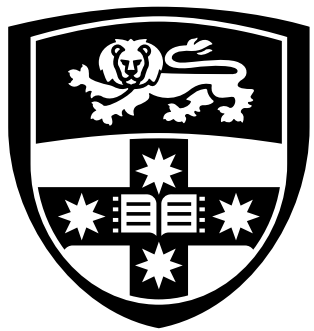
COMP2022: Formal Languages and Logic

2017, Semester 1, Week 12

Joseph Godbehere

Adapted from slides by A/Prof Kalina Yacef

May 30, 2017



THE UNIVERSITY OF
SYDNEY

Announcements

Assignment 3:

- Due on Thursday

Advanced Seminar (repeat)

- Where: here
- When: Friday 5pm
- Topic: Greg will demonstrate how to prove the *completeness* of the version of the Natural Deduction System that we used for predicate logic.
- Non-assessable, optional, but interesting!

Outline

Predicate Logic (continued)

- Prenex Normal Form

Turing Machines

- Definition
- Interesting properties
- Computability

Prenex Normal Form

Predicate logic wffs can be put into an equivalent form where all the quantifiers occur at the front, and where the only connectors are $\neg, \wedge, \vee$

i.e. they look like $Q_1 x_1 Q_2 x_2 ... Q_n x_n M$ where Q_i is either $\forall$ or $\exists$, and M is a wff without quantifiers.

For example

- $\forall x \exists z(\neg P(x, y) \wedge R(z))$ is in prenex normal form
- $\forall x \forall y \exists z(\neg P(x, y) \wedge R(z))$ is in prenex normal form
- $\neg(\exists x P(x, y) \vee \forall z \neg R(z))$ is not
- $(\forall y \exists x P(x, y) \vee \forall z R(z))$ is not

Putting a wff into Prenex Normal Form

1. Rename bound occurrences of variables which occur both bound and free, so that no quantifier uses the same variable name as a free variable.
2. Rename variables quantified more than once
3. Use equivalence laws to eliminate $\rightarrow$ and $\leftrightarrow$ connectors
4. Push negations inwards, and eliminate double negations, using equivalence laws
5. Move quantifiers outwards using equivalence laws
6. Finally:
   - For Prenex Conjunctive Normal Form: distribute $\vee$ over $\wedge$
   - For Prenex Disjunctive Normal Form: distribute $\wedge$ over $\vee$

Prenex Conjunctive/Disjunctive Normal Forms

Prenex CNF: $Q_1 x_1 Q_2 x_2 ... Q_n x_n (C_1 \wedge ... \wedge C_k)$ where C_i is a disjunction of one or more literals (or negations of literals)

- i.e. $Q_1 x_1 Q_2 x_2 ... Q_n x_n F$ where F is in CNF)

Prenex DNF: $Q_1 x_1 Q_2 x_2 ... Q_n x_n (C_1 \vee ... \vee C_k)$ where C_i is a conjunction of one or more literals.

The algorithm is the same as that used to transform wff into CNF/DNF for Propositional Logic, except that we also need to do some relabelling, and to pull all of the quantifiers to the front (outside) of the formula.

Example 1

Put $(P(f(x)) \to \forall y P(y))$ into Prenex Conjunctive Normal Form

$$
\begin{aligned}
& (P(f(x)) \to \forall y P(y)) && \\
\equiv\ & (\neg P(f(x)) \vee \forall y P(y)) && \text{(Def Imp.)} \\
\equiv\ & \forall y (\neg P(f(x)) \vee P(y)) && \text{(Q. Extraction)}
\end{aligned}
$$

Example 2

Put $(\forall x P(x) \to Q(f(x), x))$ into Prenex Conjunctive Normal Form

Note that the first 2 occurrences of x are bound, the last two are free. We need to rename the first 2.

$$
\begin{aligned}
& (\forall x P(x) \to Q(f(x), x)) && \\
\equiv\ & (\forall y P(y) \to Q(f(x), x)) && \text{(Renaming)} \\
\equiv\ & (\neg \forall y P(y) \vee Q(f(x), x)) && \text{(Def Imp.)} \\
\equiv\ & (\exists y \neg P(y) \vee Q(f(x), x)) && \text{(Q. Negation)} \\
\equiv\ & \exists y (\neg P(y) \vee Q(f(x), x)) && \text{(Q. Extraction)}
\end{aligned}
$$

