Programmation (I)

Langages et bases algorithmiques

Claude Guéganno
–
Ingénieur E.N.S.E.R.B.
Certifié en Informatique-télématique
Agrégé de Génie Électrique
–

4 septembre 2001

Table des matières

Première partie

Ordinateurs et langages de programmation

Chapitre 1

Notions de bases

1.1 Structure d'un ordinateur

Le principe des ordinateurs a été énoncé par John VON NEUMANN, dès 1946. Il indique que la mémoire dont est munie la machine doit servir non seulement à enregistrer des données, mais aussi les programmes de traitement que l'on veut appliquer à ces données.

Une telle machine est théoriquement capable de charger en mémoire et d'exécuter des programmes différents.

Un programme est une suite d'instructions élémentaires enregistrées dans des cellules consécutives de la mémoire. L'exécution de ces instructions se fait dans l'ordre ou elles apparaissent, sauf dans le cas d'un branchement [1]. Un branchement peut être :

- inconditionnel, ce qui permet à l'ordinateur, par exemple, de répéter à l'infini une séquence;
- ou conditionnel, un choix est alors efectué.

Un ordinateur est constitué, d'un point de vue fonctionnel, de trois parties principales: la **mémoire centrale**, l'**unité de traitement** et les **unités d'échanges**, qui assurent les échanges avec les périphériques standards ou non, ainsi qu'avec les mémoires additionnelles.

1.2 Les langages

1.2.1 Les langages machine

L'unité de traitement peut exécuter par ses circuits câblés un certain nombre d'opérations élémentaires. Elles se divisent en quatre catégories principales:

- les opérations de **traitement arithmétique et logique** (ex: addition, ET logique ...);

1. ou «rupture de séquence».

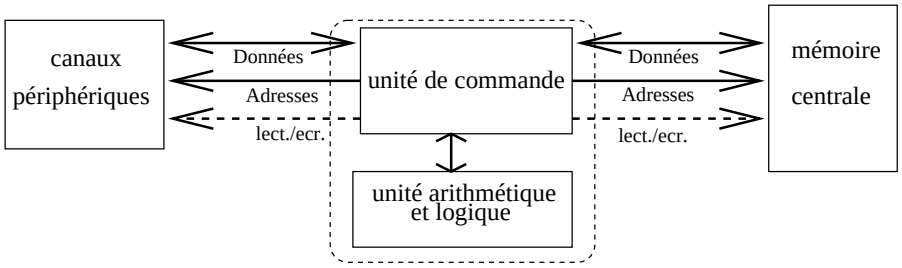


FIG. 1.1 – *Structure générale d'un ordinateur.*

- les opérations de **transfert** entre mémoire et registres de l'unité de calcul, dans un sens ou dans l'autre.
- les opérations d'**entrée** et de **sortie** entre mémoire centrale et périphériques;
- les opérations de **sauts** conditionnel ou inconditionnel qui permettent de commander le déroulement logique du programme.

Chaque ordinateur possède son propre jeu d'instructions élémentaires. C'est le **langage machine**. Il est codé en binaire (suite de 0 et de 1).

1.2.2 Les langages d'assemblage

Le langage d'assemblage reprend le même jeu d'instruction que le langage machine, mais avec un codage symbolique des différents éléments:

- les codes opérations sont des *mnémoniques* (ex: `move` pour les transferts, `add` pour l'addition ...)
- les registres sont désignés par des symboles (ex: `d0`),
- les cellules mémoires peuvent être représentées par des étiquettes.

Exemple L'instruction du processeur 68000 (`Motorola`)

`00 01 001 000 010 101` → langage machine

correspond à l'instruction mnémonique

`move.b (a5), d1` → langage assembleur

ce qui signifie que l'octet situé à l'adresse pointée par le contenu du registre `a5` doit être transféré dans le registre de données `d1`.

La traduction du langage machine en langage assembleur est confiée à un programme qui prend en entrée un fichier assembleur, et qui produit en sortie un fichier binaire.

1.3 Les langages de programmation évoluée

1.3.1 Généralités

Le concept de langage de programmation évolué date de la fin des années 1950. L'objectif consistait à rapprocher la description programmée de la description «naturelle». Deux langages ont marqué le début de la programmation évoluée :

- `FORTRAN`[2] (1956) est le premier langage évolué a avoir connu une très large diffusion. Dédié au calcul scientifique, il permet de générer un grand nombre d'instructions élémentaires à partir d'une instruction évoluée. Par exemple, la formule

$$\Delta = b^2 - 4 \times a \times c$$

  se traduit en `FORTAN` par l'expression

```
Delta = B**2 -4*A*C
```

- `COBOL`[3] est un langage orienté vers la gestion administrative.

L'apport essentiel de ces langages est la notion d'abstraction[4] des zones de mémoire physiques (registres, mémoire centrale) et de leurs adresses au profit de la notion de *variable*.

Une **variable** est repérée par un **identificateur symbolique**, caractérisée par un **type** et par une **valeur**.

À un instant donné, l'état d'un programme est parfaitement décrit par l'ensemble des valeurs de ses variables (registres ou mémoire) ainsi que par un repère d'exécution qui indique l'instruction en cours.

1.3.2 Modularité

Le concept de modularité a eu une influence déterminante sur l'évolution des langages de programmation. L'idée de base est qu'un problème complexe est plus facile à résoudre lorsqu'il est décomposé en sous-problèmes plus petits. Ainsi, la réalisation d'un gros système logiciel sera simplifiée si ce système est décomposable en modules.

Conséquence sur les langages : la possibilité de «modulariser» un gros programme est prise en compte par la plupart des langages modernes :

- les *packages* de `Ada`
- les classes de `C++ ou Java`
- `les bibliothèques du langage C`
- `les unités du Pascal`
- `les modules de Modula 2`
- `...`

2. pour FORmula TRANslator
3. COmmon Business Oriented Language
4. Dans le sens de dissimulation

Qualités apportées par la modularité:

- meilleure lisibilité;
- évolutivité facilitée;
- possibilité de de vérifier les propriétés des modules;
- réutilisation de composants pré-existants;
- compilation séparée

Contraintes pour les langages :

- Les langages doivent offrir des **constructions syntaxiques** délimitant explicitement les composants modulaires.
- Chaque module doit constituer «un tout» par lui même. Les interactions avec d'autres modules sont consignés dans un **interface**.
- Les modules doivent permettre de **dissimuler** à la vue de l'extérieur des informations, et, en particulier de cacher les détails de leur réalisation. C'est la notion d'**abstraction**. On défini les notions d'informations **publiques** ou **privées**.
- Les erreurs syntaxiques doivent pouvoir être détectées à l'intérieur d'un module donné.

1.3.3 Parallélisme et programmation concurrente

Le **parallélisme**, c'est l'exécution simultanée de plusieurs programmes. Ceci suppose que l'architecture de la machine le permet. Une telle machine dispose donc de plusieurs processeurs. Le parallélisme ne sous entend pas que les programmes exécutés simultanément participent à la même application. Ils peuvent être complètement indépendants.

Lorsqu'une machine ne dispose que d'un seul processeur (micro-informatique classique), on peut néammoins parler dans certains cas de **pseudo-parallélisme**. Chaque programme se voit affecter un processeur virtuel. Le système d'exploitation prète successivement le processeur physique aux différentes tâches pour qu'elles puissent s'exécuter convenablement en fonction des contraintes de temps qui leurs sont imposées.

Qu'il y ait un seul ou plusieurs processeurs, on parle de programmation **concurrente** lorsque plusieurs programmes (ou parties de programmes) qui participent à la même application, s'exécutent simultanément, ou quasi-simultanément.

La programmation concurrente est appliquée selon deux stratégies différentes:

1. on utilise un langage classique auquel on ajoute des composants multi-tâches (en général, ce sont des fonctions «systèmes» réalisant des requètes au noyau du système d'exploitation qui est alors nécessairement multi-tâche).
2. on programme dans un langage concurrent (`Ada` par exemple)

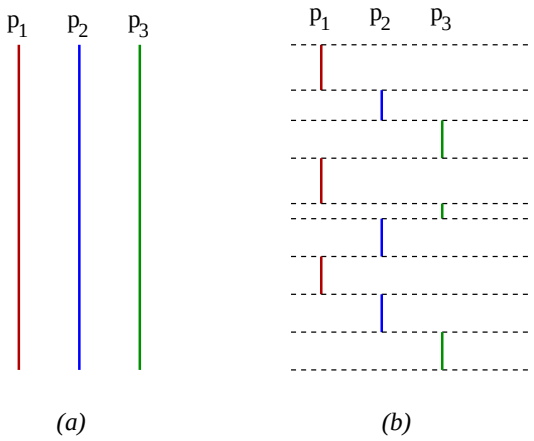


FIG. 1.2 – *Parallélisme vrai (a) et apparent (b).*

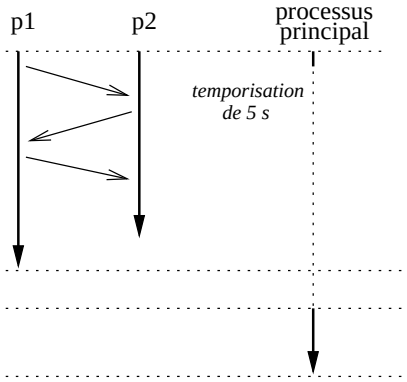
Exemple de programmation concurrente. L'exemple suivant est codé en langage `Ada`. Il met en œuvre une procédure qui se «divise» en trois tâches parallèles. Le mot clé `Task` indique que le bloc d'instructions correspondant doit être exécuté «en parallèle».

```
with Gnat.Io; use Gnat.Io;

procedure Par is
   -- processus P1
   Task P1;
   Task body P1 is
   begin
      for I in 1..30 loop
         Put("1");
         delay Duration(0.1);
      end loop;
   end P1;

   -- processus P2
   Task P2;
   Task body P2 is
   begin
      for I in 1..12 loop
         Put("2");
         delay Duration(0.2);
      end loop;
   end P2;

   -- processus principal
begin
   delay Duration(5);   -- Attendre la fin de tout
   New_Line;
   Put_Line("Fin du programme");
end Par;
```



Trace d'exécution :

```
claude@Claude:/home/claude/Classes/Sts1/langages > par
211211211211212112112112121121121111111111
Fin du programme
claude@Claude:/home/claude/Classes/Sts1/langages >
```

Cettre trace montre que les exécutions respectives de `p1` et `p2` sont bien simultanées.

