Structures de contrôle, premier cours

Naissance des structures de contrôle : du `goto` à la programmation structurée

Xavier Leroy

2024-01-25

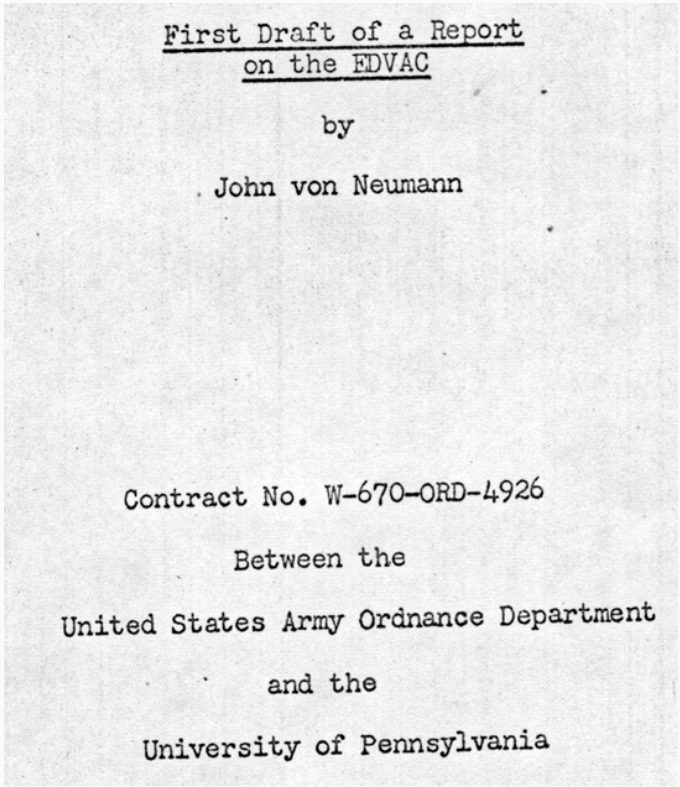
Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

Premiers ordinateurs, premiers langages

Le calculateur programmable à programme enregistré en mémoire

Une architecture décrite en 1945 par John von Neumann et semi-indépendamment par Alan Turing.

À la base de presque tous les ordinateurs construits depuis.



First Draft of a Report
on the EDVAC

by

John von Neumann

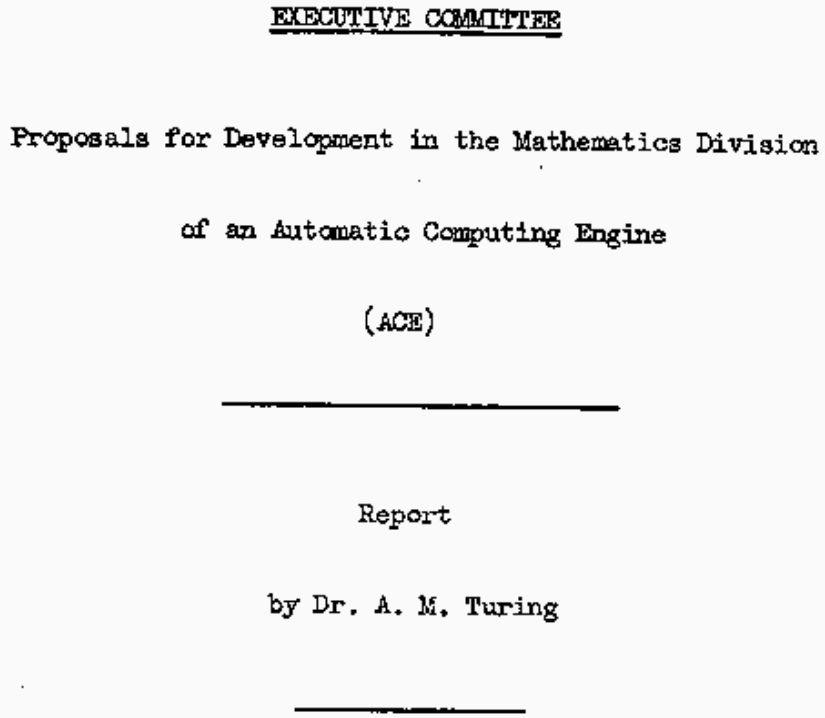
Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania



EXECUTIVE COMMITTEE

Proposals for Development in the Mathematics Division

of an Automatic Computing Engine

(ACE)

Report

by Dr. A. M. Turing

Compteur de programme et branchements

Un registre ≪PC≫, le compteur de programme (*program counter*), contient l'adresse en mémoire de la prochaine instruction.

Les instructions ≪normales≫ incrémentent le PC
$\rightarrow$ exécution en séquence.

Les instructions de branchement positionnent la valeur du PC
$\rightarrow$ saut à un point quelconque du programme.

Exemple : afficher 1!, 2!, …, n!, …

```
 0: set r1, 1                (r1 = la valeur de n!)
 4: set r2, 1                (r2 = la valeur de n)
 8: output r1
12: add r2, r2, 1
16: mul r1, r1, r2
20: branch 8
```

Branchements conditionnels

Les processeurs modernes fournissent aussi des instructions de branchement conditionnel qui

- positionnent le PC si une condition est vraie (p.ex. si un registre contient la valeur 0);
- continuent en séquence si la condition est fausse.

Exemple : une boucle comptée qui calcule $n!$

```
                                    (r1 = la valeur de n)
 0: set r2, 1
 4: mul r2, r2, r1
 8: sub r1, r1, 1
12: brnz r1, 4
                                    (r2 = la valeur de n!)
```

Alternative historique : le code auto-modifiant

Ni von Neumann ni Turing ne prévoient de branchements conditionnels, utilisant plutôt le fait que le code est stocké en mémoire vive et peut se modifier lui-même !

Exemple : branchement vers 0 si `r1` $= 0$ et vers 4 si `r1` $= 1$.
On suppose que l'instruction `branch` n est encodée par `0x76000000` $+ n$.

```
184: set r2, 0x76000000
188: mul r3, r1, 4
192: add r2, r2, r3
196: store r2, 200
200: nop
```

(instruction modifiée dynamiquement)

Le langage machine

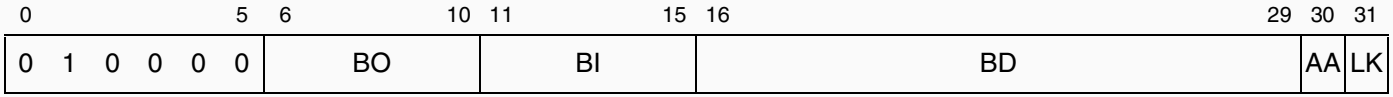
Un encodage des instructions et de leurs opérandes sous forme binaire (souvent 32 bits).

