Structures de contrôle, deuxième cours

Le contrôle non local: des subroutines aux fonctions et aux coroutines

Xavier Leroy

2024-02-01

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

Subroutines, procédures, fonctions

Partage et réutilisation de code

Certains calculs reviennent souvent !

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

Comment faire pour écrire le code une seule fois et «l'appeler» autant de fois qu'on a besoin de l'utiliser ?

Les «subroutines» en langage d'assemblage

Facile à écrire à l'aide d'une instruction de branchement calculé.

```
; Solve quadratic equation AX^2 + BX + C = 0
; Input: A in r1, B in r2, C in r3, return address in r4
; Output: solutions in r1 and r2
quadratic:
        mul r5, r2, r2      ; compute solutions
        ...
        jump r4             ; return to caller
```

Utilisation :

```
        mov r4, L100        ; set return address
        branch quadratic    ; invoke subroutine
L100:   ...                 ; execution resumes here
```

Les subroutines en langage d'assemblage

La plupart des processeurs fournissent une instruction `call` qui saute à une adresse de code donnée tout en mettant l'adresse de l'instruction suivante dans un registre ou sur une pile.

```
call quadratic, r4    ; first invocation
...
call quadratic, r4    ; second invocation
...
```

En cas d'appels imbriqués : utiliser des registres différents ou sauvegarder l'adresse de retour en mémoire, p.ex. dans une pile.

Les subroutines en FORTRAN I

Comme en assembleur, en utilisant le `goto` calculé (`ASSIGN` *lbl* `TO` *var* ... `GO TO` *var*)

```
200:   D = SQRT(B*B - 4*A*C)
       X1 = (-B + D) / (2*A)
       X2 = (-B - D) / (2*A)
       GO TO RETADDR

1000:  A = ... B = ... C = ...
       ASSIGN 1010 TO RETADDR
       GO TO 200
1010:  PRINT X1
```

Les subroutines et les fonctions en Fortran II

Fortran II (1958) introduit des constructions linguistiques pour définir des sous-programmes avec paramètres explicites.

1- Les subroutines :

```
SUBROUTINE QUADRATIC(A, B, C, X1, X2)
    D = SQRT(B*B - 4*A*C)
    X1 = (-B + D) / (2*A)
    X2 = (-B - D) / (2*A)
    RETURN
END
```

Appel : `CALL QUADRATIC(1.0, -2.0, 5.0, X1, X2)`

Arguments passés par référence si variables ou tableaux.

Toutes les variables sont locales à un sous-prog. ou au prog. principal, sauf si déclarées `COMMON`.

Les subroutines et les fonctions en Fortran II

2- Les fonctions simples : une expression avec des paramètres

```
INTPOL(X) = A * X + B * (1 - X)
X2 = INTPOL(0.5)
X3 = INTPOL(0.333333)
```

3- Les fonctions générales : subroutine + valeur de retour.

```
FUNCTION AVRG(ARR, N)
DIMENSION ARR(N)
    SUM = ARR(1)
    DO 10 I=2, N
    SUM = SUM + ARR(I)
10:  AVRG = SUM / FLOATF(N)
    RETURN
END
```

Appel : `X = AVRG(A,20) + AVRG(B,10)`

Les procédures et les fonctions en Algol 60

Proches des sous-programmes de Fortran II :

- une procédure = une commande avec des paramètres ;
- une fonction = une procédure avec une valeur de retour.

Principales différences :

- les arguments sont passés par valeur ou par nom ;
- les procédures peuvent être imbriquées et accéder aux variables des procédures englobantes ;
- la récursion est explicitement autorisée ;
- une procédure peut être passée en argument à une autre.

Une procédure en Algol 60

```
procedure quadratic(a, b, c, x1, x2);
     value a, b, c;
     real a, b, c, x1, x2;
begin
  real d;
  d := sqrt(b * b - 4 * a * c);
  x1 := (-b + d) / (2 * a);
  x2 := (-b - d) / (2 * a)
end;
```

Fonctions emboîtées, fonction comme paramètre

```
real procedure test(a, b);
    value a, b; real a, b;
    begin
        real procedure interpolate(x);
        value x; real x;
        begin
            interpolate := a * x + b * (1 - x)
        end;
        test := integrate(interpolate, 0.0, 10.0)
    end
```

Le passage par nom

La célèbre *copy rule* :

> *Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter … Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved … Finally the procedure body, modified as above, is inserted in place of the procedure statement [the call] and executed …*
>
> *(Report on the Algorithmic Language ALGOL 60)*

Proche de l'appel par nom en λ-calcul, et des macros hygiéniques de Scheme. Mais des surprises aussi !

