



COLLÈGE  
DE FRANCE  
—1530—

*Logiques de programmes, premier cours*

# **Comment raisonner sur un logiciel?**

## **La naissance des logiques de programmes**

---

Xavier Leroy

2021-03-04

Collège de France, chaire de sciences du logiciel

`xavier.leroy@college-de-france.fr`

Une revue de trois articles qui ont fondé le domaine :

- Alan Turing, *Checking a large routine*, 1949.
- Robert W. Floyd, *Assigning meanings to programs*, 1967.
- C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969.

**La découverte :**

***Checking a large routine***

**Alan Turing, 1949**

---



# How can one check a routine? That is the question!

Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

## Décomposer la vérification en étapes élémentaires

Consider the analogy of checking an addition. If it is given as:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ \underline{7768} \\ 26104 \end{array}$$

one must check the whole at one sitting, because of the carries. But if the totals for the various columns are given, as below:

$$\begin{array}{r} 1374 \\ 5906 \\ 6719 \\ 4337 \\ \underline{7768} \\ 3974 \\ \underline{2213} \\ 26104 \end{array}$$

the checker's work is much easier being split up into the checking of the various assertions  $3 + 9 + 7 + 3 + 7 = 29$  etc. and the small addition

$$\begin{array}{r} 3974 \\ \underline{2213} \\ 26104 \end{array}$$

## Le programme de Turing : calcul de la factorielle

Calculer  $n!$  en n'utilisant que des additions.

Deux boucles imbriquées.

```
int fac (int n)
{
    int s, r, u, v;
    u = 1;
    for (r = 1; r < n; r++) {
        v = u; s = 1;
        do {
            u = u + v;
        } while (s++ < r);
    }
    return u;
}
```

# L'organigramme du programme

«Unfortunately there is no coding system sufficiently generally known to justify giving the routine for this process.»

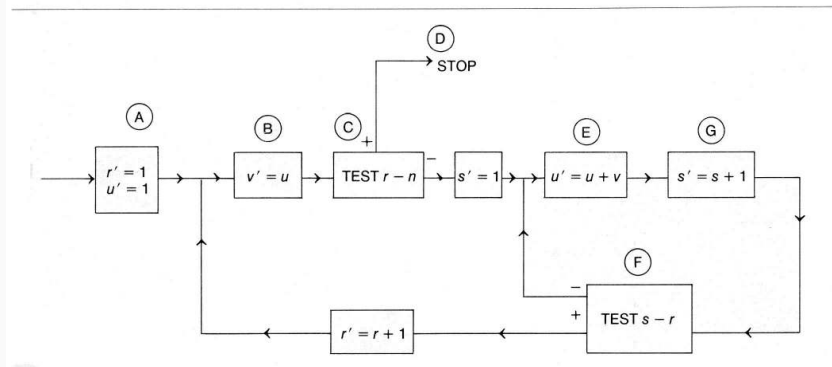


Figure 1 (Redrawn from Turing's original)

(La notation  $u/u'$  dénote la valeur de  $u$  avant/après le bloc).



# Les assertions logiques

«In order to assist the checker, the programmer should make assertions about the various states that the machine can reach.»

Documente non seulement quelle case mémoire contient quelle variable abstraite, mais surtout **les relations entre ces variables.**

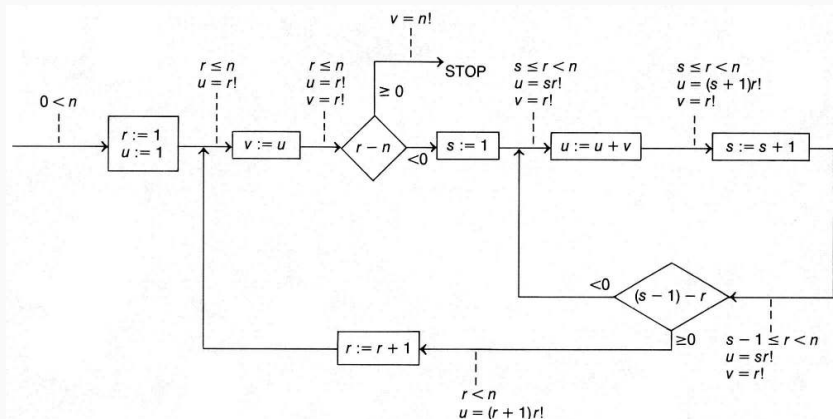
STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
27					$s$	$s + 1$	$s$
28		$r$	$r$		$r$	$r$	$r$
29	$n$	$n$	$n$	$n$	$n$	$n$	$n$
30		$\lfloor$	$\lfloor$		$s \lfloor$	$(s + 1) \lfloor$	$(s + 1) \lfloor$
31		$\lfloor$	$\lfloor$	$\lfloor$	$\lfloor$	$\lfloor$	$\lfloor$
	TO Ⓑ WITH $r' = 1$ $u' = 1$	TO Ⓒ	TO Ⓓ IF $r = n$ TO Ⓔ IF $r < n$		TO Ⓖ	TO Ⓑ WITH $r' = r + 1$ IF $s \geq r$ TO Ⓔ WITH $s' = s + 1$ IF $s < r$	TO Ⓕ

