1. Fonctions, expressions, valeurs

A. Eléments de cours

a) Types

Un *type* regroupe un ensemble de valeurs et les opérations qui lui sont associées. Au niveau du langage, on définit le nom du type, l'ensemble des valeurs du type, les conventions pour les dénoter et la spécification des opérations associées.
Types de base :

Nom du type	entier	réel	caractère	booléen
Dénotation des valeurs	563	35.4	'a' '4' ';' ' '	vrai faux
Opérations	+,-,*,/,reste,quotient	+,-,*,/		et,ou,non

Opérations de comparaison : $=$, $\neq$. Les opérandes doivent être de même type ; le résultat est un booléen. Dans le cas d'un type muni d'une relation d'ordre, on utilise aussi : $<$,$\leq$,$>$,$\geq$.

b) Désignation d'expressions

Pour désigner par le nom **x** une sous-expression **E** d'une expression **F**, on écrit **soit x=E dans F**. Cette phrase est une expression : son évaluation consiste à évaluer **E** dans le contexte où elle apparaît puis à remplacer dans l'expression **F** toute occurrence de **x** par la valeur de **E** et enfin à évaluer **F**. Ainsi l'expression **E** n'est évaluée qu'une fois avant l'évaluation de **F**.
Le nom **x** ainsi introduit est *local* à l'expression **F**, c'est-à-dire que l'association de la valeur de **E** à **x** n'a de sens que dans l'expression **F**. On dit que la *portée* du nom **x** est l'expression **F**.
Par exemple, pour 3 associations, on écrit :

soit x1 = E1, x2 = E2, x3 = E3 dans F

Ces associations sont indépendantes les unes des autres : si dans une expression **Ei** on fait référence à un nom **xj** (i≠j), le nom **xj** tire sa signification du contexte englobant la construction soit et non de l'expression **Ej**. Si l'on veut effectivement exprimer une dépendance, on doit emboîter deux ou plusieurs constructions **soit . . .dans . . .**.

c) Expression conditionnelle

Une expression conditionnelle résulte d'une analyse par cas, les cas s'excluant mutuellement et recouvrant le domaine de valeurs considéré.

La construction selon

La forme syntaxique **selon** explicite tous les cas : on énumère l'ensemble des couples <condition, expression> dans un ordre quelconque. Chaque condition est décrite par une expression à valeur booléenne. Les cas doivent couvrir le domaine et s'exclure mutuellement.
Par exemple, pour trois cas, on écrit :

```
selon { description du domaine }
    C1 : E1
    C2 : E2
    C3 : E3
```


Une telle phrase est une expression. Son évaluation comporte d'une part l'évaluation des expressions booléenne **Ci** pour déterminer celle qui a la valeur vrai et d'autre part l'évaluation de la expression **Ei** correspondant au cas ainsi identifié. La valeur alors obtenue est la valeur de l'expression conditionnelle. *L'ordre dans lequel les cas sont énumérés n'est pas pertinent vis-à-vis de l'évaluation.*
On doit avoir :

```
(C1 ou C2 ou C3)        et        non (C1 et C2) et non (C1 et C3) et non (C2 et C3)
{ recouvrement          et                exclusion mutuelle }
```

Autres notations

— Les deux expressions suivantes ont le même sens :

```
si C alors E1 sinon E2          selon { description du domaine }
                                    C : E1
                                    non C : E2
```

— Lorsqu'il y a plus de deux cas on peut aussi utiliser le mot-clé **sinon** pour exprimer que la condition correspond à tous les autres cas que ceux énumérés explicitement. Par exemple les expressions suivantes ont le même sens :

```
selon { description du domaine }        selon { description du domaine }
    C1 : E1                                 C1 : E1
    C2 : E2                                 C2 : E2
    C3 : E3                                 C3 : E3
    sinon : E4                              non C1 et non C2 et non C3 : E4
```

Les opérateurs booléens conditionnels "et puis" et "ou alors"

A et **B** étant deux expressions à valeur booléenne :
— L'expression **A et puis B** et l'expression **si A alors B sinon faux** ont le même sens : *dans ces deux expressions,* **B** *n'est évalué que si* **A** *a la valeur vrai.*
— De même les expressions **A ou alors B** et **si A alors vrai sinon B** ont le même sens : **B** *n'est évalué que si* **A** *a la valeur faux.*

Expression conditionnelle à valeur booléenne

Une expression conditionnelle à valeur booléenne peut toujours être ré-écrite sous forme d'une formule logique à l'aide de l'opérateur **et puis**. Par exemple, l'expression

```
selon
    C1 : B1
    C2 : B2
    C3 : B3
```

peut s'écrire

```
(C1 et puis B1) ou (C2 et puis B2) ou (C3 et puis B3)
```

d) N-uplets, produit de types

Un *n-uplet* de valeurs est décrit entre des chevrons ($<$ et $>$), les valeurs étant séparées par des virgules. Le type d'un n-uplet est le produit (au sens ensembliste du terme) des types de chacune des valeurs. Pour définir un produit de types, on peut de manière équivalente, ou bien décrire un n-uplet de types ou bien décrire un n-uplet de champs, un champ étant formé d'un nom et d'un type.

Description d'un type produit sans noms de champs

On décrit le type produit entre des chevrons, les types étant séparés par des virgules. On peut par exemple définir un type **date** de la manière suivante :

date : type <entier sur [1...31], entier sur [1...12], un entier ≥ 0 >
{ <j, m, a> étant de type date, j est le quantième du jour dans le mois, m est le quantième du mois dans l'année et a est l'année. }

Pour sélectionner les valeurs qui composent un objet de type produit défini sans noms de champs, il faut utiliser une expression pour nommer ces valeurs. Par exemple, si **D** est de type **date**, on écrit : soit <j, m, a> = D dans ••••

Description d'un type produit avec noms de champs

On définit le type produit en énumérant les champs entourés de chevrons et séparés par des virgules. Dans l'exemple des dates, on écrit maintenant :

date : type <qj : entier sur [1...31], qm : entier sur [1...12], qa : entier ≥ 0 >

Dans cette forme, on sélectionne les valeurs qui composent une valeur de type **date** en utilisant les noms de champs **qj**, **qm** et **a** qui font partie de la définition du type : **D** étant une date de valeur < 9, 10, 1964 >, **D.qj** vaut 9, **D.qm** vaut 10 et **D.qa** vaut 1964.

e) Textes et caractères

Le type **caractère** rassemble tous les caractères (en particulier, l'espace est un caractère comme les autres). Un caractère est dénoté en l'entourant de quotes (’). Les seules opérations attachées au type caractère sont les opérations de comparaison ($=$ et $\neq$).

Le type **texte** regroupe toutes les chaînes de caractères, quelle que soit leur longueur. Un texte est dénoté en l'entourant de guillemets ("). Ceci est nécessaire pour distinguer un caractère et le texte singleton formé du même caractère. Par exemple, **'a'** est de type **caractère** alors que **"a"** est de type **texte**. Le texte vide est noté **[]**.

Le type **texte** est muni des opérations suivantes (de manière plus générale ces opérations sont définies pour toute séquence) :

- L'opération de concaténation de deux textes est notée **&**.
- Les symboles ∘ et • dénotent respectivement l'ajout d'un caractère *à gauche* d'un texte et l'ajout d'un caractère *à droite* d'un texte : c∘t = [c] & t t•c = t&[c]
- EstVide(t) a la valeur vrai si le texte t est vide.
- premier(t) dénote le premier caractère du texte t ; fin(t) dénote le texte formé de t sans son premier caractère : t = premier(t) ∘ fin(t).
- dernier(t) dénote le dernier caractère du texte t ; début(t) dénote le texte formé de t sans son dernier caractère : t = début(t) • dernier(t).
- Les opérations premier, fin, dernier et début ne peuvent pas être appliquées à un texte vide.

B. Exemples d'exercices rédigés

Exemple E1.1 : Moyenne olympique

La moyenne olympique de n nombres est la moyenne des n-2 nombres restant après avoir enlevé deux des nombres donnés, l'un ayant la valeur maximum et l'autre la valeur minimum.
Décrire une fonction qui étant donnés quatre entiers positifs ou nuls, leur associe leur moyenne olympique. Par exemple, la moyenne olympique des quatre nombres 10, 8, 12, 24 est 11, celle des nombres 12, 12, 12, 12 est 12.

```
{ spécification des fonctions }
   mo : fonction (u, v, w, x : entier ≥ 0)  ⟶ réel ≥ 0          { moyenne olympique de 4 nombres }
   minquatre : fonction (i, j, k, l : entier ≥ 0)  ⟶ entier ≥ 0          { minimum de 4 nombres }
   maxquatre : fonction (i, j, k, l : entier ≥ 0)  ⟶ entier ≥ 0          { maximum de 4 nombres }
   maxdeux, mindeux : fonction (a,b : deux entiers ≥ 0)  ⟶ entier ≥ 0 { maximum et minimum de 2 nombres }

{ réalisation des fonctions }
   maxdeux(a,b) :
      retour : (a+b + abs(a-b)) / 2
   mindeux(a,b) :
      retour : (a+b - abs(a-b)) / 2
   maxquatre(i,j,k,l) :
      retour : maxdeux (maxdeux (maxdeux(i,j), k), l)
   minquatre(i,j,k,l) :
      retour : mindeux (mindeux (mindeux(i,j), k), l)
   mo(u, v, w, x) :
      retour : (u+v+w+x - minquatre(u,v,w,x) - maxquatre(u,v,w,x)) / 2
```

Exemple E1.2 : Maximum de trois entiers

Décrire une fonction qui détermine le maximum de trois entiers distincts deux à deux. Etudier diverses solutions selon la manière de conduire l'analyse par cas et de la formuler.

```
Maxtrois : fonction (a, b, c : entier)  ⟶ entier      { maximum de 3 nombres distincts deux à deux }
```

(i) Analyse en 3 cas fondée sur l'analyse du résultat

```
Maxtrois(a, b, c) :
   retour : selon a, b, c
            a>b et a>c : a
            b>a et b>c : b
            c>a et c>b : c
```

(ii) Analyse en 6 cas fondée sur l'analyse des données

```
Maxtrois(a, b, c) :
   retour : selon a, b, c
            a>b et b>c : a
            b>a et a>c : b
            a>c et c>b : a
            c>a et a>b : c
            b>c et c>a : b
            c>b et b>a : c
```

(iii) Analyses en deux cas emboîtées fondées sur l'analyse des données

```
Maxtrois(a, b, c) :
    retour : si a > b alors
                 si a > c alors a
                 sinon { a < c } c
             sinon { a < b }
                 si b > c alors b
                 sinon { b < c } c
```

(iv) Utilisation d'un nom local (pour décrire le maximum des deux premiers)

```
Maxtrois(a, b, c) :
    retour : soit M = si a > b alors a sinon b
                 dans si M > c alors M sinon c
```

(v) Utilisation d'une fonction plus simple (Maxdeux)

```
Maxtrois(a, b, c) :
    retour : Maxdeux(Maxdeux(a, b), c)
```

Exemple E1.3 : Somme des chiffres d'un nombre

Décrire une fonction qui à un entier compris entre 0 et 9999 associe la somme des chiffres qui le composent dans sa représentation en base 10. Exemple : donnée 9639, résultat : 27.

```
SC : fonction (n : entier sur [0...9999]) ⟶ entier ≥ 0
    { Somme des chiffres de la représentation en base 10 de n. }
```

Les chiffres formant l'entier donné peuvent être déterminés à l'aide de divisions par 10. On introduit une fonction qui à deux nombres associe le quotient et le reste de leur division :

```
QR : fonction (N : entier ≥ 0, D : entier > 0) ⟶ entier ≥ 0, entier ≥ 0
    { soit <q, r> = QR (N, D), q est le quotient et r le reste de la division de N par D }
QR (N, D) :
    retour : <N quotient D, N reste D>
        { quotient et reste sont des primitives correspondant à la division entière }
```

On peut alors réaliser la fonction **SC**, par exemple comme suit :

```
SC(X) :
    retour : soit <q1, c0> = QR (X, 10)
                 dans soit <q2, c1> = QR (q1, 10)
                     dans soit <c3, c2> = QR (q2, 10)
                         dans c0 + c1 + c2 + c3
```

Exemple E1.4 : A propos de durées

On considère ici des durées représentées par des quadruplets <j, h, m, s> où j représente un nombre de jours, h un nombre d'heures dans une journée, m un nombre de minutes dans une heure et s un nombre de secondes dans une minute. On étudie des opérations sur les durées.

