fact(x)$

1) $y = fact(z) \wedge z = x$ premise
2) $y = fact(z)$ by and_e on 1
3) $z = x$ by and_e on 1

4) $y = fact(z)$ by eq_e on 2, 3

VC1: $y = fact(z) \wedge \neg(z = x) \vdash (z+1) * y = fact(z+1)$

1) $y = fact(z) \wedge \neg(z = x)$ premise
2) $y = fact(z)$ by and_e on 1
3) $(z+1) * y = (z+1) * fact(z)$ by arith on 2
4) $(z+1) * y = fact(z+1)$ by arith on 3 /* defn of fact */

e.g.

```
assert(0 ≤ n ∧ a > 0);

s := 1;

i := 0;
assert(??)
while (i < n) do {
   assert(?? ∧ i < n) by partial_while;

   s := s * a;

   i := i + 1;
   assert(??)
};
assert(?? ∧ ¬(i<n)) by partial_while;
assert(s = a^n)
```

↳ trace behaviour of loop:

a = 2, n = 3:

s	i	i < n
1	0	true
1 x 2	1	true
1x2x2	2	true
1x2x2x2	3	false

Inv: $s = a^i$

```
assert(0 ≤ n ∧ a > 0);

s := 1;

i := 0;

while (i < n) do {


   s := s * a;

   i := i + 1;

};
assert(s = a^i ∧ ¬(i<n)) by partial_while;
assert(s = a^n) by implied on VC3;
```

VC 3: $s = a^i \wedge \neg(i < n) \vdash s = a^n$

P) $s = a^i \wedge \neg(i < n)$ premise
1) $s = a^i$ by and_e on P
2) $\neg(i < n)$ by and_e on P
3) $i = n$ by ???

add to loop invariant

Inv: $s = a^i \wedge i \leq n$

VC 3: $s = a^i \wedge i \leq n \wedge \neg(i < n) \vdash s = a^n$

P) $s = a^i \wedge i \leq n \wedge \neg(i < n)$ premise
1) $s = a^i$ by and_e on P
2) $\neg(i < n)$ by and_e on P
3) $i \leq n$ by and_e on P
4) $i = n$ by arith on 2, 3
5) $s = a^n$ by eq_e on 1, 4

```
assert(0 ≤ n ∧ a > 0);
assert (1 = a^0 ∧ 0 ≤ n) by implied on VC1;
s := 1;
assert (s = a^0 ∧ 0 ≤ n) by asn;
i := 0;
assert (s = a^i ∧ i ≤ n) by asn;
while (i < n) do {
    assert (s = a^i ∧ i ≤ n) by partial_while;
    assert (s * a = a^(i+1) ∧ i+1 ≤ n) by implied on VC2;
    s := s * a;
    assert (s = a^(i+1) ∧ i+1 ≤ n) by asn;
    i := i + 1;
    assert (s = a^i ∧ i ≤ n) by asn;
};
assert (s = a^i ∧ i ≤ n ∧ ¬(i < n)) by partial_while;
assert(s = a^n) by implied on VC3;
```

Inv: $s = a^i \wedge i \le n$

VC2: $s = a^i \wedge i \le n \vdash s * a = a^{i+1} \wedge i+1 \le n$

1) $s = a^i \wedge i \le n$ premise
2) $s = a^i$ by and_e on 1
3) $i \le n$ by and_e on 1
4) $s * a = a^i * a$ by arith on 2
5) $s * a = a^{i+1}$ by arith on 4 /* defn of exponent */
6) $i+1 \le n$ by arith on 3
7) $s * a = a^{i+1} \wedge i+1 \le n$ by and_i on 5, 6