Example 3

Put $(\neg Q(f(x), x) \to (\exists y Q(x, y) \to \forall y P(y)))$ in Prenex Conjunctive Normal Form

$$
\begin{aligned}
& (\neg Q(f(x), x) \to (\exists y Q(x, y) \to \forall y P(y))) && \\
\equiv\ & (\neg Q(f(x), x) \to (\exists y Q(x, y) \to \forall z P(z))) && \text{(Renaming)} \\
\equiv\ & (\neg\neg Q(f(x), x) \vee (\exists y Q(x, y) \to \forall z P(z))) && \text{(Def Imp.)} \\
\equiv\ & (\neg\neg Q(f(x), x) \vee (\neg\exists y Q(x, y) \vee \forall z P(z))) && \text{(Def Imp.)} \\
\equiv\ & (Q(f(x), x) \vee (\neg\exists y Q(x, y) \vee \forall z P(z))) && \text{(DNeg.)} \\
\equiv\ & (Q(f(x), x) \vee (\forall y \neg Q(x, y) \vee \forall z P(z))) && \text{(Q. Neg.)} \\
\equiv\ & (Q(f(x), x) \vee \forall y (\neg Q(x, y) \vee \forall z P(z))) && \text{(Q. Extr.)} \\
\equiv\ & \forall y (Q(f(x), x) \vee (\neg Q(x, y) \vee \forall z P(z))) && \text{(Q. Extr.)} \\
\equiv\ & \forall y (Q(f(x), x) \vee \forall z (\neg Q(x, y) \vee P(z))) && \text{(Q. Extr.)} \\
\equiv\ & \forall y \forall z (Q(f(x), x) \vee (\neg Q(x, y) \vee P(z))) && \text{(Q. Extr.)}
\end{aligned}
$$

Paradox of the Liar

By Eubulides, Greek philosopher, 600 BC:

Epimenides, the Cretan, says to a fellow Cretan:
"all Cretans are liars"
Epimenides, being a Cretan, lies.
Therefore, Cretans don't lie, and Epimenides does not either.
Hence all Cretans are liars ...?

Where is the mistake?

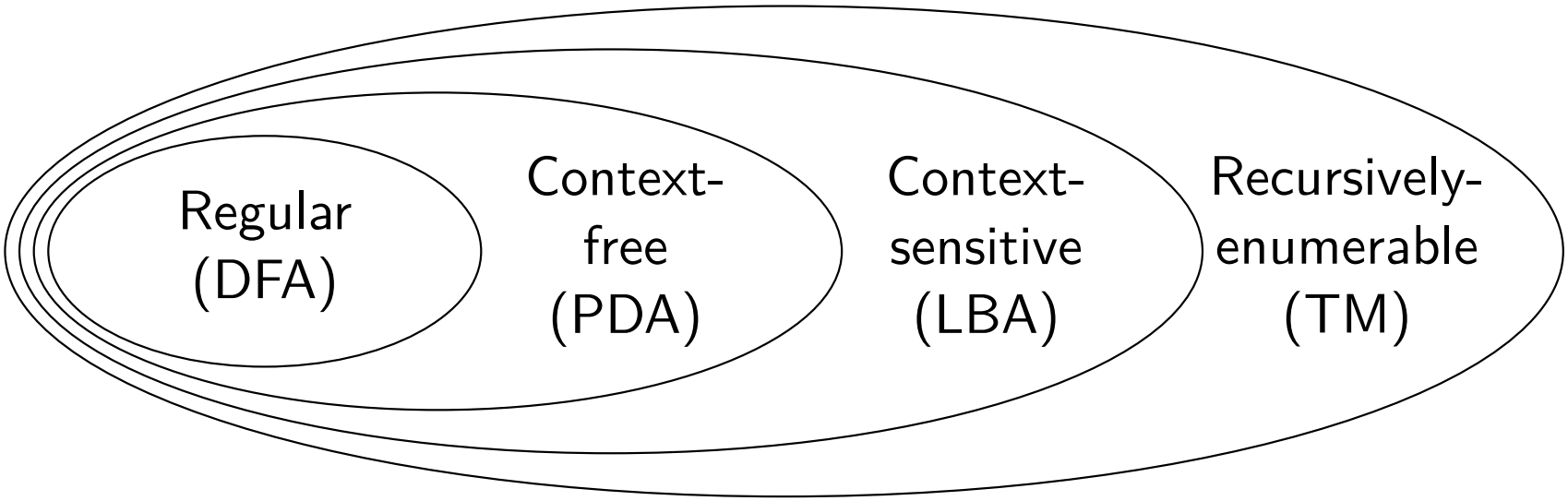
Predicate logic for program verification

- Pre- and post- conditions are predicate logic formula used to verify programs.
  - "Programming by contract"
- If all pre-conditions are true, then the post conditions must also be true.
- Assertions.