Figure 2 (Redrawn from Turing's original)

(La notation  $\lfloor n$  signifie «factorielle  $n$ ».)

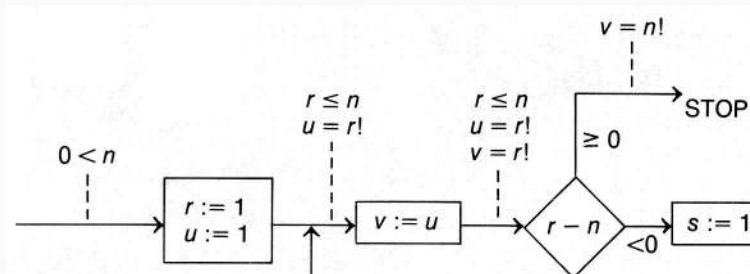
# Les assertions logiques directement sur l'organigramme

Dans la notation moderne (introduite par Floyd en 1967), on note les assertions directement sur les arcs de l'organigramme.



# La vérification

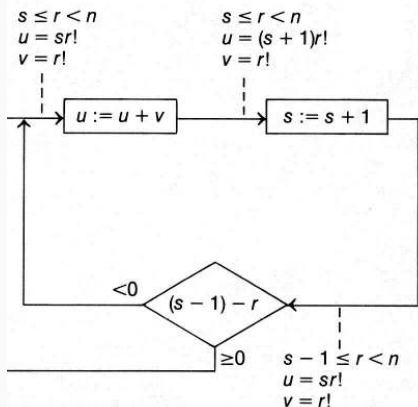
«The checker has to verify that the columns corresponding to the initial condition and the stopped condition agree with the claim that are made for the routine as a whole.»



$$r \leq n \wedge u = r! \wedge v = r! \wedge r - n \geq 0 \implies v = n!$$

# La vérification

«[The checker] also has to verify that each of the assertions in the lower half of the table is correct. In doing this the columns may be taken in any order and quite independently.»



$$s \leq r < n \wedge u = sr! \wedge v = r!$$
$$\Downarrow$$
$$s \leq r < n \wedge u + v = (s + 1)r! \wedge v = r!$$

$$s - 1 \leq r < n \wedge u = sr! \wedge v = r!$$
$$\wedge (s - 1) - r < 0$$
$$\Downarrow$$
$$s \leq r < n \wedge u = sr! \wedge v = r!$$

## Vérifier la terminaison

«Finally the checker has to verify that the process comes to an end. Here again he should be assisted by the programmer [...]. This may take the form of a quantity which is asserted to decrease continually and vanish when the machine stops.»

Turing propose de prendre l'ordinal  $(n - r)\omega^2 + (r - s)\omega + k$ , ce qui correspond à l'ordre lexicographique sur  $(n - r, r - s, k)$ .

Plus prosaïquement il suggère  $2^{80}(n - r) + 2^{40}(r - s) + k$ .