Exemple : branchement conditionel pour l'architecture Power.

bc | Base | User | **bc**

Branch Conditional [and Link] [Absolute]

bc	BO,BI,BD	(AA=0, LK=0)
bca	BO,BI,BD	(AA=1, LK=0)
bcl	BO,BI,BD	(AA=0, LK=1)
bcla	BO,BI,BD	(AA=1, LK=1)

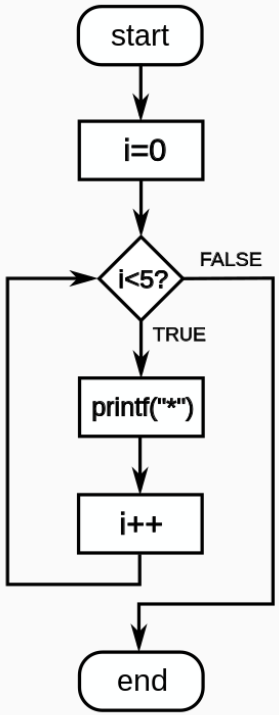



```
if ¬BO2 then CTR ← CTR - 1
ctr_ok  ← BO2 | ((CTRm:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
    if AA=1 then a ← 640 else a ← CIA
    NIA ← m0 || (a + EXTS(BD||0b00))m:63
else           NIA ← m0 || (CIA + 4)m:63
if LK=1 then LR ← CIA + 4
```

Pour les humains : très difficile à écrire… et impossible à relire !

Les organigrammes de programmes (ordinogrammes, *flow charts*)

Une représentation graphique et informelle des programmes.



(Giacomo Alessandroni, CC BY-SA 4.0)

Production du langage machine : placement des calculs en mémoire, ajout de branchements, codage binaire des instructions.

Le langage d'assemblage (1947)

Une représentation textuelle du langage machine :

- mnémoniques pour nommer les instructions ;
- étiquettes pour désigner des points de programme ;
- commentaires pour documenter le code.

Exemple : une boucle comptée qui calcule $n!$

```
;  Compute factorial of n.
;  n is passed in r1.
;  n! is left in r2.
;  r1 is clobbered.
          set r2, 1              ; initialize result to 1
  again:  mul r2, r2, r1         ; multiply result by n
          sub r1, r1, 1          ; decrement n
          brnz r1, again         ; repeat until n = 0
                                 ; here, result is n!
```

Macro-assembleurs et *autocoders* (IBM, 1955)

Des pseudo-instructions que l'assembleur expanse en des séquences d'instructions souvent utilisées.

Exemple : boucle comptée

Code source

```
BEGIN_LOOP(lbl, r1, 0)
...
...
END_LOOP(lbl, r1, 100)
```

Code après expansion

```
        set r1, 0
lbl:
        ...
        ...
        add r1, r1, 1
        cmp r1, 100
        brlt lbl
```

Premières structures de contrôle

Le langage FORTRAN (1957)

Des expressions complexes, en notation algébrique usuelle, automatiquement traduites en instructions machine (*FORmula TRANslator*).

Source FORTRAN :

```
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

Code assembleur :

```
mul t1, b, b
mul t2, a, c
mul t2, t2, 4
sub t1, t1, t2
sqrt d, t1
mul t3, a, 2
sub x1, d, b
div x1, x1, t3
neg x2, b
sub x2, x2, d
div x2, x2, t3
```

Mais aussi : la première structure de contrôle !

Le contrôle en FORTRAN I

La boucle comptée :

```
      DO 100 I = 1, N
      ...
      ...
100:  ...
```

Répète l'exécution des lignes entre `DO` et `100` (exclu)
avec `I` prenant les valeurs 1, 2, ..., `N`.

Compilation efficace : test à la fin de la boucle, déroulage, etc.

Le contrôle en FORTRAN I

Étiquettes et `GO TO`.

Branchement conditionnel ternaire `IF X L1, L2, L3` (selon que `X` $<$ 0 ou `X` $=$ 0 ou `X` $>$ 0).

```
      IF X 701, 702, 702
701:  X = -X
702:  ...
```

Branchement à une étiquette calculée.

```
ASSIGN 100 TO DEST
...
GO TO DEST
```

Fortran IV (1961) : branchement conditionnel logique.

```
IF X .LT. Y GOTO 100
```

Les commandes structurées d'Algol (1960)

Les expressions sont formées à partir de constantes et de variables, combinés à l'aide d'opérateurs. De même, Algol introduit l'idée que les commandes *s* (*statements*) d'un programme sont construits à partir de commandes élémentaires

- affectation : `x := expr`
- appels de procédures : `proc(arg1, arg2)`

en les combinant via des structures de contrôle

- séquence («blocs») : `begin` s_1`;` s_2`; ... end`
- conditionnelle : `if` *be* `then` s_1 `else` s_2
- boucle comptée : `for i :=` e_1 `step` e_2 `until` e_3 `do` s
- boucle générale : `while` *be* `do` s

Un changement de point de vue sur les programmes

Au niveau de la syntaxe : utilisation de grammaires formelles comme la ≪forme de Backus-Naur≫ (BNF) et d'algorithmes d'analyse syntaxique récursifs.

Commandes : $s ::= x$ `:=` e
$\mid proc(e_1, \ldots, e_n)$
$\mid$ `begin` s_1`;`$\ldots$`;` s_n `end`
$\mid$ `if` be `then` s_1 [`else` s_2]
$\mid$ `while` be `do` s
$\mid$ `for` i `:=` e_1 `step` e_2 `until` e_3 `do` s

Au niveau de la sémantique (intuitive, puis formelle) :
l'idée s'impose que la signification d'une commande est entièrement déterminée par celles de ses sous-commandes.

Contrôle non structuré en Algol

Algol autorise aussi le contrôle non structuré, avec des sauts vers des étiquettes placées sur des commandes :

$s ::= \ldots$

$\mid L : s$	commande étiquetée L
$\mid$ `go to` $L \mid$ `goto` $L \mid$ `go` L	saut à l'étiquette L

Les étiquettes ont une portée (*scope*) qui est le bloc englobant leur définition. On ne peut pas sauter «à l'intérieur» d'un bloc.

Même niveau ✔

