Chapitre 1 - Introduction

Programme

```
Programme bidule
Const constantes {Déclaration des constantes}
Type types {Déclaration des types}
Var variables {Déclarations des variables globales}
      Procédures et fonctions
Début
Programme principal
Fin
```

Lecture et écriture

- instruction `lire`

`lire(X)` mettre dans la case X, la valeur présente sur l'organe d'entrée de la machine

- instruction `écrire`

`écrire(X)` mettre sur l'organe de sortie de la machine, le contenu de la case X

Alternatives

A. Si alors

```
Si cond
      Alors inst
FinSi
```

B. Si alors sinon

```
Si cond
      Alors inst1
      Sinon inst2
FinSi
```

C. Selon

Cette instruction permet à la machine de faire un choix parmi n possibles, n étant plus grand que deux.

```
Selon exp
      val1 : inst1
      val2 : inst2
      ...
      valn : instn
FinSelon
```

Chapitre 2 – Les Itérations

Répéter

```
Répéter
      inst
Jusqu'à cond
```

Tantque

```
TantQue cond Faire
      inst
FinTantQue
```

Pour

```
Pour var←valinit Jusqu'à valfin Incr op Faire
      inst
FinPour
```

Chapitre 3 – Les Types scalaires

Le type entier

Il est muni des opérations suivantes :

+ - * div mod

Le type réel

Il est muni des opérations suivantes :

+ - * /

Opérations sur les propositions booléennes

Dans l'ensemble des deux propositions V et F, on définit des opérations dites opérations logiques dont les plus importantes sont :

conjonction	∧
disjonction inclusive	∨
négation	¬
implication	⇒
équivalence	⇔
disjonction exclusive	w

p	q	p∧q	p∨q	p⇒q	p⇔q	pwq
F	F	F	F	V	V	F
F	V	F	V	V	F	V
V	F	F	V	F	F	V
V	V	V	V	V	V	F

$p \Rightarrow q = \neg p \vee q$ $p \Leftrightarrow q = (\neg p \vee q) \wedge (\neg q \vee p)$

$\neg(p \vee q) = \neg p \wedge \neg q$ $\neg(p \wedge q) = \neg p \vee \neg q$

$pwq = \neg(p \Leftrightarrow q) = \neg\ [(\neg p \vee q) \wedge (\neg q \vee p)] = \neg(\neg p \vee q) \vee \neg(\neg q \vee p) = (p \wedge \neg q) \vee (q \wedge \neg p)$

Le type booléen

Il est constitué de deux éléments : vrai, faux notés V et F.

Il est muni des opérations : ∧, ∨, ¬

Il est totalement ordonné par la relation : F<V

Le type caractère

Le type caractère est l'ensemble des caractères d'imprimerie habituelle. Il est noté `car`.

Une variable de type caractère peut prendre toute valeur de ce type. Une constante de type `car` se distingue par son écriture 'O', 'A', 'o', 'a', '+'.

Un ordinateur ne pouvant manipuler que des éléments binaires, il est nécessaire de codifier les caractères, cad de faire correspondre à chacun d'eux une configuration binaire. Un des codes les plus utilisés est l'ASCII.

Les fonctions `ord` (pour obtenir le code ASCII du caractère donné), `chr` (pour obtenir le caractère du code ASCII donné), `prec` (caractère précédent) et `succ` (caractère suivant) sont définies.

Type défini par énumération

```
Type saison : (prin, été, aut, hiver)
```

On a la relation prin < été < aut < hiver

Type intervalle

`0 .. 10`	pour	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
`'a' .. 'f'`	pour	{'a', 'b', 'c', 'd', 'e', 'f'}
`été .. hiv`	pour	{été, aut, hiver}

Ordres de priorité

1 - `Non`

2 - Opérateurs multiplicatifs : `* / div mod et`

3 - Opérateurs additifs : `+ – ou`

4 - Opérateurs relationnels : `= ≠ < ≤ > ≥ dans`

Chapitre 4 – Les Séquences

Type abstrait de données d'une séquence
Sorte : Seq
Utilise : Entier, Elément
Opérations :
CréerSeq(Entier, Entier) → Seq
ChangerIème(Seq, Entier, Elément) → Seq
BorneInf(Seq) → Entier
BorneSup(Seq) → Entier
Ième(Seq, Entier) → Elément

Implémentation d'une séquence
```
Const max = 30
Type tab-entier : tableau [1..max] d'entier
Var t : tab-entier
```

