11 Contrôle de flot

- Instructions de contrôle de flot
- Sous-programmes
- Passage de paramètres
- Exceptions

Introduction

- Afin de rendre les langages utiles et flexibles, il faut, en plus des affectations de variables, des *intructions de contrôle*:
  - Un moyen de choisir entre des alternatives de code différentes,
  - Un moyen de répéter des collections d'instructions.
- De plus, afin de faciliter la conception des instructions de contrôle, il faut un mécanisme qui permette de former des collections d'instructions.
- Les langages impératifs se distinguent par l'importance des moyens de contrôle de flot

Mécanismes de structuration de base

- Il y a trois mécanismes de structuration d'instructions fondamentaux:
  - Les séquences ou instructions composées: begin-end
  - Les instructions de sélection/conditionnelles: if-then-else
  - Les répétitions/boucles: while-do
- En fait, il a été prouvé que seulement 2 mécanismes de structuration sont nécessaires: les *gotos* (branchements) et les sélections à une branche, dérivés directement des instructions disponibles dans le matériel. Mais leur utilisation mène à un code "spaghetti" où le flot de contrôle est difficile à suivre.
  - Débat 'goto considered harmful' des années 60-70
  - Une des raisons de la programmation structurée

L'instruction GOTO

- L'instruction GOTO (*aller à*) est un transfert de contrôle sans restriction à un point différent du programme.
- C'est le seul mécanisme de transfert présent dans les langages de bas niveau.
- Contrairement aux structures de contrôle plus modernes, ce mécanisme est dangereux, mauvais pour la lisibilité et doit être évité:
  - Le GOTO peut laisser derrière lui un environnement dans un état incohérent, si on ignore les règles d'implémentation du langage.
- Certains langages ont tout de même conservé le GOTO, mais restreignent l'usage et le rendent peu commode à utiliser (Java interdit son utilisation).

Equivalences entre mécanismes de structuration

- **if** C **then** S ⇔ **if** C **then** S **else** null
- **repeat** S **until** C ⇔ S; **while** ~C **do** S
- **for** i:= début **to** fin **do** S
  ⇔ i:= début; **while** i<= fin **do begin** S; i := succ(i) **end**
- **case** i **of** C1: S1; …. ⇔ **if** i=C1 **then** S1 **else** …

Séquences

- L'idée d'une instruction composée est que plusieurs instructions peuvent être abstraites en une seule.
- Dans certains langages, en plus des instructions composées, il peut y avoir des déclarations de données (types, constantes, variables). Dans ce cas on parle de *bloc*.
  - Algol, Pascal, Ada: **`begin`** … **`end`**
  - C, C++, Java: `{` … `}`
  - Algol 68: **`do`** ... **`od`**
  - Prolog: instruction composée implicite: `a :- b, c.`

Sélection

- Dans la grammaire de Pascal, il y a une ambigüité avec les *if-then-else* emboîtés:

  if C1 **then if** C2 **then** S1 **else** S2

  A quel “if” se rattache le “else” ?
  Réponse: au “then” le plus proche.

- En Modula-2 et en Ada, il n’y a pas d’ambigüité car les if-then-else doivent etres terminés par un **end** (Modula-2) ou **end if** (Ada).

- En Ada, il existe aussi le **elsif**, pour les *if-then-else* emboîtés.

Formes spéciales de sélection

- Pascal et Ada fournissent des instructions de sélection de cas équivalents à des *if-then-else* imbriqués
  - syntaxiquement plus léger
  - et surtout compilables en un code plus efficace (génération de tables de saut); l'expression conditionnelle est donc souvent plus restrictive que dans un *if-then-else*

Ada

```
case chiffre is
   when 0 => zero;
   when 1,3,5,7,9 => impair(chiffre);
   when 2,4,6,8 => pair(chiffre);
end case
```

Table de saut
(en assembleur Intel):

```
DW      offset zero
DW      offset impair
DW      offset pair
DW      offset impair
DW      offset pair
...
```

Formes spéciales de sélection (suite)

- Le *goto calculé* en Fortran consiste à brancher sur le label correspondant à la valeur de l'*expression*:

```
20 …
          go to (10, 20, 30), expression
10 …
30 …
```

- En C/C++/Java (mais pas C#), l'instruction **switch** est en fait un goto calculé:

```
switch (expression){
                case const1: S1;
                …
                case constn: Sn;
                default: Sn+1
}
```

Danger: une fois l'instruction S_i exécutée, le contrôle continue sur S_{i+1} à moins qu'une instruction "*break*" ne soit utilisée pour l'empêcher.

