# Langages et paradigmes

Olivier Lecarme

Master d'Informatique, première année

2007–2008

# Quatrième partie IV

# Énoncés et expressions

# Classification des structures de contrôle

- Séquence : les énoncés sont exécutés l'un après l'autre dans l'ordre indiqué.
- Sélection : un énoncé est choisi parmi plusieurs, d'après une condition évaluée à l'exécution.
- Itération : un groupe d'énoncés est exécuté de manière répétitive, soit un certain nombre de fois (répétition), soit jusqu'à ce qu'une certaine condition soit vraie (itération proprement dite).

# structures de contrôle

- Appel de procédure : un groupe d'énoncés est encapsulé de manière à être traité comme un tout, avec paramétrisation.
- Récursivité : une expression est partiellement définie en termes d'elle-même.
- Parallélisme : plusieurs groupes d'énoncés sont exécutés en même temps, soit grâce à plusieurs processeurs, soit par partage du temps sur l'unique processeur.

# structures de contrôle

- Non-déterminisme : l'ordre d'exécution de plusieurs groupes d'énoncés peut être choisi de manière quelconque et imprévisible.

Nous n'étudierons dans ce chapitre que les trois premières catégories.

# Évaluation des expressions

- Généralités
  - Une expression est une suite d'opérateurs et d'opérandes :
    - appel de fonction : `f(a, b, c)`
    - opérateurs infixes : `a + b * c`
    - mélange des deux : `"+"(a, b)` (Algol 68, Ada) ; `a.operator+(b)` (C++)
    - notation lispienne : `(+ a b c)`

# généralités

- Parfois extension à des opérateurs conditionnels :
  - Algol 60 : `a := if b < 0 then 1 else 0`
  - Algol 68 : `a := (b < 0 | 1 | 0)`
  - C : `a = b < 0? 1 : 0`
- très rares opérateurs postfixes :
  - Pascal : `fichier^`
  - C : `c++`

# Priorités et associativités

- L'associativité détermine l'ordre d'exécution à priorité égale :
  - de gauche à droite en général : `a - b + c`
  - pas toujours logique : `a / b / c`
  - parfois de droite à gauche pour l'exponentiation : `a ** b ** c` en Fortran
  - de droite à gauche pour les fonctions : `f(g(h(a)))`
  - de droite à gauche également en APL : `+/+/[1]T` ou `2×S+T÷2`

# priorités et associativités

- La priorité évite de placer trop de parenthèses dans les expressions compliquées :
  - `a + b * c = a + (b * c)`
  - mais `a / b * c = (a / b) * c`
  - s'il y a trop de niveaux de priorité (voir C ou Icon), le programmeur les oublie
  - s'il n'y en pas assez, des expressions naturelles sont erronées : **if** `a < b` **and** `b < c` **then** `...` en Pascal

# priorités et associativités

- Les langages sont extrêmement différents dans ce domaine :
  - `IF a < b < c THEN` en PL/I est syntaxiquement correct mais de signification stupide
  - **if** `a < b < c` **then** en Pascal est normalement sémantiquement incorrect (erreur de type)
  - pour certains langages les opérateurs de comparaison ne sont pas associatifs
  - `if a < b < c then` en Icon a la signification attendue mais ne calcule pas un booléen

# Affectation

- C'est l'énoncé caractéristique des langages impératifs, le moyen de changer l'état de la machine abstraite. Beaucoup de langages impératifs séparent plus ou moins strictement les énoncés des expressions :
  - Fortran, PL/I, Pascal : séparation totale
  - Algol 60, C : expressions conditionnelles
  - Algol 68, APL, Icon : langages d'expressions

# affectation

- La signification même de l'énoncé d'affectation n'est pas universelle :
  - en Pascal ou C, les noms sont interprétés différemment en partie gauche et en partie droite :

    ```
    a := b;
    b := c + d;
    ```

    `b` désigne une valeur en partie droite, un emplacement en partie gauche.
  - en partie gauche on peut mettre une valeur-G, qui dénote un emplacement à modifier
  - en partie droite on peut mettre une valeur-D, qui dénote une valeur à affecter

# affectation

- toute expression ne peut pas être une valeur-G :

```
a + b := c;
f(a, b) := c;
g^.a[f(b) + c]^.d := e;
```

- dans certains langages, les noms sont des références à des valeurs, plutôt que les noms d'emplacements :

```
b := 2;
c := b; // c désigne donc la même valeur
// que b, et non pas une valeur
// égale à celle de b
```

# affectation

- Dans ce cas, toute variable est une valeur-G ; quand elle apparaît là où on a besoin d'une valeur-D, on doit la dérepérer (concept introduit par Algol 68).
- Java utilise ce mécanisme pour les objets, mais l'autre mécanisme pour les types prédéfinis.