Grandeur de la *copy rule*

Une fonction de sommation très générale :

```
real procedure Sum(k, l, u, ak)
      value l, u; integer k, l, u; real ak;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

Somme du tableau `A` : `Sum(i, 1, m, A[i])`

Somme des carrés : `Sum(i, 1, n, i*i)`

Somme de la matrice `A` : `Sum(i, 1, m, Sum(j, 1, n, A[i,j]))`

Misère de la *copy rule*

```
procedure swap(a, b)
    integer a, b;
    begin
        integer temp;
        temp := a;
        a := b;
        b := temp;
    end;
```

Cette procédure n'échange pas toujours ses arguments !

Par exemple, `swap(i, A[i])` s'expanse en

`temp := i; i := A[i]; A[i] := temp`.

($\rightarrow$ passage à appel par valeur + appel par référence dans les langages postérieurs à Algol, comme Pascal, Ada, C++, …)

Dimensions du *design* des procédures et des fonctions

Quelques choix possibles :

- Sémantique du passage d'arguments
  (par valeur, par référence, par pointeur, par nom, …)
- Récursivité et réentrance (ou non)
- Fonctions emboîtées (ou non)
- Portée des variables (lexicale, dynamique)
- Durée de vie des variables (le bloc, le programme, …)
- Fonctions comme valeurs
  (de 1re classe, ou juste paramètres d'autres fonctions).

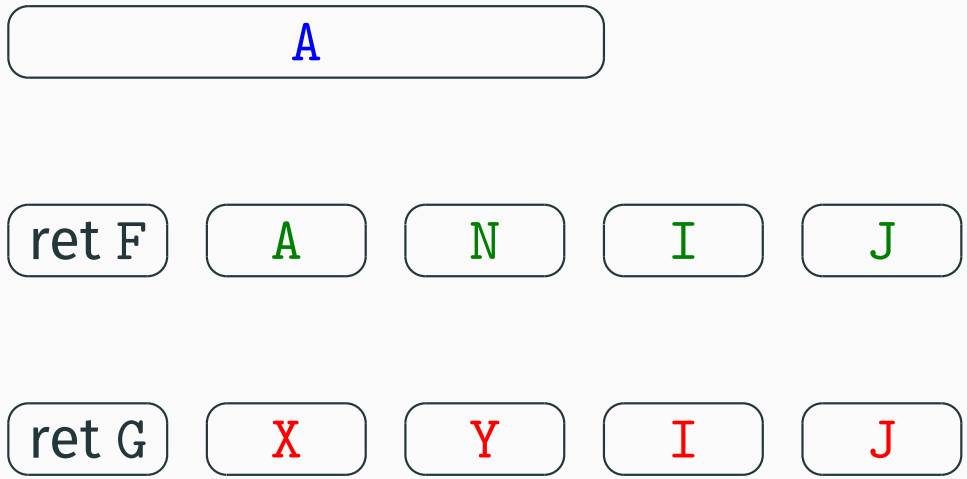
Les choix sont très liés aux techniques d'implémentation des environnements qui conservent les valeurs des variables.

Environnements préalloués statiquement (FORTRAN)

```
DIMENSION A(10)
COMMON A

SUBROUTINE F(A, N)
  ... I ... J...

SUBROUTINE G(X, Y)
  ... I ... J ...
```



Une case mémoire par variable `COMMON`.

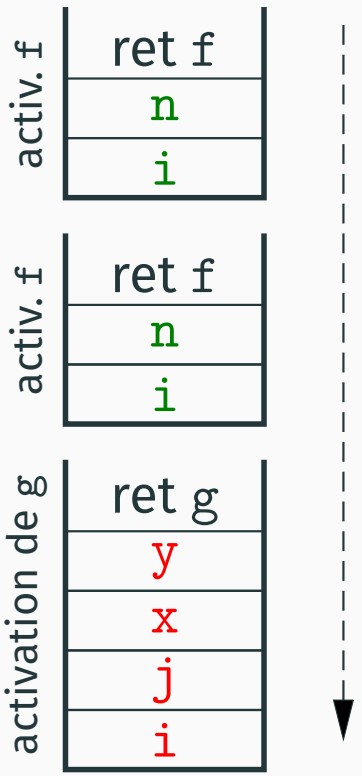
Une case mémoire par variable de chaque subroutine.

Une case mémoire par subroutine pour l'adresse de retour.

Simple et efficace, mais interdit la récursion et la réentrance.

Utilisation d'une pile de blocs d'activation (*stack frames*)

