Programming and Computational Logic
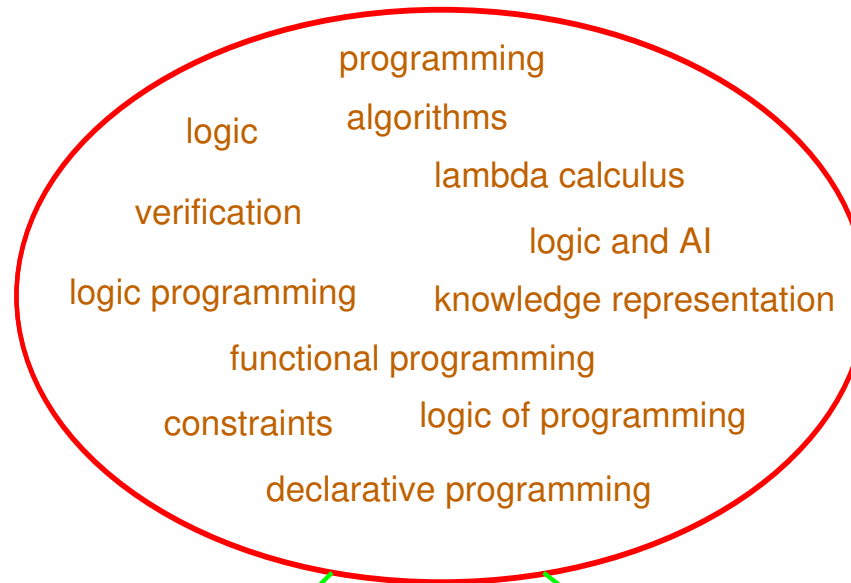# A Motivational Introduction

The following people have contributed to this course material:

Manuel Hermenegildo (editor), Technical University of Madrid, Spain and University of New Mexico, USA; *Francisco Bueno, Manuel Carro, Pedro López, and Daniel Cabeza,* Technical University of Madrid, Spain; *María José García de la Banda,* Monash University, Australia; *David H. D. Warren,* University of Bristol, U.K.; *Ulrich Neumerkel,* Technical University of Vienna, Austria; *Michael Codish,* Ben Gurion University, Israel

Computational Logic

programming

logic   algorithms

lambda calculus

verification

logic and AI

logic programming   knowledge representation

functional programming

constraints   logic of programming
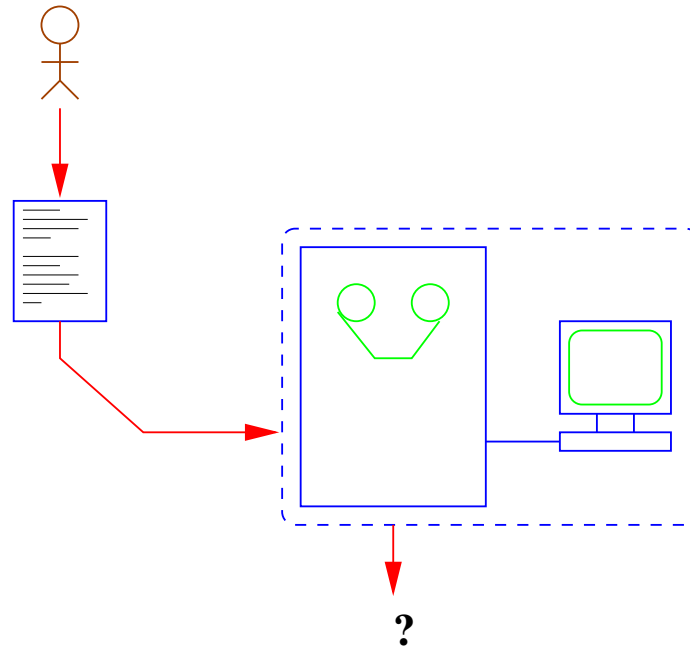
declarative programming

Logic of Computation
program verification
proving properties

Declarative Programming
direct use of logic
as a programming tool

The Program Correctness Problem

- Conventional models of using computers – not easy to determine correctness!