# affectation

- Orthogonalité de l'affectation :
  - dans un langage d'expression (Algol 68, Icon, APL), tout a une valeur, donc l'affectation aussi
  - en Pascal, PL/I ou Ada, l'affectation est un énoncé pur
  - en C, l'affectation a une valeur, qui est celle de sa partie droite ; l'inconvénient est que l'affectation est notée comme une égalité :

```
if (a = b ) {
/* ceci devrait être exécuté
quand a est égal à b */
}
```

# affectation

- en Java la syntaxe fautive est conservée, mais l'erreur est détectée

# affectation

- Opérateurs d'affectation combinés :
  - `t[i] := t[i] + 1` nécessite deux références à l'élément du tableau
  - il n'est pas facile en général de programmer une procédure qui fait le même travail
  - Algol 68 introduit les affectations combinées : `t[1] + := 1`
  - C fait la même chose : `b.c[3].d *= e;`
  - C ajoute les opérateur immédiats sur les variables : `*p++ = *q++;`
  - Icon combine la totalité des opérateurs infixes : `a ||| := b`

# affectation

- Quelques langages proposent l'affectation multiple :
  `a, b := b, a ;` permet d'échanger les valeurs de `a` et `b`.

# Ordre d'évaluation dans les expressions

- Associativité et priorité déterminent l'ordre d'évaluation des opérations, mais pas des opérandes : dans `a - f(b) - c * d` on ne sait normalement pas si `f(b)` est évalué avant toute opération, ni si la multiplication est faite avant la première soustraction.
- Même remarque pour les paramètres des sous-programmes (sauf exceptions) : dans `f(a, g(b), h(c))` on ne sait pas dans quel ordre sont évalués les deux appels de fonctions.

# ordre d'évaluation

- Deux raisons pour que ce soit important :
  - changement de signification si un appel de fonction a un effet de bord (modification de l'état de la machine abstraite)
  - possibilité de produire du code efficace si le compilateur a le droit de choisir l'ordre d'évaluation

# ordre d'évaluation

- Les définitions de langages hésitent sur ce qu'il faut faire :
  - celles de Pascal ou Ada spécifient que l'ordre n'est pas spécifié
  - celles d'Algol 60 et de Java imposent l'évaluation de gauche à droite
  - celle d'Algol 68 spécifie que l'évaluation est collatérale

# ordre d'évaluation

- Si le compilateur réordonne certaines opérations, cela peut :
  - supprimer ou provoquer des dépassements de capacité : `b - c + d` n'est pas identique à `b + d - c` si les trois valeurs absolues sont proches de l'entier maximum
  - modifier de manière significative la précision du résultat : si `b = -c`, alors `a + b + c` n'est pas égal à `a` si `a` est très petit devant `b` et que l'évaluation est faite strictement de gauche à droite

# Opérateurs booléens à court-circuit

- En algèbre de Boole, $(a > b) \vdash (b > c)$ applique l'opérateur d'intersection aux deux comparaisons, qui sont donc toutes les deux évaluées.
- En programmation, `(a > b)` **and** `(b > c)` peut ne pas évaluer la deuxième comparaison si la première est fausse.

# opérateurs à court-circuit

- Les langages diffèrent sur ce point :
  - en Algol 60, tout est évalué
  - en Pascal, on ne doit pas programmer de manière à compter sur la non-évaluation
  - en C, les opérateurs booléens font le court-circuit
  - en Ada, les deux types d'opérateurs sont fournis
- Le choix du court-circuit facilite la programmation.
- Le choix de l'évaluation totale facilite la génération de code et peut produire du code plus efficace.

