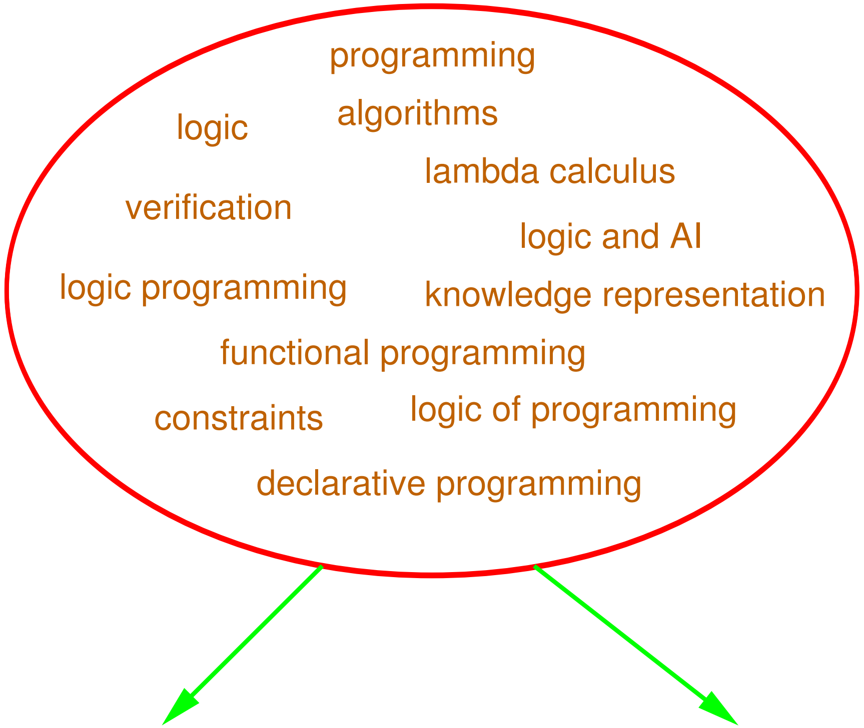
A Motivational Introduction to Computational Logic and (Constraint) Logic Programming

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



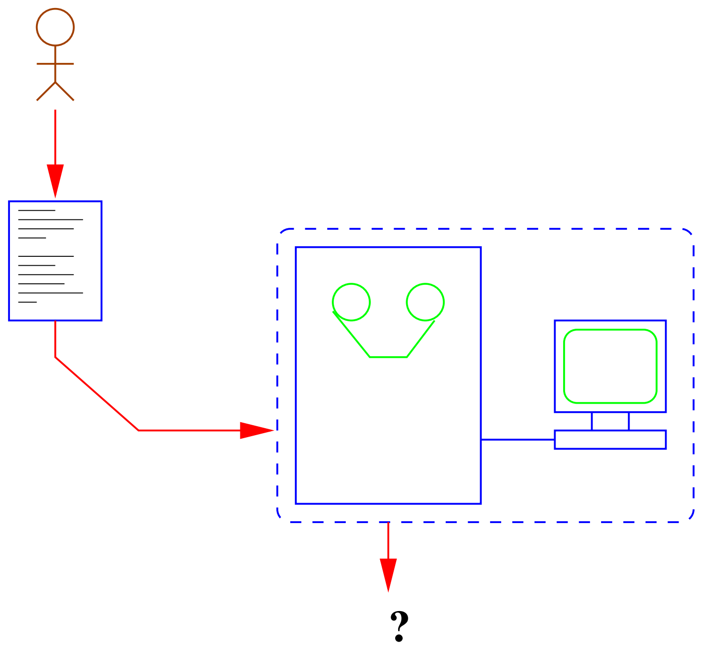
Logic of Computation

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:

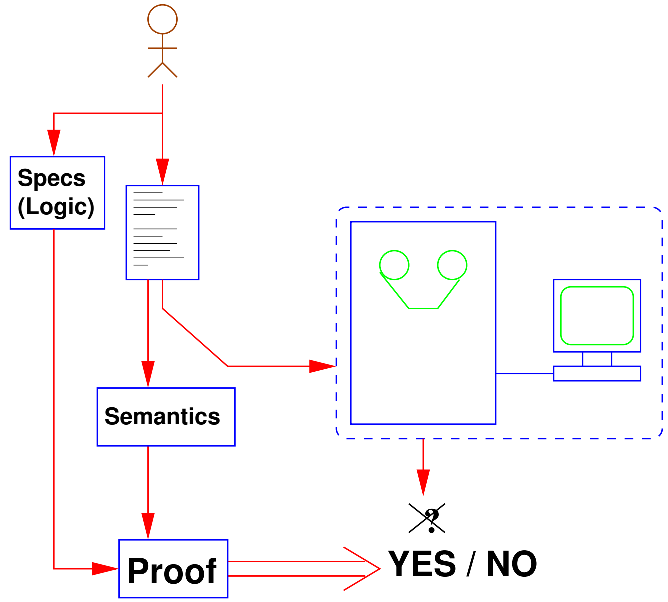
- verbose
- vague
- ambiguous
- needs context (assumed information)
- ...