  ◇ Has become a very important issue, not just in safety-critical apps.
  ◇ Components with assured quality, being able to give a warranty, ...
  ◇ Being able to run untrusted code, certificate carrying code, ...

A Simple Imperative Program

• Example:

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

• Is it correct? With respect to what?

• A suitable formalism:

   ◇ to provide *specifications* (describe problems), and

   ◇ to reason about the *correctness of programs* (their *implementation*).

   is needed.

Natural Language

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

Ideal at first sight, but:

- ◇ verbose
- ◇ vague
- ◇ ambiguous
- ◇ needs context (assumed information)
- ◇ ...

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

Logic

- A means of clarifying / formalizing the human thought process

- Logic for example tells us that (classical logic)
  Aristotle likes cookies, and
  Plato is a friend of anyone who likes cookies
  imply that
  Plato is a friend of Aristotle

- Symbolic logic:
  A shorthand for classical logic – plus many useful results:
  $a_1 : likes(aristotle, cookies)$
  $a_2 : \forall X\ likes(X, cookies) \rightarrow friend(plato, X)$
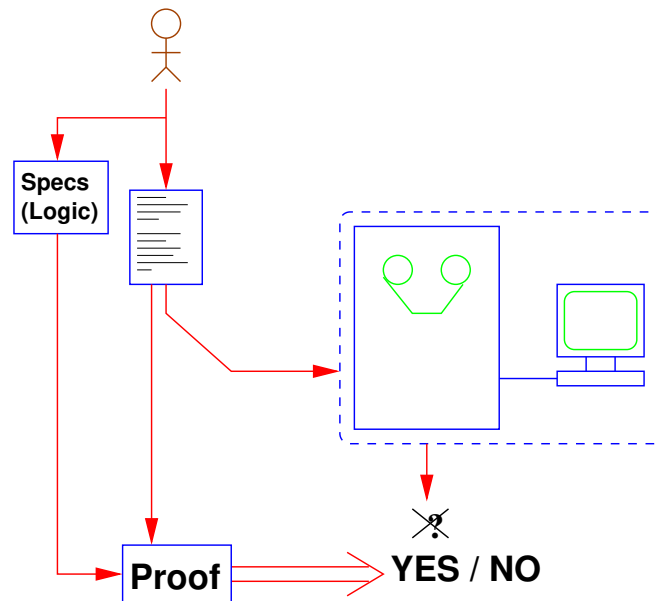 $t_1 : friend(plato, aristotle)$
  $T[a_1, a_2] \vdash t_1$

- But, can logic be used:

  ⬦ To represent the problem (specifications)?
  ⬦ *Even perhaps to solve the problem?*

Using Logic

- For expressing specifications and reasoning about the correctness of programs we need:

    ◇ Specification languages (assertions), modeling, ...

    ◇ Program semantics (models, axiomatic, denotational, fixpoint, ...).

    ◇ Proofs: program *verification* (and debugging, equivalence, ...).

Generating Squares: A Specification (I)

Numbers —we will use "Peano" representation for simplicity:

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

- Defining the natural numbers:

  0 is a natural, $\qquad$ 1 is a natural, $\qquad$ 2 is a natural, $\qquad$ . . .
  $nat(0)$ $\qquad$ $\wedge$ $nat(s(0))$ $\qquad$ $\wedge$ $nat(s(s(0)))$ $\qquad$ $\wedge$ . . .

  ◇ A better solution: $\boxed{nat(0) \quad \wedge \quad \forall X \ (nat(X) \rightarrow nat(s(X)))}$

Generating Squares: A Specification (I)

Numbers —we will use "Peano" representation for simplicity:

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

- Defining the natural numbers:

0 is a natural, $\qquad$ 1 is a natural, $\qquad$ 2 is a natural, $\qquad$ . . .