```
begin
  integer i;
L:...
  goto L
end
```

Vers l'extérieur ✔

```
begin
  integer i;
  ... goto L ...
end;
L: ...
```

Vers l'intérieur ✘

```
goto L;
begin
  integer i;
L:...
end
```

Structures de contrôle + «goto»

Ce «style Algol» avec structures de contrôle et branchements `goto` s'impose très vite comme standard pour décrire et publier les algorithmes (dès 1960 dans *Comm. ACM*), à la place des organigrammes utilisés avant.

ALGORITHM 95
GENERATION OF PARTITIONS IN PART-COUNT FORM
FRANK STOCKMAL
System Development Corp., Santa Monica, Calif.

procedure partgen(c,N,K,G); **integer** N,K; **integer array** c; **Boolean** G;
comment This **procedure** operates on a given partition of the positive integer N into parts $\leqq K$, to produce a consequent partition if one exists. Each partition is represented by the integers $c[1]$ thru $c[K]$, where $c[j]$ is the number of parts of the partition equal to the integer j. If entry is made with G = **false**, **procedure** ignores the input array c, sets G = **true**, and produces the first partition of N ones. Upon each successive entry with G = **true**, a consequent partition is stored in $c[1]$ thru $c[K]$. For $N = KX$, the final partition is $c[K] = X$. For $N = KX+r$, $1 \leqq r \leqq K-1$, final partition is $c[K] = X$, $c[r] = 1$. When entry is made with **array** c = final partition, c is left unchanged and G is reset to **false**;

```
begin integer a,i,j;
        if ¬ G then go to first;
        j := 2;
        a := C[1];
test:   if a < j then go to B;
        c[j] := 1 + c[j];
        c[1] := a − j;
zero:   for i := 2 step 1 until j − 1
        do c[i] := 0;
        go to EXIT;
B:      if j = K then go to last;
        a := a + j × c[j];
        j := j + 1;
        go to test;
first:  G := true;
        c[1] := N;
        j := K + 1;
        go to zero;
last:   G := false;
EXIT: end partgen
```

Structures de contrôle + ≪goto≫

On retrouve cette combinaison de `goto` et de structures de contrôle dans beaucoup de langages impératifs :
Algol 68, Algol W, Pascal, Ada, Simula, PL/I, C, C++, …

Des langages ≪`goto` + boucle comptée≫ comme FORTRAN et BASIC ont évolué en adoptant des structures de contrôle d'Algol (`if...else...` en FORTRAN 77, `do while` en FORTRAN 90).

Quelques langages impératifs sans `goto` :
Modula-2 (1980), Python (1991), Java (1995).

Variations sur la conditionnelle

Cascade de tests booléens :

`if` be_1 `then` s_1 `elsif` be_2 `then` s_2 `elsif` `...` `else` s_n

aussi notée (`cond` (be_1 s_1) (be_2 s_2) `...` (`t` s_n)) en Lisp.

Analyse de cas sur une valeur entière ou d'un type énuméré :

```
case (grade) of
  'A' : s1
  'B', 'C': s2
  'D' : s3
  'F' : s4
end
```

```
switch (grade) {
  case 'A': s1; break;
  case 'B':
  case 'C': s2; break;
  case 'D': s3; break;
  case 'F': s4; break;
}
```

Variations sur les boucles

Boucles générales :

- test au début : `while` *be* `do` *s*
- test à la fin : `do` *s* `while` *be* ou `repeat` *s* `until` *be*
- tests au milieu : `loop` ...`exit if` *be*...`end`

Boucles comptées avec condition d'arrêt anticipé : (PL/I, Algol 68)

[`FOR` *index*] [`FROM` *first*] [`BY` *increment*] [`TO` *last*] [`WHILE` *condition*]
`DO` *statements* `OD`

Boucles qui itèrent sur une collection de valeurs :

```
for item in collection: s                    (Python)
for (Type item : collection) { s }           (Java)
```

Sorties anticipées de boucles

Arrêter la boucle englobante :

`break` (C, C++, Java) / `exit` (Ada) / `last` (Perl)

Arrêter l'itération courante et commencer la prochaine :

`continue` (C, C++, Java) / `next` (Perl)

Extension à plusieurs niveaux de boucles imbriquées :

(*multi-level exit*)

- sortir de la *N*-ième boucle englobante — `exit` *N* (Shell)
- sortir de la boucle étiquetée *L* — `break` *L* (Java)

```
xloop: for (int x = 0; x < dimx; x++)
  yloop: for (int y = 0; y < dimy; y++) {
    ... break xloop ... break yloop ...
}
```

Programmation structurée, contrôle structuré

Le mouvement pour une programmation structurée (1965–1975)

Un changement radical de point de vue sur le logiciel.

Passer de la vision

organigramme et code machine

à une vision

texte source construit, structuré,
directement lisible (sans organigramme),
sur lequel on peut raisonner
(informellement ou mathématiquement).

Le manifeste de ce mouvement : le livre *Structured Programming* de Dahl, Dijkstra, et Hoare (1972).

La polémique autour du contrôle structuré (1965–1975)

Une querelle entre deux styles de programmation :

- Une programmation avec beaucoup de ≪goto≫,
  souvent par transcription naïve d'un organigramme.
- Une programmation utilisant principalement des structures de contrôle (conditionnelles, boucles),
  souvent écrite directement, sans organigramme.

Le slogan de cette polémique :

Go to statement considered harmful

(Titre d'une brève communication de Dijkstra dans CACM 1968. Le titre a été choisi par la rédaction de CACM.)

Premiers signes

Since the summer of 1960, I have been writing programs in outline form, using conventions of indentation to indicate the flow of control. I have never found it necessary to take exception to these conventions by using ***go*** *statements.*

I used the keep these outlines as original documentation of a program, instead of using flow charts . . . Then I would code the program in assembly language from the outlines. Everyone liked these outlines better than the flow charts.

(Dewey Val Schorre, 1966)

Premiers signes

If you look carefully you will find that surprisingly often a ***go to*** *statement which looks back really is a concealed* ***for*** *statement. And you will be pleased to find how the clarity of the algorithm improves when you insert the* ***for*** *clause where it belongs.*

If the purpose [of a programming course] is to teach Algol programming, the use of flow diagrams will do more harm than good, in my opinion.

(Peter Naur, BIT, 1963).

Le feu aux poudres

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

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.

(E. W. Dikjstra, Communications of the ACM 11(3), 1968.)

Une réflexion sur la difficulté de savoir ≪où on en est≫ dans l'exécution d'un programme, et quel chemin a été suivi depuis le début du programme.

Quelles ≪coordonnées≫ pour désigner un point dans l'exécution ?

```
if (x == 0) {
    ...
} else if (y == 0) {
    ...          ⇐ VOUS ÊTES ICI
} else {
    ...
    ...
}
```

Séquence et conditionnelles :
point de programme courant + valeur booléenne des conditionnelles précédentes

Quelles ≪coordonnées≫ pour désigner un point dans l'exécution ?

```
while (x < y) {              ⇐ 2e itération
    while (a[x] != 0) {      ⇐ 5e itération
        if (x == 0) {
            ...
        } else if (y == 0) {
            ...              ⇐ VOUS ÊTES ICI
        } else {
            ...
            ...
        }
    }
}
```

Séquence, conditionnelles, et boucles :
point de programme + booléens + compte d'itérations pour chaque boucle.

Quelles ≪coordonnées≫ pour désigner un point dans l'exécution ?