# Structures d'énoncés

- Flot de contrôle non structuré
  - Dans les langages d'assemblage, le flot de contrôle est traité à l'aide de trois concepts non structurés :
    - étiquettes
    - branchement inconditionnel
    - branchement conditionnel
  - En Fortran IV, Cobol, Algol 60 ou PL/I, ces mécanismes sont les seuls qui permettent de programmer une itération commandée par une condition logique.

# contrôle non structuré

- L'énoncé de branchement a subsisté en Pascal, Ada ou C, comme moyen de traiter des situations exceptionnelles :
  - sortie au milieu d'une boucle : une condition couvrant le reste de la boucle la remplace sans problème
  - sortie prématurée d'un sous-programme : un énoncé spécialisé est sans doute préférable, à condition de ne pas en abuser

# contrôle non structuré

- traitement d'erreurs : la sortie du sous-programme peut éventuellement sortir de plusieurs sous-programmes en cours, d'où effondrement de la pile d'exécution ; un traitement d'exception est sans doute préférable (Ada, C$^{++}$, Java)

# contrôle non structuré

- L'abandon du branchement est le résultat de l'idée de programmation structurée des années 70 (Dijkstra, puis Hoare et Wirth) : le flot de contrôle du programme doit être déterminé par des structures d'énoncés à une entrée et une sortie.
- Introduction des nouvelles structures :
  - conditionnel : Algol 60
  - répétitif : Algol 60
  - choix sélectif : Algol W
  - itératifs : Pascal
  - plusieurs langages ont tenté de proposer des structures itératives plus complexes, sans grand succès

# Séquence

- Caractéristique des langages impératifs.
- Un groupe d'énoncés est parenthésé entre **begin** et **end** ou entre accolades (C, Icon).
- L'énoncé composé peut avoir une valeur (Ada, C, Icon, Lisp), celle de la dernière expression évaluée.
- Les énoncés sont exécutés dans l'ordre indiqué, mais les bons compilateurs ne se privent pas de changer l'ordre si c'est possible et souhaitable.

# Sélection

- En Fortran, seulement branchement conditionnel : `IF (A - B) 10, 20, 30`
- Algol 60 introduit la forme utilisée par Pascal :

```
if c1 then ...
else if c2 then ...
...
else ...
```

# sélection

- Pas de symbole de fin, ce qui pose un problème syntaxique :
  - en Algol 60, l'énoncé suivant **then** ne peut être un énoncé **if**
  - en Pascal, règle en langue naturelle pour lever l'ambiguïté
  - en Algol 68, Fortran 77, etc., on ajoute un symbole de fin
  - pour éviter une cascade de **endif**, on ajoute un symbole **elsif** pour remplacer **else if**

# sélection

- Si la condition est compliquée, le compilateur peut produire du code qui court-circuite l'évaluation d'une partie :

```
if c1 and c2 then
    ...
endif
```

est équivalent à

```
if c1 then if c2 then
    ...
endif endif
```

# sélection

- S'il y a une partie **else**, l'équivalent nécessite des branchements et étiquettes, ou la duplication d'une partie du code.
- Construction assez différente en Lisp :

```
(cond
   ((= a b) (...))
   ((< a b) (...))
   ...
   (t (...)))
```

# Choix

- L'énoncé **case** est introduit par Algol W (Wirth et Hoare) :

```
case i in
    e1 ;
    e2 ;
    ...
end case
```

# choix

- La forme plus générale est due à Pascal :

```
case exp in
a : e1 ;
b,c : e2 ;
d..f : e3
end
```

- Modula-2 ou Ada offrent des variantes syntaxiques.
- L'intérêt majeur de l'énoncé est de permettre de produire du code très efficace, en plus de la clarté expressive.

# choix

- Les langages diffèrent suivant le traitement des cas non prévus :
  - cas spécial introduit par **else** ou **otherwise**
  - erreur à la compilation si un cas manque (Ada)
  - les cas absents sont ignorés (Fortran 90, C)
- C propose une syntaxe très primitive :
  - pas d'intervalles d'étiquettes ni d'étiquettes multiples
  - traitement de **switch** comme l'aiguillage d'Algol 60, où chaque cas nécessite une sortie explicite si l'on ne veut pas enchaîner avec le suivant
  - malheureusement, cet énoncé est repris tel quel par C$^{++}$et Java

# Boucles

C’est un des domaines dans lequel les auteurs de langages ont le plus laissé libre cours à leur imagination, spécialement dans les années 70. Initialement les langages ne connaissaient que les boucles commandées par la progression arithmétique d’une variable.