STORAGE LOCATION	(INITIAL) Ⓐ $k = 6$	Ⓑ $k = 5$	Ⓒ $k = 4$	(STOP) Ⓓ $k = 0$	Ⓔ $k = 3$	Ⓕ $k = 1$	Ⓖ $k = 2$
---------------------	---------------------------	--------------	--------------	------------------------	--------------	--------------	--------------

**La redécouverte et la formalisation :**  
***Assigning meanings to programs***  
**Robert W. Floyd, 1967**

---

Robert W. Floyd

### ASSIGNING MEANINGS TO PROGRAMS<sup>1</sup>

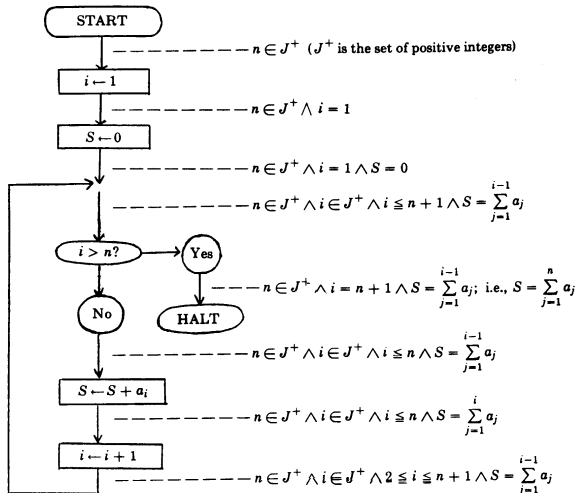
**Introduction.** This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of

*Mathematical Aspects of Computer Science*, 1967, 14 pages.

Proceedings of Symposium on Applied Mathematics, vol 19, AMS.

# Le retour des assertions logiques

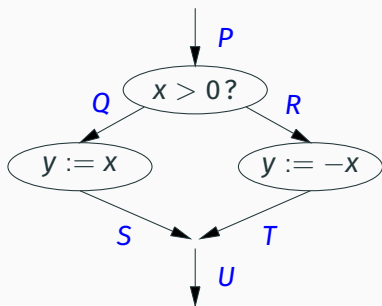
18 ans après, Floyd redécouvre l'idée de Turing :  
annoter un organigramme par des assertions logiques.





## Des conditions de vérification...

Floyd formalise les **conditions de vérification** :  
les implications logiques qui garantissent la cohérence des  
assertions qui annotent le programme.



$$\begin{aligned}P \wedge x > 0 &\Rightarrow Q \\P \wedge x \leq 0 &\Rightarrow R \\ \exists y_0, Q[y \leftarrow y_0] \wedge y = x &\Rightarrow S \\ \exists y_0, R[y \leftarrow y_0] \wedge y = -x &\Rightarrow T \\ S \vee T &\Rightarrow U\end{aligned}$$

Programme annoté  $\longrightarrow$  Conditions de vérification

## ... à la sémantique du langage de programmation

programme annoté  $\xrightarrow[\text{formelle}]{\text{sémantique}}$  conditions de vérification

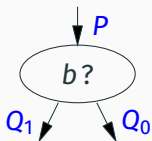
Floyd observe que les règles qui transforment un programme annoté en conditions de vérification constituent une **sémantique** du langage de programmation utilisé.

C'est la naissance de la sémantique formelle!

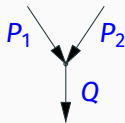
(Cf. mon cours «Sémantiques mécanisées» en 2019).

*«[T]he proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about programs in the language, appear to be novel»*

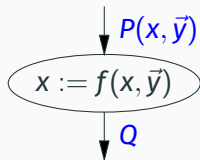
## Conditions de vérification pour les organigrammes



$$P \wedge b \Rightarrow Q_1$$
$$P \wedge \neg b \Rightarrow Q_0$$



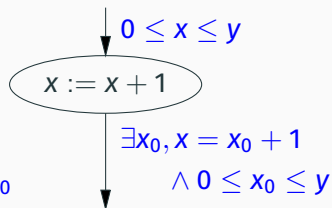
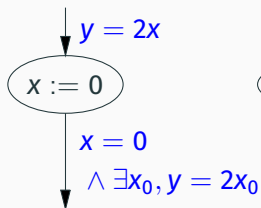
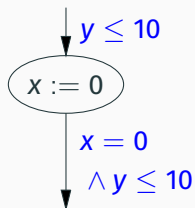
$$P_1 \vee P_2 \Rightarrow Q$$



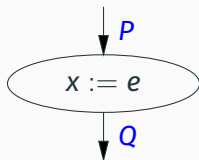
$$(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})) \Rightarrow Q$$

# La règle de Floyd pour l'affectation

Exemples :



Le cas général :



$$(\exists x_0, x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0]) \Rightarrow Q$$