La programmation concurrente sera abordée plus en détail dans le cours «Systèmes multi-tâches et temps-réel»

1.4 Compilation et interprétation

La traduction d'un langage évolué en langage machine peut se faire selon deux techniques différentes :

1. la **compilation** : le traducteur appelé compilateur traduit l'ensemble du programme écrit dans un langage évolué en un programme en langage machine. L'exécution se fait dans un second temps après chargement en mémoire du programme objet. Le programme source et le traducteur ne sont plus utiles pour l'exécution.
2. l'**interprétation** : le traducteur est cette fois un interpréteur. Il traduit chaque instruction puis l'exécute immédiatement avant de passer à l'instruction suivante.

La compilation a l'avantage de ne traduire chaque instruction qu'une seule fois. La traduction peut être optimisée pour que l'exécution du programme soit plus rapide (ou dans certains cas pour que le programme exécutable soit plus compact).

L'interprétation entraîne une lenteur certaine à l'exécution à cause du processus de traduction. De plus, une même instruction peut être traduite plusieurs fois. Par contre la mise au point est facilitée, les erreurs pouvant être prises en compte

Chapitre 2

Les familles de langages évolués

On peut répartir les langages de programmation en familles. Parmi les familles dominantes, nous aborderons :

- la programmation impérative,
- la programmation fonctionnelle,
- la programmation orientée objet,
- la programmation en logique.

2.1 La programmation impérative

C'est la programmation la plus ancienne. Il y a une description explicite de l'enchaînement des instructions. Le programme est écrit dans une logique proche de celle de la machine elle même. En particulier, les notions de donnée et d'instruction sont essentielles.

Dans les langages de programmation impérative, citons au moins :

- le `basic` (1965) : pour *Beginner's All Purpose Symbolic Instruction Code* [1]. C'est un langage interprété, facile à apprendre. La petite taille de son interpréteur, compatible avec la capacité des premiers micro-ordinateurs, a contribué à son essort. Le `Quick Basic` de `Microsoft` en est une amélioration.
- le `Fortran` (1956), créé à l'origine pour le calcul scientifique.
- le `Cobol` (1961) : pour *COmmon Business Oriented Language* [2]. Développé à l'initiative du Département de Défense des U.S.A., il est rapidement devenu un standard.
- `APL` (1962) : caractérisé par une grande richesse d'opérateurs standards (produits matriciels ...).
- `PL/1` [3]. (1964) : premier langage à vocation «universelle». Il permet aussi bien la programmation système que la programmation scientifique.

1. Langage à tout faire pour débutants.
2. Langage orienté vers la gestion
3. Programming Language Number One

- `ALGOL`[4] (1958,66,68) : d'origine, et à vocation universitaire.
- `Pascal` (1969) : langage conçu initialement pour enseigner la programmation. Il a connu un grand succès, et ses applications touchent aujourd'hui de nombreux domaines.
- `C` (1972) : développé en même temps que le système `UNIX`, ce langage est devenu le standard pour le développement de systèmes. Il couvre tous les domaines d'applications. Sa syntaxe a inspiré les langages objets `C++` et `Java`.

Exemples de programmes

Basic

```
bwBASIC: list
     5:  rem factorielle
     10: print "n =?"
     20: input N
     30: R = 1
     40: for i = 1 to N
     50:    R = R * i
     60: next i
     70: print "résultat", R
```

Langage C

```
/* factorielle */
#include <stdio.h>

void main() {
  int i, r, n;
  printf("n =? ");
  scanf("%d",&n);
  for (r=1, i=1; i<=n; i++) r = r*i;
  printf("résultat: %d\n", r);
}
```

Pascal

```
(* factorielle *)
program factorielle;

var i, r, n : integer;

begin
   r := 1;
   write('n =? ');
   read(n);
   for i := 1 to n do
      r := r*i;
   writeln('résultat:', r);
end.
```

Fortran 77

```
c factorielle
      program ex
      integer i,r,n
      r = 1
      n = 3
      read *,n
      do 10 i=1,n,1
         r = r*i
 10   continue
      print *,'résultat:',r
      end
```

Ces quelques exemples de conception mettent en évidence des différences syntaxiques, mais également une ressemblance globale de la structure. Cette remarque montre que l'approche de la programmation impérative peut - et

4. ALGOrithmic Language

peut-être doit - se faire indépendemment de toute syntaxe, en faisant appel à un langage générique théorique : l'algorithme.

2.2 La programmation fonctionnelle

La programmation fonctionnelle ou *applicative*, constitue un style de programmation aussi ancien que la programmation impérative. Elle nécessite des ressources informatiques importantes. C'est ce qui explique que son impact ne s'est véritablement accru qu'après 1980. Les micro-ordinateurs actuels supportent largement les interpréteurs de langages fonctionnels. Parmi ces langages, citons `Scheme` et `CAML`.

Principes généraux. Les langages fonctionnels font appel, comme composants de base, aux **fonctions**. On parle aussi de langages *applicatifs* pour souligner que l'on applique une fonction à une expression.

Les langages fonctionnels

- `Lisp`[5] (1958) : c'est l'un des plus anciens langages de programmation. Il est architecturé autour d'une notion de *liste* et n'est pourvu d'aucun *typage*. À l'origine, il était dédié à la recherche en intelligence artificielle.
- `Scheme` : dialecte de `Lisp`, il est aujourd'hui universellement répandu.
- `ML` (*metalanguage*) : développé à partir de 1978 par R. Milner, il gère en plus les types d'objets (variables ou fonctions).

Exemple de programmation fonctionnelle. Comment traduire en langage `Scheme` le programme de calcul de la factorielle que nous avons donné en exemple dans la section 2.1?

Il s'agit de coder la fonction définie par :

$$\begin{cases} \mathrm{N} \to \mathrm{N} \\ n \mapsto n! \end{cases}$$

avec

$$n! = fact(n) = \begin{cases} 1 \text{ si } n \leq 1 \\ \prod_{i=1}^{n} i \text{ sinon} \end{cases}$$

Soit,

$$fact(n) = \begin{cases} 1 \text{ si } n \leq 1 \\ n \times fact(n-1) \text{ sinon} \end{cases}$$

5. List Programming

De cette dernière expression on déduit la traduction en `Scheme` de la fonction factorielle:

Scheme

```
; factorielle
(define (fact n)
  (cond ((<= n 1) 1)
        (#t (* n (fact (- n 1))))
))
```

Cette écriture n'a rien de commun avec le codage séquentiel de la programmation impérative. On remarque en particulier :

- l'absence de variable,
- l'absence de structure de contrôle,
- l'utilisation du principe de *récursivité* [6],
- la concision du code.

Voici la même fonction, écrite en `ML` dans deux versions différentes.

Caml light

```
(* fonction factorielle *)

let rec fact = function (x) ->
    if x<2 then 1 else x*fact(x-1);;

let rec factbis = function
    0 -> 1
  | x -> x*factbis(x-1);;
```

2.3 La programmation orientée objet

Il s'agit d'un style de programmation visant à faciliter la conception et la maintenance des grandes applications. Les **objets** qui structurent ces programmes sont souvent la transposition des objets du monde réel. Les connaissances déclaratives (les données) et les connaissances procédurales qui caractérisent chaque objet sont regroupées dans une même entité, à la différence de ce qui se passe en programmation impérative classique.

6. Une fonction est dite *récursive* lorsqu'elle apparaît dans sa propre définition.

2.3.1 Les concepts de base

Les langages orientés objets sont des langages conçus pour mettre en œuvre la technique de programmation par objets. Cette technique prône une description des systèmes en termes d'**objets** et de **classes** manipulés, plutôt qu'en termes de fonctions assurées.

Exemple: pour une application graphique, une figure géométrique est définie par :

- une position (variable),
- des fonctions pour dessiner, déplacer, effacer, créer ...

