Call/CC, un coup d'état sur le contrôle

Yann Régis-Gianas

Séminaire Code Source – 23 mars 2016

Une lutte de pouvoir pour le contrôle du calcul

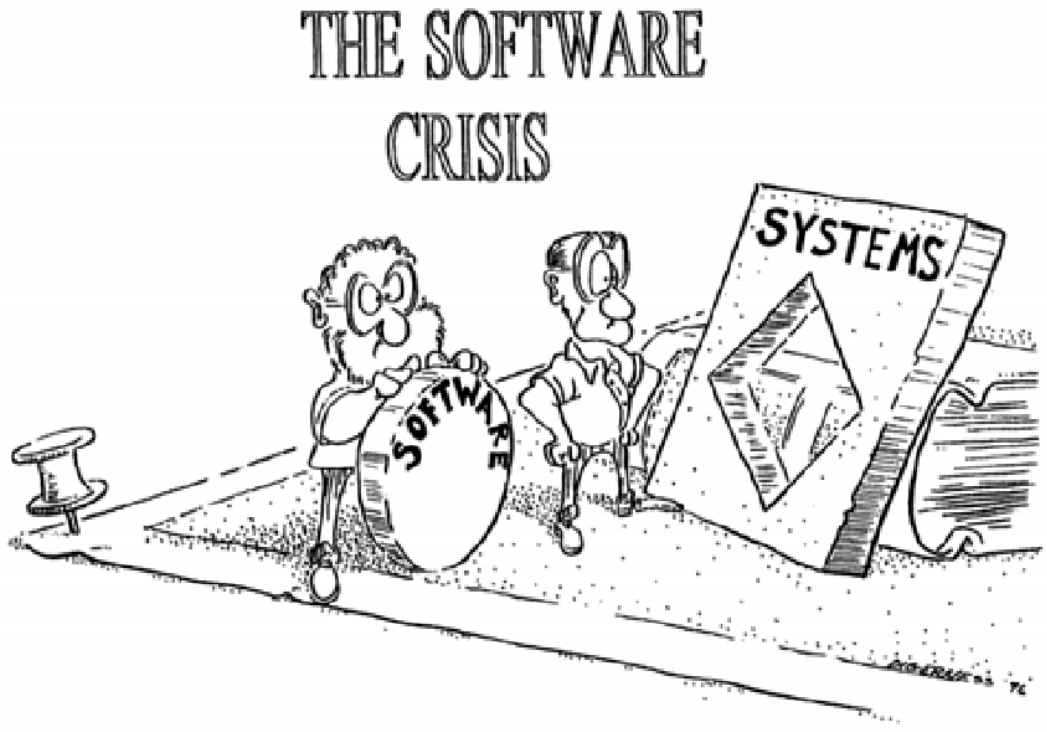
Plan

1. Les années 60
2. *Goto statement Considered Harmful*
   – E. Dijsktra (1968)
3. *Structured Programming with Goto statement*
   – D. Knuth (1974)
4. *A Correspondence Between ALGOL 60 and Church's Lambda-Notation*
   – P. Landin (1964)
5. Aujourd'hui: Compositionalité, Expressivité & Typabilité

Les années 60

(Le début de la civilisation pour les programmeurs)

La crise du logiciel



Deux méthodes pour un bon logiciel:

Raisonner localement sur chaque composant logiciel

Construire des composants **par composition**

Deux propriétés d'un bon logiciel:

La **responsabilités sont séparées**

Les **interfaces sont miminales**

Programmer, c’est communiquer avec la machine

…mais aussi avec les programmeurs.

Goto statement Considered Harmful
– E. Dijkstra (1968)
(La dictature du Puriste)

Anecdote

« Avant qu'ils n'introduisent cette fichue programmation structurée, nous étions libres! »

– Un programmeur anonyme rencontré en 2003 dans un bar de Paris 11ème.

Anecdote

« Avant qu'ils n'introduisent cette fichue programmation structurée, nous étions libres! »

– Un programmeur anonyme rencontré en 2003 dans un bar de Paris 11ème.

Nous allons voir deux exemples de programmes écrits “librement”.