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, 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:
  $nat(0) \wedge nat(s(0)) \wedge nat(s(s(0))) \wedge \ldots$
- 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)

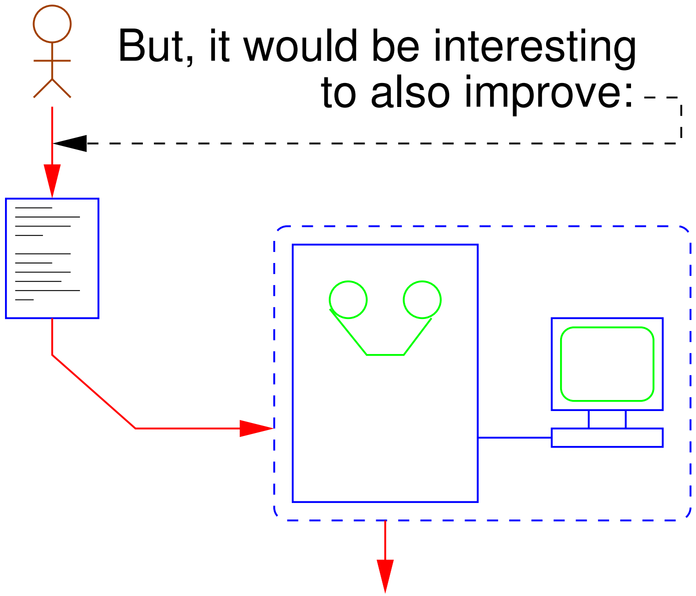
- Multiplication of naturals:
  $\forall X\ (nat(X) \rightarrow mult(0, X, 0)) \wedge$
  $\forall X \forall Y \forall Z \forall W\ (mult(X, Y, W) \wedge add(W, Y, Z) \rightarrow mult(s(X), Y, Z))$
- Squares of the naturals:
  $\forall X \forall Y\ (nat(X) \wedge nat(Y)\ \wedge mult(X, X, Y) \rightarrow nat\_square(X, Y))$

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

- *Precondition:*
  empty.
- *Postcondition:*
  $\forall X (output(X) \leftarrow (\exists Y\ nat(Y) \wedge le(Y, s(s(s(s(0))))) \wedge nat\_square(Y, X)))$

Alternative Use of Logic?

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

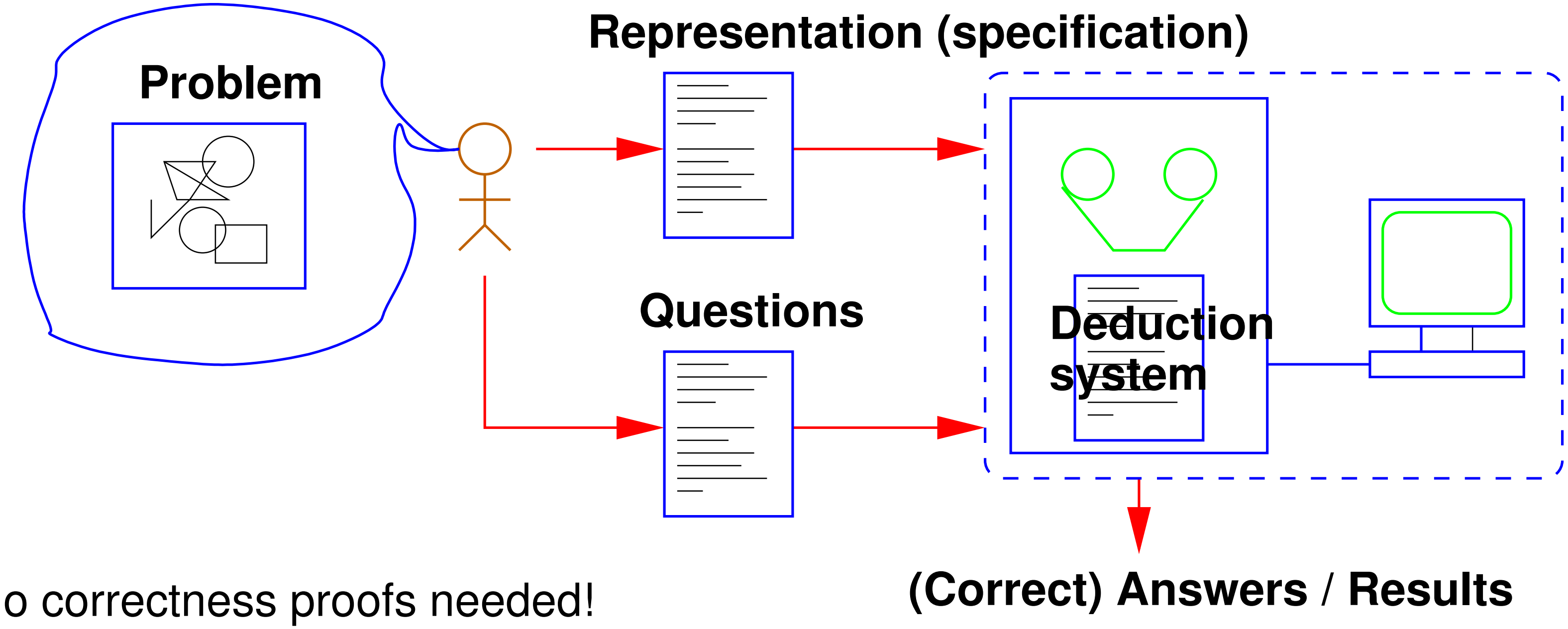


i.e., the process of implementing solutions to problems.