Logic: Summary

- Predicate logic
  - More expressive than propositional logic (terms & quantifiers)
  - But also more complex
  - Formal proofs:
    - Natural Deduction System N (Laws of Equivalence, Rules of Inference, laws and rules involving quantifiers)
  - Proof by Resolution
    - Laws of Equivalence, Indirect Proof, Resolution Rule
    - Prolog: uses Prenex CNF + Resolution
- Tautologies
- Invalidity
  - find a truth assignment for which the argument does not hold
- Soundness of the NDS, and of Resolution
  - Any wff derived from an empty set of premises is a tautology
- Completeness of the NDS, and of Resolution
  - Any tautology can be derived in the system

Chomsky Hierarchy



Type 0 grammars (unrestricted grammars) are powerful enough to describe the set of *recursively enumerable* languages.

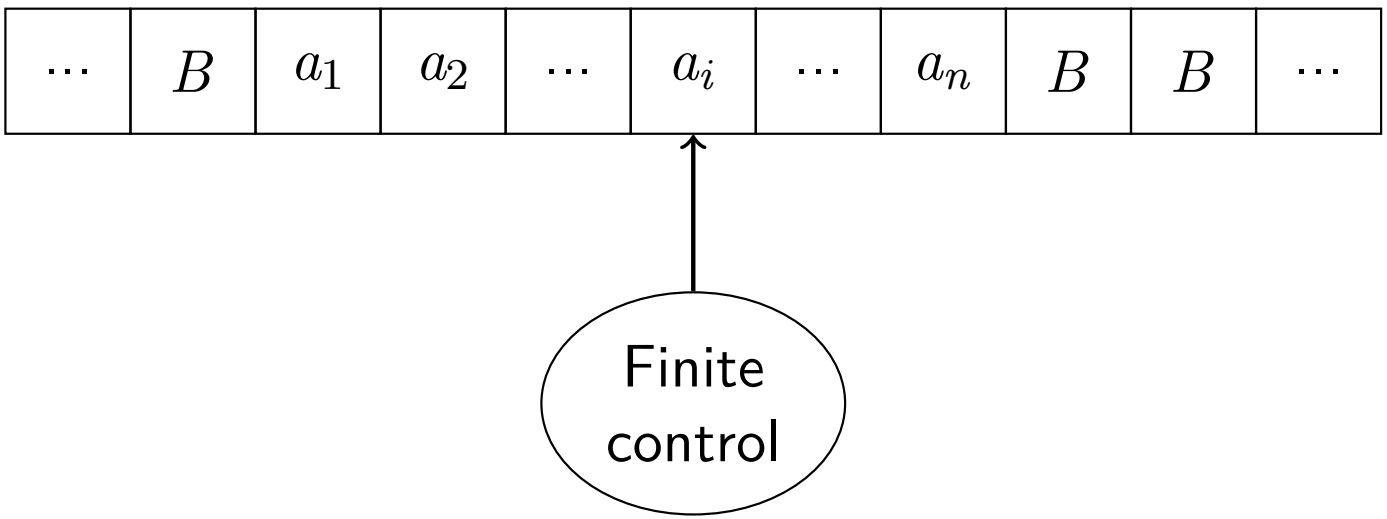
Turing Machines

- Finite automata have a finite memory. They recognise regular languages (regular grammars)
- Push Down Automata: Add a LIFO stack (infinite memory, but access is limited to the top). PDAs recognise context-free languages (context-free grammars)
- By changing the stack for a tape (infinite memory, no limitation of access) we get Turing machines: our current computers. TM recognise recursively enumerable languages (unrestricted grammars)
- Alan Turing (1912-1954) mathematician and logician
  - http://www.alanturing.net
  - Turing machines introduced in his 1936 article

Turing Machines: schematic view

Turing machines have infinite memory: a tape.