## Quelques règles générales

Notations :

- $c$  commande (morceau de programme)
- $\vec{P}$  préconditions (une par entrée dans  $c$ )
- $\vec{Q}$  postconditions (une par sortie de  $c$ )
- $V_c(\vec{P}; \vec{Q})$  conditions de vérifications pour  $\vec{P}$ ,  $c$ ,  $\vec{Q}$

**Conséquence** : si  $V_c(\vec{P}; \vec{Q})$  et  $\vec{P}' \Rightarrow \vec{P}$  et  $\vec{Q} \Rightarrow \vec{Q}'$ , alors  $V_c(\vec{P}'; \vec{Q}')$ .

**Conjonction** : si  $V_c(\vec{P}; \vec{Q})$  et  $V_c(\vec{P}', \vec{Q}')$  alors  $V_c(\vec{P} \wedge \vec{P}'; \vec{Q} \wedge \vec{Q}')$ .

**Disjonction** : si  $V_c(\vec{P}; \vec{Q})$  et  $V_c(\vec{P}', \vec{Q}')$  alors  $V_c(\vec{P} \vee \vec{P}'; \vec{Q} \vee \vec{Q}')$ .

**Quantification existentielle** : si  $V_c(\vec{P}; \vec{Q})$  alors  $V_c(\exists x. \vec{P}; \exists x. \vec{Q})$ .

Si la condition de vérification  $V_c(P_1 \dots P_n; Q_1 \dots Q_m)$  est vraie  
c s'exécute d'un état initial  $s$  vers un état final  $s'$  (sortie  $j$ )  
l'état initial  $s$  satisfait l'une des préconditions  $P_i$   
alors  
l'état final  $s'$  satisfait la postcondition  $Q_j$ .

Facile à vérifier pour les règles des organigrammes.

Corollaire : si le programme démarre dans un état initial satisfaisant sa précondition  $P$ , et s'il termine, alors l'état final satisfait sa postcondition  $Q$ .

## Plus forte conséquence vérifiable

Floyd conjecture que la condition de vérification  $V_c(\vec{P}; \vec{Q})$  peut toujours s'écrire sous la forme

$$T_c(P_1 \vee \dots \vee P_n) \Rightarrow \vec{Q}$$

où  $T_c(P)$  est la plus forte postcondition de la commande  $c$  avec la précondition  $P$ .

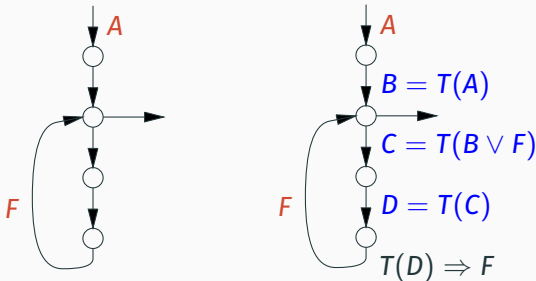
Par exemple, pour les organigrammes, on a

$$T_{x:=e}(P) = \exists x_0, x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0]$$

$$T_{\text{test}(b)}(P) = (P \wedge b, P \wedge \neg b)$$

## Vers l'automatisation

En utilisant  $T$ , on peut compléter un organigramme partiellement annoté.



« This fact offers the possibility of automatic verification of programs, the programmer merely tagging entrances and one edge in each innermost loop; the verifying program would extend the interpretation and verify it, if possible, by mechanical theorem-proving techniques. »



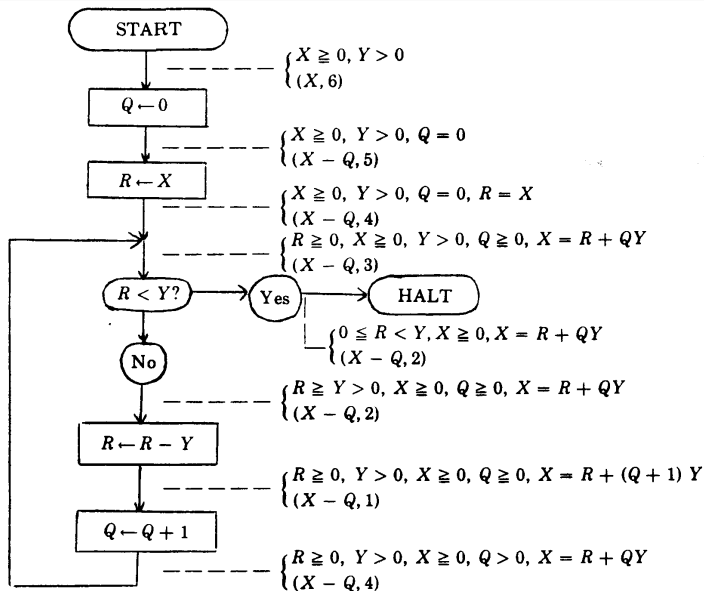
Un début de définition de  $V_c$  pour des commandes structurées à la manière d'Algol (séquence, if/then/else, boucle `for`).