Séléctions et *fall-through* (effet de chute)

```
void CompterARebours (int depuis) {
    switch (depuis) {
          case 9:     System.out.print("neuf ");
          case 8:     System.out.print("huit ");
          …
          case 1:     System.out.print("un ");
          case 0:     System.out.print("zéro ");
          default:
    System.out.println("terminé !");
    }
}
```

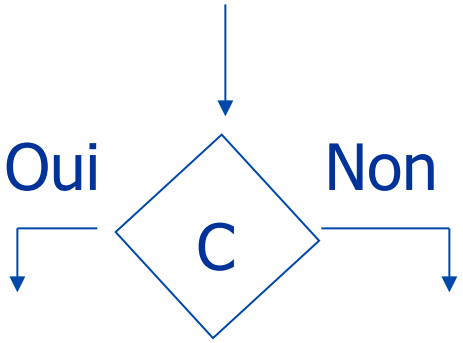
- `CompterARebours(5)` affiche:
  `cinq quatre trois deux un zéro terminé !`
- L'oubli du *break* est une source d'erreurs fréquente !

Sélection en Prolog

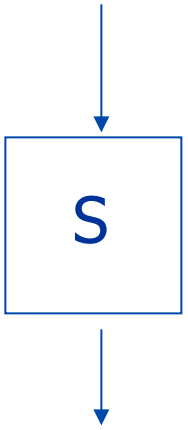
- En Prolog, plutôt que d'être conduite par ce qui est vrai ou faux, la sélection est conduite par le succès ou l'insuccès.
- La sélection est implicite dans le backtracking.

```
manger_ou_boire(Aliment) :-
  solide(Aliment), manger(Aliment).
manger_ou_boire(Aliment) :-
   liquide(Aliment), boire(Aliment).
```

Flowcharts pour représenter la sélection



Ceci représente un test: est-ce que C est vrai? Si oui, prendre le chemin de gauche. Sinon, prendre le chemin de droite.



Ceci représente une instruction S ou bien une séquence d'instructions S.

Itérations à test logique

- Il y a deux variations sur les itérations:
  les itérations ***pre-***test et les itérations ***post-***test.
  - En Pascal:
    - Pre-test: **while** C **do** S;
    - Post-test: **repeat** S **until** C;
  - En C:
    - Pre-test: **while** (expr) S;
    - Post-test: **do** S **while** (expr);

Itérations à compteur

- Les itérations à compteur (boucles for) sont apparues plus tôt, historiquement, que les itérations à test logique. Mais elles sont moins générales.
- Elles existent en Fortran, Algol 60, Pascal, Ada, C
  - La syntaxe et la sémantique sont différentes dans chacun de ces langages !
- En Prolog ou Haskell, par contre, il n'y a pas d'instructions d'itération, mais l'itération existe sous forme de définitions *récursives* (ce qui peut aussi être fait dans les langages impératifs qui permettent la récursion)
  - Les compilateurs de ce genre de langages savent remplacer les *appels récursifs terminaux* par des boucles simples

Evaluation court-circuitée

- Les langages C et Ada fournissent des opérateurs booléens permettant d'optimiser l'évaluation des conditions

```
int table[taille];
…
if (i >= 0 && i<taille && table[i] > clef) …
```

- En Pascal ce code produirait une erreur si l'indice *i* est hors des bornes du tableau, car l'ordre d'évaluation des sous-conditions n'est pas spécifiée par le langage

Commandes gardées

- Il existe une forme de sélection et de boucle complètement différente: les commandes gardées [Dijkstra, 1975].
- L'idée d'une commande gardée est la suivante.
  Etant donnés plusieurs choix:
  - Si aucun de ces choix n'est possible, il y aura une erreur à l'exécution.
  - Si un seul de ces choix est possible, l'instruction correspondant à ce choix est exécutée.
  - Si plus d'un choix est possible, l'un de ces choix est choisi de facon non-déterministe (au hasard) et l'instruction correspondant à ce choix est exécutée.

Commandes gardées (suite)

- La motivation pour ces commandes était de créer une méthodologie qui permettrait de vérifier la validité d'un programme avant, plutôt qu'après, son développement.
- Cette technologie n'a pas été adoptée à part dans le cas de calculs parallèles. Ada, Occam permettent de tels calculs et incluent les commandes gardées.
- Les commandes gardées sont pratiques lorsqu'un choix doit être fait entre plusieurs possibilités de même priorité: on laisse au langage la tâche de choisir au hasard parmi les possibilités offertes.

