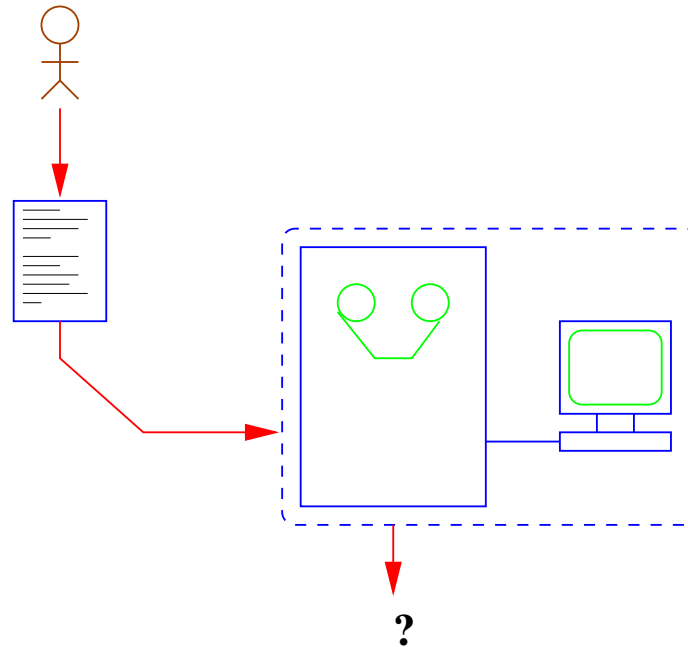**Computational Logic**

A Motivational Introduction

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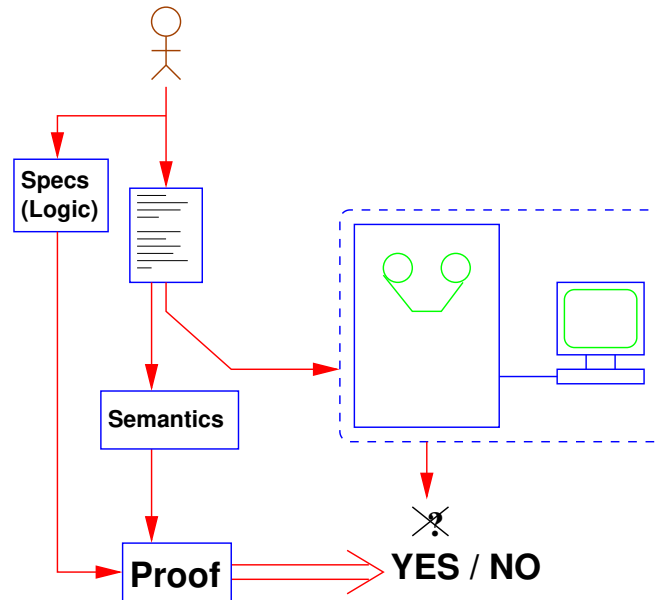 $t_1 : friend(plato, aristotle)$
  $T[a_1, a_2] \vdash t_1$

- 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:

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

- Defining the natural numbers:

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

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

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

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

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

- Addition of naturals:

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

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

Generating Squares: A Specification (II)

- Multiplication of naturals:

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

  $mult(3, 2, 6) \equiv$ adding 2 to itself 3 times is 6 $\equiv 2 + 2 + 2 = 6$

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

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

- Squares of the naturals:

  $$\boxed{\forall X \forall Y \ (nat(X) \wedge nat(Y) \ \wedge mult(X, X, Y) \to nat\_square(X, Y))}$$

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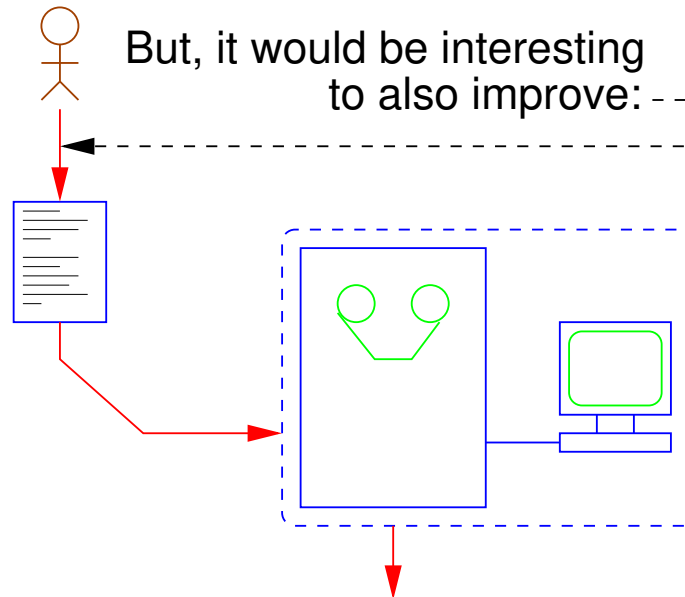
- *Precondition (empty):*