Une discussion de la complétude de la définition de  $V_c$  (voir le prochain cours).

Une méthode pour vérifier la terminaison :

- Associer à chaque arc de l'organigramme une fonction état des variables  $\rightarrow$  ensemble bien fondé  $W$  (p.ex.  $W = n$ -uplets d'entiers + ordre lexicographique)
- Vérifier que ces fonctions décroissent sur chaque transition.

## Exemple final de l'article : la division euclidienne



**Le manifeste :**

***An axiomatic basis for computer programming***

**C. A. R. Hoare, 1969**

---

# An Axiomatic Basis for Computer Programming

C. A. R. HOARE

*The Queen's University of Belfast,\* Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

*Communications of the ACM* 12(10), 1969

## Une position de principe : l'approche axiomatique

Une approche axiomatique permet de spécifier les programmes et de définir les langages **sans pour autant tout spécifier**.

Exemple de Hoare : les débordements arithmétiques (en arithmétique entière non signée).

Erreur :  $MAX + 1$  arrête le programme

Saturation :  $MAX + 1 = MAX$

Modulo :  $MAX + 1 = 0$

## Axiomatiser l'arithmétique des ordinateurs

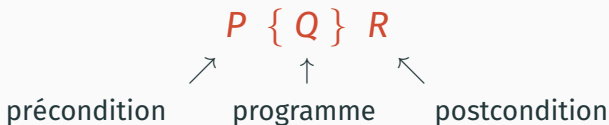
Hoare donne 9 axiomes qui sont vrais dans  $\mathbb{N}$  mais aussi dans les trois types d'arithmétique machine :

A1	$x + y = y + x$	addition is commutative
A2	$x \times y = y \times x$	multiplication is commutative
A3	$(x + y) + z = x + (y + z)$	addition is associative
A4	$(x \times y) \times z = x \times (y \times z)$	multiplication is associative
A5	$x \times (y + z) = x \times y + x \times z$	multiplication distributes through addition
A6	$y \leq x \supset (x - y) + y = x$	addition cancels subtraction
A7	$x + 0 = x$	
A8	$x \times 0 = 0$	
A9	$x \times 1 = x$	

Il montre que ces axiomes suffisent à vérifier le programme de la division euclidienne.

## Une notation : les « triplets de Hoare »

Pour axiomatiser les programmes, Hoare introduit la notation



*This may be interpreted "If the assertion  $P$  is true before initiation of a program  $Q$ , then the assertion  $R$  will be true on its completion".*

La notation utilisée partout aujourd'hui :



## Une contribution : les règles pour un langage structuré

Plus d'organigrammes, mais des structures de contrôle dans le style d'Algol 60.

$$\{ Q[x \leftarrow e] \} x := e \{ Q \} \text{ (affectation)}$$

$$\frac{\{ P \} c \{ Q \} \quad Q \Rightarrow Q'}{\{ P \} c \{ Q' \}} \text{ (conséquence 1)} \quad \frac{P' \Rightarrow P \quad \{ P \} c \{ Q \}}{\{ P' \} c \{ Q \}} \text{ (conséquence 2)}$$

$$\frac{\{ P \} c_1 \{ Q \} \quad \{ Q \} c_2 \{ R \}}{\{ P \} c_1; c_2 \{ R \}} \text{ (composition)}$$

$$\frac{\{ P \wedge b \} c \{ P \}}{\{ P \} \text{ while } b \text{ do } c \{ P \wedge \neg b \}} \text{ (itération)}$$



## La règle de Hoare pour l'affectation

$$\{ Q[x \leftarrow e] \} x := e \{ Q \}$$

Style «en arrière» : la postcondition  $Q$  détermine la précondition.