```
procedure g(x, y)
begin
  integer i, j;
  ...
end;
procedure f(n)
begin
  integer i;
  ... f( ) ... g( ) ...
end
```

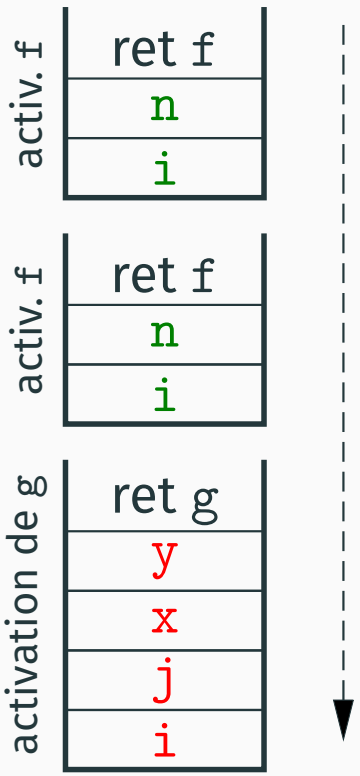


Un bloc d'activation pour une fonction contient ses variables locales (sauf si `static`) et son adresse de retour.

Appel de fonction = on empile ; retour de fonction = on dépile.

Utilisation d'une pile de blocs d'activation (*stack frames*)

```
procedure g(x, y)
begin
  integer i, j;
  ...
end;
procedure f(n)
begin
  integer i;
  ... f( ) ... g( ) ...
end
```



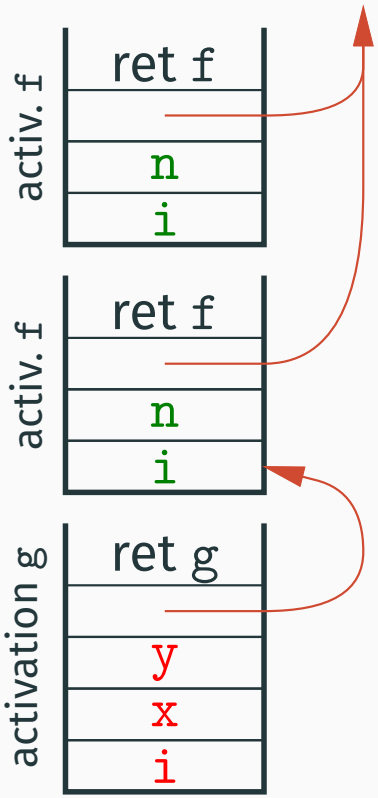
Sans emboîtement (comme en C) :

environnement = bloc d'activation courant de la fonction
+ variables globales ou `static` ;

valeur d'une fonction = pointeur vers son code.

Pile de blocs d'activation et fonctions emboîtées

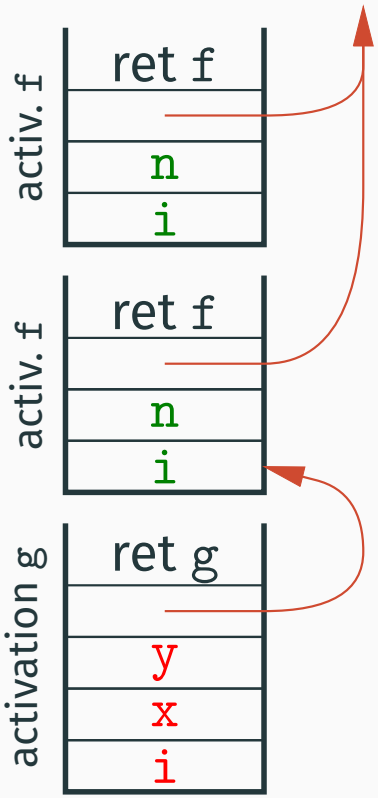
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Chaînage des blocs d'activation les plus récents des fonctions englobantes. La tête de la chaîne est passée comme un argument supplémentaire à la fonction appelée.

Pile de blocs d'activation et fonctions emboîtées

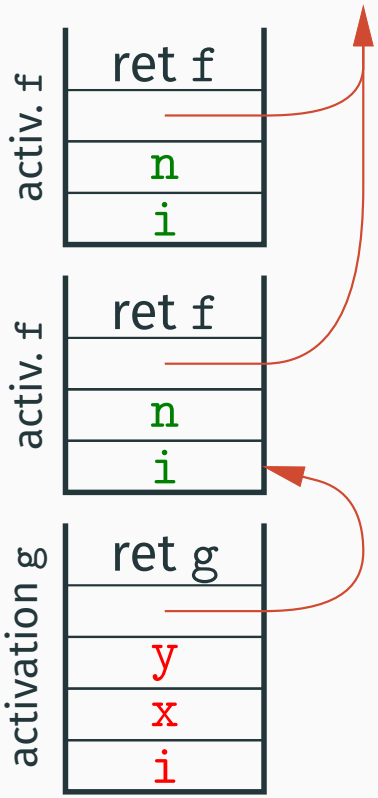
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Environnement = bloc d'activation courant de la fonction
+ blocs d'activation courants des fonctions englobantes
+ variables globales ou `static`

Pile de blocs d'activation et fonctions emboîtées

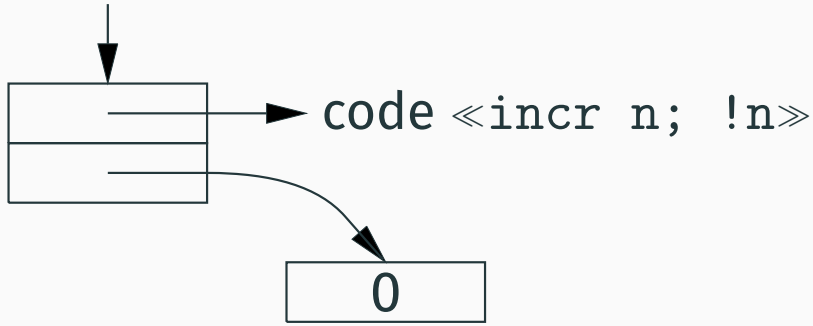
```
procedure f(n)
begin
  integer i;
  procedure g(x, y)
  begin
    integer j;
    ...
  end;
  ... f( ) ... g( ) ...
end
```



Valeur d'une fonction = pointeur de code + tête de la chaîne
($\approx$ fermeture du code par l'environnement).

Allocation en tas des fermetures et des objets