VC1: $0 \le n \wedge a > 0 \vdash 1 = a^0 \wedge 0 \le n$

1) $0 \le n \wedge a > 0$ premise
2) $0 \le n$ by and_e on 1
3) $1 = a^0$ by arith /* defn of exponent */
4) $1 = a^0 \wedge 0 \le n$ by and_i on 2, 3

e.g.

```
assert(x ≥ 0 ∧ x = x0);

y := 0;
assert(??)
while ¬(x = 0) do {
    assert(?? ∧ ¬(x = 0)) by partial_while;

    y := y + 1;

    x := x - 1;
    assert(??)
};
assert(?? ∧ x = 0) by partial_while;
assert(y = x0)
```

↳ trace behaviour: $x = 2$

y	x	¬(x = 0)
0	2	true
1	1	true
2	0	false

Inv: $x_0 = y + x$

```
assert(x ≥ 0 ∧ x = x0);
assert( x0 = 0 + x ) by implied on VC1;
y := 0;
assert( x0 = y + x ) by asn;
while ¬(x = 0) do {
    assert( x0 = y + x ∧ ¬(x = 0)) by partial_while;
    assert( x0 = y + 1 + x - 1 ) by implied on VC2;
    y := y + 1;
    assert ( x0 = y + x - 1 ) by asn;
    x := x - 1;
    assert( x0 = y + x) by asn;
};
assert( x0 = y + x ∧ x = 0) by partial_while;
assert(y = x0) by implied on VC3;
```

VC3: $x_0 = y + x \wedge x = 0 \vdash y = x_0$

1) $x_0 = y + x \wedge x = 0$	premise
2) $x_0 = y + x$	by and_e on 1
3) $x = 0$	by and_e on 1
4) $x_0 = y + 0$	by eq_e on 2,3
5) $y = x_0$	by arith on 4

VC2: $x_0 = y + x \wedge \neg(x = 0) \vdash x_0 = y + 1 + x - 1$

1) $x_0 = y + x \wedge \neg(x = 0)$	premise
2) $x_0 = y + x$	by and_e on 1
3) $x_0 = y + 1 + x - 1$	by arith on 2

VC1: $x \geq 0 \wedge x = x_0 \vdash x_0 = 0 + x$

1) $x \geq 0 \wedge x = x_0$	premise
2) $x = x_0$	by and_e on 1
3) $x_0 = 0 + x$	by arith on 3

- if program doesn't satisfy its spec, counterexample must include:
  - ↳ initial state: values for program & logical vars before program executes
  - ↳ final state: values for program & logical vars that result from executing program in initial state
  - ↳ demo that precond is satisfied by initial state
  - ↳ demo that postcond is not satisfied by initial state
- e.g. find counterexample:

```
assert(true);
a := x + 1;
if (a - 1 = 0 ) {
    y := 1;
} else {
    y := a + 1;
}
assert(y = x + 1);
```

↳ initial state:
a = doesn't matter
x = 1
y = doesn't matter
↳ final state:
a = 2, y = 3, x = 1
↳ precond:
[true] = T
↳ postcond:
[y = x + 1]
= (3 = 1 + 1)
= F

ARRAYS

- array is function mapping indices to values
  - ↳ e.g. Indices = {i · i : nat | 0 ≤ i < arraysize}
    B : Indices → Value is arr
    - function is total b/c it has value for every index value even if it isn't initialized
- commands can use arrays in 2 ways:
  - ↳ RHS of asgnt or condition : ref to value in arr
  - ↳ LHS of asgnt : assigns value to elmt in arr

if arr is used on RHS of asgnt or condition, can use pre-existing proof rules

- e.g.

```
assert(true);
if (A[4] > 4) then {

    assert (A[4] > 4) by if_then_else;
    assert (4 ≤ 4) by implied using arith;
    X := 4;
    assert (X ≤ 4) by asn;

} else {

    assert (¬(A[4] > 4)) by if_then_else;
    assert (A[4] ≤ 4) by implied using arith;
    X := A[4];
    assert (X ≤ 4) by asn;

};
assert(X ≤ 4) by if_then_else;
```