- Can read and write symbols on it, or do nothing
- Can move to the right or to the left along the tape



i.e. it is a finite state automaton attached to an infinite tape

Formal Definition

A Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

- Q is a finite set of states
- Σ is a finite set called the input alphabet. $B \notin \Sigma$
- Γ is a finite set of tape symbols. $\Sigma \subset \Gamma$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.
  - L means 'move left', R means 'move right'
- $q_0 \in Q$ is the start state
- B is a special symbol of Γ, the *blank*
- $F \subseteq Q$ is the set of accept states

Turing machine for $\{0^n 1^n \mid n > 0\}$

- $Q = \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, X, Y, B\}$
- $q_0 \in Q$ is the start state
- B
- $F = \{q_4\}$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is given by:

	0	1	X	Y	B
q_0	(q_1, X, R)			(q_3, Y, R)	
q_1	$(q_1, 0, R)$	(q_2, Y, L)		(q_1, Y, R)	
q_2	$(q_2, 0, L)$		(q_0, X, R)	(q_2, Y, L)	
q_3				(q_3, Y, R)	(q_4, B, R)
q_4					

Trying M on 0011

Semi-formal description of M

Assuming:

- The tape initially contains only 0s, 1s and blanks
- We start at the first non-blank symbol on the tape

1. If we did not start on a 0, reject
2. Replace the 0 with an X
3. Move right until a 1 occurs
4. Replace the 1 with a Y
5. Move left until we find the X marker
6. Move right once. If we are on a 0, goto (2)
7. Move right across the tape, skipping X and Y.
   - If a 1 or 0 is scanned, reject
   - If a blank is scanned, accept

Turing machine for a non-CFL: $\{a^n b^n c^n \mid n > 0\}$

- $Q = \{q_0, q_a, q_b, q_c, q_y, q_z, q_f\}$
- $\Sigma = \{a, b, c\}$
- $\Gamma = \{a, b, c, X, Y, Z, B\}$
- $q_0 \in Q$ is the start state
- B
- $F = \{q_f\}$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is given by:

	a	b	c	X	Y	Z	B
q_0	(q_a, X, R)				(q_y, Y, R)		
q_a	(q_a, a, R)	(q_b, Y, R)			(q_a, Y, R)		
q_b		(q_b, b, R)	(q_c, Z, L)			(q_b, Z, R)	
q_c	(q_c, a, L)	(q_c, b, L)		(q_0, X, R)	(q_c, Y, L)	(q_c, Z, L)	
q_y					(q_y, Y, R)	(q_z, Z, R)	
q_z						(q_z, Z, R)	(q_f, B, R)
q_f							

Trying it on *aabbcc*

Language accepted by a Turing Machine

- The language accepted by a Turing machine M is the set of strings such that M started on the left of the input and reached some state of F
- M may fail to stop!
- If M stops in some non-accepting state or never stops on input w, then M does not accept w.
- Languages accepted by a Turing Machine are "recursively enumerable"
- Unrestricted grammars (type 0)

Turing machine to add 2 numbers

A number n is represented by n 0's (*unary*)

The two numbers are separated by a 1

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{0, 1, B\}$
- $q_0 \in Q$ is the start state
- B
- $F = \{q_3\}$
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is given by:

	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 0, R)$	(q_3, B, R)
q_1	$(q_1, 0, R)$		(q_2, B, L)
q_2	(q_3, B, R)		
q_3			

Semi-formal description

1. Move to the right until we reach a 1 (skipping 0s)
2. Rewrite the 1 with a 0
3. Move to the right until we reach a blank (skipping 0s)
4. Move to the left once, to rewrite the last 0 with a blank

For example, if our tape originally read $...B000100B...$, it now reads $...B00000BB...$. i.e. $3 + 2 = 5$

Turing machines can make calculations! Note the similarity with computers.

Left-bounded tapes

Can a Turing Machine with an infinite tape in *both* directions recognise more languages than tape that is only unbounded in one direction?

Does a tape with positions $0, 1, 2, 3, ...$ have more positions than one with positions $... - 2, -1, -0, 1, 2, ...$? i.e. are there "more" integers than positive integers?

Counter-intuitively, the answer is *no*, because we can pair up every integer with a positive integer:

1	2	3	4	5	$...$
0	-1	1	-2	2	$...$

So, a tape unbounded to the left does not have more positions.

