

# Certification de logiciel

**David Cachera - David Pichardie - ENS Rennes**

**École normale supérieure de Rennes**

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ - France

[www.ens-rennes.fr](http://www.ens-rennes.fr)

# Avant de commencer

<http://coq.inria.fr/download>

<http://www.irisa.fr/celtique/pichardie/teaching/luminy14/>



# Des logiciels partout



# Des **bugs** logiciels partout



# Des **bugs** logiciels partout

```
#include <stdio.h>
int fact (int n){
    int r, i;
    r=1;
    for (i=2; i<=n; i++){
        r=r*i;
    }
    return r;
}
int main() {
    int n;
    scanf("%d",&n);
    printf("%d !=%d\n",n,fact(n));
}
```

```
% gcc fact.c -o fact.exec
% ./fact.exec
3
3!=6
% ./fact.exec
4
4!=24
% ./fact.exec
100
100!=0
% ./fact.exec
20
20! = -2102132736
```

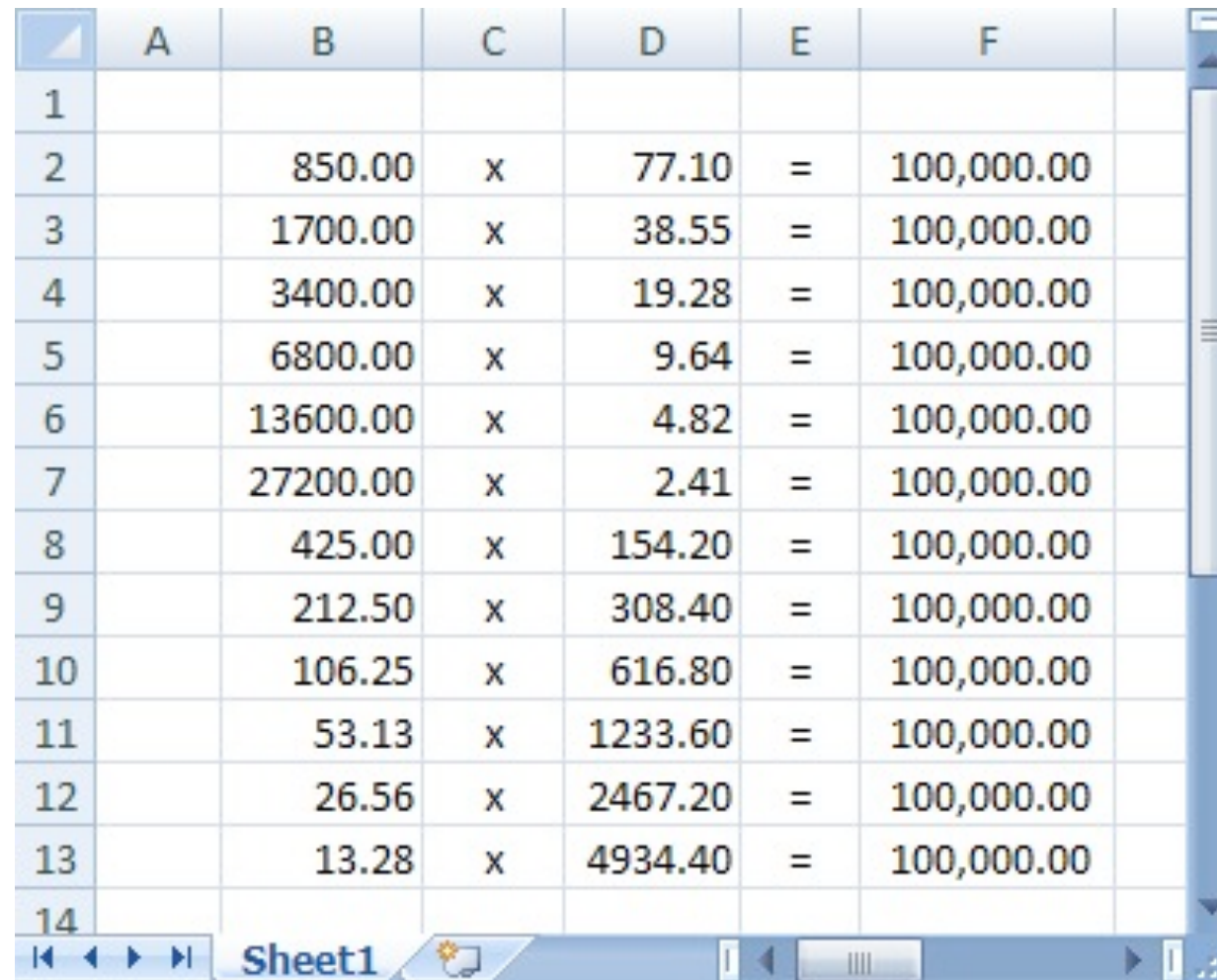
# Des **bugs** logiciels partout

```
# let rec fact n = if (n=1) then 1 else n * fact (n-1);;
val fact : int -> int = <fun>
# fact 4 ;;
- : int = 24
# fact 100 ;;
- : int = 0
# fact 20 ;;
- : int = 45350912
```