Code Spaghetti en FORTRAN

```
1      READ (*, *) I, J, K
2      If (I-J) 10, 20, 30
3  10  If (K) 30, 40, 50
4  20  I=J*K
5      If (I-25) 40, 50, 60
6  30  J=K*3
7      GO TO 20
8  40  K=K+1
9      GO TO 10
10 50  WRITE (*, *) I, J, K
11     GO TO 70
12 60  WRITE (*, *) J, K
13 70  END
```

http://www.nku.edu/~foxr/CIT130/tutorials/spaghetti1.html

Duff's device en C

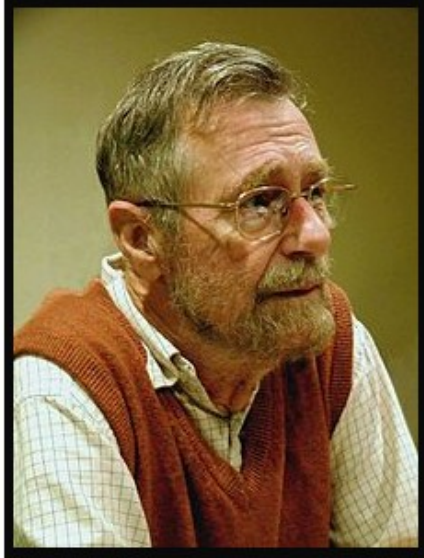
```
1 send(to, from, count)
2 register short *to, *from;
3 register count;
4 {
5     do { /* count > 0 assumed */
6       *to = *from++;
7     } while(--count > 0);
8 }
```

Duff's device en C

```
1  send(to, from, count)
2  register short *to, *from;
3  register count;
4  {
5      register n = count / 8;
6      do {
7          *to = *from++;
8          *to = *from++;
9          *to = *from++;
10         *to = *from++;
11         *to = *from++;
12         *to = *from++;
13         *to = *from++;
14         *to = *from++;
15     } while (--n > 0);
16 }
```