CréerSeq(i,j) : fait par la déclaration du type et de la variable ;
par contre les initialisations restent à faire
BorneInf(t) : 1
BorneSup(t) : 30
Ième(t,i) : t[i]
ChangerIème(t,i,e) : t[i] ← e

Type de données abstrait d'une séquence doublement indicée
Sorte : SeqDouble
Utilise : Entier, Elément
Opérations :
CréerSeq(Entier, Entier, Entier, Entier) → SeqDouble
ChangerIème(SeqDouble, Entier, Entier, Elément) → SeqDouble
BorneInf(SeqDouble, Entier) → Entier
BorneSup(SeqDouble, Entier) → Entier
Ième(SeqDouble, Entier, Entier) → Elément

Implémentation d'une séquence doublement indicée par tableau à deux dimensions
```
Const maxi = 30
      maxj = 50
Type mat-entier : tableau [1..maxi, 1..maxj] d'entier
Var m : mat-entier
```

CréerSeq(i,j,k,l) : fait par la déclaration du type et de la variable ;
par contre les initialisations restent à faire
BorneInf(m, 1) : 1
BorneSup(m, 1) : 30
BorneInf(m, 2) : 1
BorneSup(m, 2) : 50
Ième(m,i,j) : m[i,j]
ChangerIème(m,i,j,e) : m[i,j] ← e

Chapitre 5 – Les procédures et les fonctions

Les procédures
```
Procédure nom-proc (E variables avec types ; E/S variables avec types ;
                    S variables avec types)
```
- E : paramètre de donnée, ne subit aucune altération au cours de l'exécution de la procédure (pas d'affectation de ce paramètre dans la procédure).
- S : paramètre résultat de l'action, sa valeur n'est pas significative avant l'exécution de la procédure (initialisation obligatoire de ce paramètre dans la procédure).
- E/S : paramètre de donnée-résultat, sert de donnée et de résultat.

Les fonctions
- procédure : instruction
- fonction : expression

Dans une fonction, les paramètres formels sont par définition des paramètres données.
```
Fonction nom-fonc (paramètres entrées avec types) : type du param. résultat
```

Visibilité des objets (constantes, types, variables, paramètres formels)
Pour un module (procédure ou fonction)
- ses *objets locaux* sont définis dans le module, ils ne peuvent pas être utilisés en dehors,
- ses *objets globaux* sont définis en dehors du module.

Des objets globaux ne peuvent pas porter le même nom. Les objets locaux à un module ne peuvent pas porter le même nom. Par contre des objets locaux à des modules différents peuvent porter le même nom. Enfin, un objet local à un module peut porter le même nom qu'un objet global. Dans le module, ce nom désigne alors l'objet local tandis que qu'à l'extérieur du module, ce nom désigne l'objet global.

Chapitre 6 – Les chaînes de caractères

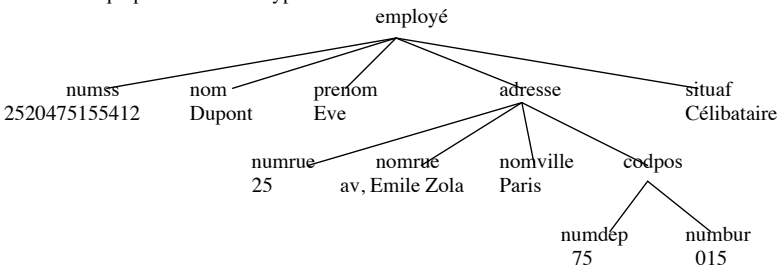
Soient A et B deux chaînes : on dit que A≤B ssi
soit l(A)≤l(B) et∀i 1≤i≤l(A) Ai=Bi
soit ∃i 1≤i≤min(l(A),l(B)) tq Ai<Bi et ∀j 1≤j<i Aj=Bj
La relation d'ordre Ai≤Bi dans l'ensemble des lettres est celle de l'ordre alphabétique habituel.
L'ordre lexicographique étendu s'étend à l'ensemble des chaînes construites sur l'ensemble des caractères du code ASCII. Il suffit de prendre pour définition de la relation Ai≤Bi
Ai≤Bi⇔ ORD(Ai)≤ORD(Bi)