Les objets (appelés aussi *acteurs*, *unités*, *schémas*, «*frames*», selon les langages, rassemblent les connaissances globales liées au problème. Ces connaissances sont de deux sortes, on distingue:

1. une partie **déclarative** : l'objet contient des données aussi nommées *variables*, *champs*, *aspects*, *attributs*, «*slots*»
2. une partie **procédurale** : l'objet contient des programmes ou méthodes.

Les objets sont décrits sur le modèle de **structures de données abstraites** (S.D.A.). Ils constituent des «boîtes noires», dissimulant leur implantation, avec une **interface publique** visible des autres objets; toutes les interactions s'établissent à travers cet interface.

Ces aspects de base sont réalisables avec les langages impératifs fortement modulaires comme `Modula 2` ou `ADA`. Les langages orientés objets leur ajoutent des concepts supplémentaires permettant de tirer un meilleur avantage du découpage en objets. On trouve en particulier :

- **les concepts de classe et d'instanciation :** une classe est un «moule à objets» à partir duquel on fabrique (on instancie) des exemplaires (des instances). Cette opération est appelée l'**instanciation**.
- **le concept de gestion dynamique des objets :** les objets sont créés et gérés explicitement à l'exécution. L'espace libéré par les objets devenus inutiles peut être automatiquement récupérée par un ramasse-miette (*garbage collector*). Cette propriété est mise en œuvre dans le langage `Java`; en `C++`, la gestion dynamique des objets est laissée à l'initiative du programmeur.
- **le concept d'héritage :** une classe n'est pas forcément décrite à partir de zéro. Elle peut être une spécialisation ou une extension d'une classe existante. La nouvelle classe **hérite** automatiquement des données et opérations de sa parente en lui ajoutant ou en lui substituant ses propres données et opérations.
- **le concept de polymorpisme :** permet d'utiliser le même nom de fonction pour des actions équivalentes mais s'appliquant à des classes différentes.

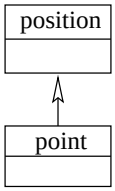


FIG. 2.1 – *Exemple d'héritage.*

– **le concept d'héritage multiple :** permet à une classe donnée d'hériter de plusieurs parents. Ce concept n'a pas été retenu pour le langage `Java`.

2.3.2 Quelques langages orientés objets

On trouve, parmi les plus répandus :

– `Smalltalk` est le résultat d'un effort ambitieux de recherche mené dans le but d'améliorer la relation entre les ordinateurs et les utilisateurs. Les concepts manipulés par l'utilisateur (fenêtres, clics de souris, icônes, ...) ont été modélisés. Il est le précurseur de la programmation objet appliquée aux *G.U.I.*[7]

– `Eiffel`: c'est un langage fortement typé disponible sous `Unix` depuis 1982. Chaque classe construite est assimilée à un type et fait l'objet d'un fichier séparé. Ce langage a posé les concepts essentiels de la programmation objet.

– `C++`: c'est un langage de programmation à usage général dérivé du `C`. Il ajoute à son langage père un grand nombre de caractéristiques, dont les plus importantes sont le support des données abstraites et de la programmation par objets. Le `C++` conserve la plus grande partie du `C`; il en adopte les types de base, les opérateurs, la syntaxe des instructions et la structure de programme. Des caractéristiques nouvelles permettent d'utiliser de nouvelles techniques de programmation.

– `Java` est un langage orienté objet conçu sur le modèle de `C++`. Avec le langage, une machine virtuelle est fournie, ce qui rend `Java` portable au niveau des sources et des binaires. Par rapport au `C++`, il y a de nombreuses simplification qui visent à fiabiliser le langage (pas d'héritage multiple, pas de pointeur ...).

2.3.3 Exemple

L'exemple suivant met en œuvre deux types abstraits : le type `Point` et le type `Rectangle`. Le type `Rectangle` est obtenu par héritage à partir du type `Point`, conformément au schéma de la figure 2.2.

7. Graphic User Interface

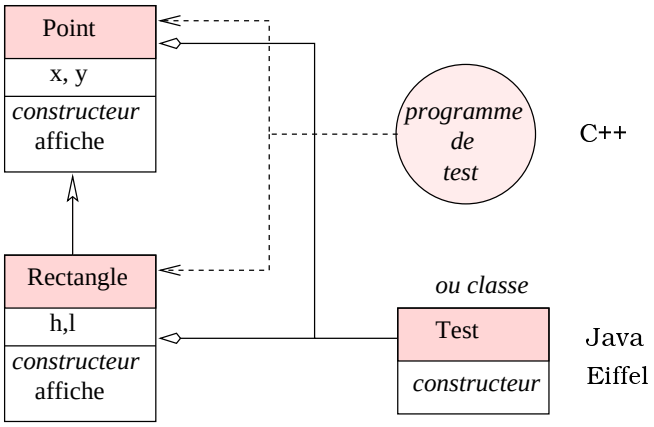


FIG. 2.2 – *Diagramme de classe.*

Eiffel

```
-- *** fichier : point.e
class POINT

creation {ANY}
  makep

feature {ANY}
   x,y : INTEGER;

   makep(xx,yy : INTEGER) is do
        x := xx; y := yy;
   end;

   affiche is do
     std_output.put_string("(");
     io.put_integer(x);
     std_output.put_string(",");
     io.put_integer(y);
     std_output.put_string(")%N");
   end;
end -- POINT
```

```
-- *** fichier : rectangle.e
class RECTANGLE
      inherit  POINT
          redefine affiche end;

creation {ANY}
   maker

feature {ANY}
   l,h : INTEGER;

   maker(xx,yy,ll,hh : INTEGER) is do
        x := xx; y := yy;
        l := ll; h := hh;
   end;

   affiche is do
     std_output.put_string("(");
     io.put_integer(x);
     std_output.put_string(",");
     io.put_integer(y);
     std_output.put_string(") [");
     io.put_integer(l);
     std_output.put_string(",");
     io.put_integer(h);
     std_output.put_string("]%N");
   end;
end -- RECTANGLE
```

```
-- *** fichier : test.e
class TEST
creation {ANY}
   make
feature {ANY}
   point : POINT;
   rectangle : RECTANGLE;

   make is
      do
        !!point.makep(2,3);
        !!rectangle.maker(1,2,3,4);
        point.affiche;
        rectangle.affiche;
      end;
end  -- TEST
```

Remarque : le langage `Eiffel` impose d'écrire chaque classe dans un fichier séparé.

C++

```
// classes 'point' et 'rectangle'
#include <iostream.h>

class point {
  public:
  int  X,Y;
  point() {}
  point(int xx, int yy){
    X = xx;
    Y = yy;
  }
  void affiche() {
    cout <<"(" << X << ","
         << Y << ")";
    cout << '\n';
  }
};

class rectangle : public  point {
  int H,L;
  public:
  rectangle() {}
  rectangle(int xx, int yy,
            int hh, int ll){
    X = xx;
    Y = yy;
    H = hh;
    L = ll;
  }

  void affiche(){
    cout <<"(" << X << ","
         << Y << ")";
    cout << "[" << H << ","
         << L << "]";
    cout << '\n';
  }
};

void main() {
  point P(2,3);
  rectangle R(1,2,3,4);
  P.affiche();
  R.affiche();
}
```

Java

```
// classes 'point' et 'rectangle'
import java.io.*;
class point {
  private int  X,Y;

  public point(int xx, int yy){
    X = xx;
    Y = yy;
  }

  public void affiche() {
    System.out.print("("+X+","+Y+")");
  }
}

class rectangle extends point {
  private int H,L;

  public rectangle(int xx, int yy,
                   int hh, int ll){
    super(xx,yy);
    H = hh;
    L = ll;
  }

  public void affiche() {
    super.affiche();
    System.out.println("["+H+","+L+"]");
  }
}

class test {
  public static void main(String[] args) {
    point P = new point(2,3);
    rectangle R = new rectangle(1,2,3,4);
    P.affiche();
    System.out.println();
    R.affiche();
  }
}
```

Trace d'exécution, commune aux trois versions :

```
(2,3)
(1,2) [3,4]
```

2.4 La programmation en logique

2.4.1 Présentation

La programmation en logique est historiquement liée au langage `Prolog`. L'utilisation de la logique comme base d'un langage de programmation bouleverse la nature même de la programmation. Le style de programmation obtenu est très différent de la programmation impérative. Il ne s'agit plus du tout de décrire pas à pas les étapes pour résoudre le problème. On ne raisonne plus en *fonctions* mais en *relations*. C'est un langage **déclaratif** caractérisé par trois niveaux d'expressions :

1. la spécification des connaissances permanentes relatives au domaine concerné,
2. la spécification du problème à résoudre,
3. un mécanisme de résolution opérant sur les deux autres niveaux, mais indépendant de ceux-ci.

On trouve là les fondements des **systèmes experts**.

Un programme `Prolog` est constitué d'un ensemble de **clauses**. L'interpréteur prouve une propriété (un résultat) à partir de ces clauses. Les mécanismes de base de cette démonstration sont l'**unification** et la **réfutation par résolution**.

2.4.2 Exemple

L'exemple consiste à rechercher une solution pour résoudre le problème d'un aventurier.

