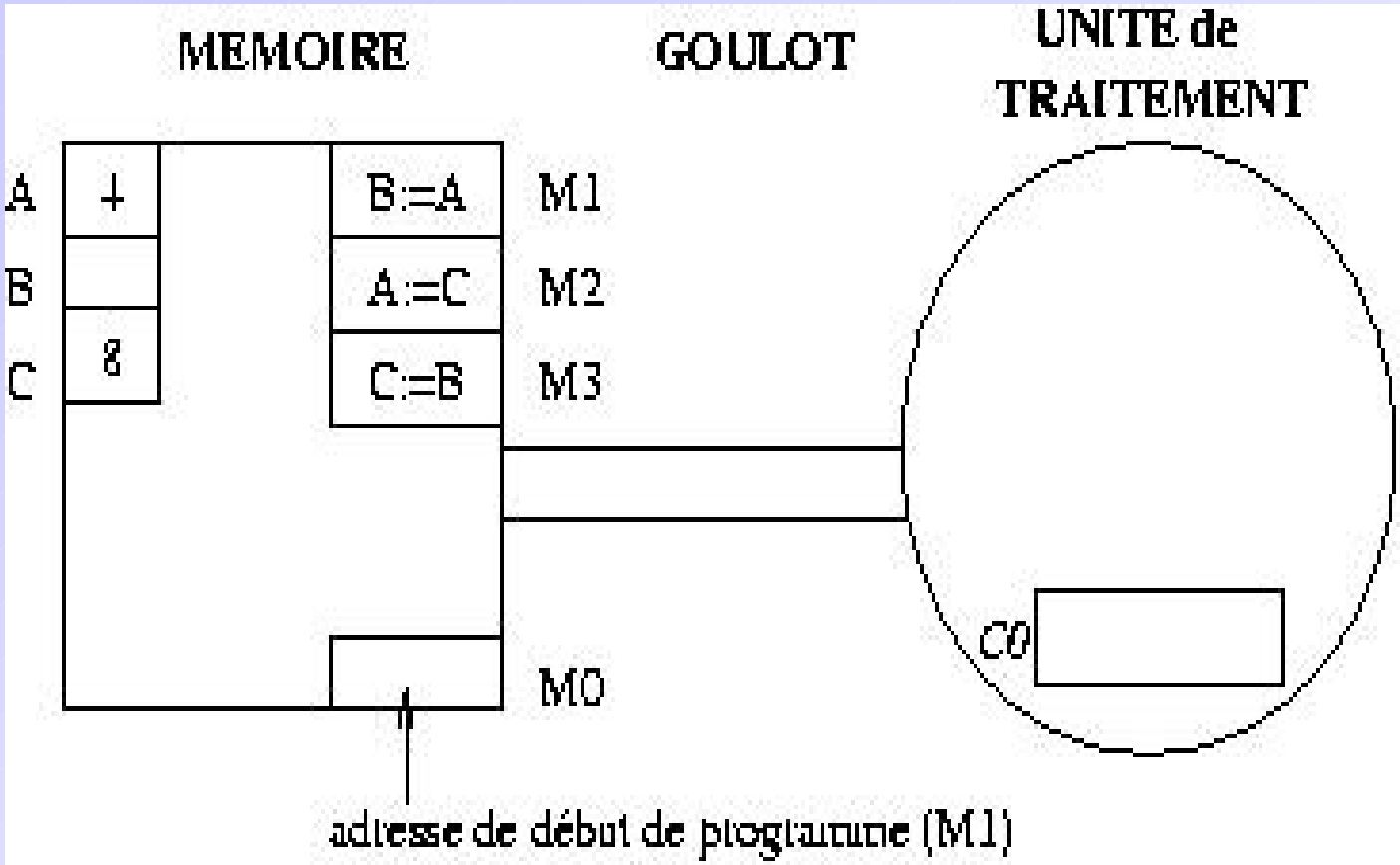
Les structures de contrôle

Chapitre 3

le modèle Von Neumann

Les machines actuelles, dans leur forme la plus rudimentaire, s'apparentent à un modèle qui date de la fin des années 40, le modèle Von Neumann. Selon ce modèle, un ordinateur est constitué de 3 parties :

- une unité centrale de traitement (CPU)
- une mémoire
- un tube qui connecte CPU et mémoire et permet de transmettre des mots de l'une vers l'autre.