Duff's device en C

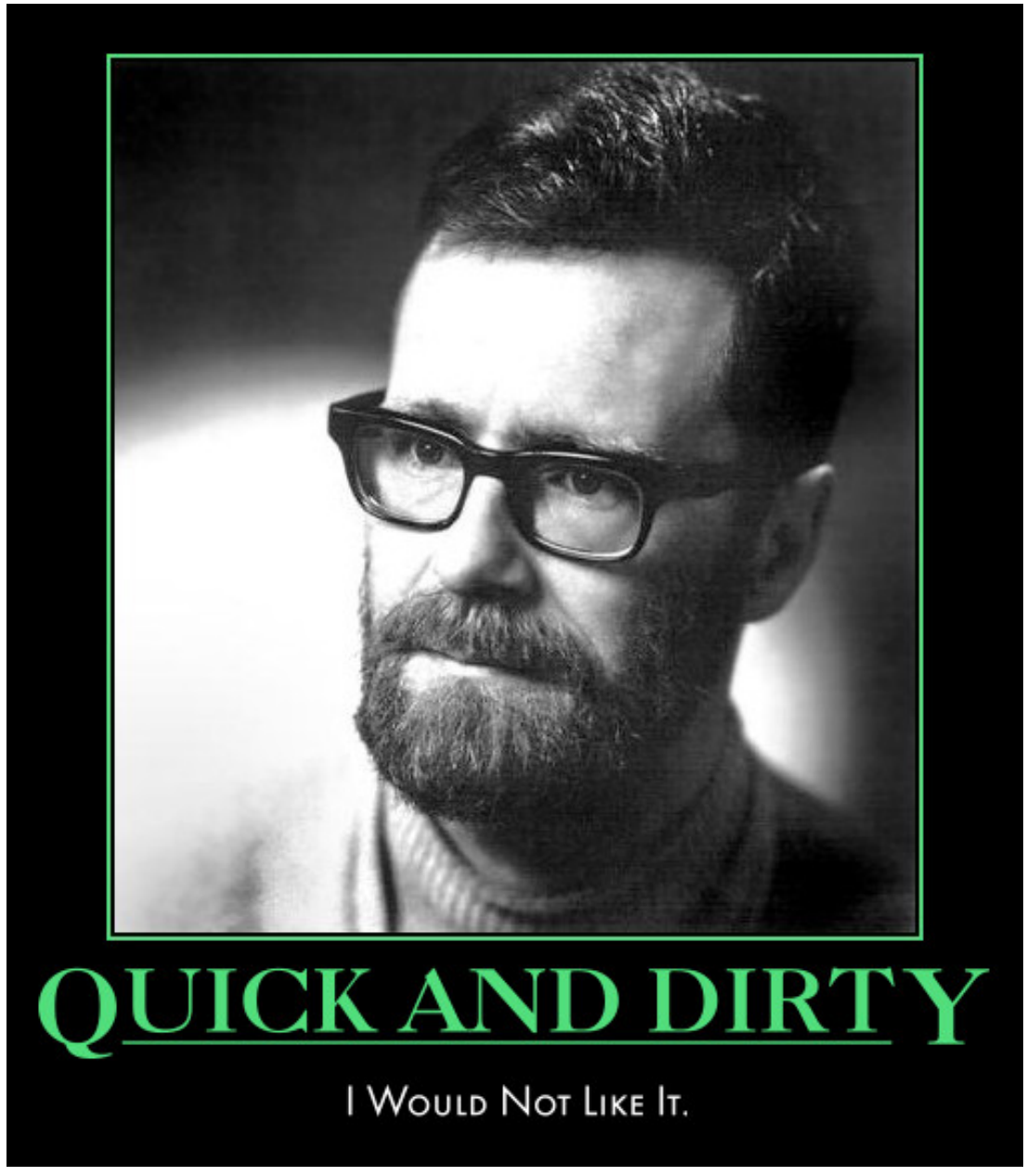
```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```



If 10 years from now, when you are doing something quick and dirty, you suddenly visualize that I am looking over your shoulders and say to yourself: 'Dijkstra would not have liked this', well that would be enough immortality for me.

(Edsger Dijkstra)

izquotes.com


QUICK AND DIRTY
I Would Not Like It.

Goto Statement Considered Harmful

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and ***I became convinced that the go to statement should be abolished from all "higher level" programming languages*** *(i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.*

Goto Statement Considered Harmful

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and ***I became convinced that the go to statement should be abolished from all "higher level" programming languages*** *(i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.*

(L'état d'urgence.)

Goto Statement Considered Harmful

L'argument principal de Dijkstra

[...] our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason ***we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process****, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.*

Localité du raisonnement

*En l'absence de **goto**, on peut définir un système de coordonnées pour caractériser le chemin d'exécution menant à un certain point du programme.*

- Si cette instruction est précédée par une autre instruction dans le code source alors le contrôle provient de celle-ci.
- Si cette instruction se situe dans une branche d'une conditionnelle alors il suffit d'observer quelle est la condition qui a permis de suivre cette branche.
- Si cette instruction se trouve en début d'une procédure alors il suffit de préciser quelle imbrication d'appels de procédure a pu conduire à l'appel de la procédure considérée.