(i) Spécification des types et des fonctions

```
{ types et fonctions sur les entiers }
    Ent60 : type entier sur [0...59] ; Ent24 : type entier sur [0...23]
    QR : fonction (N : entier ≥ 0, D : entier > 0) ⟶ entier ≥ 0, entier ≥ 0 { quotient et reste, cf. E1.3 }
    LeNbDeSec : fonction (j, h, m, s : entier ≥ 0) ⟶ entier ≥ 0
        { Equivalent en secondes de j jours, h heures, m minutes et s secondes (sans contrainte sur leurs valeurs). }
{ le type Durée et les fonctions associées }
    Durée : type <entier ≥ 0, Ent24, Ent60, Ent60>
        { durée sous forme canonique composée d'un nombre de jours, d'un nombre d'heures, d'un nombre de minutes et d'un nombre de secondes. }
{ constructeurs }
    LaDurée_E : fonction (E : entier ≥ 0) ⟶ Durée
        { construction d'une durée à partir d'un nombre de secondes }
    LaDurée_4E : fonction (j, h, m, s : entier ≥ 0) ⟶ Durée
        { construction d'une durée à partir de quatre entiers représentant un nombre de jours, d'heures, de minutes et de secondes, (sans contrainte sur leurs valeurs) }
{ sélecteur }
    LesSec : fonction (D : Durée) ⟶ entier ≥ 0          { nombre total de secondes d'une durée }
{ autres fonctions sur les durées }
    EgD : fonction (D1, D2 : Durée) ⟶ booléen           { égalité de deux durées }
    InfD : fonction (D1, D2 : Durée) ⟶ booléen          { relation "inférieur strict" sur les durées }
    SomD : fonction (D1, D2 : Durée) ⟶ Durée            { somme de deux durées }
```

(ii) Réalisation des fonctions sur les durées

```
{ égalité de deux durées }
{ version 1 }
    EgD(D1, D2) :
        retour : LesSec(D1) = LesSec(D2)
{ version 2 }
    EgD(D1, D2) :
        retour : soit <j1, h1, m1, s1> = D1, <j2, h2, m2, s2> = D2
                 dans j1 = j2 et h1 = h2 et m1 = m2 et s1 = s2

    Une autre formulation :
        EgD(<j1, h1, m1, s1>, <j2, h2, m2, s2>) :
            retour : j1 = j2 et h1 = h2 et m1 = m2 et s1 = s2

{ relation inférieur sur les durées }
{ version 1 }
    InfD (D1, D2) :
        retour : LesSec(D1) < LesSec(D2)
{ version 2 }
    InfD (<j1, h1, m1, s1>, <j2, h2, m2, s2>) :
        retour : si j1 ≠ j2 alors j1 < j2
                 sinon { j1=j2 } si h1 ≠ h2 alors h1 < h2
                       sinon { j1=j2 et h1=h2 }
                           si m1 ≠ m2 alors m1 < m2
                           sinon { j1=j2 et h1=h2 et m1 = m2 } s1 < s2

{ somme de deux durées }
{ version 1 }
    SomD (D1, D2) :
        retour : LaDurée_E(LesSec(D1) + LesSec(D2) )
```

```
{ version 2 }
   SomD (<j1, h1, m1, s1>, <j2, h2, m2, s2>) :
         retour : soit <m,ss> = QR(s1+s2, 60)
                  dans soit <h, sm> = QR(m+m1+m2, 60)
                     dans soit <j, sh> = QR(h+h1+h2, 24)
                        dans <j+j1+j2, sh, sm, ss>
```

(iii) Réalisation des constructeurs et sélecteurs sur les durées

```
LaDurée_E(n) :
   retour : soit <j, rj> = QR(n, 86400)
            dans soit <h, rh> = QR(rj, 3600)
               dans soit <m, s> = QR(rh, 60)
                  dans <j, h, m, s>
LaDurée_4E(j, h, m, s) :
   retour : LaDurée_E(LeNbDeSec(j, h, m, s))
LesSec(<j, h, m, s>) :
   retour : LeNbDeSec(j, h, m, s)
```

(iv) Réalisation des fonctions sur les entiers

```
QR(a, b) :
   retour : <a quotient b, a reste b>
LeNbDeSec(j, h, m, s) :
   retour : 86400*j + 3600*h + 60*m + s
```

C. Exercices

E1.5 : A propos de booléens

- Donner les tables de vérité des opérateurs booléens **non**, **et**, **ou**. Puis étudier les tables de vérité définissant les autres opérateurs booléens binaires et identifier les opérateurs connus.
- Exprimer les opérateurs de comparaison $\leq$, $\neq$, $>$, $\geq$ en termes des opérateurs non, $=$, $<$.
- Pour représenter le type **booléen** à l'aide du type **entier**, on décide que les constantes **vrai** et **faux** sont respectivement représentées par les entiers **1** et **0**. Donner les expressions correspondant aux opérateurs booléens usuels en termes des opérateurs arithmétiques.

E1.6 : Signe du produit

Etant donnés deux entiers, on veut, **sans calculer leur produit**, fournir, l'un des deux messages suivants : "le produit des deux nombres est positif ou nul" ou "le produit des deux nombres est négatif".

```
SignProd : fonction (x, y : entier)  ⟶ texte
```

Compléter chacune des réalisations suivantes de la fonction **SignProd**.

(i) Version 1 : analyse en deux cas

```
SignProd (x, y) :
   retour : "le produit des deux nombres est " & (si ●●●●●● alors "positif ou nul" sinon "négatif")
```


(ii) Version 2 : analyses par cas emboîtées

```
SignProd (x, y) :
   retour : "le produit des deux nombres est " &
               si x > 0 alors ●●●●●●
               sinon si x < 0 alors ●●●●●●
                     sinon { x=0 } ●●●●●●
```

E1.7 : Une date est-elle correcte ?

On considère une date représentée par deux entiers, un numéro de jour dans le mois et un numéro de mois dans l'année. On veut déterminer si deux entiers **j** et **m** caractérisent respectivement le jour et le mois d'une date d'une année non bissextile. Par exemple, c'est vrai pour $j = 28$ et $m = 1$; mais cela ne l'est pas pour $j = 31$ et $m = 4$, ni pour $j = 18$ et $m = 13$.

```
EstDate : fonction (j, m : entier)  ⟶ booléen
   { vrai ⟺ j et m caractérisent respectivement le jour et le mois d'une date d'une année non bissextile
   }
```

- Compléter la réalisation suivante de la fonction :

```
EstDate(j,m) :
   retour : 1 ≤ ●●● et ●●● ≤ 12 et
            1 ≤ j et j ≤ (si ●●● alors 31
                  sinon si ●●● alors 28
                     sinon ●●●●●●●●)
```

- Proposer d'autres réalisations de la fonction.

E1.8 : Codage d'un texte : code de César

On désire coder un texte ne comportant que des lettres majuscules et des espaces, en remplaçant chaque caractère par un autre caractère selon les règles suivantes (code dit "de César") :

- À un espace correspond un espace,
- À la lettre 'A' correspond la lettre 'D', à 'B' correspond 'E', ..., à 'W' correspond 'Z', à 'X' correspond 'A', à 'Y' correspond 'B', à 'Z' correspond 'C'.

Inversement, on veut pouvoir décoder un texte codé selon ce qui précède.
Pour cela on étudie deux fonctions, de codage et de décodage d'un caractère :

```
CodeCar : fonction (C : caractère { lettre majuscule ou espace })  ⟶ caractère
DécodeCar : fonction (C : caractère { lettre majuscule ou espace })  ⟶ caractère
```

(i) Une solution adaptée directement au problème posé

Si l'on veut procéder à la main, on place deux exemplaires de l'alphabet (dans l'ordre), l'un sous l'autre, en décalant le deuxième de trois positions à droite, et en complétant la deuxième ligne par les trois premières lettres de l'alphabet.
Codage et décodage se font alors aisément, une lettre et son code se trouvant alignés sur la même colonne. De même, pour réaliser chacune des deux fonctions **CodeCar** et **DécodeCar** il suffit de les décrire par une expression conditionnelle à 27 cas : 1 cas pour l'espace et 1 cas pour chacune des 26 lettres de l'alphabet.

(ii) Une solution plus générale
Le code de César est fondé sur une translation sur l'alphabet : les lettres étant rangées de manière circulaire dans l'ordre alphabétique (A puis B puis C, ..., puis Z, puis A), le code d'une lettre est la lettre située 3 positions après elle.
L'idée de cette deuxième version est d'assurer le codage en associant à chaque lettre un entier, par exemple son rang dans l'alphabet et à utiliser les opérations sur les entiers pour exprimer la translation.
Ceci est illustré par le schéma suivant :

- L dénote une lettre quelconque et L' dénote le code de L ;
- E dénote l'entier associé à L, E' dénote l'entier associé à L' ;
- Les flèches sont étiquetées par un nom de fonction. Les flèches en pointillé montrent la manière de réaliser les fonctions étiquetant les flèches pleines par composition des fonctions étiquetant les flèches en pointillé.

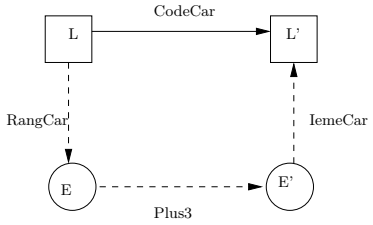


Figure 1.1 – Principe du codage de César

Les trois fonctions intermédiaires apparaissant sur le schéma sont spécifiées comme suit :

Plus3 : fonction (E : entier sur [1...26]) ⟶ entier sur [1...26]
{ E étant associé à L, Plus3(E) est l'entier associé à L' }
RangCar : fonction (L : lettre majuscule) ⟶ entier sur [1...26]
{ Rang de la lettre donnée dans l'alphabet }
IèmeCar : fonction (E : entier sur [1...26]) ⟶ lettre majuscule
{ Lettre de rang E dans l'alphabet }

Q1

- Donner une réalisation de la fonction **CodeCar** en utilisant les fonctions intermédiaires.
- Donner une réalisation de la fonction **Plus3**.
- Donner de même une réalisation de la fonction **DécodeCar** en se basant sur un principe analogue à ce qui a été fait pour **CodeCar**.