Un trésor a été caché dans une cave qui est en réalité un véritable labyrinthe composé de salles reliées entre elles par des galeries. Parmi ces salles, certaines sont dangereuses et y rentrer conduirait inévitablement à une mort certaine.

Le plan du sous-terrain est donné par la figure 2.3 à la page 20.

Quelle route faut-il prendre pour aller de l'entrée à la sortie en passant par la salle au trésor et sans passer par la salle des voleurs ni par celle des monstres ?

Solution `Prolog`: on représente la carte du sous-terrain par une liste de **faits**. Les **prédicats** `route` et `va` donnent les règles. Il ne reste plus qu'à interroger l'interpréteur `Prolog`.

Le programme de la page 20 propose une solution. Le programme s'arrête dès qu'une route convenable est trouvée. Une modification mineure permettrait d'extraire toutes les solutions.

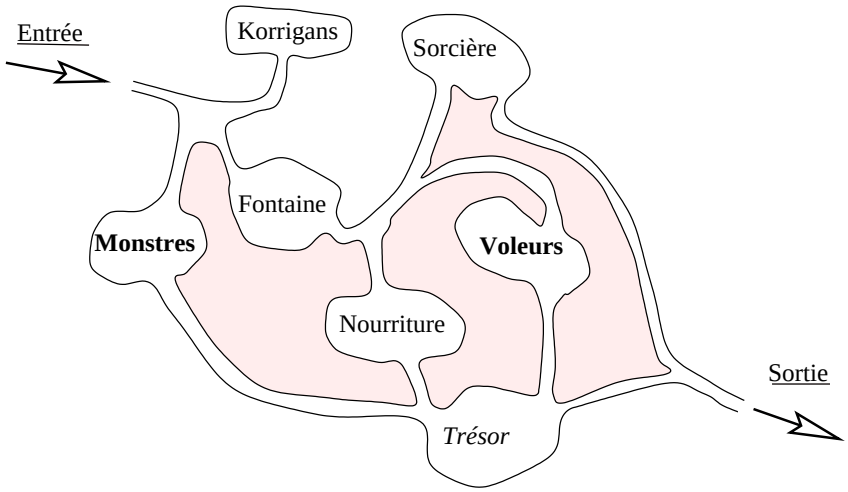


FIG. 2.3 – *Plan du sous-terrain.*

Prolog

```
galerie(entrée, monstres).      galerie(entrée, fontaine).
galerie(fontaine, korrigans).   galerie(fontaine, sorcière).
galerie(fontaine, voleurs).     galerie(fontaine, nourriture).
galerie(nourriture, trésor).    galerie(nourriture, trésor).
galerie(trésor, voleurs).       galerie(trésor,sortie).
galerie(sorcière,sortie).



sallevoisine(X,Y) :- galerie(X,Y).
sallevoisine(X,Y) :- galerie(Y,X).

eviter([monstres, voleurs]).

va(Ici, Labas) :- route(Ici, Labas, [Ici]).
va(_,_).

route(Salle, Salle, Dejavu) :-
  member(trésor, Dejavu), write(Dejavu), nl.

route(Salle, But, Dejavu) :-
  sallevoisine(Salle, Sallesuivante),
  eviter(Sallesdangereuses),
  not (member(Sallesuivante, Sallesdangereuses)),
  not (member(Sallesuivante, Dejavu)),
  route(Sallesuivante,But,[Sallesuivante|Dejavu]).
```

Trace d'exécution :

```
3 ?- va(entrée, sortie).
[sortie, trésor, nourriture, fontaine, entrée]
Yes
4 ?-
```

Remarques

- Cet exemple présente `Prolog` comme une «base de données intelligente».
- `Prolog` permet l'écriture de règles. Ces règles sont écrites de façon très synthétique; par exemple, au lieu d'écrire :

  `Si A et B, alors C;`

  on écrit :

  `C :- A,B.`
- Grâce à son mécanisme d'unification, `Prolog` se débrouille avec les règles énoncées, dès l'instant où le programme ne boucle pas.
- Le programmeur ne s'occupe pas de savoir comment trouver la solution, mais de donner tous les éléments pour qu'elle puisse être déduite.
- L'usage de variables est nécessaire pour l'écriture des règles.

2.5 La programmation des applications «temps réel»

On parle de *système temps réel* ou de *programmation temps réel* dès l'instant où l'exécution du programme est soumise à une ou plusieurs contraintes de temps. (exemple : temps de réponse à un signal issu d'un capteur, période d'échantillonnage d'un asservissement numérique, ...). La conception d'une telle application peut se faire avec deux approches :

1. On utilise un langage de programmation classique (impératif ou ou objet), mais l'exécution du programme est confiée à un système d'exploitation temps-réel (généralement multi-tâches) Des fonctions sont mises à la disposition du programmeur pour tenir compte des propriétés additionnelles du système temps-réel. Parmi les noyaux temps-réel, on trouve (entre autres) `VxWorks` de `Wind River System`, `VRTX` de `Ready System`, `OS9` de `Microware`, `SoftKernel`, ...
2. On programme dans un langage qui tient compte des spécificités du temps-réel. Mais là aussi le système d'exploitation sous-jacent dispose des propriétés requises. De tels langages permettent de tenir compte des différentes contraintes de temps de l'application. et de la découper en processus. `Ada` et dans une certaine mesure `Modula2` sont des langages présentant ces caractéristiques.

2.6 Conclusion

Il n'est pas question, dans le cadre de ce cours d'entreprendre une description détaillée de chaque famille de langage, mais cette présentation moins que sommaire donne une idée de la richesse et de la diversité des langages de programmation.

Nous remarquons également que passer d'une famille de langage à une autre demande de «penser différemment». D'autre part, tous les langages ne sont pas nécessairement adaptés à toutes les applications. Pour une application «temps-réel» on préférera `Ada` à `Prolog`. Pour la conception d'un système expert ce sera le contraire.

Depuis quelques années, la progression exponentielle des performances des ordinateurs a rendu accessible à tous la programmation dans la plupart de ces langages. Depuis le début des années 90, l'initiation à l'informatique se fait, en `Prolog`, en `Scheme`, en `CAML`, en `Java`, ...

Concernant l'informatique industrielle, les références restent les langages impératifs, dont essentiellement le langage `C`, quelques langages objets (`C++` et un peu `Java`) et enfin le langage de programmation concurrente `Ada`.

2.7 Exercices

Exercice 1

Pour chacun des langages suivant, donner la famille à laquelle il appartient: `C`, `C++`, `Lisp`, `Eiffel`, `Smalltalk`, `Pascal`, `Java`, `Fortran`, `Prolog`, `Scheme`, `Ada`.

Exercice 2

Quel langage conviendra le mieux pour la conception d'un système expert ? Même question pour la conception d'une application temps-réel.

Exercice 3

En procédant par imitation, écrire en `Scheme` un programme qui calcule la somme des n premiers entiers.

Deuxième partie

Programmation impérative

Chapitre 1

Éléments de base

1.1 Introduction

Cette présentation des bases algorithmiques de la programmation impérative est accompagnée d'une illustration en langage `C`. Ceci permet de tester très simplement les éléments d'algorithmes. Mais il ne s'agit pas ici d'un cours de `C`.

Un **algorithme** pourrait, en première approximation être défini comme étant un programme écrit dans aucun langage précis, mais pouvant être adapté à tous les langages d'une famille donnée. Ceci demande de définir, puis de respecter, certaines conventions d'écriture. Malheureusement, ce conventions ne font l'objet d'aucune normalisation et pouvent être très variables d'un endroit à l'autre.

1.2 Les données élémentaires

Les langages impératifs, dont les concepts sont très étroitement liés à l'architecture des ordinateurs, manipulent des données dont le **type** est défini au préalable.

Dans un algorithme, toutes les données, ou **variables** utilisées devront faire l'objet d'une **spécification**. Spécifier une variable, c'est donner :

- son **type**
- son **nom**
- de manière optionnelle, sa **valeur**

Les types élémentaires sont : ***entier*** , ***réel*** , ***caractère*** , ***chaîne*** , ***booléen***

1.2.1 Les type numériques

Les types ***entier*** et ***réel*** correspondent à des données compatibles avec les ensembles mathématiques $\mathbb{Z}$ et $\mathbb{R}$ usuels. La nécessité de faire une différence au niveau d'un programme a pour origine le stockage et le traitement de l'information au niveau de la machine. En effet, un ***entier*** est directement codé

en binaire dans la mémoire de l'ordinateur et peut être directement exploité par l'unité arithmétique et logique du processeur. Un ***réel*** , par contre, doit faire l'objet d'un stockage particulier et un calcul sur les ***réels*** fait appel à une unité de traitement spécifique[1].

Exemples de déclarations

entier i ← 5;
réel x, y ← 3.14;

Codage en `C`

```
int i=5;
float x, y=3.14;
```

L'entier i est initialisé à 5, le réel y à 3.14 et le réel x n'est pas initialisé.

1.2.2 Le type *booléen*

Une variable de type ***booléen*** peut prendre deux valeurs : *Vrai* ou *Faux* . La notion de booléen sera détaillée au moment de l'étude des opérateurs relationnels.