# Répétition

- Énoncé primitif en Fortran :
  - La forme est :

```
      DO 10 I = 1, 20, 2
      ...
10    CONTINUE
```

  - Ici 10 est une étiquette, et l'énoncé étiqueté ne fait rien.
  - `I` est la variable de commande de la boucle, et prend les valeurs de 1 à 19 par pas de 2.
  - `I` est une variable simple, locale et entière.
  - Les bornes et le pas sont des constantes ou variables entières.

# répétition

- Les problèmes cachés sont nombreux :
  - rien n'empêche les énoncés dans la boucle de changer la valeur de `I`, ce qui change le nombre de répétitions
  - des branchements peuvent entrer dans la boucle ou en sortir
  - la valeur de `I` au sortir de la boucle n'est pas définie par le langage
  - même si l'intervalle est vide, le corps de la boucle est exécuté au moins une fois

# répétition

- Questions générales : Soit la forme plus agréable de Modula-2 :

```
FOR i := un TO deux BY trois DO
    ...
END
```

  - peut-on modifier `i`, `un`, `deux` et `trois` dans le corps de la boucle ? quel est l'effet ?
  - que se passe-t-il si `un` est supérieur à `deux` quand `trois` est positif ?
  - quelle est la valeur de `i` à la fin de la boucle ?
  - `i` peut-elle être autre chose qu'une variable simple ?

# répétition

- Réponses :
  - Algol 60 définit son énoncé répétitif sous une forme si générale qu'elle permet n'importe quoi :

    **for** `t[i] := j` **to** `t[k]` **do** `i := i + 1;`
  - La plupart des langages (mais pas C) imposent des restrictions :
    - pour permettre la production de code efficace
    - pour permettre de déterminer le nombre de répétitions dès l'entrée de la boucle

# répétition

- suite des réponses :
  - la variable de commande doit être simple, locale et non modifiée dans la boucle (y compris par un appel de sous-programme)
  - la valeur finale et le pas ne sont évalués qu'une seule fois, avant l'entrée dans la boucle
  - la condition d'achèvement est vérifiée avant l'entrée dans la boucle
  - la valeur de la variable de commande au sortir de la boucle n'est pas définie

# Itération

- L'énoncé **while** est introduit par Algol W.
- Pascal ajoute l'énoncé **repeat**.
- Modula-2 ajoute un énoncé **loop**, avec sortie au milieu par un énoncé **exit**.
- Certains langages ajoutent des formes symétriques des précédentes.

# itération

- Plusieurs langages proposent des mécanismes variés de sortie de boucle :
  - sortie de plusieurs boucles à la fois (nécessite d'étiqueter les boucles)
  - saut immédiat à la prochaine itération
  - l'effet majeur est de rendre moins clair l'effet de la boucle, de rendre difficile la définition d'un invariant

# itération

- Quelques langages proposent une combinaison de répétition et d'itération :
  - Algol 60 :

    ```
    for i := 1, 4 to j, i + 1 while i < k do ...
    ```
  - PL/I :

    ```
    FOR i := j BY k TO m WHILE t[i] < n DO; ... END;
    ```
  - C :

    ```
    for (i = un; i <= deux; i += trois) { ... }
    for (;;) { ...; break; ... }
    ```
  - L'énoncé est compliqué à comprendre, et il est très difficile de produire du code efficace.

# Autres formes de boucles

- Pascal étendu fait parcourir à la variable de commande un ensemble de valeurs :

```
for v in ensemble do ...
```

- Le langage le plus achevé est SETL, qui manipule directement les ensemble et les quantificateurs.
- Icon propose le concept de générateur :

```
every i := un to deux by trois do { ... }
every write(upto(' ', chaine))
every element :=!liste do { ... }
```

# autres boucles

- On peut construire des mécanismes similaires dans des langages plus classiques grâce au passage de procédure en paramètre.