(iii) Généralisation
On généralise le principe du codage en se basant sur une translation de n positions dans l'alphabet (n = 3 étant le cas particulier précédent).

Q2

- Spécifier les fonctions de codage et de décodage.
- Réaliser la fonction de codage en spécifiant, réalisant et utilisant les fonctions intermédiaires adéquates.
- Réaliser la fonction de décodage en n'utilisant que la fonction de codage.

E1.9 : Relations sur des intervalles d'entiers
On caractérise un intervalle fermé d'entiers par la donnée de ses bornes inférieure et supérieure.

Intervalle : type <entier, entier>
{ <i, s> décrit un intervalle de borne inférieure i et de borne supérieure s.
Contrainte : $i \leq s$. }

(i) Relations entre points et intervalles
Pour situer un entier par rapport à un intervalle, on spécifie les trois fonctions suivantes :

Précède : fonction (x : entier, I : Intervalle) ⟶ booléen *{ vrai ⟺ x < borne inférieure de I }*
Dans : fonction (x : entier, I : Intervalle) ⟶ booléen *{ vrai ⟺ x appartient à I }*
Suit : fonction (x : entier, I : Intervalle) ⟶ booléen *{ vrai ⟺ x > borne supérieure de I }*

- En utilisant des opérateurs logiques donner une réalisation des fonctions **Précède** et **Dans**.
- Donner une réalisation de la fonction **Suit** en utilisant les deux fonctions précédentes.

(ii) Relations entre intervalles
Donner une réalisation de chacune des deux fonctions suivantes :

CoteAcote : fonction (I1, I2 : Intervalle) ⟶ booléen
{ vrai ⟺ I1 et I2 sont contigus. Exemples : <1,3>, <4, 10> ; <2, 8>,<-1,1> }
Chevauche : fonction (I1, I2 : Intervalle) ⟶ booléen
{ vrai ⟺ I1 et I2 ont des points communs, ils n'ont pas de bornes communes et aucun n'est inclus dans l'autre. }

E1.10 : Fractions
On étudie un type correspondant à la notion de fraction. On veut pouvoir construire des fractions à partir de deux entiers correspondant au numérateur et au dénominateur, fournir le texte d'une fraction et disposer des opérations sur les fractions. Les valeurs manipulées sont donc des entiers, des textes et des fractions.

Q1. Spécification et réalisation d'un type Fraction
Il s'agit de spécifier le type **Fraction** et les fonctions de construction et de sélection associées selon les indications ci-dessous. On travaillera par analogie avec l'exemple **E1.4**.

Un objet de type **Fraction** est un couple d'entiers (produit de types) formé du numérateur et du dénominateur de la **forme irréductible** de la fraction considérée.

La fonction de construction d'un objet de type **Fraction**, nommée **Frac** a pour paramètre deux entiers correspondant au numérateur et au dénominateur de la fraction considérée éventuellement sous forme non réduite : par exemple, Frac(21, 60) construit le couple <7, 20>, tout comme Frac(42, 120) ou Frac(7, 20).

Pour mettre une fraction sous forme irréductible, on dispose d'une fonction de calcul du plus grand commun diviseur de deux nombres, dont le profil est le suivant :

pgcd : fonction (n1. n2 : entier > 0) ⟶ entier > 0.

Q2. Utilisation du type Fraction
Réaliser les fonctions suivantes :

SomFrac : fonction (F1, F2 : Fraction) ⟶ Fraction *{ somme de deux fractions }*
TexteFrac : fonction (F : Fraction) ⟶ texte
{ texte décrivant une fraction, par exemple TexteFrac (Frac (4, 6)) = "2 / 3"
Exemple d'utilisation : l'expression TexteFrac (SomFrac (Frac (3, 4), Frac (15, 18))) a pour valeur "19/12". }

On dispose de la fonction **LeTexteE** construisant le texte de la représentation décimale d'un entier :

LeTexteE : fonction (E : entier) ⟶ texte
{ texte de la représentation décimale d'un entier. Par exemple : LeTexteE (125) = "125" }

D. Problèmes dirigés

E1.11 : A propos d'une nomenclature de pièces
On considère ici un aspect de l'informatisation d'un stock de pièces d'un magasin de fournitures, pour automobiles par exemple, et plus particulièrement la manière de référencer ces pièces.

Une *nomenclature* rassemble l'ensemble des conventions permettant de regrouper les pièces en différentes catégories et d'associer à chaque pièce une *référence* unique servant à la désigner. Ces conventions sont ici les suivantes :
- A un premier niveau, les pièces sont réparties en *familles*. Chaque famille est codée par une lettre majuscule. On peut donc répertorier au plus 26 familles.
- A un deuxième niveau, on code les pièces dans chaque famille, par un entier entre 0 et 999. On peut donc répertorier au plus 1000 pièces dans chaque famille.

La forme externe d'une référence est ainsi formée de 4 caractères, la lettre codant la famille, puis les trois chiffres codant la pièce (avec des zéros en tête si nécessaire). Par exemple, la référence **B053** désigne la pièce de numéro **053** dans la famille **B** alors que la référence **C053** désigne la pièce de numéro **053** dans la famille **C**.

(i) Type associé à la notion de référence

Pour représenter les références, on définit le type **Réf**, selon le lexique suivant :

LettreMaj : type caractère *{ restreint aux lettres majuscules }*
Réf : type <CFam : LettreMaj, NumP : entier sur [0. . .999]>

Par exemple, <**'X', 4**> est la valeur de type **Réf** correspondant à la référence **X004**.
On considère une fonction, nommée **TexteVersRéf**, spécifiée comme suit :

TexteVersRéf : (T : texte) ⟶ Réf
{ Valeur de type Réf associée à T.
Par exemple : TexteVersRéf ("X340") = <'X', 340>, TexteVersRéf ("X041") = <'X', 41>
Pré-condition *: T est formé d'une lettre suivie de 3 chiffres. }*

Pour réaliser la fonction **TexteVersRéf**, on introduit une fonction nommée **CvE** :

Chiffre : type caractère *{ restreint aux caractères dénotant des chiffres }*
CvE : fonction (C : Chiffre) ⟶ entier
{ valeur entière correspondant au chiffre. Par exemple, CvE('2') = 2 ; CvE('0') = 0 }

Q1. Donner une réalisation de la fonction **TexteVersRéf** en termes de **CvE**.

Q2. Réalisation de la fonction **CvE**
- Donner une réalisation de la fonction **CvE** sous forme d'une expression conditionnelle.
- On dispose de plus des éléments de lexique suivant :

  AlphabetChif : texte "0123456789"
  Rang : fonction (C : caractère, T : texte non vide de caractères distincts deux à deux) ⟶ entier > 0
  { Rang de C dans T. Le rang du premier élément de T est 1.
  Pré-condition : C appartient à T. }

  Donner une autre réalisation de la fonction **CvE** en termes de la fonction **Rang**.

(ii) Relation d'ordre sur les références
L'ordre sur les références que nous étudions ici est défini de la manière suivante : la priorité est donnée aux codes de familles par rapport aux numéros de pièces. L'ordre sur les familles est l'ordre alphabétique, l'ordre sur les numéros de pièce est celui des entiers. Par exemple la référence **B300** est avant la référence **C230**, (parce que **B** précède **C** dans l'ordre alphabétique) et la référence **U817** est après la référence **U653** (parce que **817** est strictement supérieur à **653**).

Pour représenter cette relation d'ordre, on spécifie les deux fonctions suivantes :

EgR : fonction (R1, R2 : Réf) ⟶ booléen *{ vrai ⟺ les deux valeurs données sont identiques }*
InfR : fonction (R1, R2 : Réf) ⟶ booléen *{ vrai ⟺ R1 est strictement avant R2 }*

Pour comparer des références, on doit donc pouvoir comparer des lettres de l'alphabet. On fait abstraction du codage interne des lettres et on suppose que les seules opérations de comparaison prédéfinies sur les lettres sont l'égalité (notée $=$) et la différence (notée $\neq$).

On spécifie ainsi une fonction nommée **InfL** comme suit :

InfL : fonction (L1, L2 : LettreMaj) ⟶ booléen
{ vrai ⟺ L1 précède strictement L2 dans l'ordre alphabétique. Par exemple InfL('L', 'X') = vrai ;
InfL('U', 'U') = faux }

Q3. Réalisation des fonctions **EgR** et **InfR**
- Donner une réalisation de chacune des deux fonctions **EgR** et **InfR**, en termes des opérations de comparaison sur les entiers, et de celles sur les lettres de l'alphabet ($=$, $\neq$, **InfL**).
- Spécifier les autres fonctions permettant de comparer des références, analogues aux opérateurs habituels sur les entiers ($\neq$, $>$, $\leq$, $\geq$) puis en donner une réalisation (en exploitant les relations qui existent entre les opérateurs de comparaison).

Q4. Réalisation de la fonction **InfL**
Pour réaliser la fonction **InfL**, on introduit le lexique suivant :

TexteMaj : type texte *{ formé de LettreMaj en ordre alphabétique }*
AlphabetMaj : TexteMaj = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

- Donner une réalisation de la fonction **InfL** en utilisant la fonction **Rang** (cf **Q2**).
- **InfL(L1, L2)** a la valeur vrai si et seulement si **L1** se trouve avant **L2** dans une liste des lettres de l'alphabet classée en ordre alphabétique. Pour réaliser **InfL** selon cette idée, on dispose de la fonction suivante (que l'on ne demande pas de réaliser) :

EstAvant : fonction (L1, L2 : LettreMaj différentes, T : TexteMaj de longueur ≥ 2) $\longrightarrow$ booléen
{ vrai $\Longleftrightarrow$ L1 est avant L2 dans T. **Contrainte** *: L1 et L2 doivent appartenir à T. }*

Donner une deuxième réalisation de la fonction **InfL** en utilisant la fonction **EstAvant**.

E1.12 : Quelle heure est-il ?
(i) Le type HeureDuJour

On étudie l'information portée par une montre concernant l'heure dans la journée. Elle est décrite ici par un type nommé **HeureDuJour**, quadruplet formé de trois entiers et un booléen. Les entiers représentent une durée de 0 à 12 heures, exprimée en heures, minutes et secondes. Le booléen indique si l'heure du jour est située entre minuit et midi (avant midi, en latin ante meridiem, en abrégé am) ou entre midi et minuit (après midi, en latin post meridiem, en abrégé pm).

```
HeureDuJour : type
    < nbh : entier sur [0...11]                    { nombre d'heures }
      nbm, nbs : entier sur [0...59]               { nombres de minutes et de secondes }
      avant : booléen >                            { vrai si avant midi, faux sinon }
```

Par exemple, le quadruplet <3,0,0,faux> représente l'heure exprimée habituellement par "3h-pm" ou par "15h". De même, <2,25,30,vrai> représente "2h25mn30s-am". On note que <0,0,0,vrai> correspond à minuit et que <0,0,0,faux> correspond à midi.