1.2.3 Les types *caractère* et *chaîne*

Un ***caractère*** correspond à un chiffre, une lettre, un symbole ou à un code. En réalité, c'est un entier codé sur 8 *bits*[2] conformément au code A.S.C.I.I.[3], pour les codes allant de 0 à 127. Les codes allant de 128 à 255 sont réservés aux besoins spécifiques des pays. Voir l'annexe A (page 46).

Une ***chaîne*** est une suite de ***caractères*** formant un mot ou un message.

Exemples de déclarations

caractère c ← 'U';
chaîne s ← "Tom Bombadil";

Codage en `C`

```
char c = 'U';
char *s = "Tom Bombadil";
```

Remarque : le type ***chaîne*** n'existe pas directement en `C`. Une chaîne du `C`est assimilée à une adresse de caractère ou bien à un tableau de caractères.

1.3 Entrée et sortie standards

Ce sont le clavier et l'écran. Tous les langages proposent une ou plusieurs fonctions permettant d'écrire sur l'écran et de lire les informations à partir du clavier. Dans un algorithme, on utilisera simplement les fonctions abstraites ***écrire*** et ***lire*** .

1. Sur certaines machines, cette unité est séparée du processeur: on parle de coprocesseur arithmétique.
2. binary digit
3. American Standard Code for Information Interchange.

Exemple: le programme suivant lit un entier au clavier, lui ajoute 10 puis affiche le résultat sur l'écran :

```
entier i,j;
chaîne s ← "Le résultat est : "
lire i;
j ← i + 10;
écrire s,j;
```

Traduction C

```
int i,j;
char *s = "Le résultat est : "
scanf("%d", &i);
j = i + 10
printf("%s%d", s, j);
```

1.4 Les constantes

On distingue les constantes

1. de type ***entier*** : 1, 2, 275, -31321, ...
2. de type ***réel*** : 1.0, 3.1415, -9.01, 3×10^{-4} que l'on peut noter également 3e-4 ...
3. de type ***caractère*** : 'a', 'b', ***caractère*** (13) qui donne le caractère de code ASCII 13 ...
4. de type ***chaîne*** : "Bonjour", "1234" ...
5. de type ***booléen*** : *Vrai* , *Faux*

La notion de constante est intuitive. Nous en avons utilisé dans la section précédente.

Remarque : il faut faire la différence entre les constantes 1234, 1234. et "1234".

1234	→	***entier*** 1234
1234.	→	***réel*** 1234.000
"1234"	→	suite des ***caractères*** '1', '2', '3' et '4'.

1.5 Opérateurs et expressions

Une **expression** est une composition de valeurs de types compatibles produisant une nouvelle valeur. Les expressions sont construites à partir des constantes, des variables et des opérateurs.

Les **opérateurs** se répartissent en 4 catégories :

- les opérateurs arithmétiques
- les opérateurs relationnels
- les opérateurs logiques
- les opérateurs fonctionnels

1.5.1 Opérateurs et expressions arithmétiques

Ils sont résumés dans le tableau suivant :

Opérateur	Opération	Type des opérandes	Type du résultat
$\times$	multiplication	***réel*** ou ***entier***	***entier*** si les deux opérandes sont de type ***entier*** ; ***réel*** sinon
/	division	***réel*** ou ***entier***	***réel***
div	division entière	***entier***	***entier***
modulo	reste de la division entière	***entier***	***entier***
+ -	addition soustraction	***entier*** ou ***réel***	***entier*** si les deux opérandes sont de type ***entier*** ; ***réel*** sinon

TAB. 1.1 – *Les opérateurs arithmétiques.*

Remarque : l'opérateur ***modulo*** délivre le reste de la division entière. Il ne doit pas être appliqué avec l'un des opérandes non entier.

Sémantique : Une expression arithmétique indique la manière de calculer des valeurs de type ***entier*** ou ***réel*** à partir d'éléments de base pouvant être :

1. des valeurs courantes des variables
2. des valeurs des constantes
3. des valeurs intermédiaires obtenues par composition des précédentes au moyen d'opérateurs ou de fonctions.

Des règles de priorité sont traduites par la **syntaxe** des expressions. Elles sont basées sur la priorité de traitement des opérateurs. Dans l'ordre décroissant de priorité, on trouve :

1. $\times$ / ***div*** ***modulo***
2. + -

Lors de l'évaluation d'une expression de type arithmétique, les opérations de même priorité sont exécutées de gauche à droite.

$$\text{j } \textbf{\textit{modulo}} \text{ k} \times \text{x} \equiv (\text{j } \textbf{\textit{modulo}} \text{ k}) \times \text{x}$$

Afin de clarifier l'écriture des programmes, il conviendra d'utiliser convenablement les parenthèses.

1.6 Conversions de type

Pour des raisons de commodité, le mélange des types est autorisé dans les expressions arithmétiques, mais il doit être explicite dans la plupart des

cas.

Conversion implicite : la combinaison d'***entiers*** et de ***réels*** dans un calcul entraîne la conversion automatique des ***entiers*** en ***réels*** .

Exemple : l'expression arithmétique 6×1.5 est décomposée de la manière suivante :

$6 \times 1.5 \quad \rightarrow$ expression originale (***entier*** $\times$ ***réel***)
$6.0 \times 1.5 \quad \rightarrow$ l'***entier*** est converti en ***réel*** (***réel*** $\times$ ***réel***)
$9.0 \quad \rightarrow$ le calcul est effectué. (***réel***)

Conversion explicite Dans le tableau suivant, on prend :

caractère c;
entier i;
réel x;

Type initial	**Type final**	**Syntaxe**	**Commentaire**
entier	***réel***	$x =$ ***réel*** (i);	
réel	***entier***	$i =$ ***entier*** (x);	$i \leftarrow$ partie entière de x
caractère	***entier***	$i =$ ***entier*** (c);	$i \leftarrow$ code A.S.C.I.I. de c
entier	***caractère***	$c =$ ***caractère*** (i);	$c \leftarrow$ caractère dont le code A.S.C.I.I. est i

Application : Le programme :

```
entier i ← 85;
caractère c ← 'a';
réel x ← 1.234;
écrire c, i, x, entier (c), caractère (i), entier (x)
```

affichera : a 85 1.234 97 U 1

1.6.1 Expressions booléennes

Les expressions booléennes (*Vrai* ou *Faux*) utilisent en général des opérateurs **relationnels** et/ou **logiques**.

Les opérateurs relationnels

Opérateur	**Opération**	**Type des opérandes**	**Type du résultat**
$= \neq$ $\leq \geq$ $< >$	comparaison entre ***booléens*** , ***caractères*** , ***chaîne*** , ***entiers*** ou ***réels***	***booléens*** , ***caractères*** , ***chaîne*** , ***entiers*** ou ***réels***	***booléen***

TAB. 1.2 – *Les opérateurs relationnels.*

Exemple:

```
entier i,j;
caractère c;
booléen A, B;
...
A ← (i+4) ≤ j;
B ← c < 'E';
```

Les opérateurs logiques

Sémantique: une expression booléenne indique la manière de calculer une valeur de type booléen à partir :

- de relations arithmétiques,

A ***et*** B		B	
		Faux	*Vrai*
A	*Faux*	*Faux*	*Faux*
	Vrai	*Faux*	*Vrai*

A ***ou*** B		B	
		Faux	*Vrai*
A	*Faux*	*Faux*	*Vrai*
	Vrai	*Vrai*	*Vrai*

A	***non*** A
Faux	*Vrai*
Vrai	*Faux*

TAB. 1.3 – *Les opérateurs logiques.*

- de relations entre ***caractères*** ,
- de relations entre ***chaîne*** ,
- des valeurs courantes de variables (de type ***booléen***),
- des valeurs intermédiaires obtenues par composition des précédentes au moyen d'opérateurs ou de fonctions.

Le type d'un expression booléenne est ***booléen*** . La syntaxe des expressions booléennes traduit les règles de priorité entre les opérateurs. Dans l'ordre décroissant de priorité, on a :

1. ***non***
2. $\times$ / ***modulo div et***
3. $+$ - ***ou***
4. $= \neq < > \leq \geq$

Le tableau des priorités comporte des opérateurs arithmétiques et de relation car ils peuvent intervenir dans le calcul de la valeur d'une expression booléenne.

Lors de l'évaluation d'une expression booléenne, les opérations de même priorité sont exécutées de gauche à droite.

1.6.2 Les opérateurs fonctionnels

La plupart des langages proposent un ensemble de fonctions générales. Dans un algorithme nous utiliserons communément les fonctions décrites dans la table 1.4

Désignateur	**Description**	**Type du paramètre**	**Type du résultat**
`abs`(x)	$\lvert x \rvert$	***entier*** ou ***réel***	celui du paramètre
$\arccos(x)$	arc cosinus	***entier*** ou ***réel***	***réel***
$\arcsin(x)$	arc sinus	***entier*** ou ***réel***	***réel***
$\arctan(x)$	arc tangente	***entier*** ou ***réel***	***réel***
$\cos(x)$	cosinus	***entier*** ou ***réel***	***réel***
$\exp(x)$	e^x	***entier*** ou ***réel***	***réel***
$\ln(x)$	logarithme népérien	***entier*** ou ***réel***	***réel***
$\log(x)$	logarithme décimal	***entier*** ou ***réel***	***réel***
`sqr`(x)	x^2	***entier*** ou ***réel***	celui du paramètre
`sqrt`(x)	$\sqrt{x}$	***entier*** ou ***réel***	celui du paramètre
$\tan(x)$	tangente	***entier*** ou ***réel***	***réel***

TAB. 1.4 – *Les fonctions.*

Remarque : selon le langage, certaines de ces fonctions peuvent porter des noms différents, voire ne pas être implémentées.