Problème de représentation des entiers (arithmétique modulaire)



# Des **bugs** logiciels partout



	A	B	C	D	E	F
1						
2		850.00	x	77.10	=	100,000.00
3		1700.00	x	38.55	=	100,000.00
4		3400.00	x	19.28	=	100,000.00
5		6800.00	x	9.64	=	100,000.00
6		13600.00	x	4.82	=	100,000.00
7		27200.00	x	2.41	=	100,000.00
8		425.00	x	154.20	=	100,000.00
9		212.50	x	308.40	=	100,000.00
10		106.25	x	616.80	=	100,000.00
11		53.13	x	1233.60	=	100,000.00
12		26.56	x	2467.20	=	100,000.00
13		13.28	x	4934.40	=	100,000.00
14						

Passage de 16 à 32 bits

# Des **bugs** logiciels partout

Calculs en nombres « à virgule » (flottants ou virgule fixe)

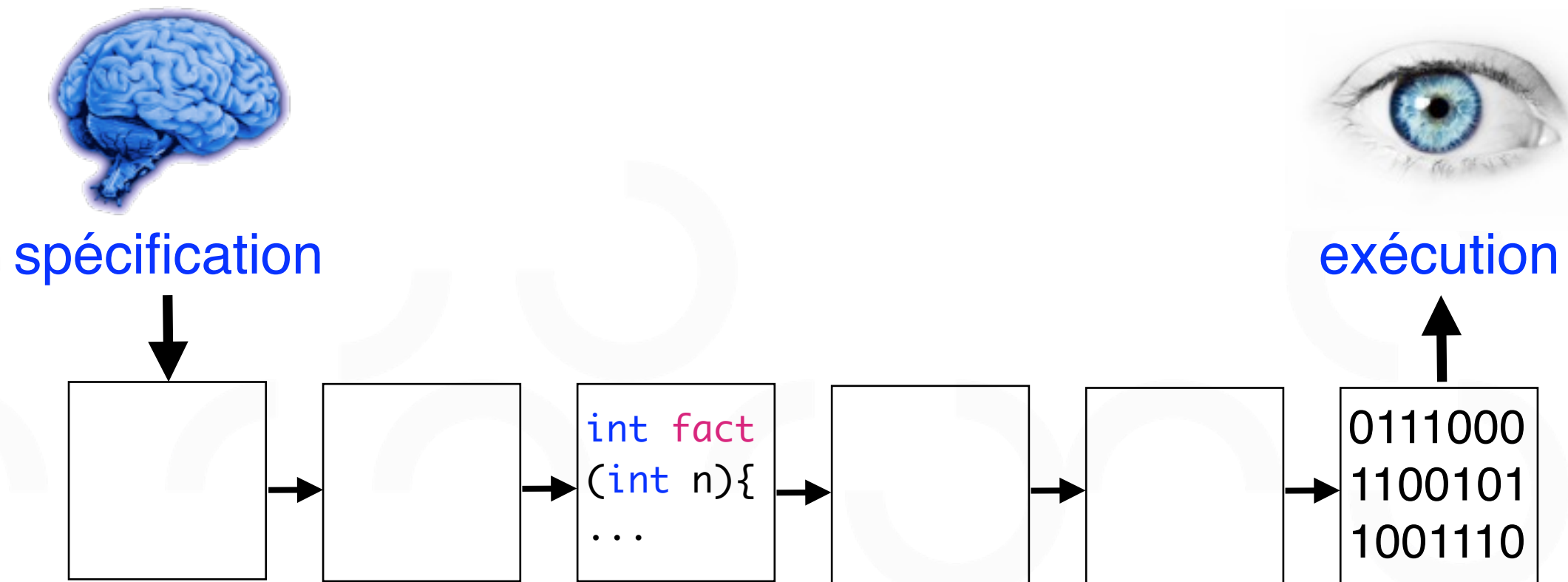
- Problèmes bien connus avec les arrondis
- 0,1 en base 10 = 0,0001100110011001100... en base 2  
bug du missile Patriot (1991)