- if arr is used on LHS of asgnt, it doesn't satisfy previously given asgnt axiom
  - ↳ if elmt index is var, value of var determines which elmt is assigned a value
  - ↳ e.g. assert (A[y] = 0)
    A[x] := 1;
    assert (A[y] = 0) by asn; ⟵ won't be true if x is y

- arr asgnt provides pair that overrides prev asgnts

↳ use rltnal overriding defn: $B \oplus S = ((\text{dom}(S)) \lhd B) \cup S$

↳ when B is function & S contains 1 pair, defn of rltnal overriding simplifies to

$$(B \oplus \{(i,e)\})[k] = \begin{cases} e & \text{if } i=k \\ B[k] & \text{if } \neg(i=k) \end{cases}$$

array assignment rule format:

assert$(P[B \oplus \{(i,e)\}/B])$;
B[i] := e;
assert(P) by array_asn;

- e.g. assert $(B \oplus \{(k,14)\})[4] = c)$;
  B[k] := 14;
  assert (B[4] = c) by array_asn;
  ↳ this works b/c: $B \oplus \{(k,14)\})[4] =$
  if k=4 then 14
  else B[4]
  - CASE 1: k=4
    assert (14 = c);
    B[4] := 14;
    assert (B[4] = c);
  - CASE 2: $k \neq 4$
    assert (B[4] = c);
    B[k] := 14;
    assert (B[4] = c);
- e.g. assert $((A \oplus \{(i, i+1)\})[j+1] = c)$;
  A[i] := i + 1;
  assert (A[j+1] = c) by array_asn;
  ↳ CASE 1: i = j+1
  assert (i+1 = c);
  A[i] := i+1;
  assert (A[j+1] = c);
  ↳ CASE 2: $i \neq j+1$
  assert (A[j+1] = c);
  A[i] := i+1;
  assert (A[j+1] = c);
- e.g. program shifts elmts in arr down by 1 location

assert $((\forall i \cdot 0 \le i \wedge i < n \Rightarrow B[i] = B0[i]) \wedge 0 < n)$;
assert (Inv0) by implied on VC1;
k := 0;

assert (Inv) by asn;
while (k < n-1) do {
  assert (Inv ∧ k<n-1) by partial_while;
  assert (InvBK) by implied on VC2;
  k := k + 1;
  assert (InvB) by asn;
  B[k-1] := B[k];
  assert (Inv) by array_asn;
};
assert (Inv ∧ ¬(k<n-1)) by partial_while;
assert $(\forall i \cdot 0 \le i \wedge i < n-1 \Rightarrow B[i] = B0[i+1])$ by implied on VC3;

Trace loop: n=4

k	B	k < 3
0	[4, 7, 16, 3]	T

1	[7, 7, 16, 3]	T
2	[7, 16, 16, 3]	T
3	[7, 16, 3, 3]	F

Loop inv:

$Inv = (\forall i \cdot 0 \le i \wedge i < k \Rightarrow B[i] = BO[i+1]) \wedge (\forall i \cdot k \le i \wedge i < n \Rightarrow B[i] = BO[i])$

accounts for all $0 \le i < k$ (has value of next elmt in arr)

accounts for all $k \le i < n$ (unchanged value from original arr)

other labels:

↳ $InvB = Inv[B \oplus \{(k-1, B[k])\}/B]$

↳ $InvBK = Inv[B \oplus \{(k-1, B[k])\}/B][k+1/k]$

↳ $Inv\ 0 = Inv[0/k]$

VC1:

$(\forall i \cdot 0 \le i \wedge i < n \Rightarrow B[i] = BO[i]) \wedge 0 < n$

$\vdash$

$(\forall i\ 0 \le i \wedge i < 0 \Rightarrow B[i] = BO[i+1]) \wedge (\forall i \cdot 0 \le i \wedge i < n \Rightarrow B[i] = BO[i])$