```
let counter () =
  let n = ref 0 in
  fun () -> incr n; !n
```



Permet de renvoyer en valeur résultat
des fermetures (fonctions avec variables libres) ou
des objets (méthodes partageant des variables d'instance).

Découple durée de vie des variables et discipline de pile d'appels.

Les flux de contrôle autour des appels de fonctions

Le flux de contrôle pour un appel de procédure / de fonction

En Fortran II comme dans beaucoup de langages, le flux de contrôle autour d'un appel de procédure est simple :

- au retour de la procédure, l'exécution continue avec la commande qui suit l'appel ;
- les étiquettes sont locales à chaque procédure
  $\rightarrow$ pas de sauts `goto` d'une procédure vers une autre.

Autrement dit, l'appel `CALL` $proc(e_1, \ldots, e_n)$ est une commande de base, au même titre qu'une affectation $x := e$
(sauf que l'appel peut ne pas terminer).

Procédures avec plusieurs points de retour

En Fortran 77, une procédure peut avoir d'autres points de retour que le point après le CALL. Ce sont des étiquettes passées en arguments supplémentaires.

```
SUBROUTINE QUADRATIC(A, B, C, X1, X2, *)
    D = B*B - 4*A*C
    IF (D .LT. 0) RETURN 1
    D = SQRT(D)
    X1 = (-B + D) / (2*A)
    X2 = (-B - D) / (2*A)
    RETURN
END