$nat(0) \qquad \wedge \quad nat(s(0)) \qquad \wedge \quad nat(s(s(0))) \qquad \wedge \quad . . .$

$\diamond$ A better solution: $\boxed{nat(0) \quad \wedge \quad \forall X \ (nat(X) \rightarrow nat(s(X)))}$

- Order on the naturals (less or equal than):

$le(0,0) \qquad le(0,s(0)) \qquad le(0,s(s(0))) \qquad . . .$

$le(s(0),s(0)) \quad le(s(0),s(s(0))) \quad le(s(0),s(s(s(0)))) \quad . . .$

$. . . \qquad\qquad . . . \qquad\qquad . . . \qquad\qquad . . .$

$\boxed{\forall X \ (nat(X) \rightarrow le(0, \ X)) \qquad \wedge \qquad \forall X \forall Y \ (le(X,Y) \rightarrow le(s(X),s(Y)))}$

Generating Squares: A Specification (I)

Numbers —we will use "Peano" representation for simplicity:

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

- Defining the natural numbers:

  0 is a natural, $\qquad$ 1 is a natural, $\qquad$ 2 is a natural, $\qquad$ ...

  $nat(0)$ $\qquad$ $\wedge$ $nat(s(0))$ $\qquad$ $\wedge$ $nat(s(s(0)))$ $\qquad$ $\wedge$ ...

  ◇ A better solution: $\boxed{nat(0) \quad \wedge \quad \forall X \ (nat(X) \rightarrow nat(s(X)))}$

- Order on the naturals (less or equal than):

  $le(0,0)$ $\qquad$ $le(0, s(0))$ $\qquad$ $le(0, s(s(0)))$ $\qquad$ ...

  $le(s(0), s(0))$ $\quad$ $le(s(0), s(s(0)))$ $\quad$ $le(s(0), s(s(s(0))))$ $\quad$ ...

  ... $\qquad$ ... $\qquad$ ... $\qquad$ ...

  $\boxed{\forall X \ (nat(X) \rightarrow le(0, \ X)) \quad \wedge \quad \forall X \forall Y \ (le(X,Y) \rightarrow le(s(X), s(Y)))}$

- Addition of naturals:

  $add(0,0,0)$ $\qquad$ $add(0, s(0), s(0))$ $\qquad$ $add(0, s(s(0)), s(s(0)))$ $\qquad$ ...

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

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

Generating Squares: A Specification (II)

- Multiplication of naturals:

  "Multiply X and Y" is "add Y to itself X times," e.g.

  $mult(3, 2, 6) \equiv$ "three times 2 is 6" (adding 2, 3 times is 6)$\equiv 2 + 2 + 2 = 6$

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

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

- Squares of the naturals:

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

Generating Squares: A Specification (II)

- Multiplication of naturals:

  "Multiply X and Y" is "add Y to itself X times," e.g.

  $mult(3, 2, 6) \equiv$ "three times 2 is 6" (adding 2, 3 times is 6)$\equiv 2 + 2 + 2 = 6$

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

  $$\boxed{\begin{array}{l} \forall X \ (nat(X) \rightarrow mult(0, X, 0)) \ \land \\ \forall X \forall Y \forall Z \forall W \ (mult(X, Y, W) \land add(W, Y, Z) \rightarrow mult(s(X), Y, Z)) \end{array}}$$

- Squares of the naturals:

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

We can now write a *specification* of the (imperative) program, i.e., conditions that we want the program to meet:
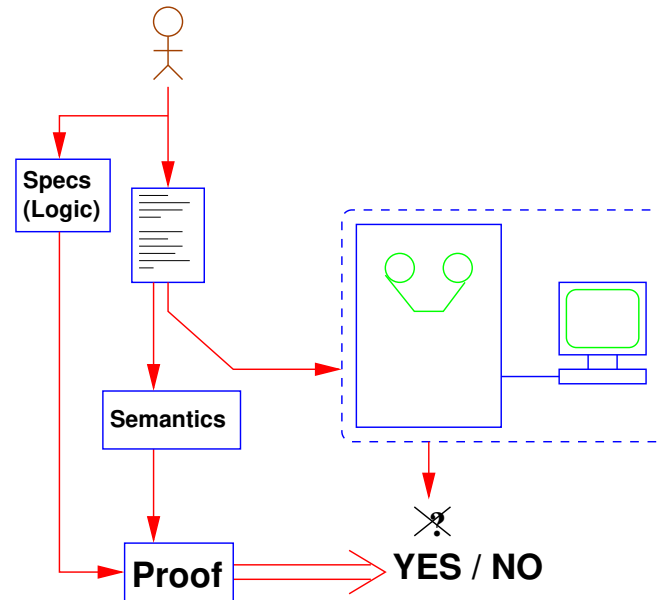
- *Precondition (empty):*
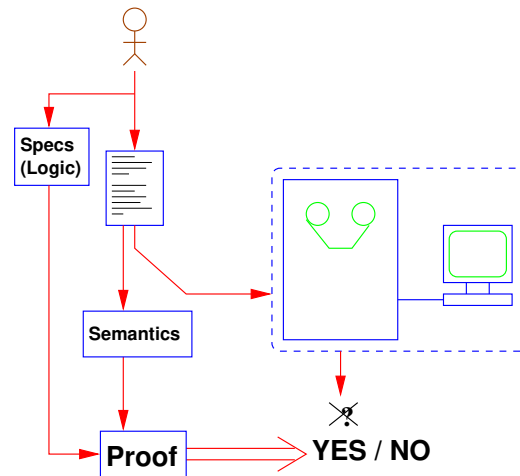
  $$\boxed{true}$$

- *Postcondition:*

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

Use of Logic

- For expressing specifications and reasoning about the correctness of programs we need:

  ◇ Specification languages (assertions), modeling, ...

  ◇ Program semantics (models, axiomatic, fixpoint, ...).

  ◇ Proofs: program *verification* (and debugging, equivalence, ...).

Semantic Tasks

- Semantics:

  ◇ A *semantics* associates a meaning (a mathematical object) to a program or program sentence.

- Semantic tasks:

  ◇ Verification: proving that a program meets its specification.

  ◇ Static debugging: finding where a program does not meet specifications.

  ◇ Program equivalence: proving that two programs have the same semantics.

  ◇ etc.

Styles of Semantics

- **Operational:**

  The meaning of program sentences is defined in terms of the steps (transformations from state to state) that computations may take during execution (derivations). Proofs by induction on derivations.

- **Axiomatic:**

  The meaning of program sentences is defined indirectly in terms some axioms and rules of some logic of program properties.

- **Denotational (fixpoint):**

  The meaning of program sentences is given abstractly as elements of some suitable mathematical structure (domain).

- **Model (declarative) semantics:**

  The meaning of programs is given as a minimal model ("logical meaning") of the logic that the program is written in.

Alternative Use of Logic?

- So, logic allows us to *represent problems* (program specifications).


But, it would be interesting to also improve:
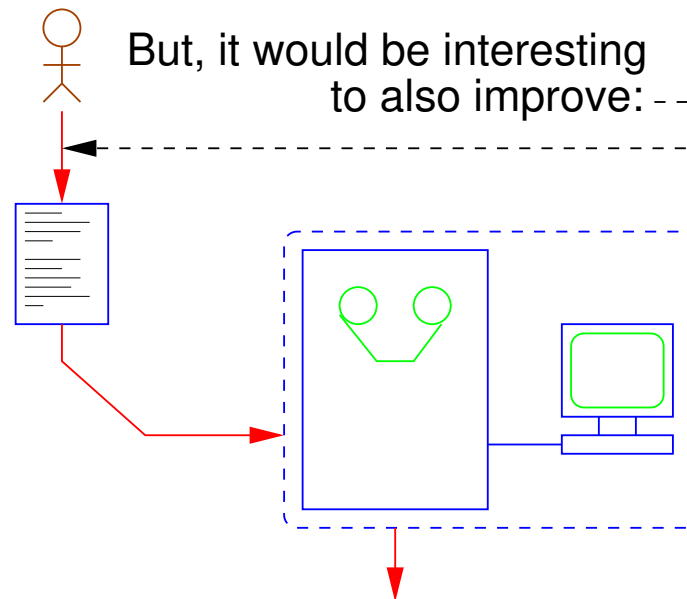
i.e., the process of implementing solutions to problems.

