

# Programming and Computational Logic

## A Motivational Introduction

---

**The following people have contributed to this course material:**

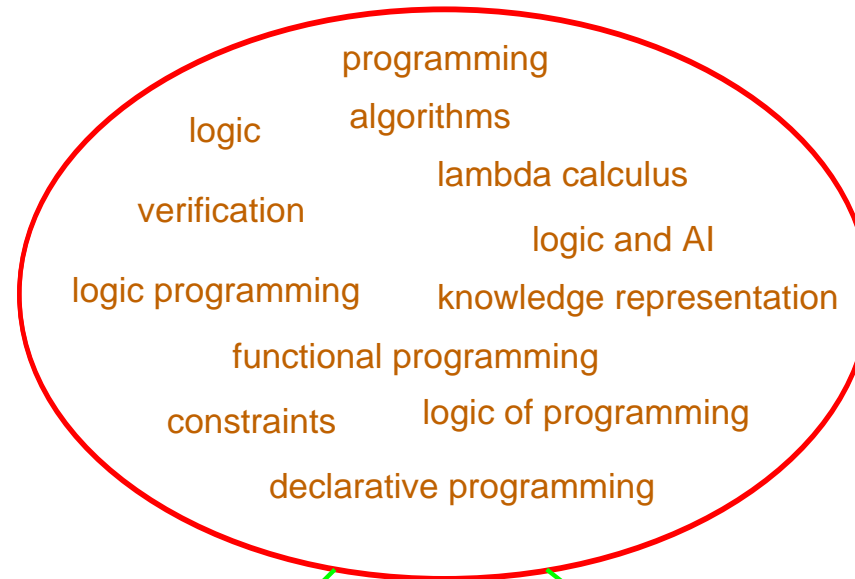
*Manuel Hermenegildo (editor), Technical University of Madrid, Spain and University of New Mexico, USA; Francisco Bueno, Manuel Carro, Pedro López, and Daniel Cabeza, Technical University of Madrid, Spain; María José García de la Banda, Monash University, Australia; David H. D. Warren, University of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

---

## Course General Topic

---

### Computational Logic



#### Logic of Computation

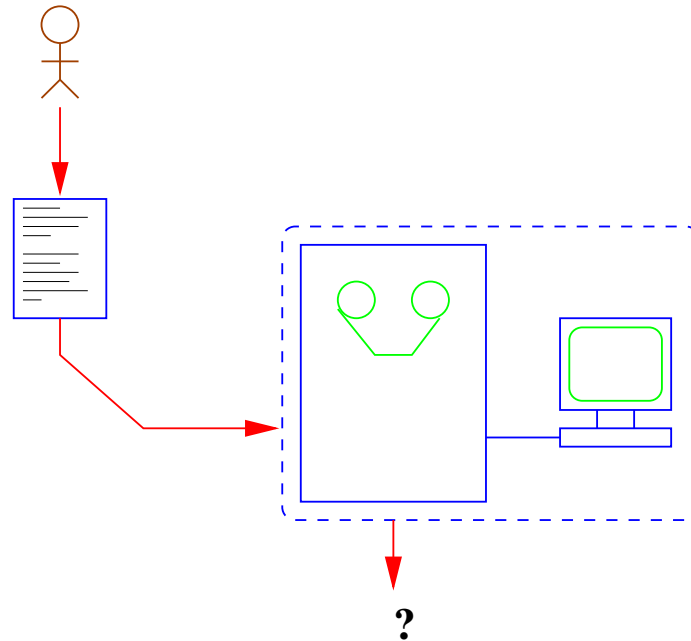
program verification  
proving properties

#### 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...

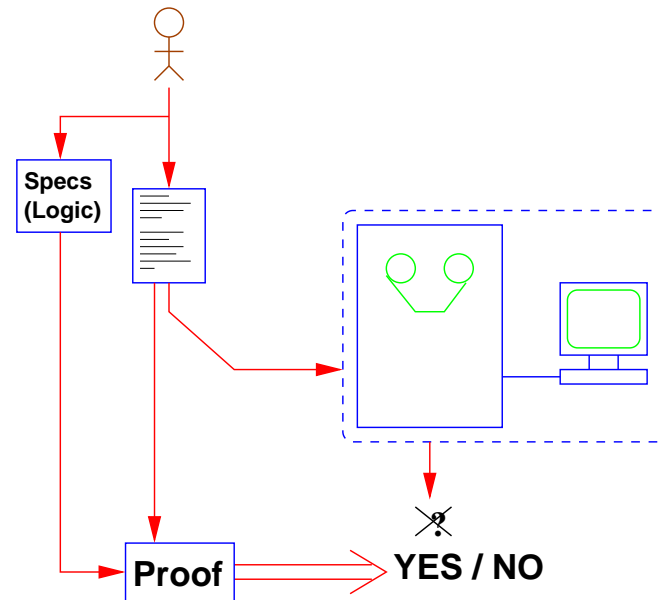
## Logic

---

- 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:  
 $a_1 : \text{likes}(\text{aristotle}, \text{cookies})$   
 $a_2 : \forall X \text{ likes}(X, \text{cookies}) \rightarrow \text{friend}(\text{plato}, X)$   
 $t_1 : \text{friend}(\text{plato}, \text{aristotle})$   
 $T[a_1, a_2] \vdash t_1$
- 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, fixpoint, ...).
  - ◇ Proofs: program *verification* (and debugging, equivalence, ...).