    ...
    CALL QUADRATIC(1.0, -2.0, 12.5, X1, X2, *99)
    ...
99: WRITE (*,*) 'Error - no real solutions'
    STOP
```

Le ≪goto≫ non local

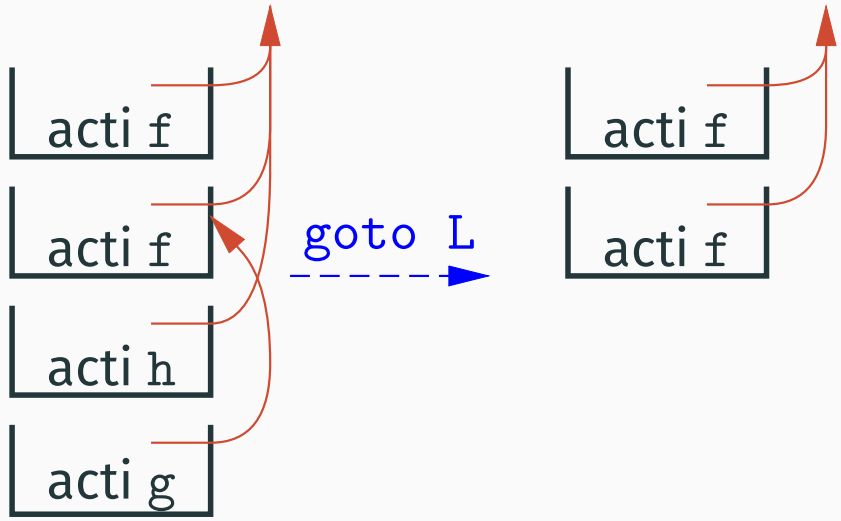
En Algol et en Pascal, un `goto L` peut sortir d'un ou plusieurs blocs englobants, tant que le `goto` est dans la portée de la définition de l'étiquette `L`.

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

Cela fonctionne aussi si `goto L` est dans une procédure définie dans la portée de `L`...

Le «goto» qui sort d'une procédure

```
procedure h(p)
begin
L: ... p() ...
end;
procedure f(n)
begin
  procedure g()
  begin goto L end;
  ... f() ... h(g) ...
L:...
end
```



L'effet d'un `goto L` non local est de terminer l'appel de la procédure et les appels précédents jusqu'à revenir à l'activation qui définit `L`.

Exemple : les erreurs fatales dans le code Pascal de TeX

```
label end_of_TEX, final_end;

procedure jump_out;
begin goto end_of_TEX;
end;

begin
  ...
end_of_TEX: close_files_and_terminate;
final_end: ready_already:=0;
end.
```

Retours multiples et `goto` non locaux

On ne peut pas passer une étiquette `L` en paramètre à une procédure, mais on peut passer une procédure qui fait `goto L`.

```
procedure quadratic(a, b, c: real; var x1, x2: real;
                    esc: procedure ());
variable d: real;
begin
    d := b * b - 4 * a * c;
    if d < 0 then esc();
    d := sqrt(d);
    x1 = (-b + d) / (2*a);
    x2 = (-b - d) / (2*a)
end;
```

Retours multiples et `goto` non locaux

```
procedure solve(a, b, c: real);
variable x1, x2: real;
label error, done;

    procedure goto_error;
    begin goto error end;

begin
    quadratic(a, b, c, x1, x2, goto_error);
    writeln('Solutions:', x1, x2);
    goto done;
  error:
    writeln('No real solutions');
  done:
end;
```

Résultat supplémentaire vs. points de retour multiples

Une approche plus classique : renvoyer un résultat supplémentaire (code de retour, code d'erreur) qui indique comment la fonction s'est terminée (normalement ou pas).

```
int quadratic(double a, double b, double c,
              double * x1, double * x2)
{
    double d = b * b - 4 * a * c;
    if (d < 0) return -1;
    d = sqrt(d);
    *x1 = (-b + d) / (2 * a);
    *x2 = (-b - d) / (2 * a);
    return 0;
}
```

Utilisation des codes d'erreur

```
void solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);
    if (rc < 0) {
      printf("Error - no real solutions\n");
      exit(2);
    }
    printf("Solutions: %f %f\n", x1, x2);
}
```

✔ Traitement de l'erreur «sur place».

Utilisation des codes d'erreur

```
int solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);
    if (rc < 0) {
      return -1;

    }
    printf("Solutions: %f %f\n", x1, x2); return 0;
}
```

✔ Traitement de l'erreur «sur place».

✔ Propagation d'un code d'erreur vers l'appelant.

Utilisation des codes d'erreur

```
void solve(double a, double b, double c)
{
    double x1, x2;
    int rc = quadratic(a, b, c, &x1, &x2);




    printf("Solutions: %f %f\n", x1, x2);
}
```

✔ Traitement de l'erreur «sur place».

✔ Propagation d'un code d'erreur vers l'appelant.

✘ Ignorer l'erreur et continuer comme si de rien n'était.

Les types «option» et «result»

Empêcher d'ignorer les erreurs en utilisant des types sommes et le typage fort.

Idiome courant dans les langages fonctionnels et en Rust.
P.ex. en OCaml :