1) $(\forall i \cdot 0 \le i \wedge i < n \Rightarrow B[i] = BO[i]) \wedge 0 < n$ premise
2) $\forall i \cdot 0 \le i \wedge i < n \Rightarrow B[i] = BO[i]$ by and_e on 1
3) $\forall i \cdot 0 \le i \wedge i < 0 \Rightarrow B[i] = BO[i]$ by arith /* vacuously true */
4) $(\forall i\ 0 \le i \wedge i < 0 \Rightarrow B[i] = BO[i+1]) \wedge (\forall i \cdot 0 < i \wedge i < n \Rightarrow B[i] = BO[i])$ by and_i on 2, 3

VC2:

$(\forall i \cdot 0 \le i \wedge i < k \Rightarrow B[i] = BO[i+1]) \wedge (\forall i \cdot k \le i \wedge i < n \Rightarrow B[i] = BO[i]) \wedge (k < n-1)$

$\vdash$

$(\forall i \cdot 0 \le i \wedge i < k+1 \Rightarrow (B \oplus \{(k, B[k+1])\})[i] = BO[i+1]) \wedge (\forall i \cdot k+1 \le i \wedge i < n \Rightarrow (B \oplus \{(k, B[k+1])\})[i] = BO[i])$

1) $(\forall i \cdot 0 \le i \wedge i < k \Rightarrow B[i] = BO[i+1]) \wedge (\forall i \cdot k \le i \wedge i < n \Rightarrow B[i] = BO[i]) \wedge (k < n-1)$ premise
2) $\forall i \cdot 0 \le i \wedge i < k \Rightarrow B[i] = BO[i+1]$ by and_e on 1
3) $\forall i \cdot k \le i \wedge i < n \Rightarrow B[i] = BO[i]$ by and_e on 1
4) $k < n-1$ by and_e on 1
5) for every ig {
   6) assume $0 \le ig \wedge ig < k+1$ {
      7) $0 \le ig \wedge ig < k \vee ig = k$ by arith on 6
      8) case $0 \le ig \wedge ig < k$ {
         9) $\neg(ig = k)$ by arith on 8
         10) $(B \oplus \{(k, B[k+1])\})[ig] = B[ig]$ by set on 9 /*override*/
         11) $0 \le ig \wedge ig < k \Rightarrow B[ig] = BO[ig+1]$ by forall_e on 2
         12) $B[ig] = BO[ig+1]$ by imp_e on 6, 11
         13) $(B \oplus \{(k, B[k+1])\})[ig] = BO[ig+1]$ by eq_e on 10, 12
      }
      14) case $ig = k$ {
         15) $(B \oplus \{(k, B[k+1])\})[ig] = B[k+1]$ by set on 14 /*override*/
         16) $k \le k+1 \wedge k+1 < n \Rightarrow B[k+1] = BO[k+1]$ by forall_e on 3
         17) $k \le k+1$ by arith
         18) $k+1 < n$ by arith on 4
         19) $k \le k+1 \wedge k+1 < n$ by and_e on 17, 18
         20) $B[k+1] = BO[k+1]$ by imp_e on 16, 19

21) (B⊕{(k, B[k+1]})[ig] = BO[k+1] by eq_e on 15, 20
22) (B⊕{(k, B[k+1]})[ig] = BO[ig+1] by eq_e on 14, 21
}
23) (B⊕{(k, B[k+1]})[ig] = BO[ig+1] by cases on 7, 8-13, 14-22
}
24) 0≤ig ∧ ig<k+1 ⇒ (B⊕{(k, B[k+1]})[ig] = BO[ig+1] by imp_i on 6-23
}
25) ∀i · 0≤i ∧ i<k+1 ⇒ (B⊕{(k , B[k+1])})[i] = BO[i+1] by forall_i on 5-24
26) for every ig {
27) assume k+1≤ig ∧ ig<n {
28) k+1≤ig by and_e on 27
29) ig<n by and_e on 27
30) ¬(ig = k) by arith on 28
31) (B⊕{(k, B[k+1])})[ig] = B[ig] by set on 30 /*override*/
32) k≤ig ∧ ig<n ⇒ B[ig] = BO[ig] by forall_e on 3
33) k≤ig by arith on 28
34) k≤ig ∧ ig<n by and_i on 29, 33
35) B[ig] = BO[ig] by imp_e on 32, 34
36) (B⊕{(k, B[k+1])})[ig] = BO[ig] by eq_e on 32, 35
}
37) k+1≤ig ∧ ig<n ⇒ (B⊕{(k, B[k+1])})[ig] = BO[ig] by imp_i on 27-36
}
38) ∀i k+1≤i ∧ i<n ⇒ (B⊕{(k, B[k+1])})[i] = BO[i] by forall_i on 26-37
39) (∀i · 0≤i ∧ i<k+1 ⇒ (B⊕{(k , B[k+1])})[i] = BO[i+1]) ∧ (∀i · k+1≤i ∧ i<
⇒ (B⊕{(k, B[k+1])})[i] = BO[i]) by and_i on 25, 38