```
while (x < y) {                ⇐ 2e itération
    while (a[x] != 0) {        ⇐ 5e itération
        if (x == 0) {
            L:...
        } else if (y == 0) {
            ...                ⇐ VOUS ÊTES ICI
        } else {
            ...; if (y < 10) goto L;
            ...
        }
    }
}
```

Avec des `goto` arbitraires : *it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress* (Dijkstra).

Ça donne à réfléchir!

> *The tide of opinions first hit me personally in 1969, when I was teaching an introductory programming course for the first time. I remember feeling frustrated at not seeing how to write programs in the new style; I would run to Bob Floyd's office asking for help, and he usually showed me what to do.*
>
> (D. E. Knuth, 1974)

Suite au brûlot de Dijkstra, de nombreuses études

- pour savoir si la programmation sans `goto` est vraiment la meilleure manière d'exprimer la structure du programme;
- pour savoir éliminer les `goto`, au cas par cas ou systématiquement par transformation de programme.

Le survey de Knuth

D. E. Knuth, *Structured Programming with **go to** Statements*, Computing Surveys 6(4), 1974.

Après un historique de la polémique, Knuth étudie sur des exemples bien choisis

- l'impact de l'élimination des `goto` sur la clarté du code (notamment si des *flags* booléens doivent être ajoutés);
- l'impact de l'élimination des `goto` sur les performances;
- quelles structures de contrôle (au delà de celles d'Algol) aideraient à mieux exprimer la structure du programme.

Exemple : table de hachage

Deux tableaux : `A[N]` contenant les clés, `B[N]` les valeurs.

```
void add(key k, data d)
{
    int i = hash(k);
    while (1) {
        if (A[i] == 0) goto notfound;
        if (A[i] == k) goto found;
        i = i + 1; if (i >= N) i = 0;
    }
  notfound: A[i] = k;
     found: B[i] = d;
}
```

Code très concis! Mais avec deux `goto`...

Exemple : table de hachage

Avec une boucle `while` :

```
void add(key k, data d)
{
    int i = hash(k);
    while (! (A[i] == 0 || A[i] == k)) {
        i = i + 1; if (i >= N) i = 0;
    }
    if (A[i] == 0) A[i] = k;
    B[i] = d;
}
```

Duplication du test `A[i] == 0`.

Exemple : table de hachage

En utilisant la construction `break` pour terminer immédiatement la boucle :

```
void add(key k, data d)
{
    int i = hash(k);
    while (1) {
        if (A[i] == 0) { A[i] = k; B[i] = d; break; }
        if (A[i] == k) {           B[i] = d; break; }
        i = i + 1; if (i >= N) i = 0;
    }
}
```

Petite duplication de code `B[i] = d`, mais sans impact sur les performances.

Exemple : rattrapage d'erreurs et libération de ressources

Un idiome de programmation système en C

```
        int retcode = -1;   // error
        int fd = open(filename, O_RDONLY);
        if (fd == -1) goto err1;
        char * buf = malloc(bufsiz);
        if (buf == NULL) goto err2;
        ...
        if (something goes wrong) goto err3;
        ...
        retcode = 0;    // success
err3:   free(buf);
err2:   close(fd);
err1:   return retcode;
```

Évite toute duplication du code qui libère les ressources.

Exemple : rattrapage d'erreurs et libération de ressources

Une version sans `goto` mais avec duplication de code :

```
int fd = open(filename, O_RDONLY);
if (fd == -1) return -1;
char * buf = malloc(bufsiz);
if (buf == NULL) { close(fd); return -1; }
...
if (something goes wrong) {
    free(buf); close(fd); return -1;
}
...
free(buf); close(fd); return 0;
```

Exemple : rattrapage d'erreurs et libération de ressources

Une version à base de blocs `do ... while(0)` et de `break`.

Serait plus robuste avec un `break` multi-niveau.

```
int retcode = -1;   // error
do { int fd = open(filename, O_RDONLY);
     if (fd == -1) break;
     do { char * buf = malloc(bufsiz);
          if (buf == NULL) break;
          do { ...
               if (something goes wrong) break;
               ...
               retcode = 0;   // success
          } while (0); free(buf);
     } while (0); close(fd);
} while(0); return retcode;
```

Expressivité du contrôle structuré

Le problème de l'élimination des ≪goto≫

Une question technique qui est au centre de la polémique sur le contrôle structuré.

Peut-on toujours transformer un programme non structuré (utilisant uniquement `goto` et `if...goto`) en un programme structuré équivalent ?

Avec ou sans duplication de code ?

Avec ou sans introduction de variables supplémentaires ?

Un résultat bien connu

Théorème

*Tout programme non structuré (ou, de manière équivalente, tout organigramme de programme) est équivalent à un programme structuré contenant une seule boucle **while** et une variable entière supplémentaire.*

Une démonstration simple du résultat bien connu

```
1: s1;
   if (b1) goto 3;
2: s2;
   goto 4;
3: s3;

4: s4;
   if (b4) goto 1;
```

Une démonstration simple du résultat bien connu

```
1: s1;
   if (b1) goto 3; else goto 2;
2: s2;
   goto 4;
3: s3;
   goto 4;
4: s4;
   if (b4) goto 1; else goto 0;
```

Mettre le programme sous forme de blocs de base (*basic blocks*) : des séquences d'affectations toujours terminées par `goto` L ou `if` be `then goto` L_1 `else goto` L_2.

Une démonstration simple du résultat bien connu

```
1: s1;
   if (b1) goto 3; else goto 2;
2: s2;
   goto 4;
3: s3;
   goto 4;
4: s4;
   if (b4) goto 1; else goto 0;
```

Remplacer les étiquettes par des nombres 0, 1, 2, …, avec 0 l'étiquette de la fin du programme.

Une démonstration simple du résultat bien connu

```
1: s1;
   if (b1) pc = 3; else pc = 2;
2: s2;
   pc = 4;
3: s3;
   pc = 4;
4: s4;
   if (b4) pc = 1; else pc = 0;
```

Remplacer chaque `goto` L par une affectation $\texttt{pc} = L$ où $\texttt{pc}$ est une nouvelle variable entière.

Une démonstration simple du résultat bien connu

```
switch (pc) {
   case 1: s1;
           if (b1) pc = 3; else pc = 2; break;
   case 2: s2;
           pc = 4; break;
   case 3: s3;
           pc = 4; break;
   case 4: s4;
           if (b4) pc = 1; else pc = 0; break;
}
```

Faire de chaque bloc de base un cas d'un `switch` sur le `pc`.

Une démonstration simple du résultat bien connu