```
type 'a option = Some of 'a | None

type ('a, 'e) result = Ok of 'a | Error of 'e

let quadratic a b c : (float * float) option =
    let d = b *. b -. 4. *. a *. c in
    if d < 0.0 then None else
        let d = sqrt d in
        Some((-. b +. d) /. (2. *. a), (-. b -. d) /. (2. *. a))
```

Les types «option» et «result»

Le typage statique et l'exhaustivité des filtrages empêchent d'oublier de gérer les erreurs :

```
let solve a b c =
    match quadratic a b c with
    | Some(x1, x2) ->
        printf "Solutions: %f %f\n" x1 x2
    | None ->
        printf "Error - no real solutions\n"
```

Propager l'erreur «vers le haut» s'obtient avec des clauses
`| None -> None` ou `| Err reason -> Err reason`'

Haskell, OCaml, Rust fournissent des syntaxes légères pour cela (notations monadiques, etc).

Exceptions et gestionnaires d'exceptions

Une exception = une structure de données qui décrit une condition exceptionnelle (erreur, absence de valeur résultat, ...).

Deux constructions du langage :

- Levée d'exception (*throw*, *raise*) : `throw` *exn*

  interrompt les calculs en cours et transmet l'exception jusqu'au premier gestionnaire englobant.

- Gestion d'exception (*catch*, *try*) : `try` s_1 `catch(...)` s_2

  intercepte les exceptions levées pendant l'exécution d'une commande s_1 et exécute une autre commande s_2.

Exemple de gestion d'exceptions en Java

```
static double[] quadratic(double a, double b, double c)
throws NoSolution
{
   ... throws (new NoSolution()); ...
}
static void solve(double a, double b, double c)
{
    try {
      double[] sols = quadratic(a, b, c);
      System.out.println(
          "Solutions: " ++ sols[0] ++ ", " ++ sols[1]);
    } catch (NoSolution e) {
      System.out.println("No real solutions");
    } finally {
      System.out.println("I'm done!");
    }
}
```

Sémantique intuitive des exceptions structurées

`throw` dans le corps du `try` :

- ≈ `break` sortant prématurément d'un bloc (*multi-level exit*)
- ≈ `goto` en avant.

`throw` dans une fonction sans `try` :

- recherche dynamique dans la pile d'appel d'un appelant avec un `try` qui sait gérer l'exception ;
- exécution des clauses `finally` des `try` que l'on a sauté.

Par rapport à un `goto` non local : le gestionnaire est déterminé dynamiquement, au lieu d'être fixé par le code qui lève l'exception.

Une brève histoire des exceptions structurées

1972 MacLisp : `THROW`, `CATCH`, puis `UNWIND-PROTECT` ($\approx$ `try...finally`).

1975 J. B. Goodenough. *Exception handling : issues and a proposed notation*, CACM 18(12).

1975 CLU (B. Liskov, MIT).
(Déclaration des exceptions qui peuvent s'échapper d'une fonction ; vérification dynamique.)

1978 LCF ML et ses descendants (SML, Caml, …).
(Sans déclaration.)

1980 Ada
(Sans déclaration.)

1990 C++
(Déclaration optionnelle, obsolète en C11, supprimée en C17.)

1995 Java
(Déclaration obligatoire ; vérification statique)

Controverses autour des exceptions

Pour :

- Pas de code à écrire pour obtenir le comportement le plus courant, c.à.d. la propagation des exceptions vers l'appelant.
- Sépare clairement le code qui détecte l'erreur du code qui sait la gérer.

Contre :

- Engendre des flux de contrôle invisibles dans le code source.
- Trop facile d'oublier de gérer les exceptions.
- Difficile de finaliser les ressources en présence d'exceptions.

(Voir la note de Stroustrup donnée en biblio, et le cours n° 7.)

Inverser ou symétriser le contrôle : itérateurs, générateurs, coroutines

Exemple : afficher une liste chaînée d'entiers

En C :

```c
for (list l = lst; l != NULL; l = l->next)
    printf("%d\n", l->val);
```

En OCaml :

```ocaml
List.iter (fun n -> printf "%d\n" n) lst
```

En Java :

```java
for (Iterator<Int> i = lst.iterator(); i.hasNext(); ) {
   System.out.println(i.next())
}
```

En Python :

```python
for n in lst: print(n)
```

Deux manières d'abstraire le parcours d'une structure de données

Itérateur «interne» :

des fonctions d'ordre supérieur qui appellent le code utilisateur.