### Exemple

$$\begin{array}{lll} \{ 0 = 0 \wedge y \leq 10 \} & x := 0 & \{ x = 0 \wedge y \leq 10 \} \\ \{ 1 \leq x + 1 \leq 10 \} & x := x + 1 & \{ 1 \leq x \leq 10 \} \end{array}$$

À comparer au style «en avant» de la règle de Floyd :

$$\{ P \} x := e \{ \exists x_0, x = e[x \leftarrow x_0] \wedge P[x \leftarrow x_0] \}$$

# La règle de Hoare pour l'itération

$$\frac{\{P \wedge b\} c \{P\}}{\{P\} \text{ while } b \text{ do } c \{P \wedge \neg b\}} \quad (\text{itération})$$

La précondition  $P$  doit être un **invariant de boucle** :  
vrai au début du corps de boucle  $c$  à chaque itération;  
rétabli à la fin du corps  $c$  pour l'itération suivante.

## Exemple (boucle comptée)

```
x := 0;
```

```
  { 0 ≤ x ≤ 10 }
```

```
while x < 10 do
```

```
  { 0 ≤ x ≤ 10 ∧ x < 10 } x := x + 1 { 0 ≤ x ≤ 10 }
```

```
done
```

```
  { 0 ≤ x ≤ 10 ∧ ¬(x < 10) } ⇒ { x = 10 }
```

## Dernier exemple de l'article : la division euclidienne

```
r := X;  
q := 0;  
while y ≤ r do  
  r := r - y;  
  q := q + 1  
done
```

Line number	Formal proof	Justification
1	<b>true</b> $\supset x = x + y \times 0$	Lemma 1
2	$x = x + y \times 0 \{r := x\} x = r + y \times 0$	D0
3	$x = r + y \times 0 \{q := 0\} x = r + y \times q$	D0
4	<b>true</b> $\{r := x\} x = r + y \times 0$	D1 (1, 2)
5	<b>true</b> $\{r := x; q := 0\} x = r + y \times q$	D2 (4, 3)
6	$x = r + y \times q \wedge y \leq r \supset x = (r - y) + y \times (1 + q)$	Lemma 2
7	$x = (r - y) + y \times (1 + q) \{r := r - y\} x = r + y \times (1 + q)$	D0
8	$x = r + y \times (1 + q) \{q := 1 + q\} x = r + y \times q$	D0
9	$x = (r - y) + y \times (1 + q) \{r := r - y; q := 1 + q\} x = r + y \times q$	D2 (7, 8)
10	$x = r + y \times q \wedge y \leq r \{r := r - y; q := 1 + q\} x = r + y \times q$	D1 (6, 9)
11	$x = r + y \times q \{ \text{while } y \leq r \text{ do } (r := r - y; q := 1 + q) \} \neg y \leq r \wedge x = r + y \times q$	D3 (10)
12	<b>true</b> $\{ ((r := x; q := 0); \text{while } y \leq r \text{ do } (r := r - y; q := 1 + q)) \} \neg y \leq r \wedge x = r + y \times q$	D2 (5, 11)

## Une discussion de tout ce qui reste à faire :

- Vérifier la terminaison et l'absence d'erreurs à l'exécution.
- Arithmétiques (virgule flottante), tableaux, enregistrements, procédures, fonctions, récursion, goto, pointeurs.

## Un argumentaire : pourquoi vérifier les programmes ?

- Coût élevé du test.
- Coût très élevé de l'erreur.
- Documentation ; portabilité.

*When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of electronics.*

*The cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, a crashed aeroplane, or a world war. Thus, the practice of program proving is not only a theoretical pursuit, followed in the interest of academic responsibility, but a serious recommendation for the reduction of the costs associated with programming error.*

*However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs. As in other areas, reliability can be purchased only at the price of simplicity.*

## **Point d'étape**

---



Dès 1969, les grandes lignes de la vérification déductive sont déjà tracées dans les travaux de Floyd et de Hoare.

Beaucoup de travail reste à faire :

- Années 1970, 1980 : approfondissement des bases de la «logique de Hoare» (→ cours n° 2)
- À partir des années 1990 : mise en pratique sous forme d'outils de vérification déductive (→ séminaires n° 1 et 2)

Le prochain grand tournant du domaine sera vers 2000...