Exemple : le programme suivant lit deux réels a et b, puis affiche

$$\tan\left(\frac{a}{\sqrt{a^2+b^2}}\right)$$

réel a,b,r;
écrire "a = "; ***lire*** a;
écrire "b = "; ***lire*** b;
r $\leftarrow$ tan(a/`sqrt`(`sqr`(a) + `sqr`(b)))
écrire "résultat = ", r

Traduction `C`

```
double a,b,r;
printf("a ="); scanf("%lf",&a);
printf("b ="); scanf("%lf",&b);
r = tan (a / sqrt( a*a + b*b))
printf("Résultat = %lf",r);
```

Cet exemple montre que l'implémentation des opérateurs fonctionnels dans un langage peut être différente de l'écriture algorithmique. Lorsqu'un nouveau langage est abordé, il convient de s'informer de la liste et des noms associés aux différents opérateurs fonctionnels existants.

1.7 Exercices sur le chapitre 1

Pour chacun des exercices, on considérera les déclarations suivantes :
caractère c, d;
entier i, j, k;
réel x, y;
c $\leftarrow$ 'U'; d = ***caractère*** (13);
i $\leftarrow$ 10; j $\leftarrow$ 4; k $\leftarrow$ -4;
x $\leftarrow$ 1.5, y $\leftarrow$ -3e1;

Exercice 4

Donner le résultat de l'évaluation de chacune des expressions suivantes, ainsi que le type du résultat :

5a. (i ***modulo*** 3) $\times$ x;

5b. 'U' + ***caractère*** (32);

5c. (i $\times$ 5 ***div*** 3) ***modulo*** 7;

5d. ***entier*** (x $\times$ 10) ***div*** 2;

Exercice 5

Donner le résultat de l'évaluation de chacune des expressions booléennes suivantes :

6a. (c $\leq$ 'V') ***et*** (i $>$ 23)

6b. ***non*** (c = d) ***ou*** (x $\times$ y $<$-2000)

6c. (y / x / 2 $\neq$ -10) ***ou non*** (i ***modulo*** k $\geq$ 2)

Chapitre 2

Les structures de commande

Les **structures de commande** (*control structure*) ont pour objectif de **structurer** les programmes. En utilisant les constructions standardisées, on améliore la lisibilité des programmes et on facilite l'étude rigoureuse de leurs **propriétés**.

Le choix des structures est devenu universel. On distingue, parmi les constituants de base :

† l'**enchaînement séquentiel**

† la **répétition**

† le **choix**

† l'appel de **sous programmes**

2.1 L'enchaînement séquentiel

L'enchaînement séquentiel correspond à l'exécution à la suite les une des autres (en séquence) de plusieurs instructions.

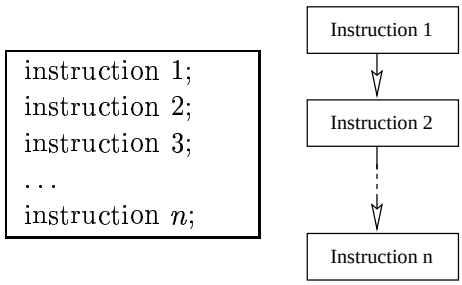


FIG. 2.1 – *Enchaînement séquentiel.*

Les instructions sont exécutées dans l'ordre où elles sont écrites.

Exemple :

```
entier a ← 1, b ← 2, e;
e ← a; a ← b; b ← e;
écrire "a = ", a, "b = ", b;
```

Traduction C

```
int a=1, b=2, e;
e = a; a = b; b = e;
printf("a = %d, b = %d\n", a, b);
```

La deuxième ligne est une séquence qui réalise la permutation des valeurs des variables a et b.

Bloc d'instructions. Un bloc d'instructions est un enchaînement séquentiel doté de déclarations de variables. Les variables déclarées à l'intérieur d'un bloc ne sont visibles que dans ce bloc.

Dans le programme précédent, on voit que la variable entière e n'a d'intérêt que pour l'échange. Elle peut donc être déclarée dans un bloc d'instructions qui ne comporte que les instructions de l'échange :

```
entier a=1, b=2;
début
    entier e;
    e = a; a = b; b = e;
fin
écrire "a = ", a, "b = ", b;
```

Traduction C

```
int a=1, b=2;
{
  int e;
  e = a; a = b; b = e;
}
printf("a = %d, b = %d\n", a, b);
```

Remarques:

1 Dans la dernière ligne, on ne pourrait pas afficher la valeur de e. En effet, la variable e nest plus visible au moment de l'affichage.

2 La notion de bloc est très importante dans des langages comme `Pascal`, `C` ou `Java`. En `Basic`, le fait que les variables n'ont pas besoins d'être déclarées rendent cette notion superflue.

3 Un bloc d'instructions peut contenir d'autres blocs d'instructions. Il n'y a pas de limite en degré d'imbrication des blocs. Une variable est utilisable dans tous les blocs internes au bloc ou elle est déclarée.

2.2 La répétition

La **répétition** (ou **boucle** ou **itération**) correspond à l'exécution répétée d'une suite d'instructions. Cette répétition est finie ou infinie. On trouve des répétitions inifinies en contrôle de processus industriels (on parle alors de tâches). Plus couramment, il y a arrêt après un nombre fini de répétitions.

2.2.1 Boucle *tant que* ... *répéter*

Principe

La répétition se poursuit tant qu'une condition (de continuation) demeure vraie. Si la condition est fausse dès l'origine, les instructions ne sont

pas exécutées.

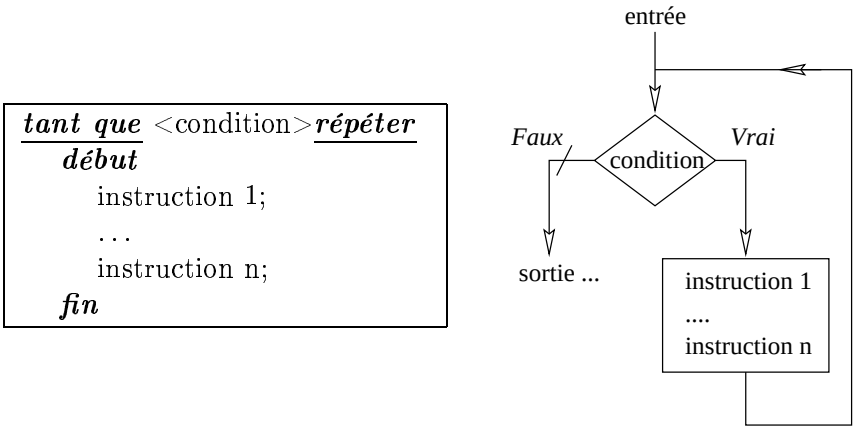


FIG. 2.2 – *Boucle* ***tant que*** .

Pour sortir de la boucle, il est nécessaire que l'une des instructions au moins vienne faire évoluer l'évaluation de la condition.

Exemple

On désire concevoir un programme qui lit des entiers strictement positifs entrés au clavier et qui en fait la somme. Le processus s'arrête lorsqu'un entier négatif est entré par l'utilisateur.

On peut réaliser ce programme avec la boucle ***tant que*** . En effet, intuitivement, la condition de continuation de la répétition est :

«***tant que*** l'entier entré est strictement positif»

Un algorithme possible est alors :

```
entier n, S;
lire n;
S ← 0;
tant que n >0 répéter
   début
      S ← S + n;
      lire n;
   fin
écrire "La somme est :", S
```

Remarques

† La variable n testée dans la condition doit être initialisée.

† Dans la boucle, la variable S est réutilisée. Elle intègre la somme des valeurs successives de n. Dans la partie droite de l'affectation, c'est l'ancienne valeur de S. À gauche, c'est la nouvelle valeur.

Application

Voir les exercices en fin de chapitre (page 40).

2.2.2 Boucle *répéter* ... *tant que*

Principe

C'est une variante de la boucle précédente, mais cette fois les instructions sont exécutées au moins une fois.

```
répéter
   début
      instruction 1;
      ...
      instruction n;
   fin
tant que <condition>
```

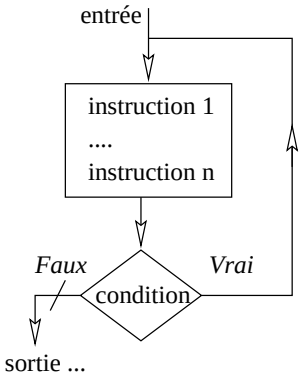


FIG. 2.3 – *Boucle* ***répéter*** ... ***tant que*** .