VC3:

(∀i · 0≤i ∧ i<k ⇒ B[i] = BO[i+1]) ∧ (∀i · k≤i ∧ i<n ⇒ B[i] = BO[i]) ∧
(k<n-1)
⊢
∀i · 0≤i ∧ i<n-1 ⇒ B[i] = BO[i+1]

1) (∀i · 0≤i ∧ i<k ⇒ B[i] = BO[i+1]) ∧ (∀i · k≤i ∧ i<n ⇒ B[i] = BO[i]) ∧
¬(k<n-1) premise
2) ∀i 0≤i ∧ i<k ⇒ B[i] = BO[i+1] by and_e on 1
3) ∀i · k≤i ∧ i<n ⇒ B[i] = BO[i] by and_e on 1
4) ¬(k<n-1) by and_e on 1
5) for every ig {
6) assume 0≤ig ∧ ig<n-1 {
7) 0≤ig ∧ ig<k ⇒ B[ig] = BO[ig+1] by forall_e on 2
8) 0≤ig by and_e on 6
9) ig<n-1 by and_e on 6
10) k≥n-1 by arith on 4
11) ig<k by arith on 9, 10
12) 0≤ig ∧ ig<k by and_i on 8, 11
13) B[ig] = BO[ig+1] by imp_e on 7, 12
}
14) 0≤ig ∧ ig<n-1 ⇒ B[ig] = BO[ig+1] by imp_i on 6-13
}
15) ∀i · 0≤i ∧ i<n-1 ⇒ B[i] = BO[i+1] by forall_i on 5-14

- in proofs of VCs, can use 3 set rules for arr asgnt override:
  - ↳ $(B \oplus \{(i, e)\})[i] = e$ by set /* override */
  - ↳ 1) $i = j$
    2) $(B \oplus \{(i, e)\})[j] = e$ by set on 1 /* override */
  - ↳ 3) $\neg(i = j)$
    4) $(B \oplus \{(i, e)\})[j] = B[j]$ by set on 3 /* override */

FUNCTIONS AND PROCEDURES

- function returns value but doesn't change value of any vars
- procedure doesn't return anything but can change values of vars that are passed as params
- formal params are identifiers used to rep values passed to a func/proc
  - ↳ appear in func/proc declaration
- actual params are actual values passed to func/proc
  - ↳ appear in call to func/proc
- function rule is specific to a function call ; it provides us w/ tautology that can be used to prove VCs
  - ↳ not used in annotated program
  - ↳

```
fun f(x1, x2) {
    assert(R);
    C;
    assert(S);
    return ret;
}

...

assert(P);
assert(Q[f(a1, a2)/y])     by implied on VCi;
y := f(a1,a2);
assert(Q)     by asn;
```

For function call $f(a1, a2)$, the function rule gives us:
Lemma 1: $(R \Rightarrow S)[a1/x1, a2/x2, f(a1, a2)/ret]$

VCi: $P \vdash Q[f(a1, a2)/y]$ (which will be proven using Lemma 1)