Le TAD chaîne
Sorte : Chaîne
Utilise : Entier, Caractère, Booléen

Opérations :
CréerChaîne() → Chaîne
ChangerIème(Chaîne, Entier, Caractère) → Chaîne
Concaténer(Chaîne, Chaîne) → Chaîne
Copier(Chaîne, Entier, Entier) → Chaîne
Effacer(Chaîne, Entier, Entier) → Chaîne
Insérer(Chaîne, Chaîne, Entier) → Chaîne
Ième(Chaîne, Entier) → Caractère
Egal(Chaîne, Chaîne) → Booléen
Lg(Chaîne) → Entier
Pos(Chaîne, Chaîne) → Entier

Chapitre 7 – Les enregistrements

Un enregistrement est une structure de donnée hiérarchisée. On peut la voir comme une cellule composée d'éléments qui peuvent être de types différents.




```
Type   code-postal = Enregistrement
                        numdep, numbur : chaine de caract.
                     FinEnregistrement
       adresse-pos = Enregistrement
                        numrue, nomrue, nomville : chaine de caract
                        codpos : code-postal
                     FinEnregistrement
       employé = Enregistrement
                   numss, nom, prénom, situaf : chaine de caract
                   adresse : adresse-pos
                 FinEnregistrement

Var e : employe
```

Numéro du département où habite l'employé e : `e.adresse.codpos.numdep`

Chapitre 8 – Les fichiers

Type abstrait de données FAS

Sorte : FAS
Utilise : Chaîne, Elément
Opérations :
- CréerFichier(Chaîne) → FAS
- OuvrirEnEcriture(Chaîne) → FAS
- OuvrirEnLecture(Chaîne) → FAS
- Fermer(FAS) → FAS
- Lire(FAS) → FAS, Elément
- Ecrire(FAS, Elément) → FAS
- FinFichier(FAS) → Booléen {observateur}

Type abstrait de données FT

Sorte: FT sous-type de FAS[Chaine]
Utilise : Chaîne
Opérations :
- LireChaîne(FT) → FT, Chaîne
- EcrireChaîne(FT, Chaîne) → FT

Type abstrait de données FAD

Sorte: FAD sous-type de FAS
Utilise : Chaîne, Entier
Opérations :
- OuvrirEnLectureEcriture(Chaîne) → FAD
- AllerEn(FAD, Entier) → FAD

Chapitre 9 – La récursivité

Schéma général d'un algorithme récursif

```
Module A(p1,..,pn)
Début
      Si cond
      Alors I1...
            Sinon I2
                  A(f(p1,..,pn))
                  I3
                  A(g(p1,..,pn))
                  ...
                  Im
      FinSi
Fin
```

Un algorithme est dit *simplement récursif* s'il ne contient qu'un seul appel récursif.
Un appel récursif est *terminal* s'il n'est jamais suivi par l'exécution d'instructions dans le module.
On dit qu'on a une *récursivité indirecte* ou *croisée* quand un module A sans appel récursif appelle un module B qui appelle A.

Chapitre 10 – Notion de complexité

Notation *O* et *Ω*
Pour une complexité algorithmique en *O(f(n))*, il existe deux constantes positives c et n_0 :
$\forall\ n \geq n_0$, $T(n) \leq cf(n)$
Dire que T(n) est en O(f(n)), c'est garantir que f(n) est un majorant de la fonction de complexité T(n).
g(n) est un minorant de T(n) : T(n) est en $\Omega(g(n))$, $\exists$ c (constante positive) t.q. $T(n) \geq cg(n)$.

Les programmes peuvent être comparés sur la base de leur fonction de complexité, à la constante de proportionnalité près.
Règle de la somme :
Supposons que deux modules P_1 et P_2 aient une complexité $T_1(n)$ en O(f(n)) et $T_2(n)$ en O(g(n)). Alors la complexité de P_1 suivi de P_2 est $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$.

Analyse des programmes
Le seul paramètre acceptable pour l'évaluation de la complexité d'un programme est *n*, la taille des données.
- La complexité de toute affectation, opération de lecture ou d'écriture peut en général se mesurer par O(1).
- La complexité d'une suite d'instructions est déterminée par la règle de la somme.
- La complexité conditionnelle (si) est celle des instructions exécutées (règle de la somme pour un si-alors-sinon), plus celle de l'évaluation de la condition (généralement égale à O(1)).
- La complexité d'une boucle est la somme cumulée, sur toutes les itérations, de la complexité des instructions exécutées dans le corps de la boucle plus le temps consacré à l'évaluation de la condition de sortie de la boucle (généralement égale à O(1)). Assez souvent, cette complexité est le produit du nombre d'itérations par la plus grande complexité rencontrée dans l'exécution d'une boucle (règle du produit).