## Generating Squares: A Specification (I)

---

Numbers —we will use “Peano” representation for simplicity:

$0 \rightarrow 0$        $1 \rightarrow s(0)$        $2 \rightarrow s(s(0))$        $3 \rightarrow s(s(s(0)))$       ...

- Defining the natural numbers:

$$nat(0) \wedge nat(s(0)) \wedge nat(s(s(0))) \wedge \dots$$

- A better solution:

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

- Order on the naturals:

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

- Addition of naturals:

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



## Generating Squares: A Specification (II)

---

- Multiplication of naturals:

$$\forall X (nat(X) \rightarrow mult(0, X, 0)) \wedge$$

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

- Squares of the naturals:

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

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

- *Precondition:*

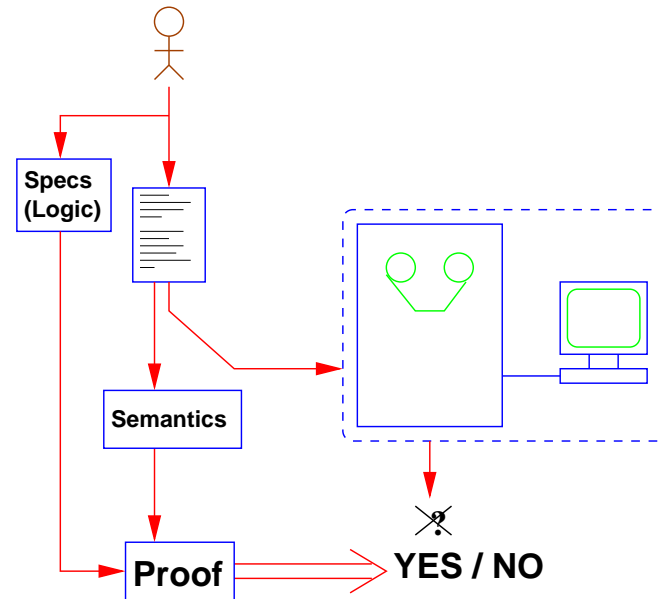
empty.

- *Postcondition:*

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

## Use of Logic

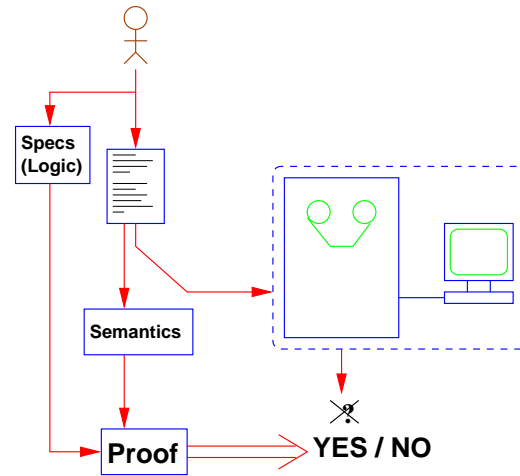
---



- 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).

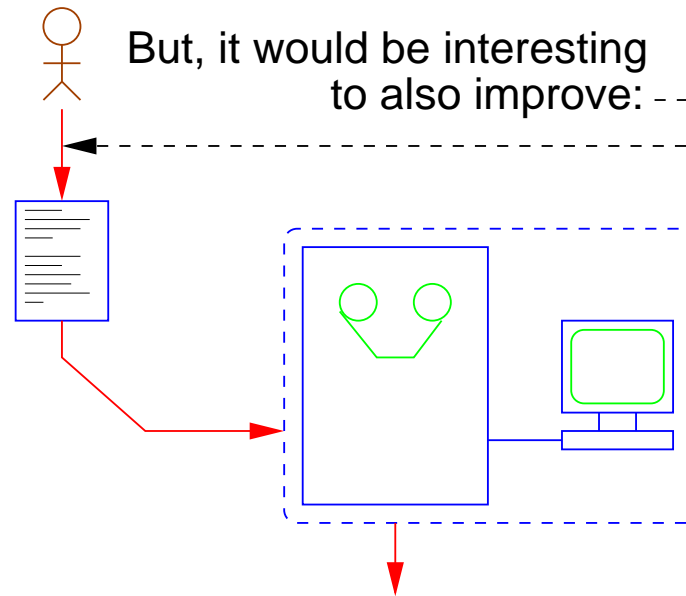
- **Model (declarative) semantics:**

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).