This is what we mean when we say a set is *enumerable*.

Left-bounded tapes

Can a Turing Machine with an infinite tape in *both* directions recognise more languages than tape that is only unbounded in one direction?

We could make a machine that jumped around to seek the correct positions according to the bijection of integers to positive integers, but there's a much simpler way:

Add states to the TM such that whenever we want to move past the left edge of the tape, we simply shift everything on the tape one position to the right instead.

- We will need to place a *marker* at the far right of the input, so we know when to stop shifting.

Multi-tape Turing Machines

Can a multi-tape Turing Machine recognise more languages than a single-tape TM?

Multi-tape Turing Machines

Can a multi-tape Turing Machine recognise more languages than a single-tape TM?

NO

Proof: We can modify a single tape TM to simulate having multiple tapes:

- Add markers symbolising the start and end of each tape
- Add markers symbolising the current TM position on each
- Add states which simulate switching between the tapes
- Add states which extend these (finite) stretches of tape (by shifting everything beyond it by one space)

Context-free languages

Can a TM recognise every context-free language?

Context-free languages

Can a TM recognise every context-free language?

YES

Proof: A TM can simulate a PDA.

- Write the PDA input to the tape
- Mark a section of tape beyond the end of the input as the stack
- Repeatedly move between the tape and the end of the stack
- i.e. each state in the PDA will have two states in the TM
  - One for the current input symbol, one for the top of stack

Context-free languages

Can a TM recognise every context-free language?

YES

Proof: A TM can simulate a PDA.

- Write the PDA input to the tape
- Mark a section of tape beyond the end of the input as the stack
- Repeatedly move between the tape and the end of the stack
- i.e. each state in the PDA will have two states in the TM
  - One for the current input symbol, one for the top of stack

Note: we will need a *non-deterministic* TM (recall that N-PDA are more powerful than D-PDA!)

Non-deterministic TM

Is a Non-deterministic TM more powerful than a deterministic one?

Non-deterministic TM

Is a Non-deterministic TM more powerful than a deterministic one?

NO

Proof: We can simulate a NTM with a 3-tape DTM

- Tape 1 contains the original input
- Tape 2 simulates a particular computation of the TM
- Tape 3 encodes the path through the tree of possible paths through the NTM

Non-deterministic TM

Is a Non-deterministic TM more powerful than a deterministic one?

NO

Proof: We can simulate a NTM with a 3-tape DTM

- Tape 1 contains the original input
- Tape 2 simulates a particular computation of the TM
- Tape 3 encodes the path through the tree of possible paths through the NTM

This DTM will (very slowly!) explore all the possible paths through the NTM, from shortest to longest.

Universal Turing machine

- It is possible to encode a Turing machine or to write a "program" to describe a Turing machine.
- The Universal Turing Machine accepts an encoded Turing machine M together with a string w and simulates M on w.
- Exactly what general computers do:
  - They accept a program P
  - and the input to that program
  - and produces the output of P on the given input
- Again, note the similarity between TM and computers

Church-Turing Thesis

"Every effectively calculable function is a computable function."

Essentially this means that the set of *algorithms* is equivalent to the set of *Turing Machine* algorithms.

Further, the notion of an "effectively calculable algorithm" can be reduced to "decidable predicate" – Turing Machines implement predicate logic!

So:

- TM are a very precise model for definition of algorithm.
- TM can do *everything* that a computer can do.
- However, even a Turing machine cannot solve all problems:
  - some are beyond theoretical limits of computation

Computability, Decidability, Intractability

Decidable $\subset$ TM computable

- A function that can be computed by a Turing machine is said to be *computable* (therefore the language is *enumerable*)
- It is *decidable* if and only if the Turing machine will always halt on any input string (the Halting Problem)
- *Intractability*: The efficiency problem – can we solve the problem in a reasonable time (e.g. polynomial vs exponential)?

Example of undecidable problem: The Halting problem

- Halting problem: Is there an algorithm that can decide whether the execution of an arbitrary program halts on an arbitrary input?
- e.g.
  - $f(x)\{\ return\ 2x+1;\}$, will f halt on any number?
  - $h(x)\{\ while(true)\ \{x+=1;\}\ ...\}$?
  - $g(x)\{\ for(i=x;\ i<10;\ i-=1);\ ...\}$?
- Answer: NO (Church 1936, Turing 1937)
- How can you restrict the halting problem to be decidable?