```
int pc = 1; while (pc != 0) {
   switch (pc) {
      case 1: s1;
              if (b1) pc = 3; else pc = 2; break;
      case 2: s2;
              pc = 4; break;
      case 3: s3;
              pc = 4; break;
      case 4: s4;
              if (b4) pc = 1; else pc = 0; break;
   }
}
```

Introduire une boucle `while` pour répéter le `switch` tant que $\texttt{pc} \neq 0$.

Itération d'une fonction de transition

Si on voit l'organigramme / le programme d'origine comme un automate fini, la démonstration précédente revient à construire sa fonction de transition

$$\text{état courant (dans } \texttt{pc}) \longrightarrow \text{état suivant (dans } \texttt{pc})$$

sous la forme d'une analyse de cas sur la valeur de `pc`

```
switch (pc) { ... }
```

puis à itérer cette fonction jusqu'à atteindre l'état final

```
while (pc != 0) { ... }
```

Variantes :

une variable entière *pc* $\rightarrow$ plusieurs variables booléennes

un `switch` $\rightarrow$ une cascade de `if ... then ... else`.

Le folklore autour de ce résultat

(David Harel, *On Folk Theorems*, CACM 23(7), 1980)

Résultat souvent attribué à Böhm et Jacopini, *Flow diagrams, Turing machines, and languages with only two formation rules*, CACM 1966.

Cependant, cet article montre un résultat différent (plusieurs boucles `while`) par des techniques plus subtiles de réécriture locale de graphes.

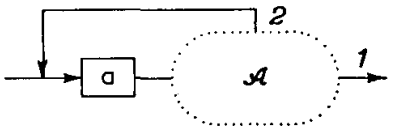


FIG. 13. Structure of a type I diagram

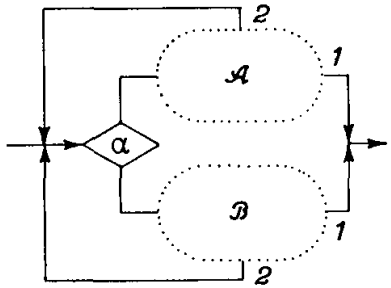


FIG. 14. Structure of a type II diagram

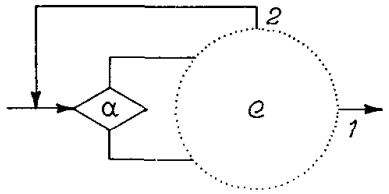


FIG. 15. Structure of a type III diagram

Le folklore autour de ce résultat

La démonstration simple du résultat bien connu apparaît en 1967 dans une lettre de D. C. Cooper à la rédaction de CACM.

Elle est reprise ensuite dans des dizaines d'articles et de livres, souvent attribuée à Böhm et Jacopini, ou sans attribution…

Harel fait remonter le résultat à Kleene (1936)!
(Toute fonction récursive partielle est la ≪minimisation≫ d'une fonction récursive primitive.)

Réductions entre structures de contrôle

(Notion introduite par S. Rao Kosaraju, *Analysis of Structured Programs*, JCSS 9, 1974.)

Soient L_1 et L_2 deux langages qui ont les mêmes commandes de base (affectations, appels de fonctions, …) mais diffèrent sur leurs structures de contrôle.

L_1 est réductible à L_2 si pour tout programme de L_1, il existe un programme de L_2 qui

- a les mêmes commandes de base (pas de duplication de code);
- n'utilise pas de variables supplémentaires.

Exemple de réduction : la boucle `do-while`

La boucle `do-while` (avec test à la fin des itérations) n'est pas réductible à la boucle `while-do` (avec test au début) :
pour traduire

```
do s while be
```

il faut ou bien dupliquer *s*

```
begin s; while be do s end
```

ou bien introduire une variable booléenne

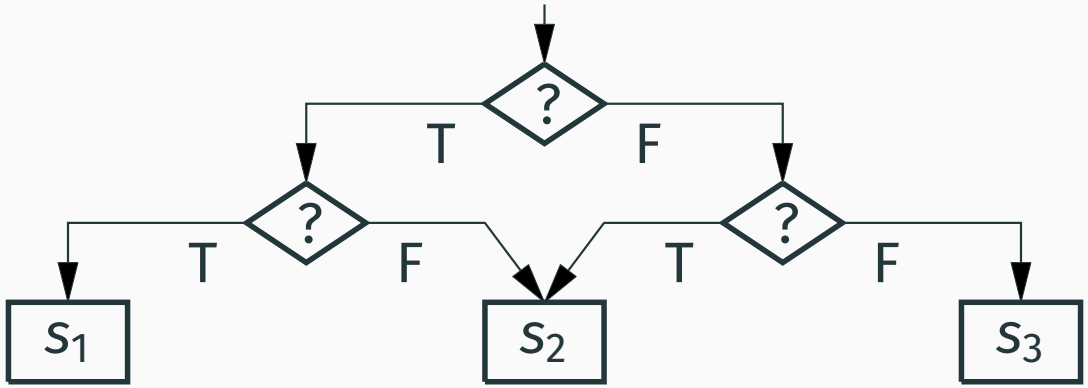
```
loop := true; while loop do begin s; loop := be end
```

En revanche, `do-while` est réductible à `while-do` + `break` :

```
while true do begin s; if not be then break end
```

Exemple de réduction : organigrammes sans cycles

Les organigrammes sans cycles ne sont pas réductibles à `if-then-else`.

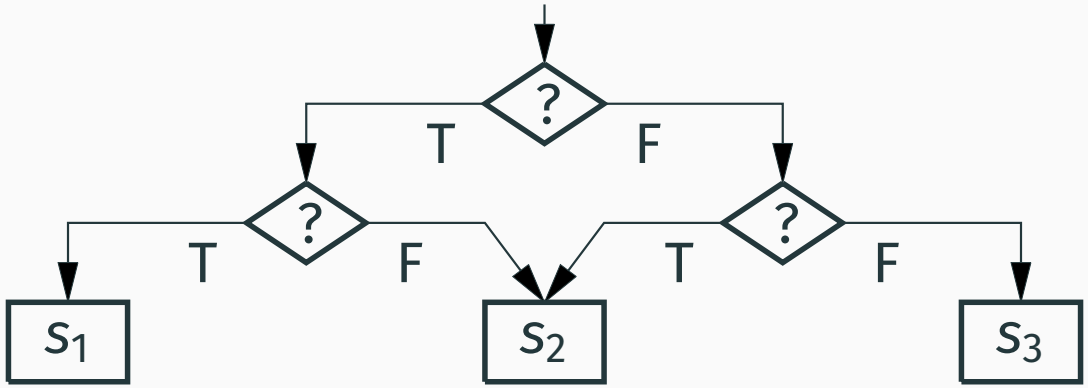


Ou bien on duplique s_2 :