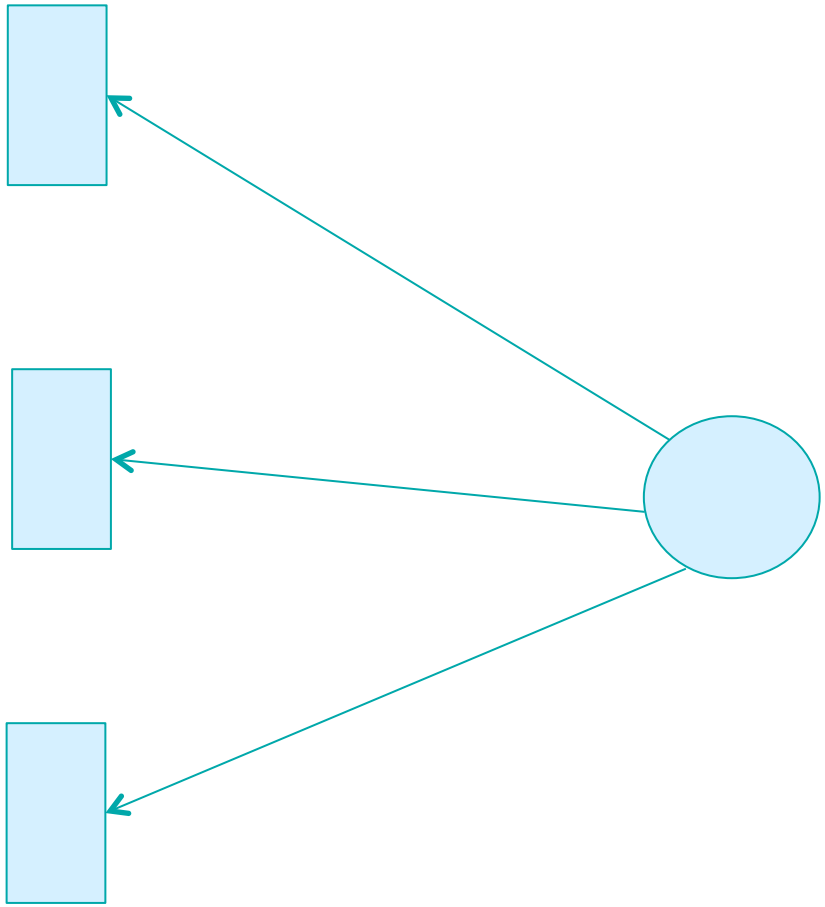
Commande gardée en Ada

- Une tâche serveur gère un tampon partagé. Des tâches concurrentes peuvent y lire ou y écrire:

```
task body serveur is
    tampon: string := "";
begin
    loop
        select
            when nonPlein(tampon) =>
                accept remplir(c: character) …
        or
            when nonVide(tampon) =>
                accept vider(s: out string) …
        end select;
    end loop;
end serveur;
```

Choix non-déterministe

- Cf. sémantique Réseaux de Petri



Les sous-programmes

- Un *sous-programme* est un bloc nommé encapsulant (cachant) un algorithme; il est évalué en suspendant l'appelant, en exécutant les instructions du bloc, puis en retournant le contrôle au programme appelant
- Au départ, les *sous-programmes* (procédures et fonctions) ont été conçus comme mécanisme de réutilisation de code.
- Ils peuvent aussi être considérés comme un mécanisme fondamental d'abstraction: on parle d'*abstraction de processus*.
- Au niveau sémantique, un sous-programme est une opération complexe qui peut être initiée (appelée, invoquée) comme une opération élémentaire.

Eléments d'un sous-programme

- *L'entête* (ou *profil*) d'un sous-programme consiste en
  - un nom
  - des paramètres (nombre ou *arité*, types, modes de passage)
  - si le sous-programme est une fonction: le type de valeur retourné (le *paramètre-résultat*)
- Le corps d'un sous-programme est représenté par:
  - une séquence d'instructions
- Terminologie: un sous-programme est
  - *défini* par un profil et un corps de sous-programme
  - *déclaré* par un profil uniquement (pour signaler l'existence d'une définition externe ou à venir)

Passage de paramètres

- Le mode de passage d'un *paramètre effectif* (celui de l'appelant) à un *paramètre formel* (la vue offerte au sous-programme) détermine:
  - Quelle partie de l'argument est donnée au sous-programme:
    - Seulement sa **valeur**
    - Seulement son **addresse**
    - Et sa **valeur** et son **adresse**.
  - Quelles restrictions sont appliquées sur l'usage de l'argument:
    - Permission de **lire**
    - Permission **d'écrire**
    - Permission de **lire** et **d'écrire**.