- Si cette instruction se situe dans une boucle, alors il suffit de caractériser pour quelle itération de la boucle l'instruction peut être exécutée.

Localité du raisonnement

*En l'absence de **goto**, on peut définir un système de coordonnées pour caractériser le chemin d'exécution menant à un certain point du programme.*

- Si cette instruction est précédée par une autre instruction dans le code source alors le contrôle provient de celle-ci.
- Si cette instruction se situe dans une branche d'une conditionnelle alors il suffit d'observer quelle est la condition qui a permis de suivre cette branche.
- Si cette instruction se trouve en début d'une procédure alors il suffit de préciser quelle imbrication d'appels de procédure a pu conduire à l'appel de la procédure considérée.
- Si cette instruction se situe dans une boucle, alors il suffit de caractériser pour quelle itération de la boucle l'instruction peut être exécutée.

(Une version édulcorée des règles de raisonnement formel à la Floyd-Hoare-Dijsktra.)

Goto Statement Considered Harmful

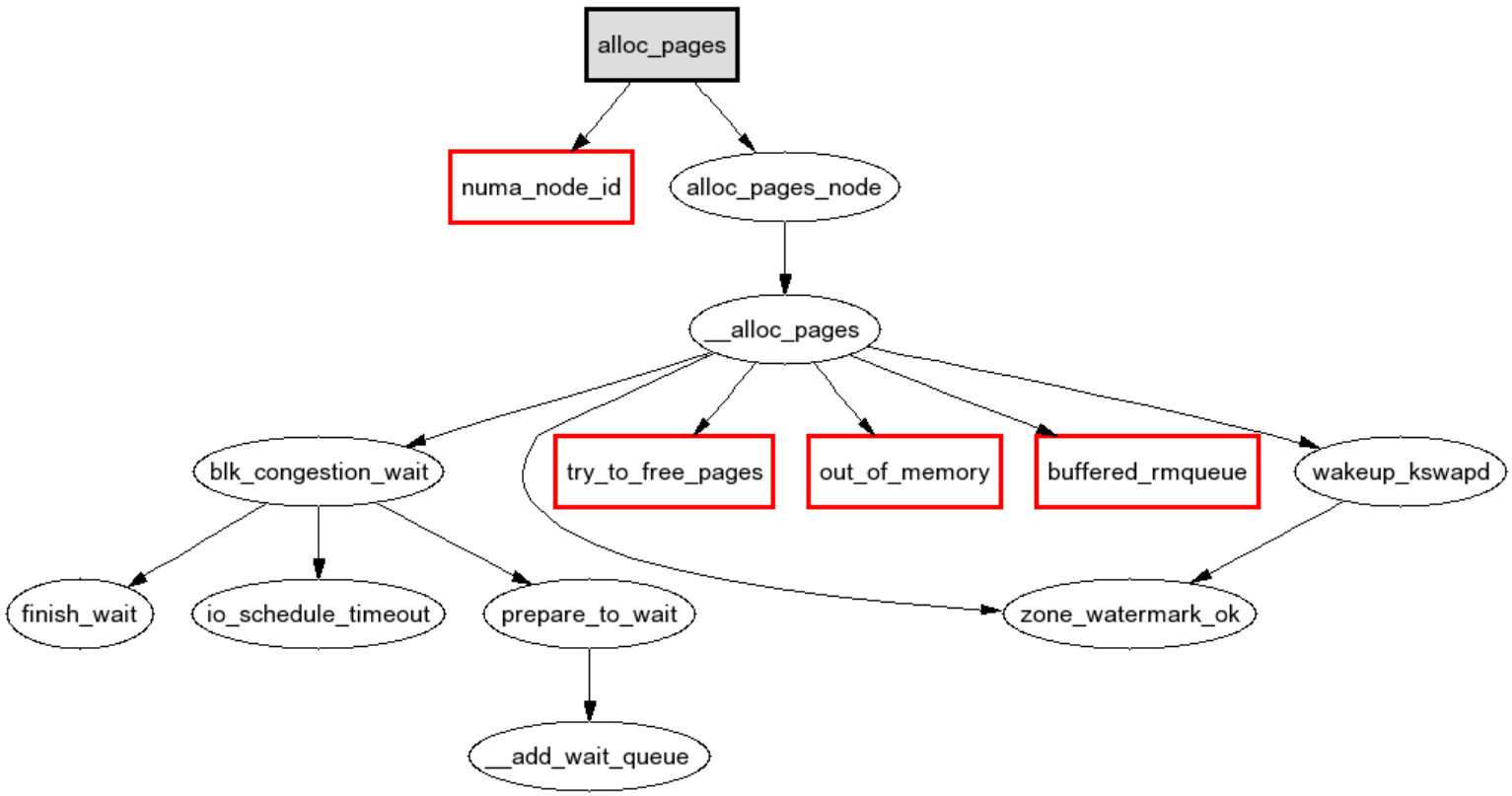
The go to statement as it stands is just ***too primitive****; it is* ***too much an invitation to make a mess of one's program****. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.*

La programmation structurée

Restreindre les constructions du langage de programmation
à un sous-ensemble suffisamment expressif
et pour lequel il existe des systèmes de preuve de programmes.

- Structures de contrôle de haut-niveau: **while**, **if-then-else**,…
- Procédures et fonctions (de seconde classe).
- Suppression du **goto**
  (Un point unique de sortie et d'entrée pour les boucles.)
- Minimisation des variables globales au profit de variables locales.

L'arbre des appels de fonctions

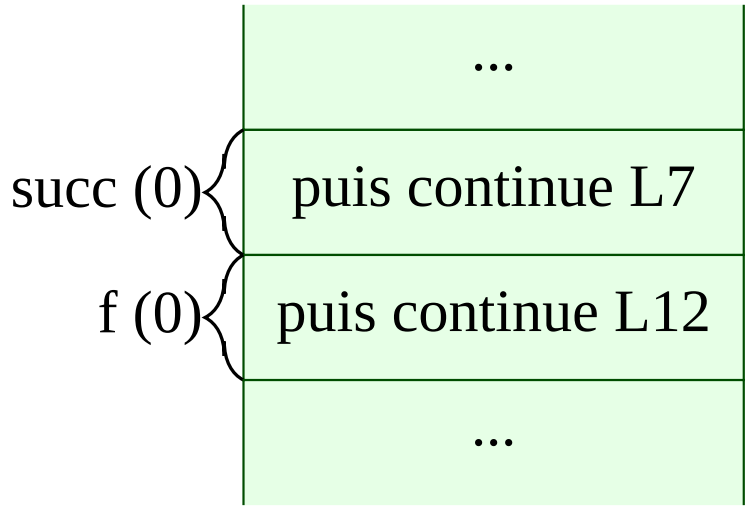


Graphe des appels de `alloc_pages()` dans le noyau Linux 2.6.12-rc2 [1]

[1] `http://www.csn.ul.ie/~mel/projects/codeviz/`

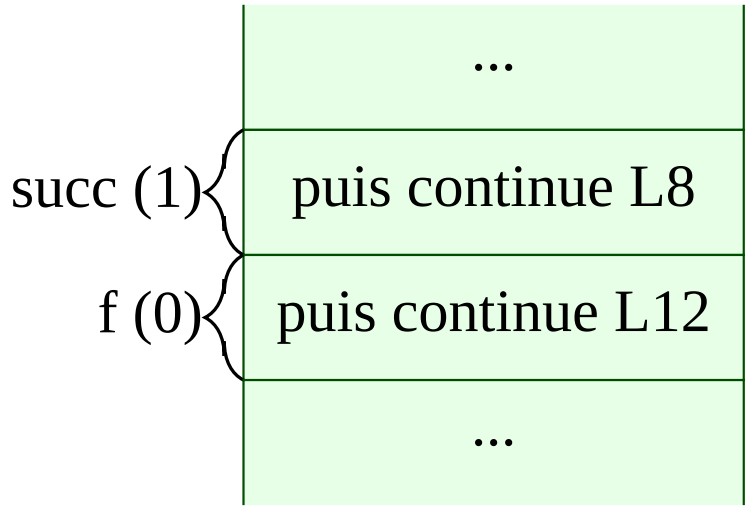
Discipline de pile