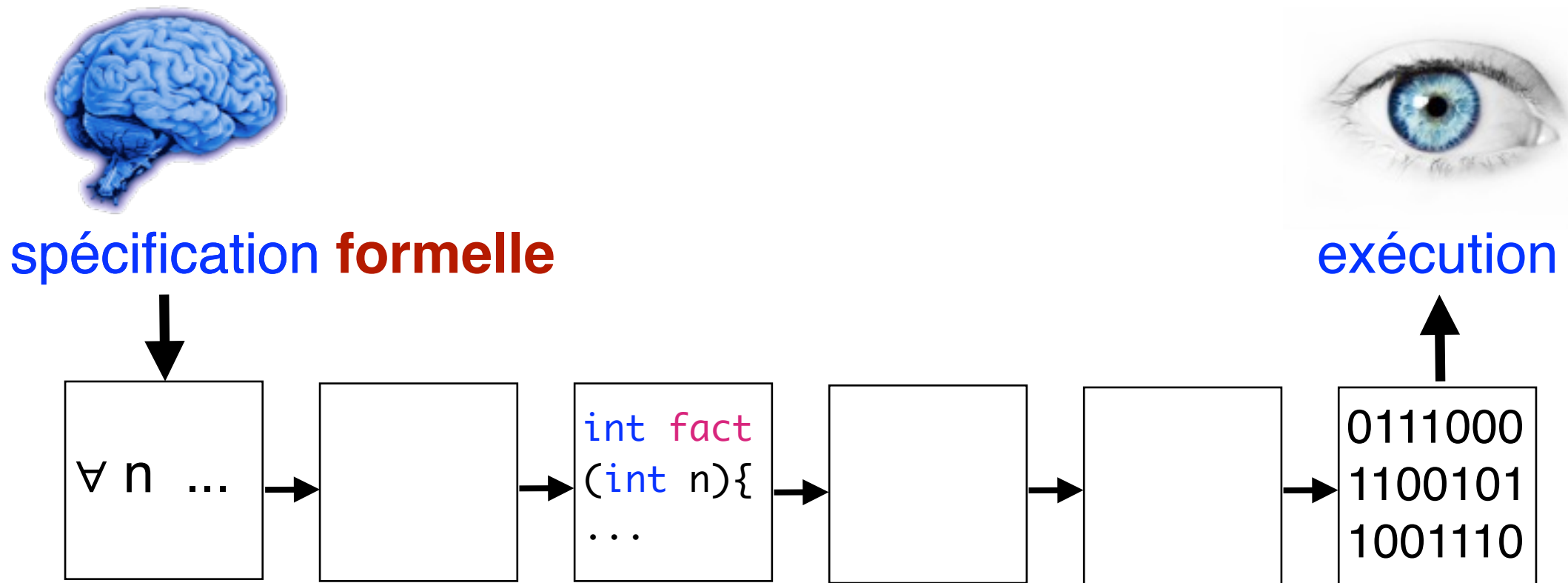
# Bugs ?

Un « bug » est (presque) toujours dû à un décalage entre

- ce qu'on attend du programme... **spécification**
- son comportement réel **exécution**