Programme de permutation des 2 valeurs 4 et 8 dont les noms en mémoire sont, à l'instant initial *A* et *C* :

```
B:=A;
A:=C;
C:=B;
```

actions	passages dans le goulot	commentaires
M0->C0	1	adresse de début → dans le compteur ordinal
C0->M1	1	transfert de l'adresse de la 1ère instruction
B:=A->UAL	1	transfert de l'instruction dans l'UAL
UAL->A	1	transfert de l'adresse A
4->UAL	1	transfert de la valeur 4 dans l'UAL
UAL->B	1	transfert de la valeur 4 dans B
C0->M2	1	transfert de l'adresse de la 2ème instruction
A:=C->UAL	1	transfert de l'instruction dans l'UAL
UAL->C	1	transfert de l'adresse C
8->UAL	1	transfert de la valeur 8 dans l'UAL
UAL->A	1	transfert de la valeur 8 dans A
C0->M3	1	transfert de l'adresse de la 3ème instruction
C:=B->UAL	1	transfert de l'instruction
UAL->B	1	transfert de l'adresse B
val(B)->UAL	1	transfert de la valeur de B dans l'UAL
UAL->C	1	transfert de 4 dans C
	=16	

Variable informatique

Les cellules mémoires nommées A, B, C, M0, M1, etc... sont modélisées dans les langages de programmation par des *variables informatiques*. Une variable informatique prend donc des valeurs différentes au cours de l'exécution d'un programme.

On notera la différence entre une variable informatique et une variable au sens mathématique qui n'est qu'une entité abstraite ne pouvant désigner qu'une unique valeur.

Affectation

Le moyen d'"imiter" les actions d'"aller chercher dans la mémoire" et de "mettre en mémoire" est donné par **l'instruction d'affectation** notée `:=` en Ada (comme dans l'exemple précédent).

C'est l'instruction qui sert essentiellement à modifier la valeur d'une variable.

Syntaxe

```
<affectation>::=
      <expression_gauche>:=<expression droite>;
<expression gauche> dénote une adresse
<expression droite> dénote une valeur
```

Exemple

```
X:=Y+2;
X:=X/4;
```

On va chercher la valeur de X en mémoire, on la divise par 4 dans l'UAL, puis on transfert cette valeur à l'adresse notée symboliquement X

Sémantique de l'affectation

La valeur de la partie droite est enregistrée à l'adresse de la variable en partie gauche.

L'affectation sépare la programmation en 2 mondes :

- Le premier, concerné par la partie droite de l'affectation, est un monde d'expressions avec leurs propriétés algébriques, un monde dans lequel ont lieu les calculs.
- Le second, concerné par la partie gauche est un monde d'adresses permettant de localiser des données (valeurs ou instructions) dans la mémoire.

Séquence

L'idée imposée par le modèle Von Neumann de penser un programme en terme de trafic à travers le goulot amène à organiser séquentiellement la suite d'affectations.

En Ada, une séquence d'instructions est obtenue par simple juxtaposition textuelle de plusieurs instructions.

Entrées/Sorties

Un autre moyen de modifier la valeur d'une variable est réalisé par la lecture à partir d'un support externe par la commande :

```
get(X);
```

où X dénote une adresse

La commande :

```
put(expression);
```

Evalue l'expression et réalise une affectation de cette valeur sur le fichier standard de sortie qui est l'écran. L'affichage de la valeur peut aussi s'appliquer à un fichier de sortie quelconque.

On parle ici de commandes et non d'instructions. En effet, elles ne sont pas partie intégrante du langage mais fournies par une bibliothèque.

Structures de contrôle

La seule instruction qui a des effets sur le contenu de la mémoire est l'affectation.

Les autres instructions ne sont là que pour permettre d'organiser de manière à la fois plus concise et plus lisible la suite des affectations.