```
1  int succ (int x) {
2      ▷  return (x + 1);
3  }
4  int f (int y) {
5    int i, j = 0;
6    i = succ (0);
7    j = succ (1);
8    return i + j;
9  }
10 int h () {
11   int x = f (0);
12   return 2 * x;
13 }
```



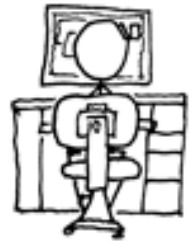
Discipline de pile

```
1  int succ (int x) {
2      ▷  return (x + 1);
3  }
4  int f (int y) {
5    int i, j = 0;
6    i = succ (0);
7    j = succ (1);
8    return i + j;
9  }
10 int h () {
11   int x = f (0);
12   return 2 * x;
13 }
```




I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.


EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE



Structured Programming with Goto statement
– D. Knuth (1974)
(La révolte)


Donald
Knuth '60
50th Reunion

Exemple

Soit une table A contenant des valeurs $A[0] \ldots A[m]$ et une table B servant à compter combien de fois l'élément x de A a été recherché. Lorsque l'on recherche un élément dans A et qu'il ne s'y trouve pas, on souhaite le rajouter.

```
1 for (i = 0; i <= m; ++i)
2   if (A[i] == x) goto found;
3 not_found: i = m + 1; m = i; A[i] = x; B[i] = 0;
4 found: B[i] = B[i] + 1;
```

```
1 i = 0;
2 while (i <= m && A[i] != x) i++;
3 if (i > m) { m = i; A[i] = x; B[i] = 0; }
4 B[i] = B[i] + 1
```

Lequel de ces programmes est le plus lisible? Le plus efficace?

La gestion des erreurs système en C

```
1 #define unix_call(X) X; if (errno != 0) goto
    unix_error;
2 void do_some_unix_stuff () {
3     unix_call (fd = fopen ("foobar.txt", "a"));
4     return
5 unix_error:
6     // If the control flows here, an Unix error occured.
7     // Report the error.
8 }
9
```

Structures de contrôle manquantes

```
1  compare:
2  if A[i] < x
3  then if L[i] != 0
4    then i := L[i]; goto compare;
5    else L[i] := j ; goto insert fi;
6  else if R[i] != 0
7    then i := R[i]; goto compare ;
8    else R[i] := j ; goto insert fi;
9  fi;
10 insert: A[j] := x;
11 L[j] := 0; R[j] := 0; j := j + 1;
```

Structures de contrôle manquantes

```
1 t := true;
2 while t do
3   begin if A[i] < x
4     then if L[i] != 0 then i := L[i];
5        else L[i] := j; t := false; fi;
6     else if R[i] != 0 then i := R[i];
7        else R[i] := j; t := false; fi
8   end
9 A[j] := x;
```

Structures de contrôle manquantes