Exemple

On souhaite qu'un utilisateur entre un code à quatre chiffres compris entre 1000 et 9999. Un code erroné est refusé. Il s'agit donc de lire des valeurs jusqu'à ce que l'une d'entre elles répond au critère d'acceptation.

```
entier code;
répéter
   début
      lire code;
tant que code <1000 ou code >9999
écrire "Code acepté";
```

2.2.3 La boucle *pour* avec compteur

Principe

Lorsque le nombre d'itérations est connu avant la boucle, on préfère souvent une boucle avec compteur à la boucle ***tant que*** qui est plus générale.

```
pour <indice> de <valeur initiale> à <valeur finale> par pas de <pas>
   répéter
      début
         instruction 1;
         ...
         instruction n;
      fin
```

FIG. 2.4 – *Boucle* ***pour*** .

Remarque : la boucle ***pour*** peut être traduite par une boucle ***tant que*** :

```
<indice>← <valeur initiale>
tant que <indice>≤ <valeur finale> répéter
   début
      <Instruction_1>
      ...
      <Instruction_n>
      <indice>← <indice>+ <pas>
   fin
```

Exemple

L'affichage des carrés des 100 premiers entiers est réalisé par l'algorithme suivant :

```
entier i,n,carré;
pour i de 1 à 100 par pas de 1 répéter
      début
         carré ← i × i;
         écrire carré;
      fin
```

2.3 Le choix

2.3.1 Choix simple

Le choix correspond à un aiguillage du de l'exécution du programme vers différents traitements en fonction d'une ou plusieurs conditions. Le choix le plus simple est l'**alternative** :

si <Condition> ***alors*** <Instruction$_1$>
sinon <Instruction$_2$>

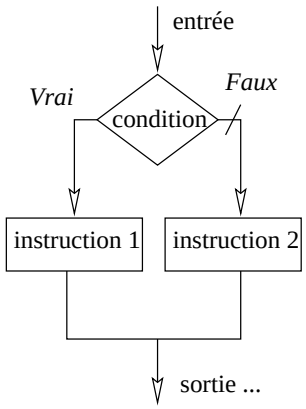


FIG. 2.5 – *Choix simple (alternative).*

<Instruction$_1$> et <Instruction$_2$> peuvent être des instructions simples des blocs d'instructions ou d'autres structures de commande. Ainsi, plusieurs structures alternatives peuvent être imbriquées :

si <Condition$_1$> ***alors*** <Instruction$_1$>
sinon
  si <Condition$_2$> ***alors*** <Instruction$_2$>
  sinon <Instruction$_3$>

Exemple

On désire tester la parité d'un nombre entier lu au clavier :

```
entier n;
lire n;
si n modulo 2 = 0 alors
    écrire "Entier pair";
sinon "Entier impair";
```

2.3.2 Choix multiple

Principe

Pour éclaircir le programme, dans le cas d'un grand nombre d'imbrications d'alternatives, le langages de programmation proposent en général une

forme plus complète dite **choix multiple**. En général les conditions traitées sont exclusives (deux d'entre elles ne peuvent être vraies simultanément).

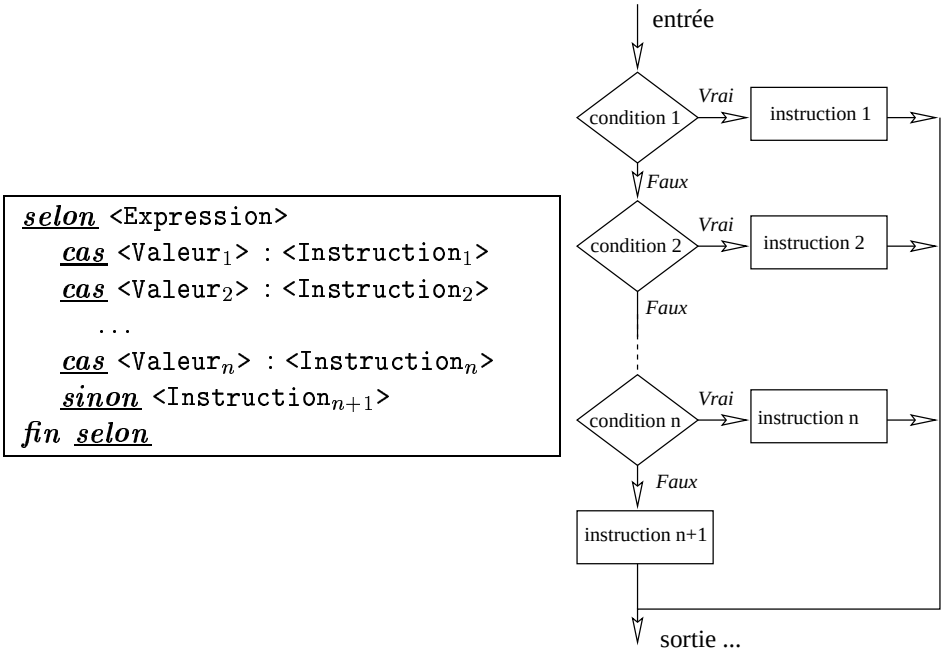


FIG. 2.6 – *Choix multiple. Dans la figure de droite , <Condition$_n$> est équivalente à « <Expression> = <Valeur$_n$> »*

Exemple

Pour gérer un menu interactif, des actions différentes sont menées selon le choix entré par l'utilisateur :

```
entier choix;
écrire "Entrer un nombre entre 1 et 4:";
lire choix;
selon choix
   cas 1: écrire "Action 1 engagée ...";
   cas 2: écrire "Action 2 engagée ...";
   cas 3: écrire "Action 3 engagée ...";
   cas 4: écrire "Action 4 engagée ...";
   sinon écrire "Mauvais choix!";
fin selon
```

2.4 Exercices sur le chapitre 2

Exercice 6

On lâche une balle d'une hauteur h. Elle rebondit sur place et chaque rebond a pour hauteur les $\frac{5}{7}^{\text{ème}}$ de la hauteur du rebond précédent. Elle s'arrête lorsque la hauteur devient inférieure à 1 mm.

6a. Écrire un algorithme qui lit au clavier la hauteur initiale, puis qui calcule la distance parcourue par la balle.

6b. Coder l'algorithme en `C`.

6c. Modifier l'algorithme puis compléter le programme pour qu'il affiche en plus le nombre de rebonds.

Exercice 7: valeur approchée de $\sin x$

On désire calculer la valeur approchée de $\sin x$, x étant exprimé en radians. On part pour cela la formule suivante :

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \times \frac{x^{2k+1}}{(2k+1)!}$$

que l'on utilisera au rang n:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Remarques :

† Pour cette dernière expression, on dit que $\sin x$ est calculé au rang n.

† La variable k indique un compteur qui va de 0 à n.

7a. Écrire un algorithme qui lit un entier n au clavier, qui calcule $\sin x$ au rang n et enfin qui affiche le résultat.

7b. Écrire un algorithme pour lequel le calcul de $\sin x$ s'arrête dès que le terme de rang k calculé devient inférieur à une valeur ϵ entrée au clavier.

Exercice 8: la multiplication grecque

Il s'agit de multiplier deux entiers positifs en n'effectuant que des multiplications et des divisions par 2. Dans l'exemple qui suit, on souhaite effectuer la multiplication $x \times y = 31 \times 43$. On construit alors un tableau à deux colonnes :

- la colonne de droite contient les divisions successives par 2 de y;
- la colonne de gauche contient les multiplications successives par 2 de x.

Le processus s'arrête lorsque y n'est plus divisible par 2.

À chaque fois que le nouvel y calculé est impair, on souligne le x correspondant. La somme des x soulignés donne le produit recherché.

x	y
31	**43**
62	**21**
124	10
248	**5**
496	2
992	**1**

$$31+62+248+992 = 1333 = 31\times 43$$

8a. Écrire un algorithme qui multiplie deux entiers selon ce principe.

8b. Coder l'algorithme dans un langage impératif.

Exercice 9: gestion d'un menu

Écrire un programme qui affiche à l'écran un menu à 4 choix selon le modèle suivant :

```
Menu :
   1 - premier choix
   2 - second choix
   3 - troisième choix
   4 - quitter
      Choix :
```

La commande réalisée pour les choix 1, 2 et 3 seront simplement l'affichage d'un message qui confirme le choix. Le programme bouclera tant que le choix 4 n'aura pas été sélectionné.

9a. Écrire l'algorithme de cette application sur la base d'une boucle ***tant que*** .

9b. Même question avec une boucle ***répéter*** ... ***tant que*** .

9c. Comparer les deux méthodes.

Exercice 10: boucle *pour*

En utilisant une boucle ***pour*** , écrire un algorithme qui calcule la somme des n premiers entiers.

Exercice 11

Écrire l'algorithme d'un programme qui lit un entier n, un réel x puis qui affiche x^n.

Exercice 12

Écrire l'algorithme d'un programme qui prend en entrée deux entiers positifs n et p, puis qui affiche la somme

$$S = n^0 + n^1 + n^2 + n^3 + \cdots + n^p$$

Exercice 13

Écrire un algorithme qui lit un entier n, un réel x puis qui calcule la somme suivante :

$$S = x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots + \frac{x^n}{n}$$

Le résutat sera affiché à l'écran.