- Appels de procédures non récursives : on commence par celles qui n'en appellent aucune autre, puis celles qui appellent uniquement celles qu'on vient de traiter en incluant les complexités qu'on vient de calculer, et ainsi de suite.
- Appels de procédures récursives : il faut construire une relation de récurrence pour T(n).

Chapitres 11 - Algorithmes de tri

Tri fusion (mergesort)

```
Const inf = 1
      sup = 100
Type tab = tableau [inf,sup] d'entier

Procédure trier-fusion (E/S t : tab ; E inf,sup : entier) {inf ≤ sup}
Var m : entier
Début
  si inf=sup
    alors écrire(''le tableau est trié'')
    sinon
      m←(inf+sup) div 2
      trier-fusion(t,inf,m)
      trier-fusion(t,m+1,sup)
      fusionner(t,inf,m,sup)
  finSi
Fin

Procédure fusionner (E/S t : tab, E d,m,f : entier )
Var i,j,k : entier
    temp : tab
Début
  i←d
  j←m+1
  Pour k←1 à f-d+1 inc +1 Faire
    Si i≤m et j≤f
       Alors
         Si t[i]≤t[j]
           Alors
             temp[k]←t[i]
             i←i+1
Sinon
             temp[k]←t[j]
             j←j+1
         FinSi
       Sinon
         Si i≤m
           Alors
             temp[k]←t[i]
             i←i+1
           Sinon
             temp[k] ←t[j]
             j←j+1
         FinSi
    FinSi
  FinPour
  Pour k←1 à f-d+1 inc +1 Faire
     t[d+k-1]←temp[k]
  FinPour
Fin
```


Tri rapide (quicksort)

```
Procédure tri-rapide(E/S t : tab, E inf,sup : entier)
Var p : entier
Début
  si inf<sup
    Alors
      partionner(t,inf,sup,p)
      tri-rapide(t,inf,p-1)
      tri-rapide(t,p+1,sup)
  Finsi
Fin

Procédure partitionner (E/S t : tab, E d,f : entier, S p : entier )
Var i,j,pivot : entier
Début
   pivot←t[d]
   i←d
   j←f
   TantQue i≤j Faire
      TantQue t[i]≤pivot et i≤j Faire
         i←i+1
      FinTantQue
      TantQue t[j]>pivot et i≤j Faire
         j←j-1
      FinTantQue
      Si i≤j alors
         échanger(t[i],t[j])
      FinSi
   FinTantQue
   p←j
   échanger(t[d],t[j])
Fin
```

Chapitre 12 – Les pointeurs

Notion de variable dynamique : elle peut être créée ou détruite au cours de l'exécution du bloc dans lequel elle est déclarée ; l'espace-mémoire rendu libre peut être récupéré et l'accès à la valeur se fait par un pointeur.
Un pointeur est une variable dont les valeurs sont des adresses. C'est une variable statique P appelé *variable pointeur* ; par contre, dans la plupart des cas, la variable pointée par P est dynamique.
Etant donné un type quelconque T appelé *type de base*, une variable P de type pointeur est une variable scalaire qui peut prendre comme valeurs les adresses des variables de type T. Il existe donc autant de types pointeurs que de types de base.

Déclaration

```
Type pEntier = ^Entier
Var p : pEntier
```

Opérations

Sorte : Pointeur
Utilise : TypeDeBase
Operations :

Allouer (TypeDeBase) → Pointeur	{constructeur}
Récupérer(Pointeur) →	{observateur}
^Pointeur → TypeDeBase	{observateur}

« Allouer » réserve en mémoire principale un espace assez grand pour contenir un élément du type de base et renvoie un pointeur sur cet élément. C'est une opération indispensable pour la manipulation des pointeurs. La valeur de l'élément alloué est indéfinie (non nulle).
« Récupérer » rend la mémoire au système d'exploitation. P est alors inutilisable.
P^ permet d'accéder à l'élément pointé par P.

Valeur particulière
Quel que soit son type de base, un pointeur peut prendre une valeur particulière nil : signifie que « le pointeur ne pointe sur rien ». Donc, si p=nil alors p^ renvoie toujours une erreur.

Chapitre 13 – Structures séquentielles : les listes

On différencie la place d'un élément dans la liste de l'élément lui-même. Les insertions et les suppressions peuvent se faire à toute place de la liste.