```
1  loop until left leaf hit or right leaf hit:
2    if A[i] < x
3    then if L[i] != 0 then i := L[i];
4     else left leaf hit; fi;
5    else if R[i] != 0 then i := R[i];
6     else right leaf hit; fi
7    fi;
8  repeat;
9  then left leaf hit => L[i] := j
10       right leaf hit => R[i] := j;
11 fi;
12 A[j] := x; L[j] := 0; R[j] := 0; j := j + 1;
```

Dijkstra n’est pas un extrémiste

« Please don’t fall into the trap of believing that I am terribly dogmatical about [the go to statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline! »

– Dijkstra (1973), communication personnelle avec Donald Knuth

A Correspondence Between ALGOL 60 and Church's Lambda-Notation
– P. Landin (1964)
(Les lumières)

Le père de la programmation fonctionnelle?

Le λ-calcul de Church

$$
\begin{array}{rcll}
t & ::= & x & \text{Variable} \\
 & | & t\ t & \text{Application} \\
 & | & \lambda x.t & \lambda\text{-abstraction}
\end{array}
$$

Correspondance

$$
\begin{array}{ll}
\textbf{let } a = A & \{\lambda(a, b, f). \\
\textbf{and } b = B & \quad \{\lambda(g, h). \\
\textbf{and } f(x, y) = F & \quad\quad \{\lambda k.X\}[\lambda(u, v).\lambda x.K]\} \\
\textbf{let rec } g(x) = G & \quad [Y\lambda(g, h).(\lambda x.G, \lambda(y, z).H]\} \\
\quad \textbf{and } h(y, z) = H & [A, B, \lambda(x, y).F] \\
\textbf{let } k(u, v)(x) = K & \\
X &
\end{array}
$$

Programmer les structures de contrôle

Comment découper

```
1 for (i = 0; i < N; ++i) {
2   t[i] -= sqrt (t[i])
3 }
```

en deux programmes:

```
1 for (i = 0; i < N; ++i)
2   ?
```

```
1 ? {
2   t[i] -= sqrt (t[i])
3   }
```

?

Les fonctions de première classe

```
1  let iter_to n f =
2    for i = 0 to n do f i done
3
4  let main =
5    iter_to n (fun i -> t[i] := t[i] - sqrt (t[i]))
6
```

Programmer ses propres opérateurs de contrôle

```
1 let until events body init =
2   let rec aux accu =
3       match body accu with
4         | `Continue accu -> aux accu
5         | `Stop event -> event accu
6   in
7   aux init
8
```

Programmer ses propres opérateurs de contrôle

```
let find y a =
  let rec do_ init =
    until [found; not_found]
      (fun i ->
        if i = 0 then `Stop not_found
        else if !a.(i) = y then `Stop found
        else `Continue (i - 1))
  and not_found i = a := Array.append !a [|y|]; found i
  and found i = i
  in
  do_ (Array.length !a)

```

L'opérateur **J** de P. Landin

$$J(\lambda L.S)$$

L'opérateur **J** de P. Landin

$$J(\lambda L.S)$$

- L : la *continuation* du calcul sous la forme d'une fonction.
- S : l'usage de la *continuation*.
- J : la *capture de la continuation*.

Call/CC

```
1 type 'a cont
2 val callcc: ('a cont -> 'a) -> 'a
3 val throw: 'a cont -> 'a -> 'b
```

Une machine à remonter dans le temps

```
1  let now () =
2    let s = ref None in
3    callcc (fun k -> s := Some k);
4    s
5
6  let past_travel_to date =
7    match !date with
8      | None -> assert false
9      | Some date -> throw date ()
10
11 let cake = ref `Cake
12 let eat () =
13   assert (!cake = `Cake);
14   Printf.printf "I am eating the cake! Miamee!\n"
15   cake := `NoMoreCake
16
17 let _ =
18   let t = now () in
19   Printf.printf "I am in front of the cake\n.";
20   eat ();
21   past_travel_to t
```

Compositionalité, Expressivité & Typabilité
(Une démocratie participative?)

Le contrôle des données
Les données du contrôle

- Avec Call/CC, **les étiquettes sont des valeurs** de première classe.
- Les programmeurs **leur associent naturellement une sémantique** (comme à toute valeur).
- Grâce à l'**abstraction de type**, le contrôle du contrôle est programmable.

Le contrôle des données
Les données du contrôle

- Avec Call/CC, **les étiquettes sont des valeurs** de première classe.
- Les programmeurs **leur associent naturellement une sémantique** (comme à toute valeur).
- Grâce à l'**abstraction de type**, le contrôle du contrôle est programmable.

Pour finir, nous allons illustrer ces points en implémentant des coroutines.

Coroutines

Les co-routines sont une généralisation des routines inventées par Conway: l'évaluation d'une coroutine peut être interrompue et reprise plus tard.

Coroutines

Les co-routines sont une généralisation des routines inventées par Conway: l'évaluation d'une coroutine peut être interrompue et reprise plus tard.

Comment implémenter les coroutines?

En C via le “Duff’s device”