- The importance of Programming Languages (and tools).

- Interesting question: 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]:

  - ◇ program once and for all the deduction procedure in the computer,
  - ◇ find a suitable *representation* for the problem (i.e., the *specification*),
  - ◇ then, to obtain solutions, ask questions and let deduction procedure do rest:

Problem

Representation (specification)

Questions

Deduction system

(Correct) Answers / Results
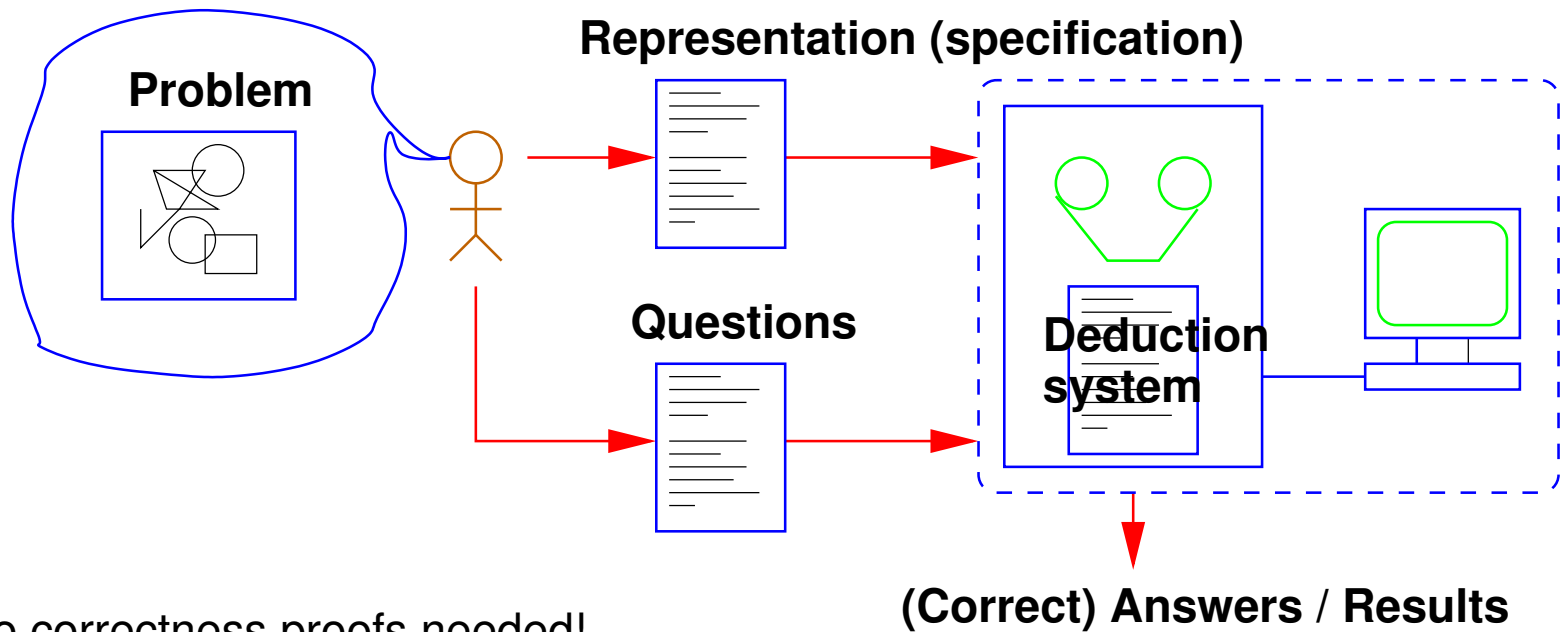
- No correctness proofs needed!