  - C is body of func f
  - $x_1, x_2, \ldots, x_n$ are formal params of f
  - $a_1, a_2, \ldots, a_n$ are acc params in call to f
  - ret is value returned by f
  - may be other local vars in body of f must R ; S can't contain any local vars except for ret
  - ignore return stmts in func body

- to use func rule:
  1) show body of func satisfies spec w/ precond R ; postcond S
  2) use func rule for each call ; conclude lemmas such as $(R \Rightarrow S)[a_1/x_1, a_2/x_2, \ldots, a_n/x_n, f(a_1, \ldots, a_n)/ret]$
  3) use lemma in proofs of VC
- simplifying assumptions:
  - ↳ will not declare vars so there'll be undeclared vars suddenly appearing in program
  - ↳ name of var containing return value is always ret

↳ last stmt in func is always return
- multiple returns not allowed
- postcond is right before return stmt

↳ funcs can't change any param values or global vars

↳ write ret := x+y; return ret; instead of return (x+y);

· e.g.

```
fun f(x,y) {
    assert(x ≥ 0);
    ...
    assert(ret = x + y);
    return ret;
    }
```

```
assert(b ≥ 0);
assert ( f(b,c) ≥ b+c )  by implied on VC1;
d := f(b,c);
assert(d ≥ b + c)  by asn;
```

↳ for function call f(b,c), we get

(R ⇒ S)[a1/x1, ..., f(a1, ...)/ret]
(x ≥ 0 ⇒ ret = x+y)[b/x, c/y, f(b,c)/ret]
} don't have to write in proof

Lemma 1: b ≥ 0 ⇒ f(b,c) = b+c by func rule for f(b,c);

VC1: b ≥ 0 ⊢ f(b,c) ≥ b+c

1) b ≥ 0 premise
2) b ≥ 0 ⇒ f(b,c) = b+c premise
3) f(b,c) = b+c by imp_e on 1,2
4) f(b,c) ≥ b+c by arith on 3

· for procedures, must know which vars are changed by them

↳ values of some params may be changed
- changed params can't be passed as expressions

↳ global vars may also be changed

· procedure rule format:

```
proc p(x1, x2) {
    assert(R);
    C;
    assert(S);
}
...
assert(P)
assert(R[a1/x1, a2/x2] ∧ H );    by implied on VCi;
p(a1,a2);
assert( S[a1/x1, a2/x2] ∧ H)    by procedure;
assert(Q)    by implied on VCj;

VC i: P  ⊢  R[a1/x1, a2/x2] ∧ H
VC j: S[a1/x1, a2/x2] ∧ H  ⊢  Q
```

↳ H is formula that must not contain any program vars changed in proc p

↳ C is body of proc p

↳ $x_1, x_2, \ldots, x_n$ are formal params of p

↳ $a_1, a_2, \ldots, a_n$ are acc params in call to p

↳ if R or S contain any logical vars, also have to sub for those

- to use procedure rule:
  1) show body of proc satisfies spec w/ precond R & postcond S
  2) use proc rule to conclude proc call satisfies spec w/ precond $R[a_1/x_1, a_2/x_2] \wedge H$ & postcond $S[a_1/x_1, a_2/x_2] \wedge H$
     - H is chosen based on P & Q & what's needed to prove VCs
  3) prove:
     - VCi: $P \vdash R[a_1/x_1, a_2/x_2] \wedge H$
     - VCj: $S[a_1/x_1, a_2/x_2] \wedge H \vdash Q$
- for any assertion that ref var w, can create logical var w_0 that equals value of w at particular point in execution of program
- log_var_intro rule:

```
assert(P[w/w0]); /* formula in terms of w only */
assert(P ∧ w0 = w) by log_var_intro;
C;
assert(Q) by proof_rule;
```

  ↳ w_0 must new name that hasn't been used before
  ↳ P is wff w/ occurrences of w_0 & possibly w in it
- introing logical var is like adding line to program for history var.

```
assert(P[w/w0] ∧ w = w);
w0 := w;
assert(P ∧ w0 = w) by asn; /* P is in terms of w0 */
C;
assert(Q) by proof_rule;
```