```c
/* Decompression code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (c == 0xFF) {
        len = getchar();
        c = getchar();
        while (len--)
            emit(c);
    } else
        emit(c);
}
emit(EOF);
```

```c
/* Parser code */
while (1) {
    c = getchar();
    if (c == EOF)
        break;
    if (isalpha(c)) {
        do {
            add_to_token(c);
            c = getchar();
        } while (isalpha(c));
        got_token(WORD);
    }
    add_to_token(c);
    got_token(PUNCT);
}
```

En C via le “Duff device”

```c
int decompressor(void) {
    static int repchar;
    static int replen;
    if (replen > 0) {
        replen--;
        return repchar;
    }
    c = getchar();
    if (c == EOF)
        return EOF;
    if (c == 0xFF) {
        replen = getchar();
        repchar = getchar();
        replen--;
        return repchar;
    } else
        return c;
}
```

```c
void parser(int c) {
    static enum {
        START, IN_WORD
    } state;
    switch (state) {
        case IN_WORD:
        if (isalpha(c)) {
            add_to_token(c);
            return;
        }
        got_token(WORD);
        state = START;
        /* fall through */

        case START:
        add_to_token(c);
        if (isalpha(c))
            state = IN_WORD;
        else
            got_token(PUNCT);
        break;
    }
}
```

```
1 int function(void) {
2     int i;
3     for (i = 0; i < 10; i++)
4         return i;
5 }
6
```

```
1  int function(void) {
2      static int i, state = 0;
3      switch (state) {
4          case 0: goto LABEL0;
5          case 1: goto LABEL1;
6      }
7      LABEL0:
8      for (i = 0; i < 10; i++) {
9          state = 1;
10         return i;
11         LABEL1:;
12     }
13 }
14
```

```
1  int function(void) {
2      static int i, state = 0;
3      switch (state) {
4          case 0:
5          for (i = 0; i < 10; i++) {
6              state = 1;
7              return i;
8              case 1:;
9          }
10     }
11 }
```

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(i,x) do { state=i; return x; case i:; } while (0)
#define crFinish }
int function(void) {
    static int i;
    crBegin;
    for (i = 0; i < 10; i++)
        crReturn(1, i);
    crFinish;
}
```

```
1 #define crReturn(x) do { state=__LINE__; return x; \
2                          case __LINE__:; } while (0)
```

En C via le “Duff device”

```c
int decompressor(void) {
  static int c, len;
  crBegin;
  while (1) {
      c = getchar();
      if (c == EOF)
          break;
      if (c == 0xFF) {
          len = getchar();
          c = getchar();
          while (len--)
              crReturn(c);
      } else
          crReturn(c);
  }
  crReturn(EOF);
  crFinish;
}
```

```c
void parser(int c) {
  crBegin;
  while (1) {
      if (c == EOF)
          break;
      if (isalpha(c)) {
          do {
              add_to_token(c);
              crReturn( );
          } while (isalpha(c));
          got_token(WORD);
      }
      add_to_token(c);
      got_token(PUNCT);
      crReturn( );
  }
  crFinish;
}
```

Par continuations

```
1  module type S = sig
2    (* Coroutines with input 'i and output 'o *)
3    type ('i,'o) coroutine
4    type ('i,'o) coroutine_body = ('i, 'o) coroutine -> 'i -> 'o
5    (* create coroutine from body function *)
6    val create : ('i, 'o) coroutine_body -> ('i,'o) coroutine
7    (* Asymmetric coroutine operator: *)
8    (* resume a suspended coroutine. *)
9    val resume: ('i,'o) coroutine -> 'i -> 'o
10   (* Complement to resume: suspend the current coroutine, return to
       caller *)
11   val yield : ('i, 'o) coroutine -> 'o -> 'i
12 end
13
```