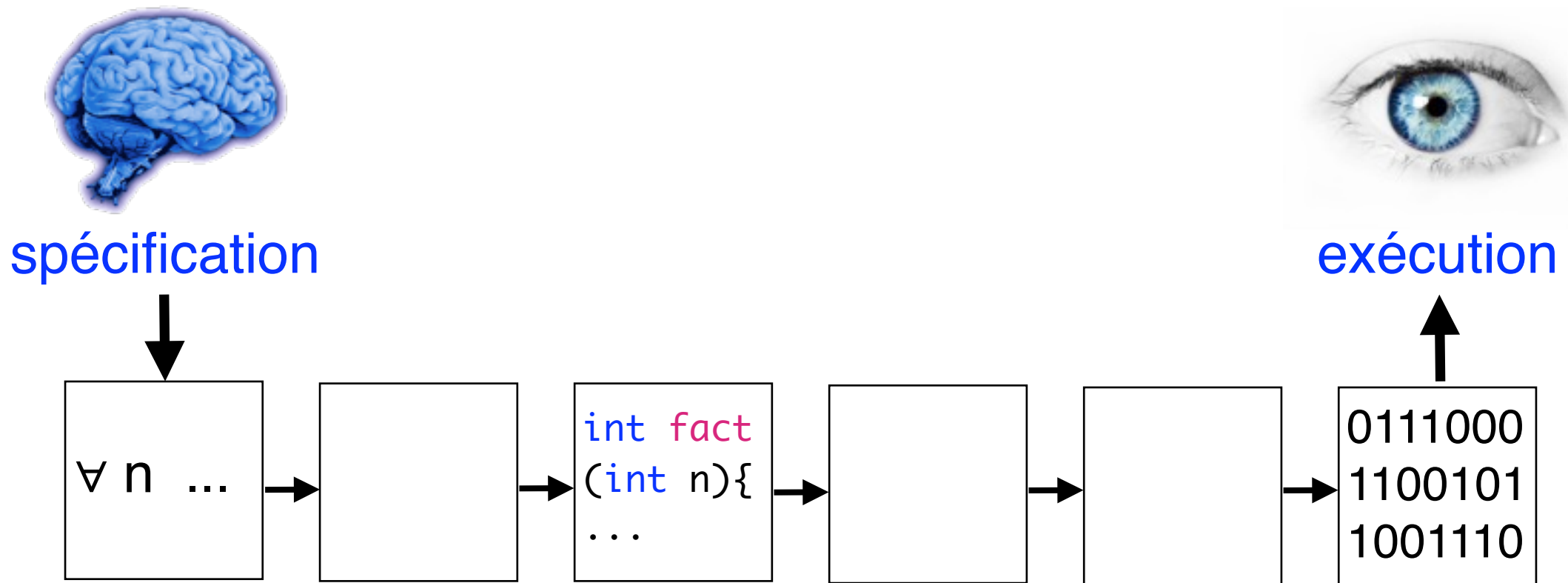
# Spécification



La spécification doit être

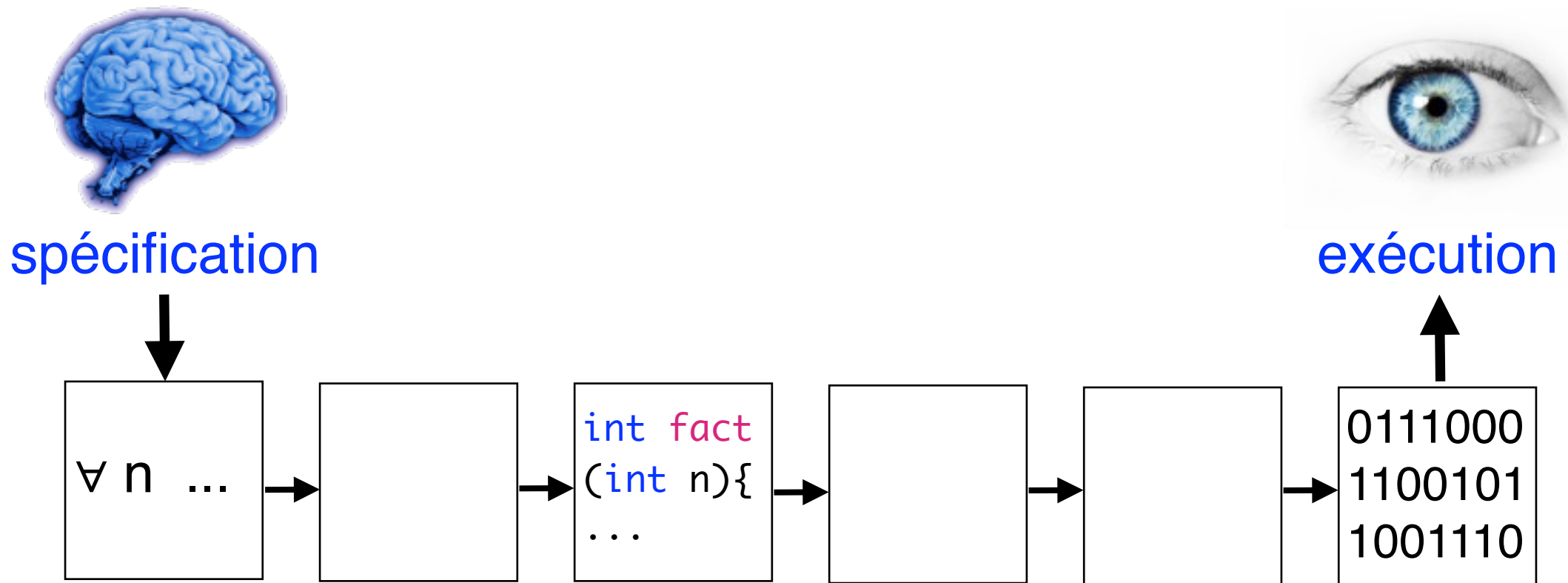
- dépourvue de toute ambiguïté
- aussi complète que possible (hypothèses sur l'environnement)

