Computational Logic

A Motivational Introduction

Note: slides with executable links. Follow the run example $\longmapsto$ links to execute the example code.

The Code Correctness Problem

- Consider a simple programming task:

> "Compute the squares of the natural numbers that are less or equal than 5"

The Code Correctness Problem

- Consider a simple programming task:

  “Compute the squares of the natural numbers that are less or equal than 5”

- In our age AI is capable of programming anything!

The Code Correctness Problem

You

write a C program that computes the squares of the natural numbers which are less or equal than 5

ChatGPT

Certainly! Below is a simple C program that computes the squares of natural numbers less than or equal to 5:

c   Copy code

```c
#include <stdio.h>

int main() {
    // Iterate through natural numbers from 1 to 5
    for (int i = 1; i <= 5; ++i) {
        // Compute the square of the current natural number
        int square = i * i;

        // Print the result
        printf("Square of %d: %d\n", i, square);
    }

    return 0;
}
```

The Code Correctness Problem

Let us consider this alternative code (generated by a **human programmer**):

```c
#include <stdio.h>
main() {
  int number, square;
  number = 0;
  while (number <= 5) {
    square = number * number;
    printf("%d\n",square);
    number = number + 1;
  }
}
```

The Code Correctness Problem

Let us consider this alternative code (generated by a **human programmer**):

```
#include <stdio.h>
main() {
  int number, square;
  number = 0;
  while (number <= 5) {
    square = number * number;
    printf("%d\n",square);
    number = number + 1;
  }
}
```

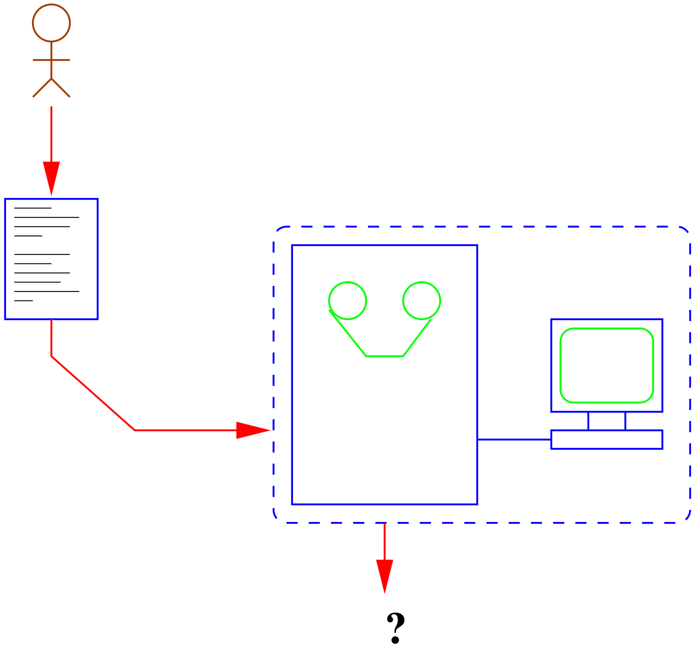
Semantically identical, except for a small detail:

- Version 1: `1 4 9 16 25`
- Version 2: `0 1 4 9 16 25`

The Code Correctness Problem

Let us consider this alternative code (generated by a **human programmer**):

```
#include <stdio.h>
main() {
  int number, square;
  number = 0;
  while (number <= 5) {
    square = number * number;
    printf("%d\n",square);
    number = number + 1;
  }
}
```



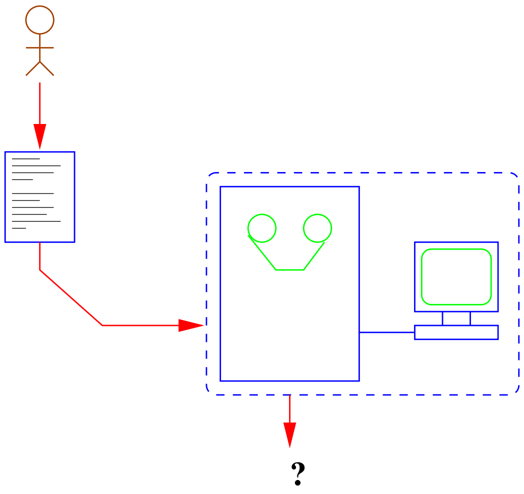
Semantically identical, except for a small detail:

- Version 1: `1 4 9 16 25`
- Version 2: `0 1 4 9 16 25`

PROBLEM: Which one should we trust? Which one is CORRECT?