```
if ...
then if ... then s1 else s2
else if ... then s2 else s3
```

Exemple de réduction : organigrammes sans cycles

Les organigrammes sans cycles ne sont pas réductibles à `if-then-else`.

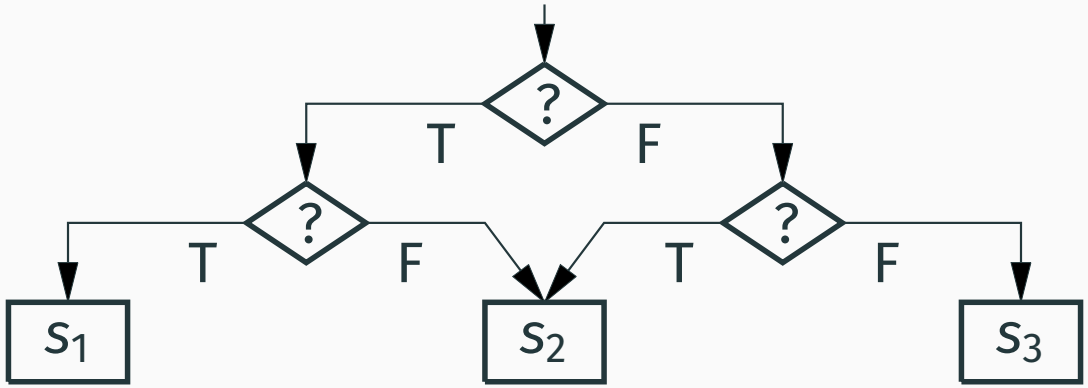


Ou bien on introduit une variable booléenne :

```
do_s2 := false
if ...
then if ... then s1 else do_s2 := true
else if ... then do_s2 := true else s3;
if do_s2 then s2
```

Exemple de réduction : organigrammes sans cycles

Les organigrammes sans cycles ne sont pas réductibles à `if-then-else`.



Mais tout va bien si on a aussi les boucles et une sortie `break` :