- e.g.

```
proc Unknown(x) { /* x is changed in Unknown */
   assert(x = x0);
   ...
   assert(x > x0);
}
```

```
assert(q = y);
assert (q = q0 ∧ q0 = y) by log_var_intro;
Unknown(q);
assert (q > q0 ∧ q0 = y) by procedure;
assert(q > y) by implied on VC1;
```

VC1: $q > q0 \wedge q0 = y \vdash q > y$

- e.g.

```
proc inc(x) { /* x is changed in inc */
   assert( x = x0);
   x := x + 1;
   assert( x = x0 + 1);
}
```

```
assert(b = b0 );
inc(b);
assert( b = b0 + 1) by procedure;
assert ( b = b1 ∧ b1 = b0 + 1) by log_var_intro;
inc(b);
assert ( b = b1 + 1 ∧ b1 = b0 + 1) by procedure;
assert( b = b0 + 2) by implied on VC1;
```

VC1: $b = b1+1 \wedge b1 = b0+1 \vdash b = b0+2$

RECURSIVE FUNCTIONS AND PROCEDURES

- for recursive func, assume it satisfies its specs for all calls to func & then prove program satisfies its spec
  - ↳ using induction in correctness proofs of recursive funcs
    - don't need to explicitly talk abt base case b/c non-recursive part of func body will be proven correct as we use normal techniques

- e.g.

Pre: $\exists k \bullet 0 \leq k \wedge k \leq i \wedge B[k] = c$

```
fun find(i) {
    assert(Pre);
    if (B[i]=c) then {
        assert ( Pre ∧ B[i] = c ) by if_then_else;
        assert ( B[i] = c)  by implied using and_e;
        ret := i;
        assert (B[ret] = c)
    } else {
        assert ( Pre ∧ ¬(B[i] = c)) by if_then_else;
        assert (B[find(i-1)] = c) by implied on VC1;
        ret := find(i-1);
        assert (B[ret] = c)
    }
    assert( B[ret] = c)  by if_then_else;
    return ret;
}
```

VC1: $Pre \wedge \neg(B[i] = c) \vdash B[find(i-1)] = c$

Lemma 1: $(\exists k \cdot 0 \leq k \wedge k \leq i-1 \wedge (B[k] = c)) \Rightarrow B[find(i-1)] = c$ by func rule for find(i-1)

To prove VC1:

lem1) $(\exists k \cdot 0 \leq k \wedge k \leq i-1 \wedge (B[k] = c)) \Rightarrow B[find(i-1)] = c$ premise

1) $(\exists k \cdot 0 \leq k \wedge k \leq i \wedge (B[k] = c)) \wedge \neg(B[i] = c)$ premise

2) $\exists k \cdot 0 \leq k \wedge k \leq i \wedge (B[k] = c)$ by and_e on 1

3) $\neg(B[i] = c)$ by and_e on 1

4) for some ku assume $0 \leq ku \wedge ku \leq i \wedge (B[ku] = c)$ {

  5) $0 \leq ku$ by and_e on 4

  6) $ku \leq i$ by and_e on 4

  7) $B[ku] = c$ by and_e on 4

  8) disprove $ku = i$ {

    9) $B[i] = c$ by eq_e on 7,8

    10) false by not_e on 3,9

  }

  11) $\neg(ku = i)$ by raa on 8-10

  12) $ku - 1 \leq i$ by arith on 6, 11

  13) $0 \leq ku \wedge ku - 1 \leq i \wedge (B[ku] = c)$ by and_i on 5, 7, 12

  14) $\exists k \cdot 0 \leq k \wedge k - 1 \leq i \wedge (B[k] = c)$ by exists_i on 13

}

15) $\exists k \cdot 0 \leq k \wedge k - 1 \leq i \wedge (B[k] = c)$ by exists_e on 2, 4-14