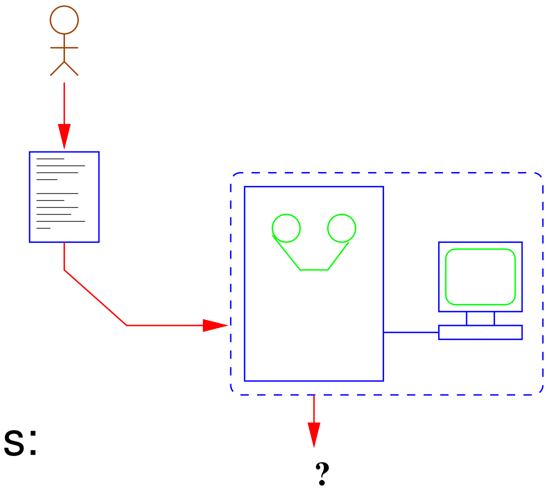
The Code Correctness Problem

- It is not easy to determine correctness:
  - ◇ Testing? – How many tests?
  - ◇ What are the correct outputs anyway? (E.g., was 0 an expected output?)
- I.e., correct, with respect to what?



The Code Correctness Problem

- It is not easy to determine correctness:
  - ◇ Testing? – How many tests?
  - ◇ What are the correct outputs anyway?
    (E.g., was 0 an expected output?)
- I.e., correct, with respect to what?

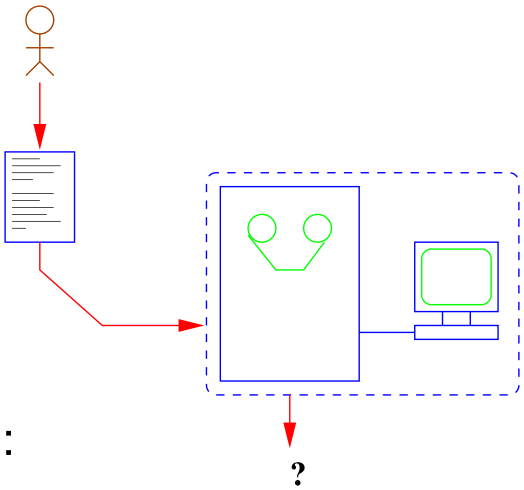


- Correctness is specially important in critical applications:
  - ◇ Medicine, aerospace, transport, energy, etc.

  Once deployed we cannot debug, there are no second chances.
- But due to society's digital transformation
  practically all software is crítical in practice (prestige, economic loss, etc.):
  e-commerce, blockchain, banking, online administration, social media, ...

The Code Correctness Problem

- It is not easy to determine correctness:
  - ◇ Testing? – How many tests?
  - ◇ What are the correct outputs anyway?
    (E.g., was 0 an expected output?)
- I.e., correct, with respect to what?



- Correctness is specially important in critical applications:
  - ◇ Medicine, aerospace, transport, energy, etc.

  Once deployed we cannot debug, there are no second chances.
- But due to society's digital transformation
  practically all software is crítical in practice (prestige, economic loss, etc.):
  e-commerce, blockchain, banking, online administration, social media, ...

AI is here to transform the world, but, how to communicate in an unambiguous form?

Natural Language

"Compute the squares of the natural numbers which are less or equal than 5."

- Would seem ideal at first sight, but:
  - verbose
  - vague
  - ambiguous
  - needs context (assumed information)
  - ...

  Philosophers and Mathematicians already pointed this out a long time ago!

- A more suitable formalism is needed:
  - to provide *specifications* (describe problems), and
  - to reason about the *correctness of programs* (their *implementation*).

Logic

- A means of clarifying / formalizing the human thought process
- *Classical* logic:

  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  implies that
  Plato is a friend of Aristotle

Logic

- A means of clarifying / formalizing the human thought process
- *Classical* logic:

  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  implies that
  Plato is a friend of Aristotle

- *Symbolic* logic:
  A shorthand for classical logic – plus many useful results:

  $a_1 : likes(aristotle, cookies)$
  $a_2 : \forall X \; likes(X, cookies) \rightarrow friend(plato, X)$
  $t_1 : friend(plato, aristotle)$
  $T[a_1, a_2] \vdash t_1$

Logic

- A means of clarifying / formalizing the human thought process
- *Classical* logic:

  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  implies that
  Plato is a friend of Aristotle

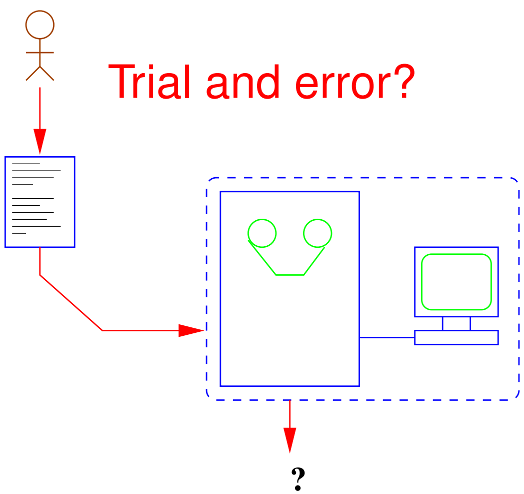
- *Symbolic* logic:
  A shorthand for classical logic – plus many useful results:

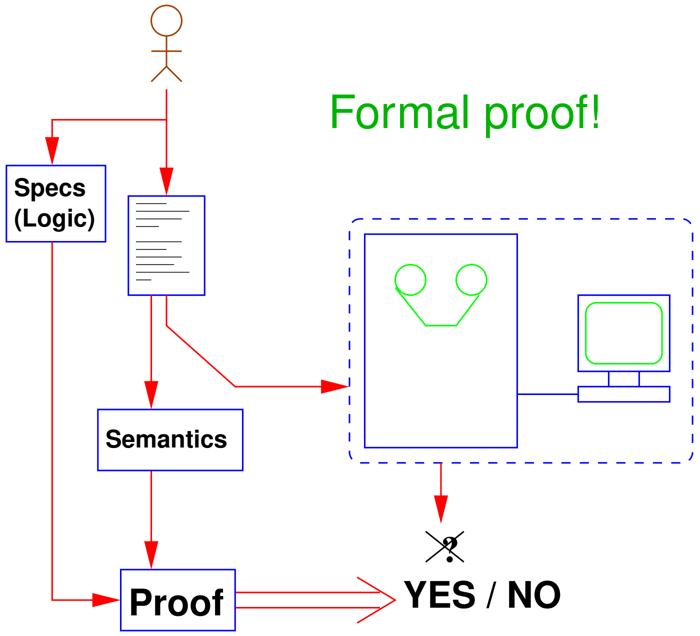
  $a_1 : likes(aristotle, cookies)$
  $a_2 : \forall X \ likes(X, cookies) \rightarrow friend(plato, X)$
  $t_1 : friend(plato, aristotle)$
  $T[a_1, a_2] \vdash t_1$

- So,
  - Can logic be used to represent problems (specifications)?
  - And even perhaps to *solve* problems?

Using Logic

- For expressing specifications and reasoning about program correctness we need:
  - ◇ Specification language (e.g., assertions about input/output), modeling, etc.
  - ◇ Semantics for the programming language (models, axiomatic, denotational, fixpoint, ...).
  - ◇ Proofs: *verification* of program correctness – *for all possible inputs!*





Generating Squares: Building a Specification (I)

“Compute the squares of the natural numbers that are less or equal than 5”

Let us formalize our problem starting from scratch –a game, to specify things fully.

Generating Squares: Building a Specification (I)

“Compute the squares of the natural numbers that are less or equal than 5”

Let us formalize our problem starting from scratch –a game, to specify things fully.

- For numbers we can use “Peano” representation:

  $0 \equiv 0 \qquad 1 \equiv s(0) \qquad 2 \equiv s(s(0)) \qquad 3 \equiv s(s(s(0))) \qquad \ldots$

  ...more or less equivalent to “caveman” numbers:

  $0 \equiv 0 \qquad 1 \equiv | \qquad 2 \equiv || \qquad 3 \equiv ||| \qquad \ldots$

Generating Squares: Building a Specification (I)

"Compute the squares of the natural numbers that are less or equal than 5"

Let us formalize our problem starting from scratch –a game, to specify things fully.

- For numbers we can use "Peano" representation:

  0 $\equiv$ 0 1 $\equiv$ s(0) 2 $\equiv$ s(s(0)) 3 $\equiv$ s(s(s(0))) $\ldots$

  ...more or less equivalent to "caveman" numbers:

  0 $\equiv$ 0 1 $\equiv$ $|$ 2 $\equiv$ $||$ 3 $\equiv$ $|||$ $\ldots$

- Let us then define the natural numbers:

  "0 is a natural, and 1 is a natural, and 2 is a natural, and $\ldots$"

  $nat(0) \quad \wedge \quad nat(s(0)) \quad \wedge \quad nat(s(s(0))) \quad \wedge \quad \ldots$

Generating Squares: Building a Specification (I)

"Compute the squares of the natural numbers that are less or equal than 5"

Let us formalize our problem starting from scratch –a game, to specify things fully.

- For numbers we can use "Peano" representation:

  0 ≡ 0 1 ≡ s(0) 2 ≡ s(s(0)) 3 ≡ s(s(s(0))) ...

  ...more or less equivalent to "caveman" numbers:

  0 ≡ 0 1 ≡ | 2 ≡ || 3 ≡ ||| ...

- Let us then define the natural numbers:

  "0 is a natural, and 1 is a natural, and 2 is a natural, and ..."