Q1. Plus 1 seconde

— Donner une réalisation de la fonction **Plus1S** spécifiée de la manière suivante :
Plus1S : fonction (X : HeureDuJour) $\longrightarrow$ HeureDuJour
{ Heure du jour à la seconde suivant l'heure X. Par exemple, Plus1S(<2,25,30,vrai>) = <2,25,31,vrai> et Plus1S(<11,59,59,faux>) = <0,0,0,vrai> }

— Donner une réalisation de la fonction **InfH** spécifiée de la manière suivante :
InfH : fonction (X1, X2 : HeureDuJour) $\longrightarrow$ booléen
{ vrai $\Longleftrightarrow$ H1 précède strictement H2 (ordre chronologique). }

(ii) Le texte représentant l'heure

On veut construire le texte représentant une heure d'horloge selon les conventions suivantes :
— On utilise les suffixes "-am" et "-pm" pour distinguer les heures avant midi de celles après midi.
— Chaque quantité est suivie de son unité : h pour heure, mn pour minute, s pour seconde. Par exemple, LeTexteH(<11,34, 20, faux) = "11h34mn20s-pm".
— Lorsque l'heure comporte une quantité nulle, l'indication correspondante est omise dans le texte. Par exemple, LeTexteH(<11,0,5,faux>) = "11h5s-pm", LeTexteH(<3,0,0,vrai>) = "3h-am".
— Toutefois, lorsque les trois entiers sont nuls, le texte est "minuit" ou "midi" selon le cas.

On étudie ainsi la réalisation d'une fonction **LeTexteH** :

```
LeTexteH : fonction (X : HeureDuJour) ⟶ texte
    { Texte décrivant X avec les conventions ci-dessus }
```

Pour construire les parties du texte correspondant à chaque unité de temps, en tenant compte du cas des quantités nulles, on introduit une fonction nommée **LeTexteU** :

LeTexteU : fonction (e : entier ≥ 0, s : texte) $\longrightarrow$ texte
{ Texte vide si e=0, sinon la représentation décimale de e suivie de s. Par exemple, LeTexteU(25,"mn")="25mn", LeTexteU(0,"mn")=[] }

De plus, on dispose de la fonction **LeTexteE** (dont la réalisation n'est pas demandée) :

```
LeTexteE : fonction (E : entier) ⟶ texte
    { texte de la représentation décimale d'un entier. Par exemple : LeTexteE (125) = "125" }
```

Q2. Donner une réalisation de la fonction **LeTexteU** en termes de la fonction **LeTexteE**. Puis donner une réalisation de la fonction **LeTexteH** en termes de la fonction **LeTexteU**.

E1.13 : A propos de dates

Parmi les diverses formes d'expression de la date d'un jour dans le calendrier grégorien, nous nous intéressons à la suivante : une date, en français, est caractérisée par un triplet d'entiers composé du quantième du jour dans le mois, du quantième du mois dans l'année et de l'année. Ainsi, le triplet 22, 9, 1990 caractérise le 22ème jour du 9ème mois de l'année 1990.

On ne considère ici que des dates prises dans la période [1900...2100]. On rappelle qu'une année est bissextile si son expression numérale est divisible par 4, comme 1968, 1972, 1976, ...Toutefois les années séculaires ne sont pas bissextiles, sauf celles dont les deux premiers chiffres forment un nombre divisible par 4, comme 1600, 2000, 2400.

Dans ce qui suit, on étudie diverses fonctions sur les dates.

(i) Informations caractérisant une date

La période considérée est caractérisée par ses années de début et de fin. Une année est dénotée par un entier compris entre ces deux années. Un *quantième d'une date* dans une année est désigné par un entier compris entre 1 et 366 ou 365 selon que l'année est bissextile ou non. Un *jour de mois* est dénoté par un entier dont l'intervalle de définition dépend du nombre de jours du mois considéré, mais il est inclu dans l'intervalle [1...31]. Un *mois* est dénoté par un entier compris entre 1 et 12.
On fixe ainsi le lexique suivant :

```
NjourM : type entier sur [1...31]              { numérotation des jours dans le mois }
quantième : type entier sur [1...366]          { numérotation des jours dans l'année }
Nmois : type entier sur [1...12]               { numérotation des mois dans l'année }
AnDeb : entier 1900 ; AnFin : entier 2100
année : type entier sur [AnDeb...AnFin]
```

De plus, on dispose de la fonction **LeTexteE** :

```
LeTexteE : fonction (E : entier) ⟶ texte
    { texte de la représentation décimale d'un entier. Par exemple : LeTexteE (125) = "125" }
```

(ii) Quantième d'une date dans son année

On étudie une fonction **QuantièmeJMA** :

```
QuantièmeJMA : fonction (J : NjourM, M : Nmois, A : année) ⟶ quantième
    { valeur du quantième dans l'année d'une date donnée sous forme de trois entiers correspondant au
    jour dans le mois, au mois dans l'année et à l'année. Par exemple, QuantièmeJMA(15,11,1993) =
    319 }
```

Pour déterminer le quantième d'une date dans son année, on se ramène au quantième du premier jour d'un mois donné dans le cas d'une année non bissextile. On spécifie ainsi la fonction :

PremierJour : fonction (M : Nmois) ⟶ quantième
{ quantième du premier jour du mois donné dans le cas d'une année non bissextile }

Q1.
- Réaliser la fonction **QuantièmeJMA** dans le cas d'une année non bissextile.
- Puis compléter la réalisation précédente pour tenir compte des années bissextiles. On spécifie pour cela une fonction supplémentaire :

  EstBis : fonction (A : année) ⟶ booléen *{ vrai ⟺ l'année est bissextile }*

- Réaliser la fonction **EstBis** en tenant compte de la période considérée.

(iii) Texte associé à une date

On considère une fonction **LeTexteD** :

LeTexteD : fonction (J : NjourM, M : Nmois, A : année) ⟶ texte *{ texte de la date donnée }*
{ Par exemple, LeTexteD(15,11,1993) = "15 Novembre 1993"; LeTexteD(1, 11, 1993) = "1er Novembre 1993". }

On introduit une fonction **LeNomDeMois** :

LeNomDeMois : fonction (M : Nmois) ⟶ texte *{ nom en français du mois de numéro donné }*

Q2. Réaliser la fonction **LeTexteD** en utilisant la fonction **LeNomDeMois**.

(iv) Date du lendemain On considère une fonction **DemainJMA** :

DemainJMA : fonction (J : NjourM, M : Nmois, A : année) ⟶ texte
{ texte d'une date correspondant au lendemain d'une date donnée , ou message si la date est la dernière de la période }

On introduit une fonction **LeLendemain** :

LeLendemain : fonction (J : NjourM, M : Nmois, A : année) ⟶ NjourM, Nmois, année
{ Date du lendemain d'une date.
Contrainte *: la date donnée n'est pas la dernière de la période. }*

Pour déterminer la date du lendemain, il faut savoir si c'est la fin d'un mois et pour cela connaître le dernier jour de chaque mois :

FinDeMois : fonction (J : NjourM, M : Nmois, A : année) ⟶ booléen
{ vrai ⟺ la date donnée est le dernier jour du mois }
DernierJour : fonction (M : Nmois) ⟶ NjourM
{ dernier jour du mois, dans le cas d'une année non bissextile }

Q3. Réaliser successivement les fonctions **DemainJMA**, **LeLendemain**, **FinDeMois**.

(v) Informations attachées à la notion de mois

Q4. Réaliser les fonctions **LeNomDeMois**, **DernierJour** et **PremierJour**.

UGA, UFR IM²AG, Algorithmique, langages et programmation Fonctions, expressions, valeurs

2. Raisonner sur les actions

A. Eléments de cours

a) Terminologie et éléments de langage

a.1. Variable

Les variables ont pour rôle de modéliser les informations des problèmes que l'on traite.

- Une *variable* exprime une *association* entre un nom et des valeurs : le nom de la variable est fixé, sa valeur peut changer au cours de l'exécution du programme considéré. Une variable peut ainsi être vue comme l'association d'un nom et d'une suite de valeurs.
- Le *type d'une variable* est celui des valeurs qu'elle peut prendre.
- L'*état d'une variable* à un instant donné est le couple <nom, valeur> à cet instant.
- Les valeurs sont décrites à l'aide du langage des expressions et des fonctions.

a.2. Constante

Une *constante* est l'association (constante) d'un nom et d'une seule valeur.

a.3. Lexique

Un lexique donne la signification d'un ensemble de noms définis pour réaliser un algorithme : à chaque nom sont associés son type et son rôle dans l'analyse du problème considéré.
Un lexique comporte des noms de types, de constantes, de variables, d'actions et de fonctions. Ces noms sont regroupés selon leurs rôles pour la solution.

a.4. Portée des noms d'un lexique

La portée d'un nom est le texte dans lequel ce nom est connu avec la signification indiquée dans le lexique.

- Dans le cas d'un *lexique global*, la portée des noms est tout le texte du programme.
- Dans le cas d'un *lexique local* à une action (ou une fonction), la portée des noms est le texte de l'action (ou de la fonction).

a.5. Affectation

La syntaxe de l'affectation est : variable $\longleftarrow$ expression
X $\longleftarrow$ 3 se lit "X prend pour valeur 3"
Le nom doit apparaître dans le lexique et le type de l'expression doit être celui de la variable.

a.6. Entrées/Sorties

La syntaxe de la **lecture** est : Lire(variable). La valeur lue doit être du type de la variable.
La syntaxe de l'**écriture** est : Ecrire(expression).

a.7. Composition séquentielle

Elle permet de décrire une action A en termes d'actions plus élémentaires A1, A2, ...et d'exprimer que ces actions doivent être exécutées l'une après l'autre (en séquence) selon un ordre indiqué.

Schéma syntaxique :

```
A :
   A1
   A2
   A3
```
ou bien
```
A :
   A1 ; A2 ; A3
```
ou encore
```
A :
   A1 ; A2
   A3
```

a.8. Composition conditionnelle

Elle permet de décrire une action A en termes d'actions élémentaires A1, A2, ...et d'exprimer que l'exécution de A consiste en l'exécution d'une seule des actions A1, A2, ...selon des conditions indiquées. Les cas doivent couvrir le domaine et s'exclure mutuellement.
Schéma syntaxique :

```
A :
   selon { liste des noms de variables dont dépend l'analyse par cas }
      C1  : A1
      C2  : A2
      C3  : A3
```

Les Ai sont des actions décrites dans le langage des actions, et les Ci sont des conditions décrites par des expressions à valeur booléenne. Les Ci décrivent une partition du domaine et s'excluent mutuellement. L'ordre dans lequel les cas sont exprimés n'est pas pertinent.

a.9. Autres formes de compositions conditionnelles

Une autre forme possible du selon est la suivante :

```
A : selon
   EB1  : A1
   EB2  : A2
   EB3  : A3
   sinon : A4
```

dans laquelle sinon correspond au cas *{ non EB1 et non EB2 et non EB3 }* pour respecter l'exclusion des cas et la couverture du domaine.