```
List.iter: ('a -> unit) -> 'a list -> unit
List.map: ('a -> 'b) -> 'a list -> 'b list
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Itérateur «externe» :

le code utilisateur appelle les méthodes d'un objet «itérateur».

```
interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

On parle d'inversion du contrôle : *don't call us, we'll call you!*

La souplesse des itérateurs externes

Facilite le parcours simultané de plusieurs structures de données.

Exemple : le *same fringe problem* (déterminer si 2 arbres binaires de recherche contiennent les mêmes valeurs).

```
boolean same_fringe(TreeSet<T> s1, TreeSet<T> s2) {
    Iterator<T> i1 = s1.iterator();
    Iterator<T> i2 = s2.iterator();
    while (i1.hasNext() && i2.hasNext()) {
        if (! i1.next().equals(i2.next())) return false;
    }
    return ! i1.hasNext() && ! i2.hasNext();
}
```

Implémenter un itérateur externe

Facile dans un langage à objets : on utilise des variables d'instance de l'objet itérateur pour «se souvenir d'où on en est».

```
class ArrayIterator<T> {
    private T[] arr;
    private int i;
    boolean hasNext() { return i < arr.length; }
    T next() { T res = arr[i]; i++; return res; }
    ArrayIterator(T [] arr) { this.arr = arr; this.i = 0; }
}
```

(En rouge : les parties du code qui proviennent d'un parcours direct par une boucle `for`.)

Implémenter un itérateur externe

Facile aussi si on a la pleine fonctionnalité (fonctions comme valeurs de première classe, avec variables libres).

```
let array_iterator (arr: 'a array) : unit -> 'a option =
    let i = ref 0 in
    fun () ->
        if !i >= Array.length arr
        then None
        else (let res = arr.(!i) in incr i; Some res)
```

Les générateurs (comme dans le langage Python)

Une manière d'écrire des itérateurs en style direct, comme des fonctions qui renvoient un nouveau résultat à chaque appel.

```
def array_elements(a):
    i = 0
    while i < len(a):
        yield a[i]
        i += 1
```

`yield` *v* : renvoie la valeur *v* à l'appelant ; l'exécution de la fonction peut reprendre juste après le `yield`.

`return` *v* : renvoie la valeur *v* à l'appelant ; termine l'exécution de la fonction.

Les générateurs (comme dans le langage Python)

Une manière d'écrire des itérateurs en style direct, comme des fonctions qui renvoient un nouveau résultat à chaque appel.

```
def array_elements(a):
    i = 0
    while i < len(a):
        yield a[i]
        i += 1
```

Exemples d'utilisation :

```
for i in array_elements((1,2,3)): print(i)

g = array_elements((1,2,3))
print(next(g))
print(next(g))
```

Produire une suite infinie à la demande

```
def primes():
    """Generator for prime numbers"""
    p = [2]; yield 2
    m = 3
    while True:
        i = 0
        while i < len(p) and p[i] * p[i] <= m:
            if m % p[i] == 0: break
            i += 1
        else:
            p.append(m); yield m
        m += 2
```

Générateurs pour le non-déterminisme et le rapport d'erreurs

Non-déterminisme $\approx$ plusieurs valeurs de retour possibles.

Erreur $\approx$ aucune valeur de retour possible.

```
def quadratic(a, b, c):
    """Generate the solutions of ax^2 + bx + c = 0"""
    d = b * b - 4 * a * c
    if d < 0: return
    d = math.sqrt(d)
    yield ((-b - d) / (2 * a))
    if d != 0: yield ((-b + d) / (2 * a))
```

Compiler un générateur

Idée : une variable rémanente de type «pointeur de code», dans laquelle on stocke l'adresse (l'étiquette) qui suit le dernier `yield`.

```
def generator():
    n = 0; while True: yield n; yield (-n); n += 1
```

En GNU C (avec étiquettes comme valeurs) :

```
int generator(void) {
        static void * pc = &&start;
        static int n;
        goto *pc;
start: n = 0; while (true) {
            pc = &&yield1; return n; yield1:
            pc = &&yield2; return (-n); yield2:
            n += 1;
        }
}
```

Générateurs avec ou sans pile d'appels

Exemple : énumérer les valeurs aux nœuds d'un arbre binaire, par un parcours infixe.