Example of undecidable problem: The Total problem

- A program/function is *total* iff it halts on all inputs
- Total problem: is there an algorithm to tell whether an arbitrary computable function is total?
- Answer: NO

Example of undecidable problem: Post-correspondence problem

- Given a collection of dominoes, such as
$$\frac{b}{ca} \quad \frac{a}{ab} \quad \frac{ca}{a} \quad \frac{abc}{c}$$
- The task is to make a list of dominoes (repetition allowed) such that the top and bottom strings match. e.g.
$$\frac{a}{ab}\frac{b}{ca}\frac{ca}{a}\frac{a}{ab}\frac{abc}{c} = \frac{abcaaabc}{abcaaabc}$$
- Naive approach: try to build a list with one domino, then with 2, 3, etc. (repetition allowed) until we find a solution.
  - If a solution is found, STOP (machine will halt)
  - However, if there is NO solution, the machine will fail to halt

Are these problems decidable?

- Is there a computable function that can tell whether an arbitrary function f always halts on input m within k units of time?

- Is there a way to decide whether two DFAs over the same alphabet are equivalent?

Partially decidable problems

- A *partially decidable* problem can be represented by a TM which always halts on "yes" answers but which may run forever on "no" answers.
- Many unsolvable problems are partially decidable.
  - e.g. the Halting problem.
- What about:
  - Total?
  - Post-correspondence?

Validity in predicate logic

Church (1936) and Turing (1937) independently proved First Order Logic (predicate logic) is partially decidable.

- If the formula is valid, there are algorithms which will halt.
- If the formula is invalid, there are no algorithms which are guaranteed to always halt.

Because predicate logic is not decidable, the Program Correctness problem cannot be decidable.

Intractability

- Tractable: a problem which can be solved efficiency (*usually* this means there is an algorithm to solve it in polynomial time or better.)

- Intractable: a problem which can be solved, but not fast enough for the algorithm to be usable (*usually* this means the algorithms are slower than polynomial time.)

Predicate Logic

- Undecidable
  - There are algorithms that find proofs when they exist, but they are not guaranteed to halt when no proof exists.

- Intractable
  - The space of possible proofs can be huge.
  - CNF and Resolution significantly help (by reducing the problem space), but they are still not efficient enough.

Program correctness problem

Can we write a program (or equivalently, devise an algorithm or Turing machine) as follows:

- Input:
  - another program
  - a predicate formula
- Output:
  - YES if the program satisfies the property represented by the formula
  - NO if it does not

Program correctness problem

Can we write a program (or equivalently, devise an algorithm or Turing machine) as follows:

- Input:
  - another program
  - a predicate formula
- Output:
  - YES if the program satisfies the property represented by the formula
  - NO if it does not

The answer depends on whether the formula is *valid* where its interpretation is dependent on the given program.

Because the problem of determining whether an arbitrary wff is valid is an *undecidable* problem, the Program Correctness problem is also *undecidable*.

Program correctness problem

The program correctness problem is undecidable, however we still want to produce correct software.

We can formulate simpler problems which may be decidable, or tractable

- relax requirement to get exact answer
- coarsen the search space (e.g. abstract numeric values to just positive, zero, negative)
- use restricted forms of formulas
  - pre/post conditions that do not contain quantifiers
- place restrictions on the types of programs
  - e.g. only check certain types of operations

Programming steps

1. Recognise whether the problem is computable
   - If not, can the problem be redefined?
2. Recognise whether the problem is decidable and tractable
   - If not, try to use heuristics to find an acceptable solution with better time efficiency
3. Analyse how efficiently the problem can be solved
   - improve time, space complexity
4. Test program correctness
   - Theorem provers, model checkers, ...

Conclusion

- Computers are sophisticated Turing machines (but *not* more powerful!)
- Church-Turing thesis
- Classify problems using Turing machines
  - Non-computable problems (no TM exists)
  - Undecidable problems
  - Decidable problems
- Time, Memory needed (FA? PDA? TM?)
- Logic: construct paradox, contradiction, error free programs
- Formal languages and logic are the essential foundations of computer science

Next week!

- Assignment 3 results (best proofs!)
- Exam advice
  - Topics
  - Structure
  - etc.
- Revision
  - Broad overview
  - Tell me in advance any topics you want revisited in detail