  $nat(0) \quad \wedge \quad nat(s(0)) \quad \wedge \quad nat(s(s(0))) \quad \wedge \quad \ldots$

  But this is infinite – a much better solution, an *inductive* definition:

  "0 is a natural number; and, if X is a natural, then the next number, $s(X)$, is too"

$nat(0) \quad \wedge \quad \forall X \ (nat(X) \rightarrow nat(s(X)))$

Generating Squares: Building a Specification (II)

"Compute the squares of the natural numbers that are less or equal than 5"

- Let us now define this less or equal order relation on the naturals:
  - ◇ We start by thinking about the table of all the facts that should be true:

$le(0, 0)$	$le(0, s(0))$	$le(0, s(s(0)))$
$le(s(0), s(0))$	$le(s(0), s(s(0)))$	$le(s(0), s(s(s(0))))$
$le(s(s(0)), s(s(0)))$	$le(s(s(0)), s(s(s(0))))$	$le(s(s(0)), s(s(s(s(0)))))$
$\ldots$	$\ldots$	$\ldots$

  - ◇ We can capture the whole first line with: "0 is less or equal than any natural"

    $\forall X(\; nat(X) \rightarrow le(0, \; X) \;)$

  - ◇ For generalizing vertically we observe that along each column: "if $x \leq y$ then $x + 1 \leq y + 1$", i.e.:

    $\forall X \forall Y(\; le(X, Y) \rightarrow le(s(X), s(Y)) \;)$

  - ◇ Putting it all together we have:

    $\forall X(\; nat(X) \rightarrow le(0, \; X)) \quad \wedge \quad \forall X \forall Y \; (le(X, Y) \rightarrow le(s(X), s(Y)) \;)$

Generating Squares: Building a Specification (III)

"Compute the squares of the natural numbers that are less or equal than 5"

- Square of a natural: "The square of X is X times X."

$$\forall X \forall Y \ (nat(X) \land nat(Y) \ \land mult(X, X, Y) \rightarrow nat\_square(X, Y))$$

where "X times Y is adding Y to itself X times" $\leadsto$ we need "times" and "add."

Generating Squares: Building a Specification (III)

"Compute the squares of the natural numbers that are less or equal than 5"

- Square of a natural: "The square of X is X times X."

  $\forall X \forall Y\ (nat(X) \land nat(Y)\ \land mult(X, X, Y) \rightarrow nat\_square(X, Y))$

  where "X times Y is adding Y to itself X times" $\rightsquigarrow$ we need "times" and "add."

- Addition of naturals:

  $add(0, 0, 0)$ $add(0, s(0), s(0))$ $add(0, s(s(0)), s(s(0)))$ $\ldots$

  $add(s(0), 0, s(0))$ $add(s(0), s(0), s(s(0)))$ $add(s(0), s(s(0)), s(s(s(0))))$ $\ldots$

  Generalizing the first line: "$0 + x = x$"

  Generalizing the columns: "if $x + y = z$ then $(x + 1) + y = z + 1$"

  $\forall X\ (nat(X) \rightarrow add(0, X, X)) \quad \land \quad \forall X \forall Y \forall Z\ (add(X, Y, Z) \rightarrow add(s(X), Y, s(Z)))$

Generating Squares: Building a Specification (III)

"Compute the squares of the natural numbers that are less or equal than 5"

- Square of a natural: "The square of X is X times X."

  $\forall X \forall Y\ (nat(X) \wedge nat(Y)\ \wedge mult(X,X,Y) \rightarrow nat\_square(X,Y))$

  where "X times Y is adding Y to itself X times" $\leadsto$ we need "times" and "add."

- Addition of naturals:

  $add(0,0,0)$ $\quad add(0,s(0),s(0))$ $\quad add(0,s(s(0)),s(s(0)))$ $\quad \ldots$

  $add(s(0),0,s(0))$ $\quad add(s(0),s(0),s(s(0)))$ $\quad add(s(0),s(s(0)),s(s(s(0))))$ $\quad \ldots$

  Generalizing the first line: "$0 + x = x$"

  Generalizing the columns: "if $x + y = z$ then $(x+1) + y = z + 1$"

  $\forall X\ (nat(X) \rightarrow add(0,X,X)) \quad \wedge \quad \forall X \forall Y \forall Z\ (add(X,Y,Z) \rightarrow add(s(X),Y,s(Z)))$

- Multiplication of naturals:

  $\forall X\ (nat(X) \rightarrow mult(0,X,0)) \wedge$
  $\forall X \forall Y \forall Z \forall W\ (mult(X,Y,W) \wedge add(W,Y,Z) \rightarrow mult(s(X),Y,Z))$

  E.g., $mult(3,2,6) \equiv 2+2+2=6$

  $mult(3,2,6) \wedge add(6,2,8) \rightarrow mult(4,2,8) \quad 2+2+2\ \ (+2) = 8$

Generating Squares: Building a Specification (IV)

- No need to despair: everything so far will typically be in libraries!