Passage par valeur

- Seule la valeur du paramètre est donnée au sous-programme. Pour sauvegarder cette valeur, on utilise une variable locale qui devient son addresse.
- Ceci est implémenté en calculant et copiant la valeur du paramètre actuel dans l'espace de mémoire du sous-programme.
- Ceci peut coûter cher si le paramètre est un objet volumineux tel qu'un tableau.
- Le passage par valeur est utilisé en Ada par le mode "in" (qui en plus considère le paramètre formel comme une constante), par le paramètre-valeur de Pascal, et par tous les paramètres en C.

Passage par référence (ou par adresse)

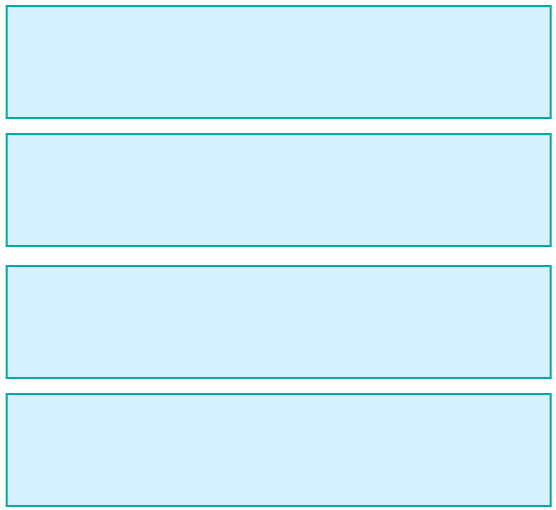
- Les adresses (et donc indirectement les valeurs) des paramètres sont fournies au sous-programme pour permettre de modifier leurs valeurs.
- Ceci est implémenté par référence indirecte: le paramètre effectif est l'addresse d'une variable (mais pas d'une expression, ni d'une valeur !) dans le programme d'appel. Il n'est pas necessaire de copier explicitement la valeur.
- L'un des grands problèmes causés par le passage par référence est la *déclaration d'alias*. Par exemple, on peut passer la même variable comme deux paramètres effectifs différents.
- Le passage par référence est signalé en Pascal par le mode "var"; il n'est pas disponible en Ada.

Passage par valeur-résultat

- Mécanisme de passage par valeur-résultat: lorsque le sous-programme est appelé, la valeur du paramètre est copiée dans une variable locale. La valeur finale de cette variable locale est ensuite recopiée (exportée) dans le paramètre effectif à la sortie.
- Ce mode de passage est plus lourd que le passage par référence, mais permet d'éviter les problèmes d'alias, particulièrement ardus dans les langages concurrents.
- Le passage par valeur-résultat est obtenu en Ada par le mode "in-out"; il n'est pas disponible en Pascal.

Résumé des modes de passage

- `in` →
-
- `in-out` →
- `référence` →



→ `out`

→ `in-out`

→ `référence`

Ownership (possession) Rust

- Les variables possèdent la ressource à l'aquelle elles sont liées.
- ```
  fn aFunction()
            {  let x=10; }
  ```
- X est lié à la ressource 10 (type primitif)
  - La valeur est alouée dans le stack
  - Si on sort du contexte la ressource est libérée
  ```

Ownership (possession) Rust

- ```
  fn aFunction() {
  let n = vec![1,2,3];}
  ```
- n est lié au vec (type non-primitif)
  - La valeur du vecteur est alouée sur le tas
  - Valeur et reference doivent être synchronisée
  ```

Déplacer les valeurs

```
fn aFunction() {
  let foo = vec![10];
 let bar = foo;
  // cannot use foo from this point!}
```

- Le vecteur était lié initialement à `foo`
- Ensuite bar à pris possession de la valeur de `foo`, valeur à été déplacée vers `bar`.
- Il est impossible ensuite d'utiliser `foo`.

Déplacer les valeurs (paramètres)

- `fn aFunction(agesParam: Vec <i32>) {}`

```
let ages= vec![10];
let newAges = aFunction(ages);
```

- Le vecteur était lié initialement à `ages`
- Ensuite agesParam à pris possession de la valeur de `ages`, valeur à été déplacée vers agesParam.