# Sémantique



À tous les stades, il faut disposer d'une **sémantique formelle** : description mathématique du comportement attendu.

# Solution



Faire une preuve formelle que la sémantique satisfait la spécification  
Utiliser l'ordinateur pour automatiser la preuve

# Indécidabilité

- Il n'existe aucun programme qui prend en entrée le texte d'un programme, et renvoie en sortie oui ou non, suivant que le programme en entrée est correct ou pas : **indécidabilité**.
- Encore vrai pour une question plus simple : le programme termine-t-il ? **Problème de l'arrêt**.
- Vrai pour toutes les définitions « raisonnables » de la notion de programme.
- Connue depuis les années 30.



Alan Turing



Kurt Gödel

# Solutions

Ce que l'ordinateur ne sait pas faire, l'homme le sait :  
raisonner « à l'extérieur »

Problème : la taille des programmes

Coopération entre

- l'expert humain : parties non décidables
- le calculateur : parties fastidieuses



# Solutions

- Exécuter le programme sur un sous-ensemble, très grand et le plus pertinent possible, des données d'entrée :

test

- Faire calculer par la machine tous les états atteignables :

vérification de modèle

- Faire calculer par la machine une approximation de la sémantique :

analyse statique

- Faire collaborer humain et machine pour démontrer que le programme respecte sa spécification :

assistant de preuve



# Assistants de preuve

Logique classique du 1er ordre ou d'ordre supérieur :

ACL-2 : calculs flottants

PVS : NASA

HOL4, HOL Light : calculs flottants

Isabelle/HOL : noyau système

Logique constructive : Coq (Agda, Matita...)

- Un programme est une preuve de la spécification
- On développe en même temps le programme et la preuve, puis on extrait automatiquement le programme
- compilateur, théorème des 4 couleurs, théorème de Feit-Thompson

# Un kit de survie en Coq

# Premiers pas en Coq



Comme tout assistant de preuve, Coq fournit

- un langage de programmation pour écrire et exécuter des programmes
- un langage de spécification pour écrire des propriétés sur les programmes
- un langage de preuve (tactiques) pour construire des preuves interactives de ces propriétés (les preuves sont ensuite vérifiées par le système)

# Premiers pas en Coq



## Originalité de Coq

- tous ces langages sont fondés sur un noyau commun : le calcul des constructions inductives (CIC), une théorie des types riche et complexe

Mais... ce qui rend Coq original le rend aussi difficile à aborder

- certains débutants essaient de comprendre le CIC avant d'utiliser Coq...
- le même mot-clé peut être utilisé pour des notions (apparemment) différentes

# Premiers pas en Coq



Nous allons présenter l'outil sans révéler ses fondements théoriques

Pré-requis : des connaissances de base en programmation  
fonctionnelle (OCaml)

Fil rouge : exemple de la structure de donnée « dictionnaire » (map)

- pour obtenir in fine un outil d'analyse performant, il faut de bonnes structures de données
- les maps sont ce que nous pouvons faire facilement de mieux en fonctionnel