Par continuations

```
1  (* A co-routine has to remember its own continuation
2     when it is suspended as well as the continuation of
3     its caller when it is resumed. *)
4  type ('i, 'o) coroutine_data = {
5    suspension : ('i cont) option ref;
6    caller     : ('o cont) option ref;
7    body       : 'i -> 'o
8  }
9
10 type ('i, 'o) coroutine =
11   Coroutine of ('i, 'o) coroutine_data
12
13 (* A co-routine is defined using a function that
14    will ultimately produce a value of type ['o]
15    from a value of type ['i]. This function can
16    interrupt itself using [yield]. *)
17 type ('i,'o) coroutine_body =
18    ('i, 'o) coroutine -> 'i -> 'o
19
```

Par continuations

```
let resume (Coroutine co) v =
  match !(co.suspension) with
    (* This is the first this co-routine is runned, so we
       directly use the initial function that was given
       at the creation of the co-routine. *)
    | None ->
      callcc (fun k -> co.caller := Some k; co.body v)

    (* The co-routine has already been suspended. It
       first captures the caller's continuation and
       reinstall its own suspended continuation. *)
    | Some suspension ->
      callcc (fun k -> co.caller := Some k; throw suspension v)

```

Par continuations

```
let yield (Coroutine co) v =
(* This time, this is the other way around: we have to save the
   co-routine continuation and restore the caller's. *)
callcc (fun k ->
  co.suspension := Some k;
  throw (unsome !(co.caller)) v
)

```

Implémentation de Call/CC

- `callcc` réifie la pile d'appels en une valeur.
- `throw` remplace la pile d'appels par une pile réifiée.

Implémentation de Call/CC

- `callcc` réifie la pile d'appels en une valeur.
- `throw` remplace la pile d'appels par une pile réifiée.

Comment faire lorsque le langage ne possède pas une telle construction?

L’expressivité des langages fonctionnels

Il suffit de savoir **représenter** un mécanisme calcul
et d’en **simuler** l’interprétation dans le langage de programmation.

Les monades

Pour embarquer un langage de programmation $\mathcal{L}$ dans un langage $\mathcal{H}$,
il suffit de définir une **monade** en $\mathcal{H}$.

```
1 (* Le type des calculs de ℒ *)
2 (* produisant une valeur de type 'a. *)
3 type 'a m
4 (* Tout calcul de ℋ est un calcul de ℒ. *)
5 val return : 'a -> 'a t
6 (* La composition de deux calculs de ℒ. *)
7 val bind : 'a t -> ('a -> 'b t) -> 'b t
```

La monade de non déterminisme

Soit le type des cartes de restaurant:

```
1 type f = M of string | Or of f list | And of f list
```

Comment calculer tous les menus possibles d'une carte donnée?

La monade de non déterminisme

```
type 'a t
val pick : 'a list -> 'a t
val return : 'a -> 'a t
val fail : 'a t
val ( >>= ) : 'a t -> ('a -> 'b t) -> 'b t
val run : 'a t -> 'a list
```

La monade de continuation

```
module Cont : sig
  type 'a t = ('a -> int) -> int
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val callCC: (('a -> 'b t) -> 'a t) -> 'a t
end = struct
  type 'a t = ('a -> int) -> int

  let return x =
    fun cont -> cont x

  let bind m f =
    fun cont -> m (fun x -> (f x) cont)

  let callCC k =
    fun cont -> k (fun x -> (fun _ -> cont x)) cont
end

```

Conclusion
(Propagande)

L’histoire d’une lutte de pouvoir pour le contrôle du calcul.

L’histoire d’une lutte de pouvoir pour le contrôle du calcul.
Mais qui en sont les véritables belligérants?

L’histoire d’une lutte de pouvoir pour le contrôle du calcul. Mais qui en sont les véritables belligérants?

- Des processus de calcul contre d’autres processus de calcul?
- Des programmeurs contre d’autres programmeurs?
- Le langage de programmation contre le programmeur?
- La programmation fonctionnelle (typée) sera-t-elle victorieuse?

Merci pour votre attention.
Avez-vous des questions?

Un passage du papier de Knuth

[...] and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages).

Un passage du papier de Knuth

[...] and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages).

C++, Caml et Coq sont nés en 1985, 1986 et 1984

Un passage du papier de Knuth

[...] and my dream is that by 1984 we will see a consensus developing for a really good programming language (or, more likely, a coherent family of languages).

C++, Caml et Coq sont nés en 1985, 1986 et 1984
Coq serait-il le rêve de Knuth?

Références

- https://en.wikipedia.org/wiki/Duff%27s_device
- http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
- http://histinf.blogs.upv.es/files/2010/12/crisis.png