Computing With Our Previous Description / Specification

Query	Answer
$nat(s(0))$ **?**	(yes)
$\exists X\ add(s(0), s(s(0)), X)$ **?**	$X = s(s(s(0)))$
$\exists X\ add(s(0), X, s(s(s(0))))$ **?**	$X = s(s(0))$
$\exists X\ nat(X)$ **?**	$X = 0 \vee X = s(0) \vee X = s(s(0)) \vee \ldots$
$\exists X \exists Y\ add(X, Y, s(0))$ **?**	$(X = 0 \wedge Y = s(0)) \vee (X = s(0) \wedge Y = 0)$
$\exists X\ nat\_square(s(s(0)), X)$ **?**	$X = s(s(s(s(0))))$
$\exists X\ nat\_square(X, s(s(s(s(0)))))$ **?**	$X = s(s(0))$
$\exists X \exists Y\ nat\_square(X, Y)$ **?**	$(X = 0 \wedge Y = 0) \vee (X = s(0) \wedge Y = s(0)) \vee (X = s(s(0)) \wedge Y = s(s(s(s(0))))) \vee \ldots$
$\exists X output(X)$ **?**	$X = 0\ \vee\ X = s(0)\ \vee\ X = s(s(s(s(0))))\ \vee\ X = s^9(0)\ \vee\ X = s^{16}(0)\ \vee\ X = s^{25}(0)$

14

Which Logic?

- We have already argued the convenience of representing the problem in logic, but

    - ◇ which logic?
        - * propositional
        - * predicate calculus (first order)
        - * higher-order logics
        - * modal logics
        - * λ-calculus, ...

    - ◇ which reasoning procedure?
        - * natural deduction, classical methods
        - * resolution
        - * Prawitz/Bibel, tableaux
        - * bottom-up fixpoint
        - * rewriting
        - * narrowing, ...

Issues

- We try to maximize expressive power.

- But one of the main issues is whether we have an **effective** reasoning procedure.

- It is important to understand the underlying properties and the theoretical limits!

- Example: propositions vs. first-order formulas.

  ◇ Propositional logic:

"spot is a dog"	*p*
"dogs have tail"	*q*

  but how can we 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:

  $$has\_tail(spot)$$

Comparison of Logics (I)

- Propositional logic:

    "spot is a dog" p
    + decidability/completeness
    - limited expressive power
    + practical deduction mechanism

    $\rightarrow$ circuit design, "answer set" programming, ...

- Predicate logic: (first order)

    "spot is a dog" $dog(spot)$
    +/- decidability/completeness
    +/- good expressive power
    + practical deduction mechanism (e.g., **SLD-resolution**)

    $\rightarrow$ classical logic programming!

Comparison of Logics (II)

- Higher-order predicate logic:

    "There is a relationship for spot"   *X(spot)*
    - decidability/completeness
    + good expressive power
    – practical deduction mechanism

    But interesting subsets $\rightarrow$ HO logic programming, functional-logic programming,
    ...

- Other logics: decidability? Expressive power? Practical deduction mechanism?
  Often (very useful) variants of previous ones:

    $\diamond$ Predicate logic + constraints (in place of unification)
        $\rightarrow$ constraint programming!
    $\diamond$ Propositional temporal logic, etc.

- Interesting case: λ-calculus

        + similar to predicate logic in results, allows higher order
        - does not support predicates (relations), only functions

    $\rightarrow$ functional programming!

- 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:
  $\neg nat(s(0))$ with $\neg nat(X) \vee nat(s(X))$ gives $\neg nat(0)$
  $\neg nat(0)$ with $nat(0)$ gives $\square$

- Answer: (yes)

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

- **Query:** $\exists X \exists Y \; add(X, Y, s(0))$ **?**

- In order to refute: $\neg add(X, Y, s(0))$

- Resolution:
  $\neg add(X, Y, s(0))$ with $\neg nat(X) \vee add(0, X, X))$ gives $\neg nat(s(0))$
  $\neg nat(s(0))$ solved as before

- Answer: $X = 0, Y = s(0)$

- Alternative:
  $\neg add(X, Y, s(0))$ with $\neg add(X, Y, Z) \vee add(s(X), Y, s(Z))$ gives $\neg add(X, Y, 0)$

```
:- module(_,_,['bf/af']).


nat(0) <- .
nat(s(X)) <- nat(X).


le(0,_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).
```

Generating Squares in a Practical Logic Programming System (II)

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."