La forme si C alors A1 sinon A2 permet d'exprimer un choix entre deux actions selon une condition C. Elle équivaut à :

```
selon
   C : A1
   non C : A2
```

La forme si C alors A en est un abrégé, propre à l'expression actionnelle, permettant d'exprimer à quelle condition C, l'action A est exécutée.

a.10. Composition itérative

Elle permet de décrire une action A en termes d'une action élémentaire A1, et d'exprimer que cette action doit être exécutée plusieurs fois au cours de l'exécution de A, tant qu'une certaine condition sur l'état du système est vérifiée. Nous utilisons quatre formes de composition itérative : tant que, répéter, répéter n fois, pour ...allant de ...à ....
Schéma syntaxique de la forme tant que :

```
A0                                          { initialisation de l'itération }
tant que CC                                 { condition de continuation }
   A1                                       { corps de l'itération }
A2                                          { sortie de l'itération }
```

CC est une condition, exprimée sous forme d'une expression à valeur booléenne, qui porte sur les *variables* du programme, c'est-à-dire les informations qui sont éventuellement modifiées (qui varient) pendant son exécution. A2 représente un exemple de contexte d'utilisation du résultat de cette itération. L'exécution d'une action itérative tant que engendre l'exécution répétitive du corps de l'action A1 tant que la condition de continuation CC est vraie. Si cette condition est fausse au départ, l'action se termine immédiatement.

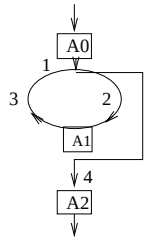


Figure 2.1 – Schéma d'exécution de la composition itérative tant que

Ce schéma met en évidence que les quatre points d'observation apparaissant dans le schéma syntaxique correspondent à un même point à l'exécution.

a.11. Formes de composition répéter

Nous les définissons en termes de la forme tant que (CA est une condition d'arrêt).

```
A0                                A0
répéter                           A1
   A1        est équivalent à     tant que non CA
jusqu'à CA                          A1
A2                                A2
```

```
                                  i ⟵ 1
répéter n fois                    tant que i ≤ n
   A          est équivalent à       A
                                     i ⟵ i + 1
```

a.12. Parcours d'un intervalle d'entiers

On peut écrire une composition itérative avec la syntaxe suivante :

```
pour k allant de i à j [, pas p] : A
```

où i et j sont des expressions à valeurs entières, k est une variable de type entier sur [i. . .j] et A une action qui peut faire référence aux valeurs de i, j et k, mais sans les modifier. k est local à l'itération : il n'apparaît pas dans le lexique. p est un entier positif ou négatif. S'il n'est pas mentionné, sa valeur par défaut est 1.

Définition en termes de la forme tant que, pour p > 0 et j ≥ i :

```
                                  k ⟵ i
pour k allant de i à j, pas p     tant que k ≤ j
   A          est équivalent à       A
                                     k ⟵ k + p
```

Définition en termes de la forme tant que, pour p > 0 et j ≥ i :

```
                                  k ⟵ j
pour k allant de j à i, pas -p    tant que k ≥ i
   A          est équivalent à       A
                                     k ⟵ k - p
```

a.13. Fonctions et actions : spécification, statut des paramètres

Une action décrit la manière de modifier un état alors qu'une fonction calcule un résultat. Il en découle une différence dans le mode de description.

Une *fonction est définie* en décrivant son profil (sa signature), c'est-à-dire la liste de ses paramètres et leur type, ainsi que le type du résultat calculé par la fonction.

Une *action est définie* en identifiant les informations qu'elle met en jeu (pertinentes pour l'action), c'est-à-dire la liste de ses paramètres.

Nous classifions les paramètres selon l'abstraction qu'ils représentent et le traitement de l'information qu'ils expriment :

- Un *paramètre donnée* est une abstraction de la *valeur* de l'expression considérée. Un paramètre donnée est uniquement consulté lors de l'exécution de l'action, jamais modifié.
- Un *paramètre résultat* est une abstraction du *nom* de l'expression considérée. Un paramètre résultat est uniquement modifié lors de l'exécution de l'action, jamais consulté.
- Un *paramètre donnée-résultat* est une abstraction du *nom* de la variable considérée. Un paramètre donnée-résultat peut être à la fois modifié et consulté lors de l'exécution de l'action.

Lors de l'utilisation, les paramètres effectifs précisent la valeur, le nom ou l'information qui permettent d'instancier la fonction ou l'action : dans le cas d'un paramètre donnée, le paramètre effectif *est une expression* ; dans le cas d'un paramètre résultat ou d'un paramètre donnée-résultat, le paramètre effectif *doit être un nom de variable.*

Une fonction ne peut être utilisée que dans le contexte d'une expression.

a.14. Transformation d'une fonction en une action

Sur un exemple simple, nous illustrons une technique systématique qui consiste à introduire un paramètre résultat. Soit les fonctions suivantes :

```
MaxDeux : fonction (a, b : entier) ⟶ entier          { maximum de deux valeurs }
   MaxDeux (a, b) :
      retour : si a>b alors a sinon b

MaxTrois : fonction (a, b, c : entier) ⟶ entier      { maximum de trois valeurs }
   MaxTrois (a, b, c) :
      retour : MaxDeux (MaxDeux (a, b), c)
```

```
{ Exemple de contexte d'utilisation }
    U, V, W : entier
    Lire (U, V, W)
    Écrire (MaxTrois (U, V, W))
```

On peut transformer ces fonctions en actions de la manière suivante : à la fonction MaxTrois correspond une action CréerMaxTrois ayant trois paramètres données et un paramètre résultat ; de même à la fonction MaxDeux correspond une action CréerMaxDeux ayant deux paramètres données et un paramètre résultat.

```
CréerMaxDeux : action (donnée a, b : entier ; résultat r : entier)
    { Construit dans r le maximum de a et b }
    CréerMaxDeux (a, b, r) :
        r ⟵ si a>b alors a sinon b

CréerMaxTrois : action (donnée a, b, c : entier ; résultat r : entier)
    { Construit dans r le maximum de a, b et c }
    CréerMaxTrois (a, b, c, r) :
        d : entier
        CréerMaxDeux (a, b, d) ; CréerMaxDeux (d, c, r)

{ contexte d'utilisation }
    U, V, W : entier ; X : entier
    Lire (U, V, W) ; CréerMaxTrois (U, V, W, X) ; Écrire (X)
```

b) Raisonner sur les états

b.1. Assertions

En français, une *assertion* est "une proposition que l'on avance et qu'on soutient comme vraie" (dictionnaire Petit Robert). Encore faut-il qu'elle soit corroborée, justifiée par les faits.

Dans le contexte de l'algorithmique et de la programmation :

- une assertion énonce une *propriété de l'état* d'une variable ou d'un ensemble de variables à un instant de l'exécution d'un programme.
- les assertions sont utilisées pour *annoter le texte des programmes* : elles énoncent les propriétés des variables lorsque l'exécution du programme passe au point du texte considéré.
- les assertions servent en particulier à spécifier les actions par une *pré-condition* et une *post-condition* : la pré-condition est une hypothèse sur l'état initial de l'action ; la post-condition est une propriété garantie à l'issue de l'exécution de l'action dans la mesure où la pré-condition est satisfaite.
- les assertions sont décrites en langue naturelle ou dans un formalisme adéquat (logique par exemple) en termes de noms de variables.

b.2. Points d'observation

On présente dans cette partie les éléments de raisonnement sur les états des diverses compositions du langage, basés sur des points d'observation précis et l'utilisation d'assertions à ces points d'observation.

Composition séquentielle
Raisonnement sur les états pour A : A1 ; A2, figure 2.2.

Chaque rectangle est l'abstraction d'un schéma d'exécution. P est la précondition de A et P' est sa postcondition. De même, Pi et P'i sont les pré et post conditions de l'action Ai. La cohérence de la composition s'exprime alors par $P \Rightarrow P1$ et $P'1 \Rightarrow P2$ et $P'2 \Rightarrow P'$.

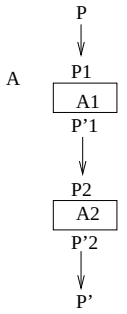


Figure 2.2 – Composition séquentielle : raisonnement sur les états

Composition conditionnelle
Raisonnement sur les états pour

```
A : selon { liste des noms de variables dont dépend l'analyse par cas }
    EB1  : A1
    EB2  : A2
    EB3  : A3
```

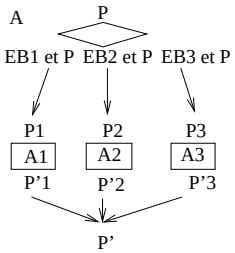


Figure 2.3 – Composition conditionnelle : raisonnement sur les états

Chaque rectangle est l'abstraction d'un schéma d'exécution. P est la précondition de A et P' est sa postcondition. De même, Pi et P'i sont les pré et post conditions de l'action Ai. La cohérence de la composition s'exprime alors par $\forall i \cdot (EBi \wedge P \Rightarrow Pi)$ et $(P'i \Rightarrow P')$.

Composition itérative
Pour raisonner sur les états d'une forme tant que, on a besoin des points d'observation suivants. Dans la suite, on fera toujours référence aux points d'observation en commentaire sous la forme {*P...*} :

```
A0                                              { initialisation de l'itération }
{ point d'observation n0 1 }
tant que CC                                     { condition de continuation }
    { point d'observation n0 2 }
    A1                                          { corps de l'itération }
    { point d'observation n0 3 }
{ point d'observation n0 4 }
A2
```

Le raisonnement est basé ensuite sur la notion d'invariant présentée dans le paragraphe 2.A.b.3..

Fonctions et actions
Pour raisonner sur des fonctions ou des actions, il faut qu'elles soient correctement spécifiées.

Une *fonction est spécifiée* en définissant la relation entre son domaine et son codomaine.

Une *action est spécifiée* en décrivant ses états initial et final en termes de ses paramètres et de leurs relations. La description de l'état initial est interprétée comme une condition *requise* pour pouvoir utiliser l'action, on parle de *pré-condition*. La description de l'état final est interprétée comme une condition *garantie* à la fin de (l'exécution de) l'action dans la mesure où l'état initial a été respecté lors de l'appel, on parle de *post-condition*.

La *réalisation d'une action* décrit le moyen de passer de l'état initial à l'état final en termes du langage des actions.

Dans le *cas d'une fonction*, on décrit une valeur en termes du langage des expressions ou du langage des actions.

b.3. Notion d'invariant

Définition : une assertion **P** placée au point d'observation n°2 est un *invariant* de l'itération si elle satisfait l'implication suivante : *CC et P a la valeur vrai au point 2* ⇒ *P a la valeur vrai au point 3* (après l'exécution du corps de l'itération A1).

Soit **P** un invariant de l'itération. Si **P** est satisfaite au point n°1 et si l'exécution se termine alors on peut déduire qu'au point n°4 l'assertion **non CC et P** est satisfaite.

Remarque :
Si A est un invariant et si B est un invariant, alors : "A et B" est un invariant ; "A ou B" est un invariant. La démonstration est évidente d'après la définition et les propriétés de l'implication.

Par contre, si "A et B" est un invariant, on ne peut rien dire sur l'invariance de A, ni sur l'invariance de B. De même, si "A ou B" est un invariant, on ne peut rien dire sur l'invariance de A, ni sur l'invariance de B.