- We can now write a *formal specification* of the (imperative) program, i.e., conditions that we want the program outputs to meet:
  - *Precondition (empty):*

    $\boxed{true}$
  - *Postcondition:*

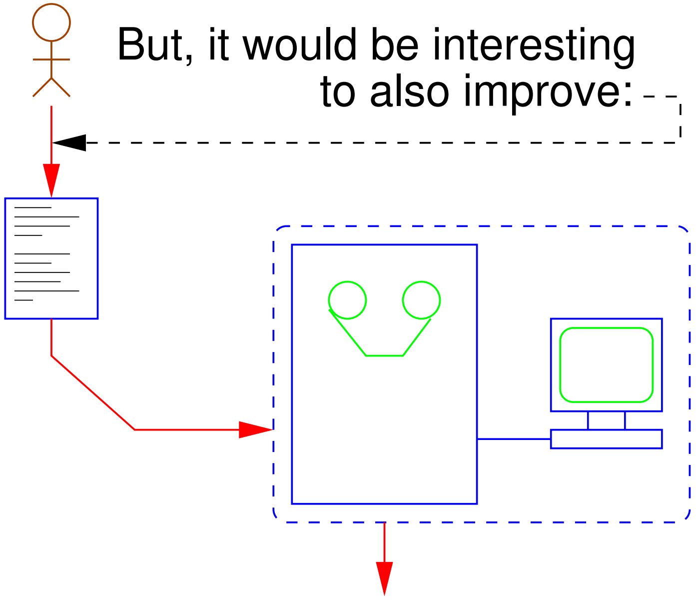
    $\boxed{\forall X(output(X) \leftarrow (\exists Y\ nat(Y) \wedge le(Y, s(s(s(s(s(0)))))) \wedge nat\_square(Y, X)))}$

    With this we express precisely our specification that "X is an output for the program if there exists a natural number Y such that X is the square of Y and Y is less or equal than 5"

Alternative Use of Logic?

- So, logic allows us to *represent / specify problems* (program specification).
- But, can we go further?
  - ◇ Given a specification,
  - ◇ can we obtain an implementation,
  - ◇ that is CORRECT 100% of the time?

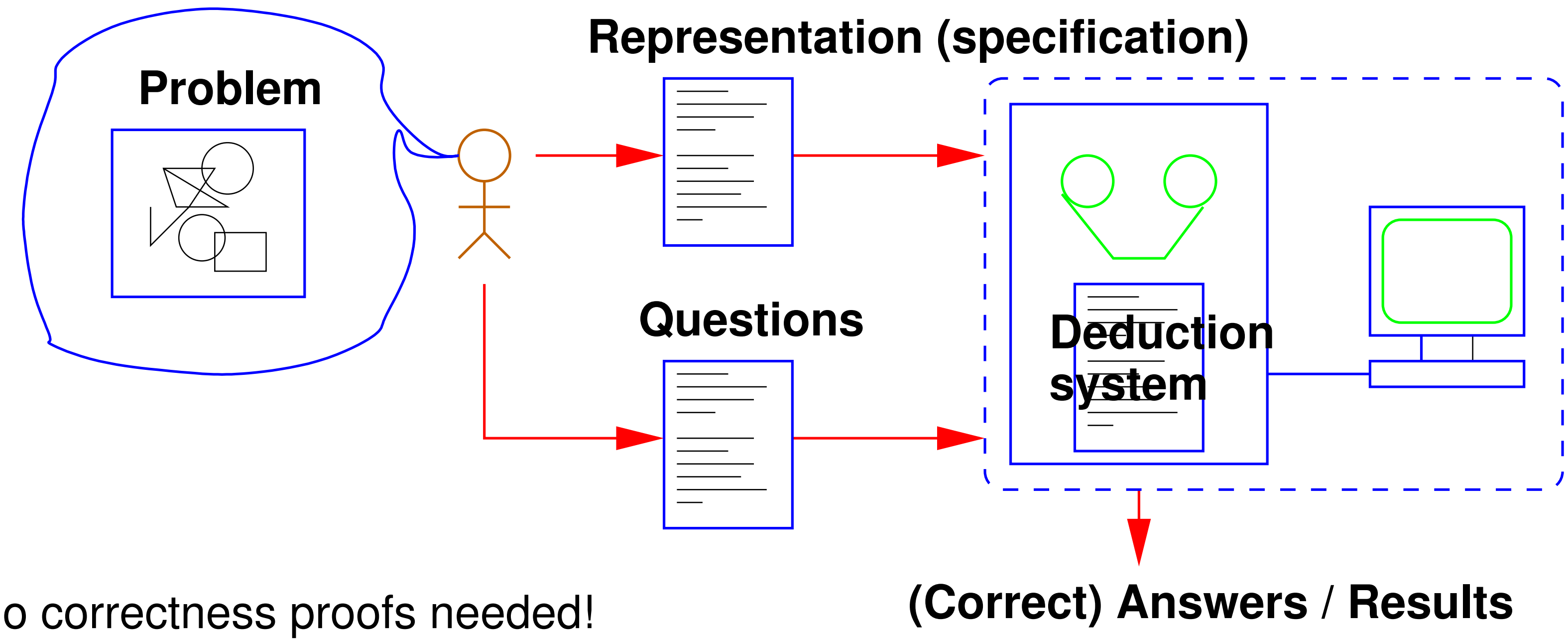
(Some relevant context – 1960's: great techniques and results were being obtained for automated theorem proving!)