On appelle ces instructions des structures de contrôle.

On distingue les structures de contrôle conditionnelles et itératives.

Conditionnelle : Syntaxe

```
<conditionnelle>::=
      if <expression_booléenne>
      then
            <suite_instructions>
      [elsif <expression_booléenne>
          then
             <suite_instructions>
          elsif <expression_booléenne>
             then
               <suite_instructions>
      ...]
            [else
                  <suite_instructions>]
      end if;
```

Conditionnelle : Sémantique

- Les parties entre crochets sont optionnelles
- Les conditions sont évaluées en séquence
- Si une condition est vraie, la suite d'instructions associée est exécutée et l'instruction conditionnelle est terminée
- Si toutes les conditions sont fausses, la partie **`else`** est exécutée et l'instruction conditionnelle est terminée

Exemple

```
if A=0.0
then
-- calcul du cas linéaire
elsif B**2-4.0*A*C>=0.0
   then
   -- calcul des racines réelles
   else
   -- calcul des racines complexes
end if;
```

Choix multiple : Syntaxe

```
<choix>::=
case <expression> is
  when <valeur> => <suite_instructions>;
  when <valeur>|<valeur> => <suite_instructions>;
  when <valeur>..<valeur> =><suite_instructions>;
  when others => <suite_instructions>;
end case;
```

Choix multiple : Sémantique

- `<expression>` est évaluée. Sa valeur appartient à un type discret
- Les valeurs utilisées en tant que filtre sont comparées de haut en bas à la valeur de `<expression>`
- Dès qu'il y a égalité entre la valeur de l'expression et celle du filtre, la suite d'instructions correspondante est exécutée
- L'ensemble des filtres d'une instruction à choix multiple représente exhaustivement l'ensemble des valeurs du type de l'expression.
- Le filtre **`others`** capte les valeurs qui n'ont pas été citées en tant que filtre

Exemple

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure calculette is
  type Operateur is (add,mult,sous,div);
  package op_io is new Enumeration_io(Operateur);use
   op_io;
  operation:Operateur; X,Y,R:Integer;
begin
 get(operation);get(X);get(Y);
 case operation is
  when add  => R:=X+Y;
  when sous => R:=X-Y;
  when mult => R:=X*Y;
  when div  => R:=X/Y;
 end case;
 put(R);
end calculette;
```

Structure itérative générale

Les instructions itératives permettent de répéter un traitement décrit par une suite d'instructions tant qu'une décision d'interruption de l'itération n'est pas rencontrée.

Syntaxe

```
<boucle>::=
      loop
            [<suite_instructions>;]
            [exit when <condition_arrêt>;]
            [<suite_instructions>;]
      end loop;
```

Sémantique

- Les parties entre crochets sont facultatives
- La condition d'arrêt est une expression booléenne
- L'instruction **`exit`** permet de sortir directement de la boucle
- Dans le cas où les boucles sont imbriquées, **`exit`** permet de sortir vers le niveau immédiatement supérieur
- Une boucle sans condition d'arrêt est une boucle infinie

Exemple : boucle infinie

```
with Ada.Text_io;use Ada.Text_io;
procedure infinie is
begin
  loop
    put( "SOS" );
  end loop;
end infinie;
```

Exemple : saisie d'entiers

```
with Ada.Text_io; with Ada.Integer_Text_io;
procedure saisie is
  X:Integer;
begin
  Ada.Text_io.put( "saisie d'entiers positifs" );
loop
  Ada.Integer_Text_io.get(X);
exit when X<=0;
  Ada.Text_io.put( "la valeur saisie est : " );
  Ada.Integer_Text_io.put(X);
 end loop;
 Ada.Text_io.put( "saisie terminée" );
end saisie;
```

Exemple : affichage

```
with Ada.Text_io;
procedure alpha is
  C:Character:='A';
begin
  Ada.Text_io.put( "Caractères alphabétiques" );
  loop
    Ada.Text_io.put(C);
  exit when C='Z';
    C:=Character'succ(C);
  end loop;
end alpha;
```

Boucle `while`

Syntaxe