B. Exemples d'exercices rédigés

Exemple E2.1 : Échange de deux valeurs

Réaliser un algorithme d'échange des valeurs de deux variables. Justifier sa correction à l'aide d'assertions.

```
lexique
    X, Y : entier
algorithme
    Lire (X, Y)      { On pose X=x0 et Y=y0 }
    X ⟵ X + Y { X=x0 + y0 et Y=y0  }
    Y ⟵ X - Y  { X=x0 + y0 et Y=x0  }
    X ⟵ X - Y  { X=y0 et Y=x0 }
    Écrire (X, Y)
```

Exemple E2.2 : Maximum de trois entiers

Réaliser un algorithme qui affiche le maximum de trois entiers donnés. On suppose que ces entiers sont distincts deux à deux. Donner plusieurs solutions.

a, b, c : entier

action conditionnelle	expression conditionnelle

solution 1

examen du domaine du résultat

```
Lire (a, b, c)                  |  Lire (a, b, c)
selon a, b, c                   |  Écrire (selon a, b, c
    a>b et a>c  : Écrire (a)    |          a>b et a>c  : a
    b>a et b>c  : Écrire (b)    |          b>a et b>c  : b
    c>a et c>b  : Écrire (c)    |          c>a et c>b  : c)
```

solution 2

réduction du domaine des données et introduction d'un nom intermédiaire

```
m : entier                                  |  Lire (a, b, c)
Lire (a, b, c)                              |  Écrire (
si a > b alors m ⟵ a sinon m ⟵ b            |     soit m = si a>b alors a sinon b
si m > c alors Écrire (m) sinon Écrire (c)  |     dans si m>c alors m sinon c )
```

Exemple E2.3 : Classement de trois entiers

Réaliser un algorithme qui étant donnés 3 entiers strictement positifs, affiche une liste ordonnée sans redondance leur correspondant. Exemples : données : 3, 1, 6 résultats : 1, 3, 6 données : 5, 1, 1 résultats : 1, 5 données : 2, 2, 2 résultat : 2

On réduit le problème en isolant les cas d'égalité dans les données et on introduit deux actions intermédiaires correspondant aux cas de deux et trois valeurs distinctes :

```
lexique
    a, b, c : entier > 0
    Saisir : action (résultat X : entier > 0)
        { Associe l'entier lu à X }

    AfficherEnOrdre2 : action (donnée a, b : entier > 0)
        { Affichage en ordre croissant des valeurs des deux entiers donnés. Pré-condition : les deux entiers
        sont différents }
```

```
AfficherEnOrdre3 : action (donnée a, b, c : entier > 0)
    { Affichage des trois valeurs en ordre croissant. Pré-condition : les trois entiers sont distincts deux à deux. }
algorithme
    Saisir (a) ; Saisir (b) ; Saisir (c)
    si a = b alors
        si a = c alors Écrire (a)                    { a=b et a=c }
        sinon AfficherEnOrdre2 (a, c)                { a=b et a≠c }
    sinon
        si a = c alors AfficherEnOrdre2 (b, c)       { a≠b et a=c }
        sinon
            si b = c alors AfficherEnOrdre2 (a, b)   { a≠b et a≠c et b=c }
            sinon AfficherEnOrdre3 (a, b, c)         { a≠b et a≠c et b≠c }
```

Réalisation des actions intermédiaires

Pour l'action Saisir, Cf. exemple **E2.4**.

```
AfficherEnOrdre2 (a, b) :
    si a < b alors Écrire (a, b) sinon Écrire (b, a)
```

Pour l'action AfficherEnOrdre3, on introduit une action intermédiaire Classer3V.

```
Classer3V : action (donnée-résultat X, Y, Z : entier > 0)
    { État initial : posons X=x0 et Y=y0 et Z=z0, distincts deux à deux ; État final : posons X=x1 et
    Y=y1 et Z=z1 : <x1,y1,z1> est une permutation de <x0,y0,z0> et x1<y1<z1 }

AfficherEnOrdre3 (a, b, c) :
    lexique
        p, s, t : entier > 0
    algorithme
        p ⟵ a ; s ⟵ b ; t ⟵ c
        Classer3V (p, s, t) ; Écrire (p, s, t)
```

Réalisation de l'action Classer3V
On place la plus petite valeur, puis on place les deux autres. On introduit une action intermédiaire d'échange :

```
Échanger : action (donnée-résultat X, Y : entier)
    { État initial : posons X = x0 et Y = y0 ; État final : X = y0 et Y = x0 }
```

et l'on obtient :

```
Classer3V (X, Y, Z) :
{ On pose X = x0 et Y = y0 et Z = z0 }
    selon X, Y, Z
        X<Y et X < Z : { rien }              { X=x0 et Y=y0 et Z=z0 et x0<y0 et x0<z0 }
        Y<X et Y < Z : Échanger (X, Y)       { X=y0 et Y=x0 et Z=z0 et y0<x0 et y0<z0 }
        Z<X et Z < Y : Échanger (X, Z)       { X=z0 et Z=x0 et Y=y0 et z0<y0 et z0<x0 }
    { posons X=x', Y=y', Z=z' : x'<y' et x'<z' et <x',y',z'> est une permutation de <x0,y0,z0> }
    si Y > Z alors Échanger (Y, Z)
{ posons X=x1 et Y=y1 et Z=z1 : x1<y1<z1 et <x1,y1,z1> est une permutation de <x0,y0,z0> }
```


Exemple E2.4 : Plus de 10 que de 15

Étant donnés deux entiers strictement positifs E1 *et* E2 *et une séquence* S *d'entiers strictements positifs, réaliser un algorithme qui détermine si* S *comporte plus d'exemplaires de* E1 *que d'exemplaires de* E2. *Les entiers de la séquence* S *sont saisis interactivement et traités à la volée. Une valeur de marque (par exemple 0) indique la fin de la saisie. On fera apparaître un invariant d'itération permettant de valider la solution.*

On construit une itération : au fur et à mesure de la saisie, on maintient les valeurs des nombres d'exemplaires de E1 et E2 déjà rencontrés. Dans les commentaires, pg désigne la partie gauche de la séquence par rapport à l'élément en cours de traitement, c'est-à-dire la séquence des éléments déjà traités. NbEx désigne une fonction qui, à deux entiers E1 et E2 et une séquence d'entiers S, associe un couple formé des nombres d'exemplaires de E1 et E2 dans S.

```
lexique
    marque : constante 0 de type entier
    E1, E2 : entier ≥ 0                                                    { données }
    nb1, nb2 : entier ≥ 0                                                  { pour élaborer le résultat }
    EC : entier ≥ 0
        { entier courant : celui que l'on vient de saisir, éventuellement la marque }
    Saisir : action (résultat : entier ≥ 0)
        { saisie d'un entier ≥0 et association de sa valeur au nom donné }
algorithme
    Saisir(E1) ; Saisir(E2)
    si E1 = E2 alors Écrire("non")
    sinon { E1 ≠ E2 }
        { état initial : aucun entier n'a été traité : pg = [] }
        nb1 ⟵ 0 ; nb2 ⟵ 0                                                  { NbEx (E1,E2,[]) = <0, 0> }
        Écrire ("donnez une suite de nombres >0 (", marque, " pour indiquer la fin)" )
        Saisir (EC)                                                        { obtention du premier élément }
        tant que EC ≠ marque                                               { la marque n'est pas traitée }
            { Invariant : <nb1, nb2>= NbEx (E1,E2,pg) }
            selon E1, E2, EC
                EC ≠ E1 et EC ≠ E2 :                                       { nb1 et nb2 ne doivent pas être modifiés }
                EC = E1 : nb1 ⟵ nb1 + 1                                    { nb2 ne doit pas être modifié }
                EC = E2 : nb2 ⟵ nb2 + 1                                    { nb1 ne doit pas être modifié }
            { <nb1, nb2>= NbEx (E1,E2,pg •EC) }
            Saisir (EC)                                                    { obtention du prochain élément }
        { état final : tous les entiers ont été traités, sauf la marque ; pg est la séquence donnée. D'après
        l'invariant, nb1 et nb2 sont les nombres d'exemplaires de E1 et E2 dans la séquence saisie. }
        si nb1 > nb2 alors Écrire("oui") sinon Écrire("non")
```

Réalisation de l'action Saisir

```
Saisir (X) :
    lexique
        Z : entier
    algorithme
        Écrire (" ?") ; Lire (Z)
        tant que Z < 0
            Écrire ("nombre < 0, recommencez") ; Lire (Z)
        X ⟵ Z
```

Exemple E2.5 : A propos d'invariant

Étant donnée une séquence **SD** *de* **LD** *entiers, on étudie un algorithme dont l'effet est d'afficher une séquence* **SR** *de* **LR** *entiers. Les entiers de* **SD** *sont traités au fur et à mesure de leur acquisition de manière à produire successivement les entiers de* **SR :**

```
lexique
   LD : entier ≥ 0                          { longueur de SD, donnée }
   E : entier                               { entrée courante lors de la lecture de SD }
   LR : entier ≥ 0                          { longueur de SR, résultat }
   S : entier                               { sortie courante lors de la production de SR }
   i : entier ≥ 0                           { pour contrôler la fin de la lecture de SD }
algorithme
(1) LR ⟵ 0 ; i ⟵ 0 ; S ⟵ 0
    {*P1*}
(2) tant que i ≠ LD
        {*P2*}
(3)     Lire(E)
        {*P3*}
(4)     si E > 0 alors
                {*P4*}
(5)             S ⟵ S + E
(6)             LR ⟵ LR+1
(7)             Écrire (S)
(8)     i ⟵ i + 1
        {*P5*}
    {*P6*}
```

L'objet de l'exercice est de déterminer l'effet de cet algorithme. Pour cela, on étudie des propriétés des variables de l'algorithme aux points d'observation {*P...*}.

Q1. *Démontrer que l'exécution du programme se termine toujours si* $\mathbf{LD} \geq \mathbf{0}$.

Si $\mathbf{LD} = \mathbf{0}$, le corps de l'itération n'est pas exécuté. Si $\mathbf{LD} > \mathbf{0}$, l'itération se termine : en effet, **i** étant initialisé à 0 et **i** augmentant de 1 une fois à chaque itération (ligne 8), **i** atteint nécessairement la valeur de **LD**.

Q2. *Caractériser les valeurs de* **SD***, pour lesquelles* **SR** *est vide.*

Si **SD** est vide, **LD**=**0** et l'itération est une action vide. A l'état final, **LR**=**0** et **SR** est vide. Sinon, l'instruction **Écrire** (ligne 7) n'est exécutée que si l'entrée courante **E** est strictement positive. Par conséquent, $\mathbf{SR} = [] \iff$ **LD**=**0 ou** $\forall \mathbf{e} \in \mathbf{SD}, \mathbf{e} \leq \mathbf{0}$.

Q3. *Soit l'assertion* **A1** *: $i \geq 0$ et $LR \geq 0$ et i éléments de SD ont déjà été lus et parmi eux LR sont strictement positifs. Démontrer que* **A1** *est un invariant de l'itération.*

Hypothèse : en **P2**, **A1** est vraie et i ≠ LD.
Démonstration : **i** est incrémenté (ligne 8) et **LR** ne peut qu'augmenter (ligne 6). Donc *$i \geq 0$ et $LR \geq 0$* reste vraie. Par ailleurs, la lecture d'un élément de **SD** (ligne 3) est suivie d'une incrémentation de **i** (ligne 8) ; l'incrémentation de **LR** (ligne 6) a lieu si l'entrée **E** est strictement positive.
Conclusion : **A1** est vraie et i ≠ LD en **P2** $\implies$ **A1** est vraie en **P5**.