  $\boxed{true}$

- *Postcondition:*

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

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:
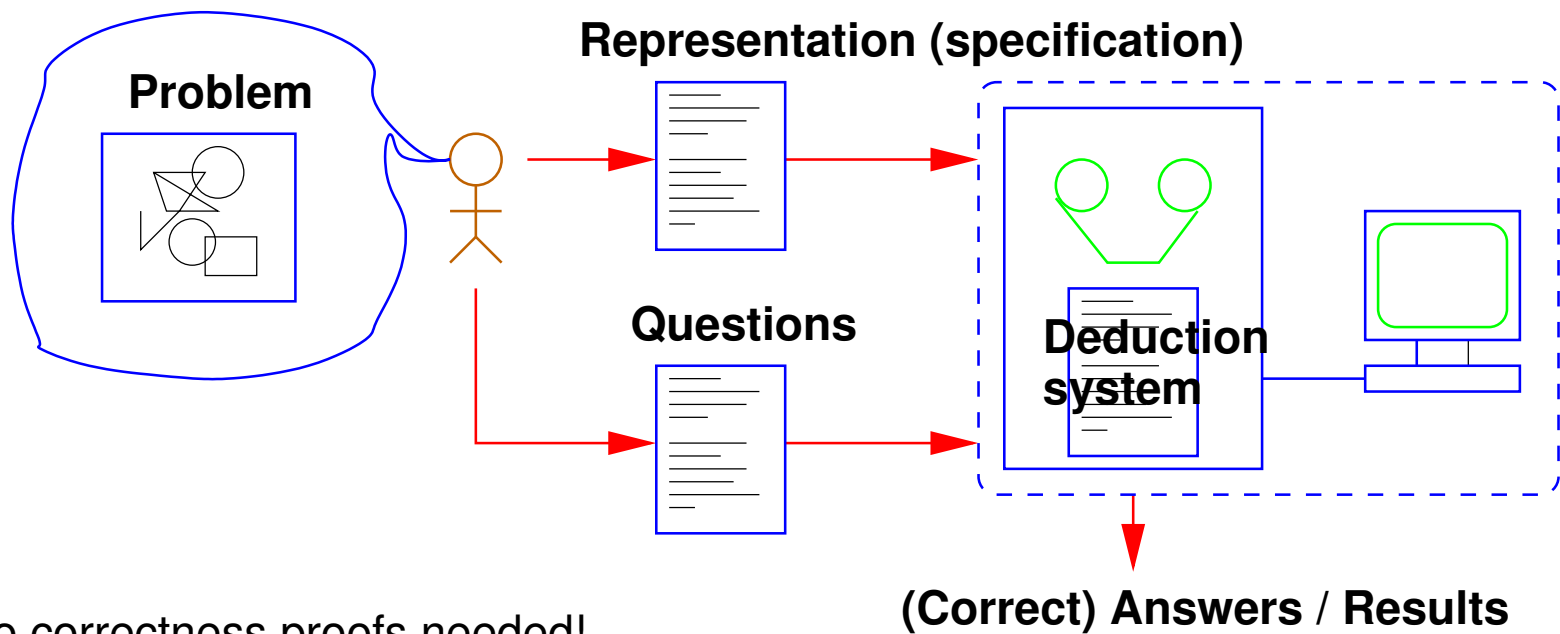
- No correctness proofs needed!

Computing With Our Previous Description / Specification

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

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**.

  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:

    $has\_tail(spot)$

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

$\rightarrow$ It is important to understand the underlying properties and the theoretical limits!

Comparison of Logics (I)

- **Propositional** logic:

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

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

- **Predicate logic**: (first order)

  "spot is a dog"   *dog(spot)*
  +/- decidability
  +/- 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
    + 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!

- 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)

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

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

- **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 $\neg nat(X) \lor add(0, X, X))$ 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 $\neg add(X, Y, Z) \lor add(s(X), Y, s(Z))$ giving $\neg add(X, Y, 0) \ldots$

17

Generating Squares in a Practical Logic Programming System (I)

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


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).
```

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)))) ; ...`

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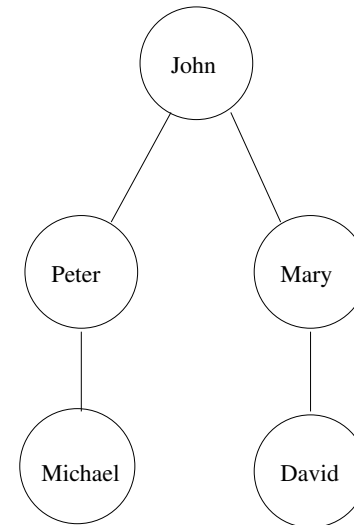
```
father_of(john, peter).
father_of(john, mary).
father_of(peter, michael).
mother_of(mary, david).

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)`?

Additional examples (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:

    $\forall X \forall L (\exists T\ (L = f(X,T) \rightarrow member(X,L)))$
    $\forall X \forall L (\exists Z \exists T (L = f(Z,T) \land member(X,T) \rightarrow member(X,L)))$

  - 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**

  - ◇ Green: programming as problem solving.
  - ◇ Robinson: **resolution**.

- **70's**

  - ◇ Colmerauer: specialized theorem prover (Fortran) embedding the procedural interpretation: First **Prolog** ("Programmation et Logique") interpreter.
  - ◇ 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.

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

- **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".

  ◇ 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.

  Also, Datalog, Answer Set Programming (ASP) – support for negation through stable models.

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

- 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.

  ⋄ ...