```
loop
    if ...
    then if ... then begin s1; break end
    else if not ... then begin s3; break end;
    s2; break
endloop
```

Graphes de flux réductibles et contrôle structuré

(W. W. Peterson, T. Kasami, N. Tokura. *On the capabilities of while, repeat, and exit statements.* CACM, 16(8), 1973.)

Le langage des graphes de flux réductibles

est réductible

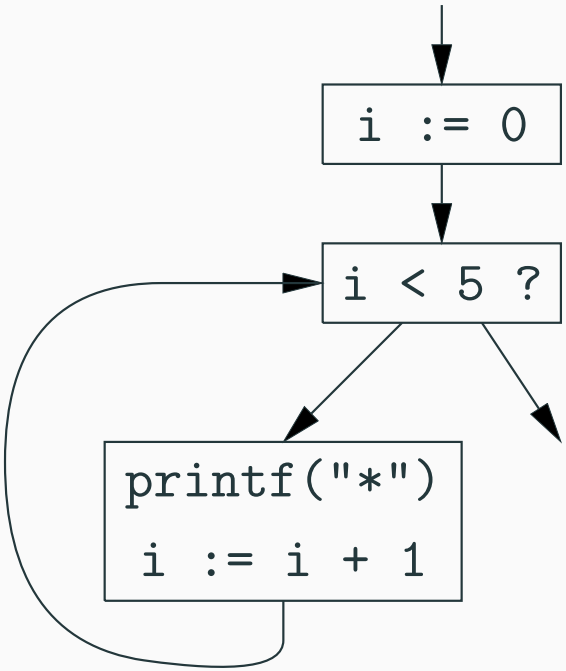
à un langage structuré avec conditionnelle, boucle infinie,
et sorties multi-niveaux (`break` *N*, évt. `continue` *N*).

Graphes de flux de contrôle (CFG, *Control-Flow Graphs*)

Une formalisation des organigrammes de programmes.

Une représentation intermédiaire utilisée par les compilateurs.

($\rightarrow$ séminaire D. Demange)

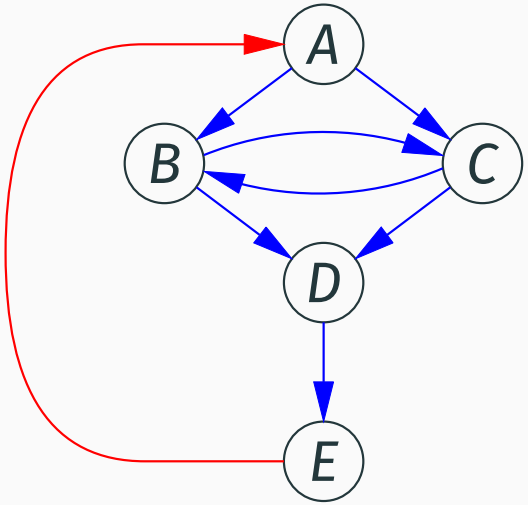


Un CFG = un graphe orienté, avec

- nœuds = blocs de base
- arcs = branchements sortant d'un bloc de base
  (1 arc sortant : `goto`;
  2 arcs : `if-then-else`;
  N arcs : `switch`)

La notion de domination

Un nœud A domine un nœud B si A est forcément exécuté avant B : tout chemin de la racine vers B passe par A.



tout nœud domine lui-même

A domine B, C, D, E

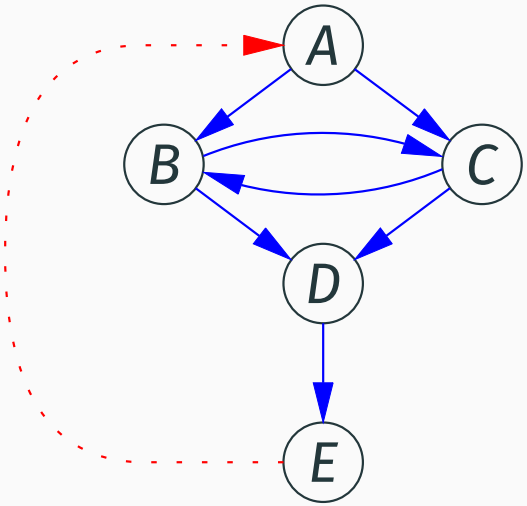
D domine E

Donne une classification des arcs $A \to B$:

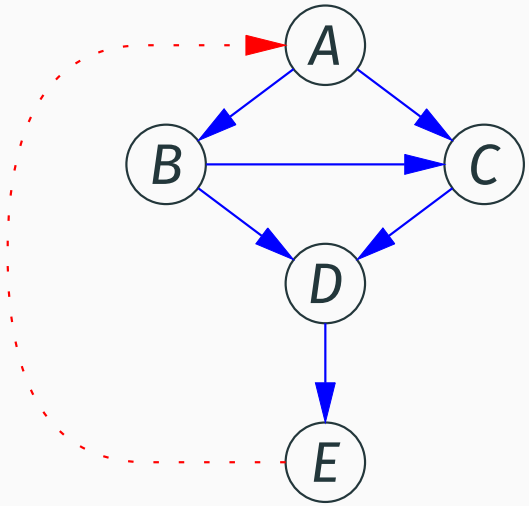
- arc arrière si B domine A ;
- arc avant sinon.

Graphes de flux réductibles

Un graphe de flux est réductible si ses arcs avant forment un graphe acyclique.



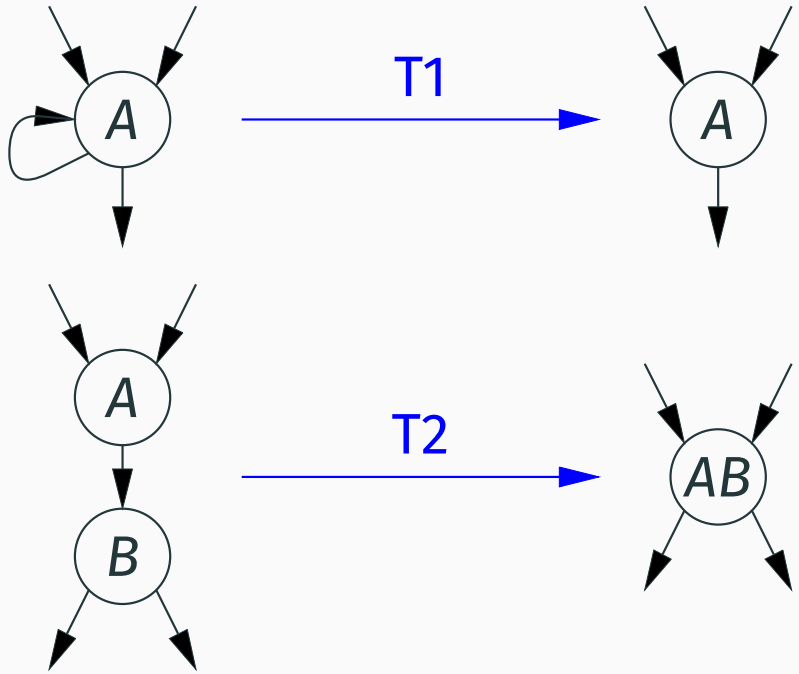
Irréductible



Réductible

Graphes de flux réductibles

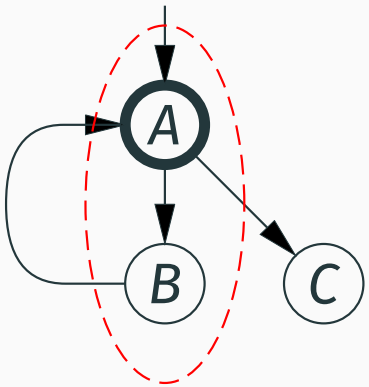
De nombreuses définitions équivalentes, notamment :
un graphe de flux est réductible s'il peut se réduire en un seul nœud en appliquant les transformations T1-T2 de manière répétée.



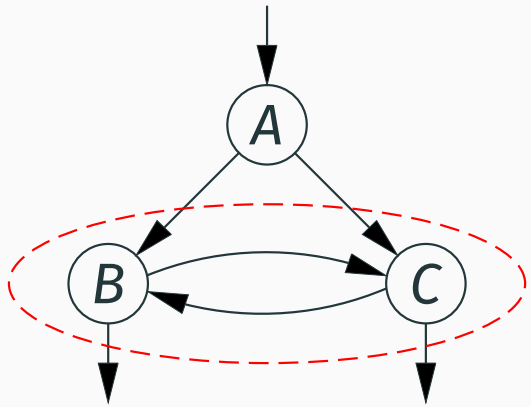
Les boucles dans les graphes de flux réductibles

Dans un graphe de flux réductible, chaque boucle X (= ensemble fortement connexe de nœuds) est une boucle naturelle :

- Elle a un seul point d'entrée $T \in X$, la tête de la boucle.
- T domine tous les nœuds de la boucle X.
- Tout nœud dans X a un chemin vers T qui reste dans X.



Boucle naturelle



Boucle non naturelle

Structures de contrôle et réductibilité

Toutes les structures de contrôle vues jusqu'ici :

- conditionnelles : `if-then-else`, `switch`
- boucles avec conditions au début / à la fin / au milieu
- sorties anticipées : `return`, `break`, `continue`

produisent des graphes de flux de contrôle qui sont réductibles.

Structures de contrôle et réductibilité

Toutes les structures de contrôle vues jusqu'ici :

- conditionnelles : `if-then-else`, `switch`
- boucles avec conditions au début / à la fin / au milieu
- sorties anticipées : `return`, `break`, `continue`

produisent des graphes de flux de contrôle qui sont réductibles.

Réciproquement : tout graphe réductible est le graphe de flux de contrôle d'un programme sans `goto`, utilisant des conditionnelles, des boucles infinies, et des sorties `break`/`continue` multi-niveaux.

Construction du programme structuré

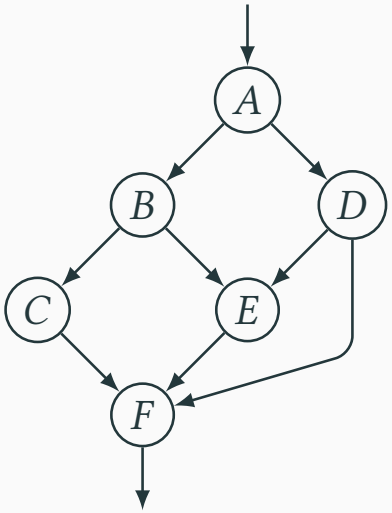
L'algorithme historique de Peterson, Kasami, et Tokura (1973) :

En 3 passes ; esquissé dans une démonstration.

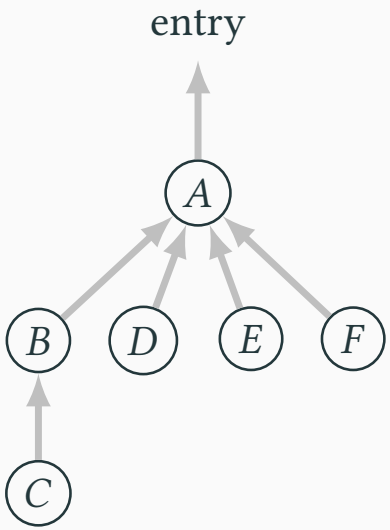
Un algorithme plus récent : N. Ramsey, *Beyond Relooper : Recursive Translation of Unstructured Control Flow to Structured Control Flow*, ICFP 2022.

Une récursion sur l'arbre de domination du graphe.

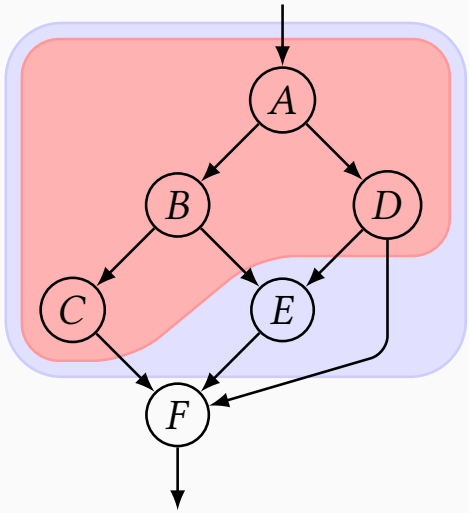
L'algorithme de Ramsey



(a) Unusual control-flow graph (node E reachable two ways)



(b) Dominator tree

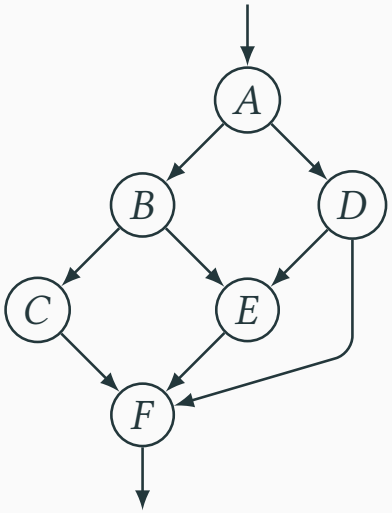


(c) Imposed nesting structure

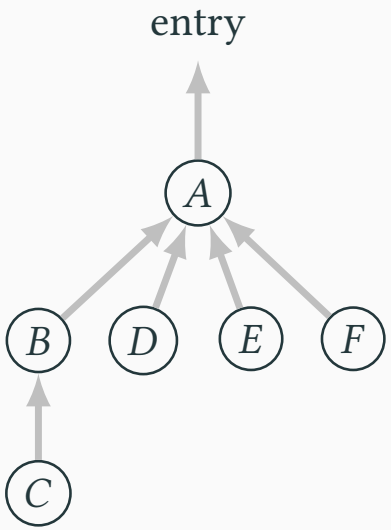
Chaque tête de boucle produit une boucle infinie.

Les fils dans l'arbre de domination qui sont aussi des successeurs produisent des conditionnelles `if-then-else`. (*A*, *B*, *C*, *D*)

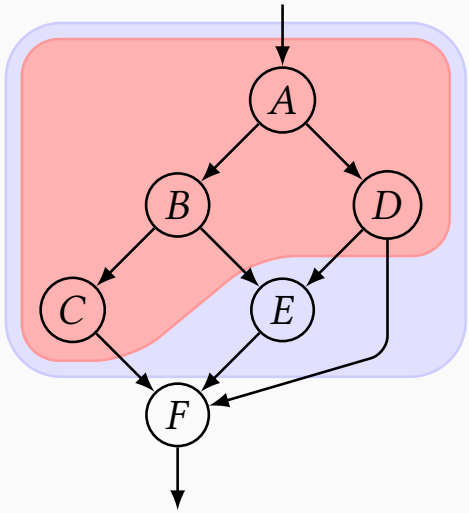
L'algorithme de Ramsey



(a) Unusual control-flow graph (node E reachable two ways)



(b) Dominator tree

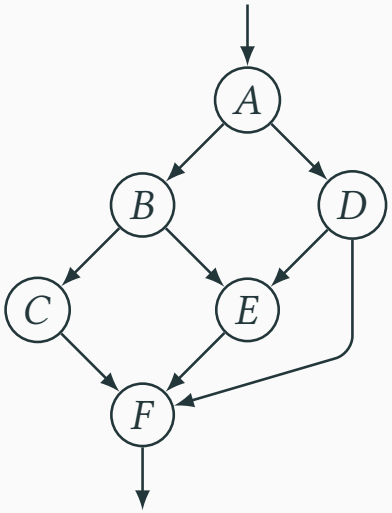


(c) Imposed nesting structure

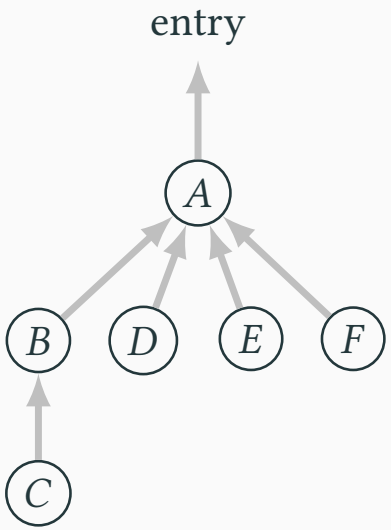
Les autres fils (E, F) sont mis dans des blocs emboîtés suivant le postordre inverse.

Les arcs non triviaux deviennent des `break` N ou des `continue` N.

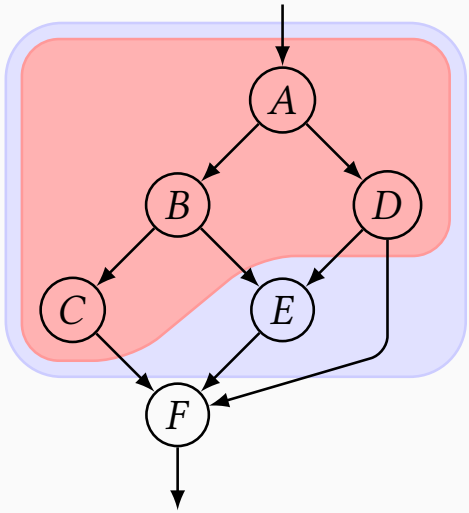
L'algorithme de Ramsey



(a) Unusual control-flow graph (node E reachable two ways)



(b) Dominator tree



(c) Imposed nesting structure