16) $B[find(i-1)] = c$ by imp_e on lem1, 15

- for recursive procedures, assume it satisfies its spec for all calls to it & then

prove program satisfies its spec

- e.g.

```
proc add(x, y) {
   /* procedure ''add'' changes x */
   /* it does not change y */
   assert((x = x0) ∧ (y ≥ 0));
   if (y!=0) then {
      assert ((x=x0) ∧ (y ≥ 0) ∧ ¬(y=0)) by if_then;
      assert ((x=x0) ∧ (y-1≥0)) by implied on VC1;
      add(x, y-1);
      assert (x = x0+y-1) by procedure
      assert (x+1 = x0+y) by implied using arith;
      x := x + 1;
      assert (x=x0+y) by asn;
   }
   assert(x = x0 + y) by if_then on VC2;
}
```

VC2: $(x = x0) \wedge (y \geq 0) \wedge (y = 0) \vdash x = x0 + y$

1) $(x = x0) \wedge (y \geq 0) \wedge (y = 0)$ premise
2) $x = x0$ by and_e on 1
3) $y = 0$ by and_e on 1
4) $x = x0 + 0$ by arith on 2
5) $x = x0 + y$ by eq_e on 3, 4

VC1: $(x = x0) \wedge (y \geq 0) \wedge \neg(y = 0) \vdash (x = x0) \wedge (y - 1 \geq 0)$

1) $(x = x0) \wedge (y \geq 0) \wedge \neg(y = 0)$ premise
2) $x = x0$ by and_e on 1
3) $y \geq 0$ by and_e on 1
4) $\neg(y = 0)$ by and_e on 1
5) $y - 1 \geq 0$ by arith on 3, 4
6) $(x = x0) \wedge (y - 1 \geq 0)$ by and_i on 2, 5

TERMINATION

- 2 ways program might not terminate:
  - ↳ infinite while loop
  - ↳ non-terminating func/proc recursion
- to prove **total correctness** of while loop, show that it terminates
- to show **termination**, identify integer expression called **bounding expression** involving vars of loop cond whose value:
  - ↳ is always ≥ 0
  - ↳ dec w/every loop iteration
  - ↳ makes loop's guard become false as it approaches 0
- e.g.

```
assert(n > 0)
sum := 0;
j := 0;
while (j != n) do {
   sum := sum + a;
   j := j + 1;
};
```

↳ bounding exp is n-j

- e.g.
```
assert(n > 0)
f := 1;
t := 0;
i := 0;
while (i != n) do {
   t := t + f;
   f := f * r;
   i := i + 1;
};
```
  - ↳ bounding exp is n - i
- to show recursive func/proc terminates:
  - ↳ determine bounding function (integer expression) involving arguments to func/proc call that's non-neg & dec in each recursive call
  - ↳ show there's section of func/proc that doesn't make recursive calls & is entered as bounding func approaches 0 (i.e. base case)
- e.g. Pre: $\exists k \bullet 0 \leq k \wedge k \leq i \wedge B[k] = c$

```
fun find(i) {
   assert(Pre);
   if (B[i] = c) then {
      ret := i;
   } else {
      ret := find(i-1);
   }
   assert( B[ret] = c)    by if_then_else;
   return ret;
}
```

  - ↳ bounding func is i
- e.g.

```
proc add(x, y) { /* procedure ''add'' changes x; */
        /* it does not change y */
   assert((x = x0) ∧ (y ≥ 0)) ;
   if (y!=0) then {
      add(x, y-1);
      x := x + 1;
   }
   assert(x = x0 + y)    by if_then_else on VC2;
}
```

  - ↳ bounding func is y

- decision problems are subset of problems w/ yes/no answers
- unsolvable/undecidable problems are decision problems where there cannot exist an algorithm to solve them
  - ↳ propositional logic is decidable
  - ↳ predicate logic is undecidable
    - more specifically, semidecidable