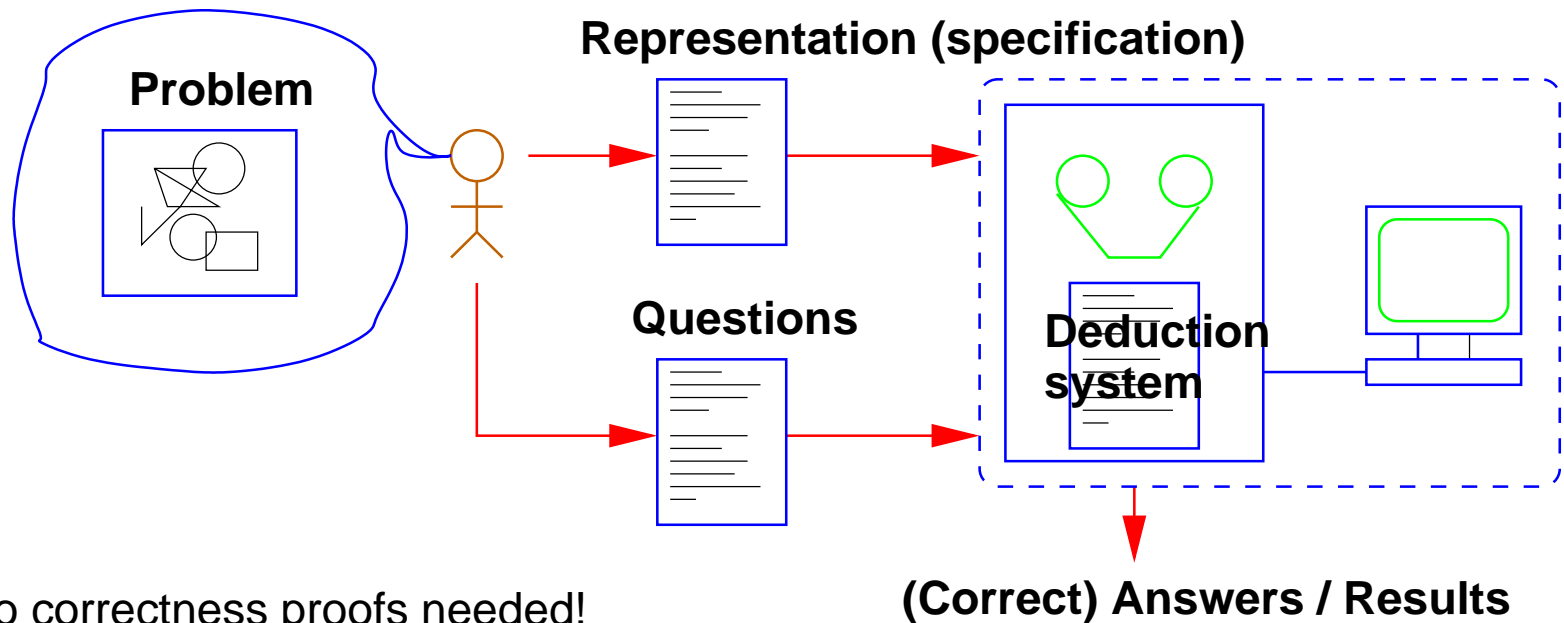


i.e., the process of implementing solutions to problems.

- 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 [Greene]:
  - ◇ 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:



- No correctness proofs needed!

## Computing With Our Previous Description / Specification

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

## 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
    - \*  $\lambda$ -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

→ 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**)

→ classical logic programming!

## Comparison of Logics (II)

---

- Higher-order predicate logic:

“There is a relationship for spot”

$X(\text{spot})$

- decidability/completeness
- + good expressive power
- practical deduction mechanism

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

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

- ◇ Predicate logic + constraints (in place of unification)  
→ constraint programming!
- ◇ Propositional temporal logic, etc.

- Interesting case:  $\lambda$ -calculus

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

→ functional programming!

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

---

- 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*)

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

---

$nat(0)$

$\neg nat(X) \vee nat(s(X))$

$\neg nat(X) \vee add(0, X, X)$

$\neg add(X, Y, Z) \vee add(s(X), Y, s(Z))$

$\neg nat(X) \vee mult(0, X, 0)$

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

$\neg nat(X) \vee \neg nat(Y) \vee \neg mult(X, X, Y) \vee nat\_square(X, Y)$

- **Query:**  $\exists X \exists Y \text{ add}(X, Y, s(0))$  ?
- **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  $\neg add(X, Y, 0)$

## Generating Squares in a Practical Logic Programming System (I)

---

```
:- 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)))) ; ...