Q4. *Soit l'assertion* **A2** *: tous les entiers de SD ont été lus et LR est le nombre d'entiers strictement positifs de SD. Démontrer que* **A2** *est vraie en* **P6**.

A1 est vraie en **P1** et **A1** est un invariant de l'itération (**Q3**) : par conséquent **A1** est vraie et non(i ≠ LD) en **P6**. Comme **i** = **LD** en **P6**, on peut conclure **A2** est vraie en **P6**.

Q5. *Soit l'assertion* **A3** *: $S \geq 0$ et S est la somme des éléments strictement positifs de SD déjà lus (0 s'il n'y en a aucun). Démontrer que* **A3** *est un invariant de l'itération.*

Hypothèse : en **P2**, **A3** est vraie et i ≠ LD.
Démonstration : on considère deux cas en **P3** :
- **E** ≤ **0** : aucun élément strictement positif supplémentaire n'est lu et **S** ne change pas.
- **E** > **0** : **E** est un nouvel élément strictement positif et **S** est augmentée de la valeur de **E** (ligne 5).

Conclusion : **A3** est vraie et i ≠ LD en **P2** $\implies$ **A3** est vraie en **P5** .

Q6. *Décrire l'effet de l'algorithme en donnant la valeur de* **SR** *en fonction de* **SD***. Justifier la réponse.*

- L'exécution se termine (**Q1**)
- En **P6**, **LR** est le nombre d'entiers positifs de SD (**Q4**)
- **LR** vaut 0 au départ (ligne 1) et augmente de 1 pour chaque appel de l'instruction **Écrire** (ligne 7). Ainsi en **P6**, **LR** est le nombre d'appels de **Écrire**, c'est-à-dire à dire la longueur de **SR**.
- **A3** est vraie en **P1**, **A3** est un invariant (**Q5**) et **S** n'est pas modifiée ligne 8 : toute valeur écrite est donc la somme des entiers strictement positifs des éléments déjà lus.

Conclusion : **LR**, longueur de **SR**, est le nombre d'entiers strictement positifs de **SD** et $\forall$ **k** $\in$ **[1. . .LR]** le k^{ème} élément de **SR** est la somme des **k** premiers éléments strictement positifs de **SD**.

C. Exercices

a) Observation des formes de composition

E2.6 : Composition conditionnelle
On considère deux actions nommées A1 et A2 réalisées de la manière suivante :

```
A1 : si EB alors A3 sinon A4
A2 : si EB alors A3 ; si (non EB) alors A4
```

A3 et A4 sont deux actions et EB est une expression à valeur booléenne : toutes les trois sont décrites en termes de variables qui appartiennent au lexique du contexte dans lequel A1 ou A2 sont utilisées.
Donner deux exemples pour A3, A4 et EB :
- un pour lequel A1 et A2 sont équivalents
- un pour lequel A1 et A2 ne sont pas équivalents

E2.7 : Composition itérative
Exprimer la construction pour ... allant de ... à ..., pas ... (parcours d'un intervalle d'entiers) en termes de la construction tant que.

b) Observation d'algorithmes

E2.8 : A propos d'échanges

On spécifie une action d'échange des valeurs associées à deux variables :

Échanger : action (donnée-résultat X, Y : entier)
{ état initial :indifférent.On pose X = x0 et Y = y0 ; état final : X = y0 et Y = x0 }

Q1.

- L'appel Échanger(X+Y,Z) est incorrect. Expliquer pourquoi.
- Soit l'extrait d'algorithme suivant :

  A, B : entier
  A ⟵ 23 ; Lire(B)
  Échanger(A, B)
  { A > 0 }

  La post-condition (assertion finale) $A > 0$ n'est pas toujours vraie. Expliquer cette situation, en observant en particulier la spécification de l'action Échanger ? Donner un contre-exemple simple illustrant la réponse.
- On considère l'extrait d'algorithme suivant :

  Échanger(A, B)
  { A > 0 }

  Quelle est la plus faible pré-condition (assertion) que l'on doit placer avant l'action de Échanger pour que la post-condition $A > 0$ soit toujours satisfaite ?

Q2. On considère l'action A réalisée de la manière suivante :

A(U, V, W) :
{ état initial : indifférent. On pose U = u0, V = v0 et W = w0 }
Échanger(U, V)
{ état intermédiaire : ●●●●● }
Échanger (V, W)
{ état final : ●●●●● }

En exploitant la spécification de l'action Échanger,

- Décrire l'état intermédiaire et l'état final,
- Donner la spécification de l'action : nom, paramètres avec leur statut (donnée, résultat, donnée-résultat) et leur type, effet (par une phrase en français).

E2.9 : Classement de trois valeurs (E2.3 suite)

On étudie un principe de réalisation d'une action Classer3V spécifiée comme suit :

Classer3V : action (donnée-résultat A, B, C : entier)
{ e.i : indifférent. On pose A = a0, B = b0, C = c0
e.f : on pose A = a1, B = b1, C = c1 :
a1 ≤ b1 ≤ c1 et <a1, b1, c1> est une permutation de <a0, b0, c0> }

Pour réaliser cette action, on part d'un principe en deux étapes, exprimé par l'ébauche d'algorithme suivante :

Classer3V(A, B, C) :
{ e.i. : indifférent. On pose A = a0, B = b0, C = c0 }
étape 1 : Classer A et B
{ état intermédiaire : On pose A = a', B = b', C=c' ●●●●● }
étape 2 : Placer C par rapport à A et B
{ e.f : on pose A = a1, B =b1, C= c1 ;
a1 ≤ b1 ≤ c1 et <a1, b1, c1> est une permutation de <a0, b0, c0> }

Q1. Pour préciser l'effet de la première étape, *indépendamment de la manière avec laquelle elle sera réalisée*, on doit décrire l'état intermédiaire souhaité à l'issue de son exécution (post-condition de la première étape).

- Pour cela, on propose l'assertion suivante :
  { état intermédiaire : On pose A = a', B = b' : a' ≤ b' }.
  Cette assertion est insuffisante pour caractériser l'état intermédiaire nécessaire à une réalisation correcte de l'action Classer3V. Expliquer pourquoi et donner un contre-exemple sous forme d'une réalisation de l'étape 1 satisfaisant l'assertion mais conduisant à une réalisation incorrecte de Classer3V. La réalisation proposée en contre-exemple doit être la plus simple possible.
- Même question pour l'assertion :
  { état intermédiaire : }
  { On pose A = a', B = b' : a' ≤ b' et <a', b'> est une permutation de <a0, b0> }
- Donner une assertion suffisante et réaliser l'action Classer3V en conséquence. On pourra utiliser l'action Échanger de l'exercice **E2.8**.

Q2. Donner les nombres minimal et maximal d'appels de l'action Échanger que peut engendrer l'algorithme obtenu. Donner les cas où il y a deux appels. De manière générale, donner pour chacun des cas possibles, le nombre d'appels de l'action Échanger engendrés par l'algorithme.

E2.10 : Lecture d'un algorithme itératif

On considère l'algorithme suivant :

```
Bit : type entier sur [0,1]
B : bit ; n : entier
Lire (B) ; Lire (n) ; ALaLigne
pour i allant de 1 à n
    répéter i fois : AfficherCaractère (si B = 1 alors '1' sinon '0')
    ALaLigne ; B ⟵ 1-B
```

AfficherCaractère a pour effet d'afficher le caractère donné à la position courante du curseur d'affichage. ALaLigne place le curseur d'affichage au début de la ligne suivant la ligne courante.

Q1. Expliquer brièvement l'effet de l'algorithme : l'exprimer par une phrase en français et/ou par des exemples significatifs. De plus, préciser l'état des variables et de l'écran d'affichage à la fin de l'exécution de l'algorithme.

Q2. Donner en fonction de la valeur lue pour n le nombre d'appels engendrés par l'algorithme, de l'action ALaLigne, puis de l'action AfficherCaractère.

E2.11 : Algorithme mystère 1

On étudie l'algorithme donné ci-dessous. Des points d'observation **{*1*}**, **{*2*}**, ... y ont été placés sous forme de commentaires.

```
Marque : constante 0 de type entier ; Secret, Résultat, EC : entier
Écrire("Taper un entier positif, ou ", Marque , " pour terminer : ") ; Lire(EC)
tant que EC < Marque
    Écrire("Valeur incorrecte. Taper un entier positif, ou ", Marque, " pour terminer : ") ; Lire(EC)
Secret ⟵ 0
Résultat ⟵ 0
```

```
{*1*}
tant que EC ≠ Marque
   selon Secret, EC {*2*}
      Secret > EC :
      Secret = EC : Résultat ⟵ Résultat + 1 {*3*}
      Secret < EC : Secret ⟵ EC ; Résultat ⟵ 1
   Écrire("Taper un entier positif, ou ", Marque, " pour terminer : ") ; Lire(EC)
   tant que EC < Marque
      Écrire("Valeur incorrecte. Taper un entier positif, ou ", Marque, " pour terminer : "") ; Lire(EC)
{*4*}
si Résultat > 0 alors Écrire("La valeur du nombre Secret est ", Secret )
Écrire("La valeur du résultat est ", Résultat )
```

Q1. Trace d'une exécution
On suppose que la séquence de valeurs suivante est entrée au clavier :

-2, 3, 1, -1, 3, 6, 2, 6, 6, 4, 0, 6

Donner une trace de l'exécution correspondante de l'algorithme, sous forme d'un tableau comportant les valeurs prises successivement par les variables EC, Secret et Résultat, lors de chaque passage à chacun des points d'observation placés dans l'algorithme.

Q2. Observation de l'algorithme
- En fonction de la taille des données et de leur nature, donner le nombre d'exécutions de l'action conditionnelle suivant le point marqué **{*2*}** dans l'algorithme.
- Expliquer l'effet de l'algorithme pour des données quelconques, en particulier dans les cas limites, et préciser le rôle de chaque variable.
- Donner un invariant de l'itération (point 2) permettant de déduire une assertion la plus forte possible au point 4.

c) A propos d'invariants

E2.12 : Algorithme mystère 2
On considère l'algorithme suivant (les commentaires indiquent des points d'observation) :

```
n : entier > 0                                                    { donnée }
i, j : entier > 0
S : entier > 0
Lire (n)
i ⟵ 1 ; j ⟵ 3 ; S ⟵ 1
{*1*}
tant que i ≠ n
   {*2*}
   S ⟵ S + j ; i ⟵ i + 1 ; j ⟵ j + 2
   {*3*}
{*4*}
Écrire (S)
```

On considère l'assertion P suivante caractérisant une propriété des variables S, i et j :

P(S,i, j) : $j = 2*i + 1$ et $S = i^2$

Démontrer que cette assertion est un invariant de l'itération. Démontrer qu'elle est vraie au point **1**. En déduire une assertion la plus forte possible au point **4**, en fonction de la donnée n.

E2.13 : Algorithme mystère 3
On étudie l'algorithme suivant :

```
x, y : entier ≥ 0                                          { données }
u, v, a : entier ≥ 0                          { intermédiaires. a est le résultat final }
Lire(x,y) ; a ⟵ 0 ; u ⟵ x ; v ⟵ y
tant que v ≠ 0
   si v reste 2 ≠ 0 alors a ⟵ a + u
   u ⟵ 2*u ; v ⟵ v quotient 2
```

Q1. Démontrer que l'algorithme se termine. Déterminer le nombre d'exécutions du corps de l'itération en fonction des données x et y. Vérifier pour quelques données et en exécutant l'algorithme à la main que la valeur finale de la variable a est le produit de x et y.