```
def inorder(t):
    if t:
        inorder(t.left)
        yield t.val
        inorder(t.right)
```

Générateurs avec ou sans pile d'appels

Exemple : énumérer les valeurs aux nœuds d'un arbre binaire, par un parcours infixe.

```
def inorder(t):
    if t:
        inorder(t.left)
        yield t.val
        inorder(t.right)
```

Perdu ! Les appels récursifs de `inorder` créent de nouveaux générateurs, non utilisés. Une seule valeur est renvoyée, celle du sommet de l'arbre. (Voir aussi : `yield from`.)

Il faudrait une autre syntaxe (distinguant «générateur» et «fonction qui fait `yield`») mais aussi une autre implémentation, avec une pile d'appels qui persiste entre les `yield`.

Coroutines asymétriques ou symétriques

Coroutines asymétriques : un autre nom pour les générateurs.

- un rapport appelé (générateur) / appelant (consommateur) ;
- `yield` revient vers l'appelant.

Coroutines symétriques : une forme de *threads* coopératifs.

- des coroutines toutes ≪au même niveau≫ ;
- `yield` passe explicitement la main à une autre coroutine.

(Simula, Modula-2)

Un exemple de coroutines symétriques

```
q = queue.Queue(maxsize = 100)

coroutine produce():
    while True:
        while not q.full(): item = build(); q.put(item)
        yield to consume

coroutine consume():
    while True:
        while not q.empty(): item = q.get(); use(item)
        yield to produce

produce()
```

Le même exemple avec des *threads* coopératifs

```
def produce():
    while True:
        while q.full(): yield
        item = build(); q.put(item)
        yield

def consume():
    while True:
        while q.empty(): yield
        item = q.get(); use(item)
        yield

spawn(produce); spawn(consume)
```

L'entrelacement des calculs est laissé en partie au choix de l'ordonnanceur (*scheduler*).

Une analyse des coroutines par de Moura et Ierusalimschy

(Ana Lúcia de Moura et Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.)

Trois dimensions de *design* :

- coroutines asymétriques ou symétriques ; (sémantique de `yield`)
- avec ou sans pile d'appels ; (position de `yield`)
- comme valeurs de 1re classe ou limitées p.ex. aux boucles `for`.

Une analyse des coroutines par de Moura et Ierusalimschy

(Ana Lúcia de Moura et Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.)

Trois dimensions de *design* :

- coroutines asymétriques ou symétriques ; (sémantique de `yield`)
- avec ou sans pile d'appels ; (position de `yield`)
- comme valeurs de 1re classe ou limitées p.ex. aux boucles `for`.

Principal résultat :

Les coroutines asymétriques de 1re classe avec pile d'appels ont la puissance des continuations délimitées à usage unique et peuvent donc encoder toutes les autres constructions de contrôle vues aujourd'hui. ($\rightarrow$ 4e et 5e cours)

Exemples de codages

Coroutines symétriques $\rightarrow$ coroutines asymétriques :
`yield to` C devient `yield` de la valeur C à un trampoline.

```
c = first generator
while True: c = next(c)
```

Threads coopératifs $\rightarrow$ coroutines asymétriques :
un ordonnanceur appelle les coroutines en séquence.

```
while not q.empty():
    c = q.get()
    try: next(c); q.put(c)
    except StopIteration: pass
```

Point d’étape

Point d’étape

≪Subroutines≫, procédures, fonctions, et méthodes restent encore aujourd’hui le principal mécanisme linguistique pour décomposer les programmes en morceaux compréhensibles et réutilisables.

Le contrôle associé à ce mécanisme (appel – calcul – retour) est simple… sauf lorsqu’il ne l’est pas :

- retours multiples, sauts non locaux, …;
- exceptions et gestionnaires d’exceptions;
- inversion du contrôle : itérateurs, générateurs;
- symétrisation du contrôle : coroutines symétriques, *threads*.

Bibliographie

Bibliographie

Une analyse et une formalisation des coroutines :

- Ana Lúcia de Moura et Roberto Ierusalimschy, *Revisiting Coroutines*, TOPLAS 31(2), 2009.

Une analyse des exceptions vs. codes de retour :

- Bjarne Stroustrup, *C++ exceptions and alternatives*, note P1947, 2019.