# Les *MAP* en OCaml

```
module type MAP =  
  
  sig  
  
    (** the type of map keys *)  
  
    type key  
  
    (** the type of map elements *)  
  
    type elt  
  
    (** the type of maps *)  
  
    type t  
  
    (** [default] is the default element in a map *)  
  
    val default : elt  
  
    (** [get m k] returns the elements that is binded with key [k] *)  
  
    val get : t -> key -> elt  
  
    (** [empty] is a map that binds every keys to [default] *)  
  
    val empty : t  
  
    (** [set m k e] returns a new map that contains the same  
        bindings as map [m] except for key [k] that is now binded with  
        the element [e] *)  
  
    val set : t -> key -> elt -> t  
  
  end
```

# Quelques tactiques de preuve



# intros



=====

**intros**

**forall** (a : A), P

a : A

=====

P

Voir terme de type **Prop**  
comme une propriété logique

=====

**intros**

P -> Q

P : **Prop**

=====

Q

=====

**intros**

P -> Q > R

P : **Prop**

Q : **Prop**

=====

R

En Coq, on utilise la forme  
 $P \Rightarrow Q \Rightarrow R$  au lieu de  $P \wedge Q \Rightarrow R$

# destruct

pour un terme de type inductif



```
Inductive nat : Set :=  
  | 0 : nat  
  | S : nat -> nat.
```

```
n : nat  
=====  
P n
```

**destruct** n

1er sous-but

2e sous-but

```
=====  
P 0
```

```
m : nat  
=====  
P (S m)
```

# simpl



**Definition** `pred (n:nat) :=`

`match n with`

`| 0 => 0`

`| S m => m`

=====

`... pred 0 ...`

**simpl**



=====

`... 0 ...`

=====

`... pred (S n) ...`

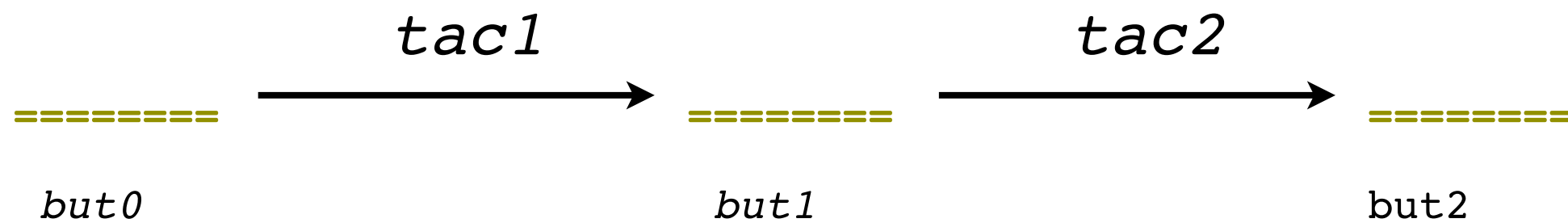
**simpl**



=====

`... n ...`

# *tac1 ; tac2*



Si *tac1* génère plusieurs sous-buts, *tac2* est appliqué sur chaque.

# induction

pour un terme de type inductif



`n : nat`

=====

`P n`

**induction** `n`

1er sous-but

`P 0`

2e sous-but

`m : nat`

`IH : P m`

=====

`P (S m)`

# apply



$H : P \rightarrow Q$

=====

$Q$

**apply**  $H$



$H : P \rightarrow Q$

=====

$P$

$H : \text{forall } x\ y, P\ y \rightarrow Q\ x\ y$

=====

$Q\ a\ (f\ a)$

**apply**  $H$



$H : \text{forall } x\ y, P\ y \rightarrow Q\ x\ y$

=====

$P\ (f\ a)$

Coq devine comment  
instancier les quantificateurs

$H : \text{forall } x\ y, P\ x\ y \rightarrow Q\ y$

=====

$Q\ (f\ a)$

**apply**  $H$



**with**  $a$

$H : \text{forall } x\ y, P\ x\ y \rightarrow Q\ y$

=====

$P\ a\ (f\ a)$

Là il faut aider un peu Coq

# rewrite



H : a = b

=====

... a ...

**rewrite** H



H : a = b

=====

... b ...

H : **forall** x y, f x y = x

=====

... (f a b) ...