Exercice 14

Pour un examen final, on dispose, pour chaque étudiant :

- de sa note au premier partiel $P1$,
- de sa note au second partiel $P2$,
- de sa note à l'examen final Ex.

Le canditat est déclaré admis si $max(\frac{P1+P2+Ex}{3}, Ex) \geq 10$

Écrire un algorithme qui lit les trois notes du candidat et qui affiche l'un des deux messages «Reçu» ou «Collé».

Exercice 15

Un marchand de vin expédie une quantité Q de bouteilles de prix unitaire P. Si le total de la commande est d'au moins 1000 F, le port est gratuit. Sinon, il est facturé 10% de la commande avec un minimum de 30 F. Écrire un algorithme qui prend en entrée P, Q et le nom du client et qui affiche en sortie une facture au nom de ce client.

Exercice 16

La SNCF a mis en place un système de réduction lié à l'âge et au nombre de membres de la famille.

50 %	pour toute personne	de plus de 60 ans
40 %	.....	de moins de 10 ans
30 %	.....	d'une famille de 5 membres
50 %	.....	d'une famille de plus de 5 membres
20 %	.....	de 11 à 25 ans

Si une personne entre dans deux catégories, c'est la réduction la plus importante qui est prise en compte.

Écrire un algorithme qui calcule la réduction pour un personne dont on connaît l'âge A et le nombre N de membres de sa famille.

Exercice 17

Connaissant les heures de départ et d'arrivée d'une *baby-sitter*, écrire un programme qui donne son salaire journalier selon le barème suivant :

† 40 F par heure avant 18 h.

† 50 F par heure après 18 h.

On suppose que les horaires ne dépassent pas minuit.

Chapitre 3

Les sous-programmes

3.1 Définition d'un sous-programme

Un sous-programme est un élément de programme **nommé** et éventuellement **paramétré**, que l'on définit afin de pouvoir ensuite l'appeler par son nom en affectant, s'il y a lieu, des valeurs aux paramètres. C'est un concept qui existe depuis le début de la programmation. Les intérêts de l'utilisation des sous-programmes sont :

- le gain de place en mémoire pour le code du programme : lorsqu'un sous-programme est appelé plusieurs fois, dans une boucle, par exemple,
- les notions d'abstraction et de modularité : utiliser un appel de sous-programme permet de d'écrire une application en faisant abstraction (en dissimulant) les détails de la fonction que réalise ce sous-programme .

Quelques règles élémentaires doivent être appliquées pour l'écriture de sous-programmes .

- Un sous-programme doit être **homogène** : il doit réaliser une tâche précise, formant un tout.
- Il doit être de **taille «raisonnable»**. La compréhension et la gestion du programme en dépend.

La **définition** du sous-programme est composée d'une **spécification** qui indique son nom, ses paramètres (ou arguments) avec leurs caractéristiques (nom, type) et d'un **corps** comprenant éventuellement des déclarations d'objets locaux au sous-programme et les instructions à exécuter.

Déclaration algorithmique d'un sous-programme

fonction `NomDeFonction`$(para_1 : type_1, \cdots, para_n : type_n) : type_{retour}$;

Le sous-programme de identifié sous le nom «`NomDeFonction`» reçoit les arguments $para_1, \cdots, para_n$ de types respectifs $type_1, \cdots, type_n$ et produit en retour une variable de type $type_{retour}$.

Déclaration d'un sous-programme (ou fonction) en C

```
type_retour NomDeFonction(type_1 para_1, ..., type_n para_n)
```

Lorsque la fonction ne renvoie rien (c'est une procédure), `type_retour` prend le type «vide» `void`.

Le Cutilise une syntaxe particulière ou le type de la valeur retournée est placé en tête de la déclaration [1].

Exemple:

fonction Max(a,b : ***entier***) : ***entier*** ;
début
  si a>b ***retourner*** a;
  sinon retourner b;
fin Max;
	`int Max(int a, int b) {`
	`  if (a>b) return a;`
	`  else return b;`
	`}`

Dans cet exemple, la ligne algorithmique :

fonction Max(a,b : ***entier***) : ***entier*** ;

traduite en Cpar

```
int Max(int a, int b)
```

est la **déclaration** (ou **spécification**) d'une fonction dont le nom est *Max*, qui reçoit deux paramètres ***entier*** *a* et *b*, et qui produit en sortie un autre ***entier*** .

a et *b* sont appelés les **paramètres formels** de la fonction *Max*. Ils ne servent qu'à l'intérieur du corps du sous-programme . Le corps du sous-programme est défini dans l'algorithme par les instructions contenues entre «***début*** » et «***fin*** Max;». En C, les instructions du sous-programme (ou **fonction**) font l'objet d'un bloc d'instructions.

3.2 Appel d'un sous-programme

L'appel d'un sous-programme se fait en mentionnant son nom, suivi des paramètres **effectifs** figurant entre parenthèses et séparés par des virgules. L'appel d'une fonction peut être une expression à part entière ou un opérande dans une expression plus complexe. Chaque paramètre effectif doit correspondre à un paramètre formel.

Exemple :

```
int k,x,y;
scanf("%d%d",&x,&y);
k = Max(x,y);
```

1. Les langages `pascal`, `modula2`, `Ada` sont plus proches de la notation algorithmique, mais les langages `C++` et `java` sont inspirés du `C`

3.3 Passage de paramètres par valeur ou par référence

Il existe deux mécanismes importants de substitution entre paramètres effectifs et paramètres formels:

- **le passage par valeur**: la valeur du paramètre efectif est recopîée dans le paramètre formel à l'entrée du sous-programme . Le sous-programme travaille sur une copie du paramètre effectif. Celui-ci n'est pas modifié à l'issue du sous-programme .
- **le passage par référence** (ou **par adresse**) : l'adresse du paramètre effectif est communiquée au sous-programme qui travaille alors directement avec lui et non sur une copie locale. Le sous-programme peut alors modifier la valeur d'une variable du programme qui l'a appelé.

Ces mécanismes peuvent conduire à des résultats différents lors d'une exécution.

En `C`, les paramètres sont passés par valeur. Pour réaliser un passage par référence, il faut explicitement passer l'adresse de la variable au sous-programme .

Exemples

† Dans le paragraphe 3.1, `a` et `b` sont les paramètres formels de la fonction `Max`. Il s'agit d'un passage par valeur.

† Dans l'instruction `scanf("%d%d",&x,&y);`, les paramètres `x` et `y` sont passés par référence. En réalité, les paramètres vraiment passés à la fonction `scanf` sont `&x` et `&y` qui sont les adresses physiques des variables `x` et `y`. Il faut que la fonction `scanf` initialise les variables `x` et `y`, donc, qu'elle les modifie.

Annexe A

Codes ASCII

Déc	Hex	Clav	Car
0	00	^@	NUL
1	01	^A	SOH
2	02	^B	STX
3	03	^C	ETX
4	04	^D	EOT
5	05	^E	ENQ
6	06	^F	ACK
7	07	^G	BEL
8	08	^H	BS
9	09	^I	HT
10	0a	^J	LF
11	0b	^K	VT
12	0c	^L	FF
13	0d	^M	CR
14	0e	^N	SO
15	0f	^O	SI
16	10	^P	DLE
17	11	^Q	DC1
18	12	^R	DC2
19	13	^S	DC3
20	14	^T	DC4
21	15	^U	NAK
22	16	^V	SYN
23	17	^W	ETB
24	18	^X	CAN
25	19	^Y	EM
26	1a	^Z	SUB
27	1b	^[	ESC
28	1c	^\	FS
29	1d	^]	GS
30	1e	^~	RS
31	1f	^_	US
32	20		SP
33	21		!
34	22		"
35	23		#
36	24		$
37	25		%
38	26		&
39	27		'
40	28		(
41	29		)
42	2a		*

Déc	Hex	Car
43	2b	+
44	2c	,
45	2d	-
46	2e	.
47	2f	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3a	:
59	3b	;
60	3c	<
61	3d	=
62	3e	>
63	3f	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4a	J
75	4b	K
76	4c	L
77	4d	M
78	4e	N
79	4f	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U

Déc	Hex	Car
86	56	V
87	57	W
88	58	X
89	59	Y
90	5a	Z
91	5b	[
92	5c	\
93	5d	]
94	5e	^
95	5f	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6a	j
107	6b	k
108	6c	l
109	6d	m
110	6e	n
111	6f	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7a	z
123	7b	{
124	7c	:
125	7d	}
126	7e	~
127	7f	DEL

Bibliographie

[1] J. Boébion. *Bases d'informatique et programmation.* 1988.
[2] Borland. *Turbo Prolog 2.0 User's Guide*, 1988.
[3] P. Lignelet. *Structures de données avec ADA.* 1990.
[4] J. Longchamp. *Les langages de programmation.* 1989.
[5] Michel Mauny. *Funtional programming using Caml Light.* january 1995.
[6] B.W. Kerninghan & D.M. Ritchie. *Le langage C.* 1987.

Index