TAD Place

Nécessité de commencer par définir le TAD `Place` qui est une case mémoire contenant un élément et un accès à la place suivante.
Sorte : Place
Utilise : Elément, TypeDebase
Opérations :

affecter-elt(Place, Elément) → Place	{constructeur}
affecter-succ(Place, Place) → Place	{constructeur}
succ(Place) → Place	{constructeur}
contenu(Place) → Elément	{observateur}

TAD Liste

Sorte : Liste
Utilise : Elément, Place, Entier, Booléen
Opérations :

créer-liste-vide() → Liste	{constructeur}
insérer(Liste, Entier, Elément) → Liste	{constructeur}
supprimer(Liste, Entier) → Liste	{constructeur}
concaténer(Liste, Liste) → Liste	{constructeur}
accès(Liste, Entier) → Place	{observateur}
longueur(Liste) → Entier	{observateur}
liste-vide(Liste) → Booléen	{observateur}
ième(Liste, Entier) → Elément	{observateur}

ième(l,i) = contenu(accès(l,i))

Représentation chaînée
On utilise des pointeurs pour lier entre eux les éléments successifs, et la liste est alors déterminée par l'adresse de son premier élément.

```
Type  liste = ^cellule
      cellule = Enregistrement
                  val : élément
                  succ :liste
                FinEnregistrement
Var l : liste
```


Représentation par faux pointeurs
Certains langages de programmation ne comportent pas de pointeur. On peut alors utiliser un tableau et des entiers pour représenter l'indice des éléments dans le tableau. Il existe un nombre assez grand de cases pour y avoir plusieurs listes à la fois. Pour pouvoir facilement insérer et supprimer des éléments dans les listes, il est important de chaîner entre elles les cases libres du tableau.

```
Type  listetab = tableau[1..n] de cellule
      cellule = Enregistrement
                  val : élément
                  succ :liste
                FinEnregistrement
Var t : listab
```

Chapitre 14 – Structures séquentielles : les piles

Une pile est un cas particulier de liste : les insertions et les suppressions se font à une seule extrémité, appelé *sommet de la pile*. LIFO : Last In First Out.

TAD Pile

Sorte : Pile
Utilise : Elément, Booléen
Opérations :

créer-pile-vide() → Pile	{constructeur}
empiler(Pile, Elément) → Pile	{constructeur}
dépiler(Pile) → Pile	{constructeur}
sommet(Pile) → Elément	{observateur}
pile-vide(Pile) → Booléen	{observateur}

Chapitre 15 – Structures séquentielles : les files

Une file est un cas particulier de liste : on fait les ajouts à une extrémité et les suppressions se font à l'autre extrémité. FIFO : First In First Out.

TAD File

Sorte : File
Utilise : Elément, Booléen
Opérations :

créer-file-vide() → File	{constructeur}
ajouter(File, Elément) → File	{constructeur}
retirer(File) → File	{constructeur}
premier(File) → Elément	{observateur}
file-vide(File) → Booléen	{observateur}

Chapitre 16 – Les arbres binaires

Structure arborescente.
Un arbre est un ensemble d'éléments appelés nœuds (dont un se singularise comme la racine) liés par une relation hiérarchique. Un nœud, comme tout élément d'une liste, peut-être de n'importe quel type. Un arbre binaire a au plus 2 fils.

TAD ArbreBinaire

Sorte : ArbreBinaire
Utilise : Noeud, Booléen, Elément
Opérations :
Créer-arbre-vide() → ArbreBinaire
Créer(Noeud, ArbreBinaire, ArbreBinaire) → ArbreBinaire
Racine(ArbreBinaire) → Noeud
Père(Noeud, ArbreBinaire) → Nœud
Fils-gauche(ArbreBinaire) → ArbreBinaire
Fils-droit(ArbreBinaire) → ArbreBinaire
Arbre-vide(ArbreBinaire) → Booléen
Affecter-val(Noeud, Elément) → Noeud
Val(Nœud) → Elément

Représentation par pointeurs

Chaque noeud contient deux pointeurs, l'un vers le sous-arbre gauche et l'autre vers le sous-arbre droit ; l'arbre est déterminé par l'adresse de sa racine.

```
Type  arbrebin = ^noeud
            noeud = Enregistrement
                        val : élément
                        gauche, droit : arbrebin
                    FinEnregistrement
Var Ab : arbrebin
fils-gauche(Ab) = Ab^.gauche
fils-droit(Ab) = Ab^.droit
```

Abre binaire de recherche (ABR)

Propriété : dans un ABR, le sous-arbre gauche (resp. droit) du noeud x ne contient que des éléments de valeur inférieure (resp. supérieure) à celle de l'élément en x.