**rewrite** H



H : **forall** x y, f x y = x

=====

Coq devine comment  
instancier les quantificateurs



# Exercice

Compléter la structure MAP avec les opérateurs suivants, la spécification de leur propriétés et les preuves correspondantes.

```
module type MAP =  
  ...  
  
  (** [remove m k] returns a new map that contains the same  
      bindings as map [m] except for key [k] that is now binded with  
      the default element *)  
  val remove : t -> key -> t  
  
  (** [map m f] returns a new map that binds each key [k] to an  
      element [(f e)], assuming [k] was binded with [e] in [m].  
      The function [f] must satisfy the property ... *)  
  val map : t -> (elt -> elt) -> t  
  
  (** [mapi m f] returns a new map that binds each key [k] to an  
      element [(f k e)], assuming [k] was binded with [e] in [m].  
      The function [f] must satisfy the property ... *)  
  val mapi : t -> (key -> elt -> elt) -> t  
  
end
```

# Prédicats inductifs

`positive` : type des entiers strictement positifs codés en binaire (poids faibles en tête)

`nat` : type des entiers naturels, codés en unaire

relation `(inf_log p n)` : `p` est codé sur moins de `n` bits.

```
Inductive positive : Set :=  
| xI : positive -> positive  
| xO : positive -> positive  
| xH : positive.
```

$$\frac{n:\text{nat}}{\text{inf\_log } xH \ (S \ n)}$$

$$\frac{p:\text{positive} \quad n:\text{nat} \quad \text{inf\_log } p \ n}{\text{inf\_log } (xO \ p) \ (S \ n)}$$

$$\frac{p:\text{positive} \quad n:\text{nat} \quad \text{inf\_log } p \ n}{\text{inf\_log } (xI \ p) \ (S \ n)}$$

```
Inductive inf_log : positive -> nat -> Prop :=
```

```
| Inf_log_xH: forall n, inf_log xH (S n)
```

```
| Inf_log_xO: forall p n, inf_log p n -> inf_log (xO p) (S n)
```

```
| Inf_log_xI: forall p n, inf_log p n -> inf_log (xI p) (S n).
```

# inv

## sur une hypothèse inductive



`p: positive`

`n: nat`

`H : inf_log p n`

=====

`P p n`

`n:nat`  


---

`inf_log xH (S n)`

`p:positive n:nat inf_log p n`  


---

`inf_log (xO p) (S n)`

`p:positive n:nat inf_log p n`  


---

`inf_log (xI p) (S n)`

**inv** H

1er sous-but

2e sous-but

3e sous-but

`n: nat`  
 =====  
`P xH (S n)`

`n : nat`  
`p : positive`  
`H : inf_log p n`  
 =====  
`P (xO p) (S n)`

`n : nat`  
`p : positive`  
`H : inf_log p n`  
 =====  
`P (xI p) (S n)`

# induction

sur une hypothèse inductive



$p$ : positive

$n$ : nat

$H : \text{inf\_log } p \ n$

=====

$P \ p \ n$

**induction**  $H$

1er sous-but

$n$ : nat

=====

$P \ xH \ (S \ n)$

2e sous-but

$n : \text{nat}$

$p : \text{positive}$

$H : \text{inf\_log } p \ n$

$IH : P \ p \ n$

=====

$P \ (xO \ p) \ (S \ n)$

3e sous-but

$n : \text{nat}$

$p : \text{positive}$

$H : \text{inf\_log } p \ n$

$IH : P \ p \ n$

=====

$P \ (xI \ p) \ (S \ n)$

# Mélanger programmes et preuves

Le type des entiers strictement positifs codés en binaire avec au plus  $n$  bits.

```
Inductive bin (n:nat) :=  
| Build_bin (p:positive) (h:inf_log p n).
```

ceci est une preuve !

# Dans les coulisses

Tous les objets que nous avons manipulés sont des termes du CIC

terme du  
CIC

terme du  
CIC

**Definition** `foo := T := def.`

terme du  
CIC

**Lemma** `foo_pr`

**Proof.**

`tac...`

**Qed.**

cette commande  
interactive construit aussi  
un terme du CIC

# Un peu de lecture

The Coq Web site: software and documentation.  
<http://coq.inria.fr/>

Coq in a hurry, Yves Bertot.  
<http://cel.archives-ouvertes.fr/inria-00001173/>

Benjamin Pierce et al. Software Foundations.  
<http://www.cis.upenn.edu/~bcpierce/sf/>

Interactive Theorem Proving and Program Development -- Coq'Art:  
The Calculus of Inductive Constructions, by Yves Bertot and Pierre  
Casteran: a comprehensive textbook on Coq.