- More concretely, can logic help here too?

From Representation/Specification to Computation

- Assuming the existence of a *mechanical proof method* (deduction procedure) *a new view of problem solving and computing is possible* [Green]:
  1. Program once and for all this deduction procedure in the computer.
  2. Then, given a problem, you just need to:
     a) Give a suitable *representation* for it to the computer (i.e., its *specification*),
     b) To obtain solutions, ask questions and let the deduction procedure do rest.



- No correctness proofs needed!

Computing With Our Previous Problem Description (Specification)

So, assuming such a proof method exists, mechanical and providing answers, then:

If we ask:	**We should obtain:**
$nat(s(0))$ **?**	yes
$\exists X\ add(s(0), s(s(0)), X)$ **?**	$X = s(s(s(0)))$
$\exists X\ add(s(0), X, s(s(s(0))))$ **?**	$X = s(s(0))$
$\exists X\ nat(X)$ **?**	$X = 0 \vee X = s(0) \vee X = s(s(0)) \vee \ldots$
$\exists X \exists Y\ add(X, Y, s(0))$ **?**	$(X = 0 \wedge Y = s(0)) \vee (X = s(0) \wedge Y = 0)$
$\exists X\ nat\_square(s(s(0)), X)$ **?**	$X = s(s(s(s(0))))$
$\exists X\ nat\_square(X, s(s(s(s(0)))))$ **?**	$X = s(s(0))$
$\exists X \exists Y\ nat\_square(X, Y)$ **?**	$(X = 0 \wedge Y = 0) \vee (X = s(0) \wedge Y = s(0)) \vee (X = s(s(0)) \wedge Y = s(s(s(s(0))))) \vee \ldots$
$\exists X output(X)$ **?**	$X = 0 \vee X = s(0) \vee X = s(s(s(s(0)))) \vee X = s^9(0) \vee X = s^{16}(0) \vee X = s^{25}(0)$

But, before going any further... which Logic should we use?

- So far we have already argued the convenience of *representing the problem specification in logic*, and seen a *new view of problem solving and computing*.
- This brings in new questions:
  - ◇ Does there indeed exist such a proof procedure?
    - * truth tables?
    - * natural deduction?
    - * resolution?
    - * Prawitz/Bibel, tableaux?
    - * bottom-up fixpoint?
    - * rewriting?
    - * narrowing? etc.
  - ◇ Which type of logic can we use?
    - * propositional?
    - * predicate calculus (first order)?
    - * higher-order logics?
    - * modal logics?
    - * λ-calculus?, etc.

Issues and tradeoffs

- We would like to **maximize expressive power**.
  Example: propositions vs. first-order formulas.
  - *Propositional* logic is useful:
    "spot is a dog" p
    "dogs have tail" q
    but limited: we can say that $p \wedge q$ is true, but not conclude that Spot has a tail.
  - *Predicate* logic extends the *expressive power* of propositional logic:
    $dog(spot)$
    $\forall X dog(X) \rightarrow has\_tail(X)$
    Now, using deduction we can conclude:
    $has\_tail(spot)$
- But we need to ensure that we have the **effective mechanical proof procedure** for that logic.

→ We recall some of the underlying properties and theoretical limits.
(Not even the most advanced AI can escape from them!)

Comparison of Logics (I)

- **Propositional** logic p – "spot is a dog" :
  - \+ decidable $\approx$ "we can prove everything to be true or false"
  - \+ we have practical deduction mechanisms (e.g., truth tables)
  - \- limited expressive power (not "Turing complete," cannot express all programs)

  $\rightarrow$ Very useful for, e.g., circuit design, "answer set" programming, ...

Comparison of Logics (I)

- **Propositional** logic p – "spot is a dog" :
  - \+ decidable ≈ "we can prove everything to be true or false"
  - \+ we have practical deduction mechanisms (e.g., truth tables)
  - \- limited expressive power (not "Turing complete," cannot express all programs)
  - $\rightarrow$ Very useful for, e.g., circuit design, "answer set" programming, ...
- **Predicate** logic (first order) ($\forall X dog(X) \rightarrow has\_tail(X)$):
  - \+ good expressive power (Turing complete, we can express "all programs")
  - \+ practical deduction mechanism (e.g., resolution)
  - +/- "semi"-decidable ≈ "we can always prove valid things in finite steps; but proving that things are not valid may sometimes not terminate"
    (this cannot be avoided: akin to the halting problem / program termination).