Copier les valeurs (types copy)

```
fn aFunction()
          {  let x=10;
             let y = x }
```

- Le type entier est un type ‘copie’
- Une copie de la valeur de x est faite pour y.

- L’utilisateur peut créer ses types ‘copie’

Emprunt de valeurs

- Pas utilisable de retourner différents types:
  - `let (v1, v2) = aFunction(v1);`
- ```
  fn aFunction(v1: &Vec<i32>, v2: &Vec<i32>)
  -> i32 {// do stuff with v1 and v2
   42 // return i32}
  ```

  ```
  let v1 = vec![1, 2, 3];
  let v2 = vec![1, 2, 3];
  let answer = aFunction(&v1, &v2);
  // we can use v1 and v2 here!
  ```
- `& signifie emprunter`
  ```

Règle de l'Emprunt de valeurs

- La durée de l'emprunt est limitée à son 'scope'
- Un type d'emprunt est autorisé
  - Lecteur multiples : &T
  - Unique rédacteur : &mut T
- Empêche les 'data races'
- Exemple de 'Scope'

```
let mut x = 5;    // 1st scope
{ let y = &mut x;// 2nd scope and &mut borrow start
*y += 1;          //
}                 //   2nd scope and &mut borrow end
println!("{}", x);// use x,  y ownership ended
```

Durée de vie

- Pour éviter les ‘dangling reference’

- Durée de vie implicite ou explicite !

- ```
  // implicit lifetime
  fn foo(x: &i32) { ... }

  // explicit lifetime - a reference to an
  i32 with a lifetime of a

  fn bar<'a>(x: &'a i32) { ... }
  ```
  ```

Passage par nom

- Le paramètre effectif remplace le paramètre formel: il faut imaginer une modification du sous-programme telle que toute occurrence du paramètre formel serait remplacée textuellement par le paramètre effectif.
- Si le paramètre effectif est une expression ou un sous-programme, il sera évalué (ou appelé) à chaque utilisation au lieu de l'être lors du passage de paramètres
- Ce mode de passage, complexe à implémenter, est abandonné dans les langages impératifs modernes, mais courant dans les langages fonctionnels
- En Algol60:

```
real procedure somme(expr, i, inf, sup); …
y := somme (3*x*x - 5*x + 2, x, 1, 10);
```

Les exceptions

- Les *exceptions* permettent de signaler des erreurs survenant pendant l'exécution, et allègent la syntaxe du programme en évitant les *if-then-else* imbriqués.
  - Le langage C, typiquement, impose de tester le statut (résultat) de tout appel de fonction pour déterminer si l'exécution progresse comme prévu, ce qui alourdit le texte du programme
  - Un mécanisme d'exceptions permet de brancher de manière transparente le flot de contrôle directement sur le code de traîte-exceptions
- C++, Ada et Java sont très riches de ce point de vue

Les exceptions (exemple Java)

```
try {
   …
   x = 0;
   d = 1/x;
   …
} catch (Exception e) {  // traitement de l'exception
   System.out.println("Exception: " + e);
}
```

```
Exception: java.lang.ArithmeticException: / by zero
```

- Le bloc "`try`" contient le code où il y a un risque de voir une exception levée
- Explicitement 'throw'
- Si une exception surgit effectivement, le code du bloc "`catch`" correspondant est exécuté, sinon pas; sans le bloc "`catch`", l'exception est propagée à l'appelant.

Les routines génériques

- Beaucoup d'algorithmes pourraient facilement s'appliquer à différents types de données. D'un autre côté, le compilateur veut connaître le type des données pour lequel un algorithme est compilé.
- La *généricité* (ou *polymorphisme paramétrique*) est un moyen de satisfaire ces deux contraintes
- Avant d'être utilisées, les routines génériques doivent être *instanciées* en fonction des types effectivement requis par le programmeur
  - la routine générique est un "patron" servant à générer des routines spécialisées => dans le code généré, il existe un code pour chaque instance
- La généricité est supportée par Ada, Java et C++

Les routines génériques (exemple Ada)

```
generic
        type T is private;
        type tabT is array (integer range <>) of T;
        with function "<"(x,y : T) return boolean;
procedure trier (tab : in out tabT);

procedure trier (tab: in out tabT) is
begin
        -- (ici, le code pour le tri)
end trier;

-- Création d'une instance pour le tri des vins millésimés:
procedure trier_vins is new trier (
        T       => vin_millesime,
        tabT    => ma_table_de_vins_millesimes,
        "<"     => critere_millesime);
```