```
<boucle_while>::=
   while not <condition_arrêt> loop
   <suite_instructions>;
   end loop;
```

avec

`<condition_arrêt>` est une expression booléenne.

Exemple : calcul de la racine carrée entière (1/2)

Spécification formelle

$$N >= 0 => \exists x,\ x^2 <= N < (x+1)^2$$

Exemple : calcul de la racine carrée entière (2/2)

```
with Ada.text_io,Ada.Integer_Text_io;
use Ada.text_io,Ada.Integer_Text_io;
procedure racine_carree is
  x:Natural:=0; y:Natural:=1; N:Natural;
  -- x<=N ∧ y=(x+1)²
begin
  get(N);
  while N>=y loop
    x:=x+1;
    y:=y+2*x+1;
  end loop;
  put( "la racine carrée entière de " );
  put(N); put( " est : " ); put(x);
end racine_carree;
```

Boucle `for` : syntaxe

```
<boucle_for>::=
  for <ident_de_compteur> in [reverse]
      <borne>..<borne> loop
      <suite_instructions>
  end loop;
```

où

`<borne>` est une valeur d'un type discret représentant une borne de l'intervalle de valeurs que peut prendre `<ident_de_compteur>`.

`reverse` est facultatif. Il indique un parcours inverse (dans l'ordre décroissant) de l'intervalle `<borne>..<borne>`

Boucle `for` : sémantique

Pour chaque valeur de l'intervalle discret parcouru dans l'ordre croissant (`in`) ou décroissant (`in reverse`) : `<suite_instructions>` est exécutée.

Le compteur de boucle `<ident_de_compteur>` est une constante créée au début de chaque pas d'itération et détruite après.

Il est implicitement déclaré.

Exemple

```
with Ada.Text_io;use Ada.Text_io;
procedure pour is
      U:Integer:=1;
begin
   for I in reverse 1..3 loop
      U:=I*U;
      put( "U=" );
      put(Integer'image(U));new_line;
   end loop;
end pour;
```

résultat

```
U= 3
U= 6
U= 6
```

Exemple

```
with Ada.Text_io;use Ada.Text_io;
procedure pour is
      U:Integer:=1;
begin
   for I in reverse 1..3 loop
      U:=I*U;I:=I+2;
      put( "U=" );put(Integer'image(U));new_line;
   end loop;
end pour;
```

erreur à la compilation :
pour.adb:6:14 assignment to loop parameter not allowed

Tirage du loto

```
with Ada.Text_io; with Ada.Integer_Text_io;
with Ada.Numerics.Discrete_Random;
use Ada.Text_io, Ada.Integer_Text_io;
procedure loto is
  subType Numeros is Integer range 1..50;
  package tirage_loto is new
   Ada.Numerics.Discrete_Random(Numeros);
  use tirage_loto;
  graine:Generator;
begin
  reset(graine);
  put_line( "les numéros de la semaine !" );
  for i in 1..6 loop
    put(Random(graine),WIDTH=>2); new_line;
  end loop;
end loto;
```

Structure de bloc

Ada est un langage dit "à structure de bloc" en ce sens qu'il permet d'associer un environnement spécifique à une suite d'instructions donnée.

Un bloc correspond ainsi à une instruction composée de plusieurs instructions qui évolue dans un environnement propre.

Un bloc est dès lors une instruction comme une autre.

Un bloc est une suite d'instructions auquel peut être attachée une partie déclarations et/ou une partie exception.

Structure de bloc : Syntaxe

```
<bloc>::=
   declare
        <liste_déclarations>
   begin
   <suite_instructions>
   [exception]
   <traitement_exceptions>
   end;
|
   begin
   <suite_instructions>
   [exception]
   <traitement_exceptions>
   end;
```

Structure de bloc

```
declare
-- partie déclarations
begin
-- instructions
  declare
  -- partie déclarations
  begin
  -- partie instructions
  [exception]
  -- traitement des exceptions
  end;
[exception]
-- traitement des exceptions
end;
```

Les blocs peuvent s'emboîter

Les blocs ne peuvent pas se chevaucher