Comparison of Logics (I)

- **Propositional** logic p – "spot is a dog" :
  - \+ decidable ≈ "we can prove everything to be true or false"
  - \+ we have practical deduction mechanisms (e.g., truth tables)
  - \- limited expressive power (not "Turing complete," cannot express all programs)
  - → Very useful for, e.g., circuit design, "answer set" programming, ...
- **Predicate** logic (first order) ($\forall X dog(X) \rightarrow has\_tail(X)$):
  - \+ good expressive power (Turing complete, we can express "all programs")
  - \+ practical deduction mechanism (e.g., resolution)
  - +/- "semi"-decidable ≈ "we can always prove valid things in finite steps; but proving that things are not valid may sometimes not terminate" (this cannot be avoided: akin to the halting problem / program termination).

This leads to practical systems: "Definite Horn clauses" + SLD-resolution:
→ **Logic Programming** (LP) – this course!

If we add constraints (in place of unification):
→ **Constraint Logic Programming** (CLP) – also this course!

Comparison of Logics (II)

Many other logics and variants:

- **Higher-order predicate logic**:

  "X is some relationship for spot" *X(spot)*

  \+ great expressive power

  \- undecidable

  \- no general deduction mechanisms

→ But interesting subsets: HO logic programming, functional-logic programming, ...

- **Other logics**: Decidability? Expressive power? Practical deduction mechanism?

- Interesting case: λ**-calculus**

  \+ similar to predicate logic in results, allows higher order

  \- does not support general predicates (relations), only functions

→ Functional programming!

Generating squares by SLD-Resolution – Logic Programming (I)

(This slide if you have seen previously resolution, otherwise you can skip.)

- We code the problem as definite (Horn) clauses:

$$nat(0)$$
$$\neg nat(X) \vee nat(s(X))$$
$$\neg nat(X) \vee add(0, X, X))$$
$$\neg add(X, Y, Z) \vee add(s(X), Y, s(Z))$$
$$\neg nat(X) \vee mult(0, X, 0)$$
$$\neg mult(X, Y, W) \vee \neg add(W, Y, Z) \vee mult(s(X), Y, Z)$$
$$\neg nat(X) \vee \neg nat(Y) \vee \neg mult(X, X, Y) \vee nat\_square(X, Y)$$

- **Query:** $nat(s(0))$ **?**
  - ◇ In order to refute: $\neg nat(s(0))$
  - ◇ Resolution:

$\neg nat(s(0))$	and	$\neg nat(X) \vee nat(s(X))$	with unifier	$\{X/0\}$	giving	$\neg nat(0)$
$\neg nat(0)$	and	$nat(0)$	with unifier	$\{\}$	giving	$\square$

  - ◇ Answer: (yes)

Generating squares by SLD-Resolution – Logic Programming (II)

(This slide if you have seen previously resolution, otherwise you can skip.)

- We code the problem as definite (Horn) clauses:

$nat(0)$
$\neg nat(X) \vee nat(s(X))$
$\neg nat(X) \vee add(0, X, X))$
$\neg add(X, Y, Z) \vee add(s(X), Y, s(Z))$
$\neg nat(X) \vee mult(0, X, 0)$
$\neg mult(X, Y, W) \vee \neg add(W, Y, Z) \vee mult(s(X), Y, Z)$
$\neg nat(X) \vee \neg nat(Y) \vee \neg mult(X, X, Y) \vee nat\_square(X, Y)$

- **Query:** $\exists X \exists Y \; add(X, Y, s(0))$ **?**
  - ◇ In order to refute: $\neg add(X, Y, s(0))$
  - ◇ Resolution:
    $\neg add(X, Y, s(0))$ and $\neg nat(X) \vee add(0, X, X))$ with unifier $\{X = 0, Y = s(0)\}$
    giving $\neg nat(s(0))$ (and $\neg nat(s(0))$ is resolved as before)
  - ◇ Answer: $X = 0, Y = s(0)$
  - ◇ Alternative:
    $\neg add(X, Y, s(0))$ with $\neg add(X, Y, Z) \vee add(s(X), Y, s(Z))$ giving $\neg add(X, Y, 0) \ldots$

Generating Squares in a Practical Logic Programming System (I)

Prolog systems implement precisely that: SLD-resolution for programs that are written in logic (using Horn clauses). Let's encode our specification in Prolog syntax...

```
nat(0).
nat(s(X)) :- nat(X).

le(0,X) :- nat(X).
le(s(X),s(Y)) :- le(X,Y).

add(0,Y,Y) :- nat(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,Y,0) :- nat(Y).
mult(s(X),Y,Z) :- add(W,Y,Z), mult(X,Y,W).

nat_square(X,Y) :- nat(X), nat(Y), mult(X,X,Y).

output(X) :- nat(Y), le(Y,s(s(s(s(s(0)))))), nat_square(Y,X).
```

... and run it on a Prolog system using using (breadth first) resolution: run example $\longmapsto$

What should we expect, given the theoretical results for first-order logic?
$\rightarrow$ to get all answers; and, if no (more) answers: we will get 'no' or not terminate.