```
block {     // zone bleue
  block {   // zone rouge
    if (A) { if (B) { C; break 2; } else { break 1; }
      else { if (D) { break 1; } else { break 2; }
  } E;
} F;
```

Point d’étape

Du goto à la programmation structurée

De 1945 à 1975, les pratiques et les langages de programmation passent

- d'une vision ≪machine≫ du contrôle (branchements et étiquettes), plus ou moins maîtrisée à l'aide d'organigrammes de programmes
- à une vision structurée du contrôle (conditionnelles, boucles, etc), qui fait du code source la principale représentation du programme.

Au prochain cours, nous verrons la structuration des programmes à plus grande échelle : subroutines, procédures, fonctions, générateurs, coroutines, …

Bibliographie

Bibliographie

Une brève histoire d'Algol :

- *ALGOL 60 at 60 : The greatest computer language you've never used and grandaddy of the programming family tree*, The Register, 15/05/2020. https://www.theregister.com/2020/05/15/algol_60_at_60/

Sur la programmation structurée :

- D. E. Knuth, *Structured Programming with **go to** Statements*, Computing Surveys 6(4), 1974.

Sur la traduction CFG $\rightarrow$ contrôle structuré :

- N. Ramsey, *Beyond Relooper : Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl)*, PACMPL 6, ICFP, 2022.