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

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

$\longrightarrow$ 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**)

$\longrightarrow$ 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 prog., ...

- Other logics: decidability? Expressive power? Practical deduction mechanism? Often (very useful) variants of previous ones:
  - ◇ Predicate logic + constraints (in place of unification) $\rightarrow$ constraint programming!
  - ◇ 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!

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

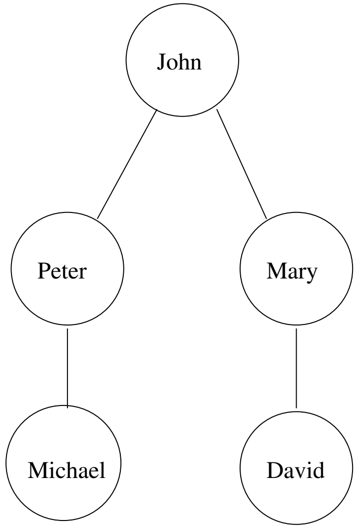
Introductory example (I) – Family relations

$father\_of(john, peter)$
$father\_of(john, mary)$
$father\_of(peter, michael)$
$mother\_of(mary, david)$
$\forall X \forall Y (\exists Z (father\_of(X, Z) \wedge father\_of(Z, Y)) \rightarrow grandfather\_of(X, Y))$
$\forall X \forall Y (\exists Z (father\_of(X, Z) \wedge mother\_of(Z, Y)) \rightarrow grandfather\_of(X, Y))$

```
father_of(john, peter).
father_of(john, mary).
father_of(peter, michael).
mother_of(mary, david).

grandfather_of(L,M) :- father_of(L,K),
                       father_of(K,M).

grandfather_of(X,Y) :- father_of(X,Z),
                       mother_of(Z,Y).
```



- How can `grandmother_of/2` be represented?
- What does `grandfather_of(X,david)` mean? And `grandfather_of(john,X)`?

Introductory example (II) - Testing membership in lists

- Declarative view:
  - ◇ Suppose there is a functor $f/2$ such that $f(H,T)$ represents a list with head H and tail T.
  - ◇ Membership definition: $X \in L \leftrightarrow \begin{cases} X \text{ is the head of } L \\ \textbf{or } X \text{ is member of the tail of } L \end{cases}$
  - ◇ Using logic:

    $\forall X \forall L (\exists T \ (L = f(X,T) \rightarrow member(X,L)))$
    $\forall X \forall L (\exists Z \exists T (L = f(Z,T) \wedge member(X,T) \rightarrow member(X,L)))$
  - ◇ Using Prolog:

```
member(X, f(X, T)).
member(X, f(Z, T)) :- member(X,T).
```

- Procedural view (but for checking membership only!):
  - ◇ Traverse the list comparing each element until X is found or list is finished

```
/* Testing array membership in C */
int member(int x,int list[LISTSIZE]) {
   for (int i = 0; i < LISTSIZE; i++)
     if (x == list[i]) return TRUE;
   return FALSE;
}
```

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

- **60's**
  - ◇ Greene: problem solving.
  - ◇ Robinson: linear resolution.
- **70's**
  - ◇ **(early)** 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
  - ◇ **(early)** Colmerauer: specialized theorem prover (Fortran) embedding the procedural interpretation: Prolog (Programmation et Logique).
  - ◇ In the U.S.: "next-generation AI languages" of the time (i.e. planner) seen as inefficient and difficult to control.
  - ◇ **(late)** D.H.D. Warren develops DEC-10 Prolog compiler, almost completely written in Prolog. Very efficient (same as LISP). Very useful control builtins.

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

- **Late 80's, 90's**
  - Major research in the basic paradigms and advanced implementation techniques: Japan (Fifth Generation Project), US (MCC), Europe (ECRC, ESPRIT projects).
  - Numerous commercial Prolog implementations, programming books, and a *de facto* standard, the Edinburgh Prolog family.
  - First parallel and concurrent logic programming systems.
  - **CLP** – Constraint Logic Programming: Major extension – many new applications areas.
  - **1995:** ISO Prolog standard.

Currently

- Many commercial CLP systems with fielded applications.
- Extensions to full higher order, inclusion of functional programming, ...
- Highly optimizing compilers, automatic parallelism, automatic debugging.
- Concurrent constraint programming systems.
- Distributed systems.
- Object oriented dialects.
- Applications
  - Natural language processing
  - Scheduling/Optimization problems
  - AI related problems
  - (Multi) agent systems programming.
  - Program analyzers
  - ...