Generating Squares in a Practical Logic Programming System (II)

run example ⟼

Query	Answer
?- `nat(s(0)).`	`yes`
?- `add(s(0),s(s(0)),X).`	`X = s(s(s(0)))`
?- `add(s(0),X,s(s(s(0)))).`	`X = s(s(0))`
?- `nat(X).`	`X = 0 ; X = s(0) ; X = s(s(0)) ; ...`
?- `add(X,Y,s(0)).`	`(X = 0 , Y=s(0)) ; (X = s(0) , Y = 0)`
?- `nat_square(s(s(0)), X).`	`X = s(s(s(s(0))))`
?- `nat_square(X,s(s(s(s(0))))).`	`X = s(s(0))`
?- `nat_square(X,Y).`	`(X = 0 , Y=0) ; (X = s(0) , Y=s(0)) ; (X = s(s(0)) , Y=s(s(s(s(0))))) ; ...`
?- `output(X).`	`X = 0 ; X = s(0) ; X = s(s(s(s(0)))) ; ...`

Applications (I)

- Many applications:
  - Natural language processing
  - Scheduling/Optimization problems
  - Many AI-related problems, (Multi) agent programming
  - Heterogeneous data integration
  - Program analyzers and verifiers
  - ...

  Many in combination with other languages.

- Some examples:
  - The IBM Watson System (2011) has important parts written in Prolog.
  - Clarissa, a voice user interface by NASA for browsing ISS procedures.
  - The first Erlang interpreter was developed in Prolog by Joe Armstrong.
  - The Microsoft Windows NT Networking Installation and Configuration system.
  - The Ericsson Network Resource Manager (NRM).
  - "A flight booking system handling nearly a third of all airline tickets in the world" (SICStus).
  - The java abstract machine specification is written in Prolog.
  - ...

Applications (II)

The IBM Watson system (from WikipediA):

> *"Watson is a question-answering computer system capable of answering questions posed in natural language, developed in IBM's DeepQA project... it competed on Jeopardy! against champions Brad Rutter and Ken Jennings, winning the first place prize of $1 million."*

Adam Lally, John M. Prager, Michael C. McCord, Branimir Boguraev, Siddharth Patwardhan, James Fan, Paul Fodor, Jennifer Chu-Carroll: *Question analysis: How Watson reads a clue.* IBM J. Res. Dev. 56(3): 2:

> *"Prior to our decision to use Prolog for this task, we had implemented custom pattern-matching frameworks over parses. These frameworks ended up replicating some of the features of Prolog but lacked the full feature set of Prolog or the efficiency of a good Prolog implementation. Using Prolog for this task has significantly improved our productivity in developing new pattern-matching rules and has delivered the execution efficiency necessary to be competitive in a Jeopardy! game."*

A (very brief) History of Logic Programming (I)

- **60's**
  - ◇ Green: **programming as problem solving**.
  - ◇ Robinson: **resolution**.
- **70's**
  - ◇ Kowalski: **SLD resolution** (very efficient).
  - ◇ Colmerauer: **Prolog** ("Programmation et Logique"). Interpreter in Fortran.
  - ◇ Kowalski: **procedural interpretation** of Horn clause logic. Read:
    A if B_1 and B_2 and $\cdots$ and B_n as:
    to solve (execute) A, solve (execute) B_1 and B_2 and,..., B_n
    Algorithm = logic + control.
  - ◇ D.H.D. Warren: develops first **compiler**, DEC-10 Prolog.
    * Almost completely written in Prolog.
    * Very efficient (same as Lisp).
    * Top-level, debugger, very useful builtins, ... becomes the standard.
- **80's, 90's**
  - ◇ Major research in the basic paradigms and advanced implementation techniques: Japan (Fifth Generation Project), US (MCC), Europe (ECRC, ESPRIT projects), leading to the current EU "framework research programs".

A (very brief) History of Logic Programming (II)

- **80's, 90's continued**
  - Numerous commercial Prolog implementations, programming books, using the *de facto* standard, the Edinburgh Prolog family.
  - Leading in **1995** to The ISO Prolog standard.
  - Parallel and concurrent logic programming systems.
  - Constraint Logic Programming (**CLP**): A major extension – opened new areas and even communities:
    - Commercial CLP systems with fielded applications.
    - Concurrent constraint programming systems.
- **2000-...**
  - Many other extensions: full higher order, support for types/modes, concurrency, distribution, objects, functional syntax, ...
  - Highly optimizing compilers, automatic, automatic parallelism, automatic verification and debugging, advanced environments.
  - Many applications

  Variations: Datalog, Answer Set Programming (ASP – support negation through stable models).
- **Today**
  "Verse" logic/functional language at **Epic Games** (Fortnite), Logica language at **Google**, ...