Q2. Démontrer que u * v + a = x * y est un invariant de l'itération. En déduire une assertion la plus forte possible à la fin de l'exécution.

d) Algorithmes à compléter

E2.14 : Les trois plus grandes valeurs
On considère une séquence d'entiers strictement positifs. On veut afficher les trois plus grandes valeurs de cette séquence, en ordre décroissant. Quelques exemples précisant le résultat attendu :
- *Résultat affiché : 25, 17, 9.* Les trois valeurs sont distinctes : chacune des deux premières (25 et 17) n'apparaît qu'une fois dans la séquence ; la troisième (9) y apparaît au moins une fois.
- *Résultat affiché : 19, 19, 16.* Deux des trois valeurs sont égales (19) : la séquence donnée contient exactement deux exemplaires de la plus grande (19) et au moins un exemplaire de la troisième (16).
- *Résultat affiché : 35, 35, 35.* Les trois valeurs sont égales (35). Cette valeur apparaît au moins trois fois dans la séquence donnée.

La séquence est saisie interactivement. Elle est terminée par l'entier 0 et on suppose qu'elle **contient au moins trois éléments** (sans compter la marque).

L'algorithme est un parcours de la séquence donnée. On introduit trois variables **P** (premier), **D** (deuxième) et **T** (troisième) qui contiennent les trois plus grandes valeurs déjà rencontrées, en ordre décroissant.
Compléter l'ébauche suivante de cet algorithme, en respectant les commentaires :

```
marque : constante 0 de type entier
P, D, T : entier > 0                                   { pour décrire le résultat }
EC : entier ≥ 0                                        { élément courant lors du parcours }
{ Classer les deux premiers nombres. }
Lire(P) ; Lire(D)
{ Posons P = p0, D = d0 }
si D > P alors D ⟷ P          { A ⟷ B dénote l'échange des valeurs des variables A et B }
{ Posons P = p1, D = d1 : p1 ≥ d1 et (p1,d1) permutation de (p0,d0) }
{ Placer par échange, le troisième nombre par rapport aux deux premiers. }
Lire(T)
{ Posons T = t0 }
si T > P alors ••••••••••
sinon si T > D alors ••••••••••
```

```
{ Posons P = p2, D = d2, T = t1 : p2 ≥ d2 ≥ t1 et (p2,d2,t1) permutation de (p0,d0,t0) }
Lire(EC)
tant que EC ≠ marque
    { P ≥ D et D ≥ T : le triplet courant (P,D,T) est le résultat pour les nombres déjà traités. }
        { Situer l'élément courant par rapport aux valeurs du triplet courant et, le cas échéant, modifier
        la valeur du triplet en conséquence. }
    selon EC, P, D, T
        ••••••••••
    Lire(EC)
Écrire(P, D, T)
```

E2.15 : Somme des chiffres d'un entier

Étant donné un entier E ≥ 0, on définit ici le *poids* de E comme étant l'entier somme des entiers correspondant aux chiffres (caractères '0', '1', '2', ...,'9') de la représentation décimale de E. Par exemple, 19 est le poids de l'entier 12556 ; 7 est le poids de l'entier 10123. On spécifie une fonction nommée Poids :

```
Poids : fonction (n : entier ≥ 0)  ⟶ entier ≥ 0
    { Somme des chiffres de la représentation décimale de l'entier donné }
```

Étant donnée une suite **non vide** d'entiers strictement positifs, on désire déterminer si tous les entiers de cette suite ont même poids.
La suite est saisie de manière interactive, la fin de la saisie étant déterminée par la donnée d'une marque de valeur 0.

Q1. On propose la solution suivante :

```
marque : constante 0 de type entier
EC : entier ≥ 0 ; B : un booléen ; P : entier ≥ 0

Lire(EC) ; P  ⟵ Poids(EC) ; B  ⟵ vrai ; Lire(EC)
tant que EC ≠ marque
    B  ⟵ Poids(EC) = P ; Lire(EC)
Écrire(si B alors "oui" sinon "non")
```

Cette solution est incorrecte. Quel est son effet ? Donner un invariant de l'itération permettant de justifier la réponse.

Q2. Pour corriger la solution précédente, compléter l'ébauche suivante de l'algorithme y compris l'invariant. Préciser le rôle de chaque variable. Donner le nombre d'exécutions du corps de l'itération.

```
EC : entier ≥ 0 ; B : booléen ; P : entier ≥ 0
Lire(EC) ; P  ⟵ Poids(EC) ; B  ⟵ •••••• ; Lire(EC)
tant que EC ≠ marque
    { Invariant : B a la valeur vrai  ⟺ •••••• }
    B  ⟵ •••••• ; Lire(EC)
Écrire(si B alors "oui" sinon "non")
```

Q3. Donner une autre solution, de manière à arrêter l'itération dès qu'il est possible de conclure.

Q4. Donner une réalisation itérative de la fonction Poids en utilisant les opérations quotient et reste. Donner le nombre d'appels de ces fonctions engendrés par un appel initial de la fonction Poids.

D. Machine-tracés

Nous définissons une machine-tracés, c'est-à-dire un ensemble de primitives élémentaires de tracés. Nous l'utilisons pour illustrer la notion d'action, le raisonnement sur les états et les diverses formes de composition d'actions.

a) Spécification de la machine-tracés

a.1. description de l'état de la machine

L'état de la machine-tracés est caractérisé par l'état de deux composants : l'*écran* et la *plume* de tracé. L'*écran* est un ensemble de points qui peuvent être éteints ou allumés. Une figure est matérialisée par des points allumés. Les points sont désignés à l'aide de coordonnées définies dans un système d'axes Ox, Oy dont l'origine, nommée centre (de l'écran) appartient au lexique de la machine-tracés. Un état de l'écran, c'est-à-dire son état à un instant donné, est caractérisé par les figures qu'il comporte. L'état particulier dans lequel tous les points sont éteints est nommé écran vide. Lorsque nous voulons indiquer qu'aucune contrainte particulière n'est imposée à l'écran, nous disons qu'il est dans un état *indifférent*.
La machine-tracés manipule une plume dont l'état est défini par trois attributs :

- Une *position d'écriture*, en abrégé *pe* : elle peut être *basse* ou *haute*.
- Une *position dans le plan*, en abrégé *pp* : c'est un point du plan. Le cas échéant, nous le décrivons par un couple de coordonnées.
- Un *cap* : c'est le sens de la plume, défini par un angle par rapport à l'axe Ox, orienté dans le sens trigonométrique. Cet angle est donné en degrés.

Lorsque nous voulons indiquer qu'aucune propriété particulière n'est attachée à l'état de la plume ou à l'un de ses attributs, nous disons qu'elle est dans un état *indifférent* ou que tel attribut a une valeur *indifférente*.
Les primitives sont regroupées selon le type de modifications qu'elles apportent à l'état de l'écran ou de la plume.

a.2. modification globale de l'écran et de la plume

- Vider vide l'écran et place la plume au centre, en position haute, cap Ouest-Est (0°).
- Effacer vide l'écran *sans modifier* l'état de la plume.

a.3. modification de la position d'écriture de la plume

- Lever met la plume en position "haute" (si elle n'y est pas déjà).
- Baisser met la plume en position "basse" (si elle n'y est pas déjà).

a.4. modification de la position de la plume dans le plan

- Avancer et Reculer, en abrégé Av et Re, font avancer ou reculer la plume d'une distance donnée, selon le cap de la plume. La distance est donnée sous forme d'un nombre entier ou réel, par exemple Av (50) ou Re (30.5). Notons que Av (-50) est équivalent à Re (50). L'interprétation exacte de l'unité de mesure dépend des caractéristiques physiques de l'environnement dans lequel la machine sera réalisée.
- Placer, en abrégé Plc, permet de mettre la plume en un point du plan donné. Par exemple, si un point est caractérisé par ses coordonnées cartésiennes, Plc (-10, 35.3) place la plume au point d'abscisse -10 et d'ordonnée 35.3.
- Les trois primitives Avancer, Reculer et Placer modifient la position de la plume, même si la plume est haute. Elles provoquent un tracé rectiligne dès que la plume est basse : elles modifient donc l'état de l'écran dans ce cas. Elles ne modifient jamais la position d'écriture ni le cap de la plume.

a.5. modification du cap de la plume

- TournerGauche et TournerDroite, en abrégé TrG et TrD, changent le cap selon une valeur d'angle donnée en degrés. Par exemple, TrG (35) augmente modulo 360 le cap de la plume de 35 degrés. TrD (45) le diminue modulo 360 de 45 degrés. TrG (35) a le même effet que TrD (325).
- Orienter, en abrégé Ort, donne à la plume le cap donné en degrés. Par exemple, Ort (90) donne à la plume le cap SudNord.

a.6. un exemple d'utilisation de la machine tracé

Exemple E2.16 : Tracé d'enveloppe

On veut réaliser un algorithme qui dessine une enveloppe comme dans la figure 2.4 : elle est composée d'un carré, de ses deux diagonales et d'un triangle équilatéral placé au-dessus du carré.

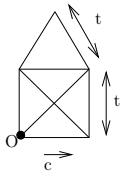


Figure 2.4 – Enveloppe

On spécifie l'action de tracé suivante :

Tenveloppe : action (donnée t : réel > 0)
{ Trace une enveloppe comme dans la figure 2.4 : t est la taille des côtés du carré et du triangle. La position et le cap initiaux de la plume correspondent au point **O** *et au cap* **c** *de la figure. Quel que soit l'état initial, à l'état final, l'écran ne comporte que la figure et la plume est haute dans le même cap qu'au départ. }*

Réaliser cette action de telle manière que le tracé de l'enveloppe soit effectué sans lever la plume et sans passer deux fois sur le même trait.

Une réalisation possible est :

```
Tenveloppe (t) :
    Vider ; Baisser
    répéter 4 fois
        Avt(t) ; TrG(90)
    TrG(45) ; Av(√2*t)
    TrG(75) ; Av(t)
    TrG(120) ; Av(t)
    TrG(75) ; Av(√2*t)
    TrG(45) ; Lever
```

b) Exercices

On s'intéresse ici au tracé de figures géométriques simples et à la manière de les composer dans des dessins plus élaborés. On s'intéresse notamment à la manière de caractériser une figure et de paramétrer l'action de tracé.

E2.17 : Tracé de carrés

On convient ici qu'un carré est défini par son sommet, sa taille et son cap : le *sommet du carré* est l'un des sommets du carré ; la *taille du carré* est la taille des côtés du carré ; le *cap du carré* est le cap du côté passant par le sommet du carré tel qu'un observateur placé au sommet du carré et regardant dans le sens de ce côté, voit tous les côtés à sa gauche et devant lui (voir figure 2.5).

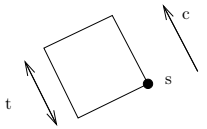


Figure 2.5 – Le carré défini par un sommet s, une taille t et un cap c

Q1. On considère la spécification suivante du tracé d'un carré :

Tcarré : action (donnée t : réel >0)
{ État initial : pe=basse, cap = a0, pp=p0 ; état final : la plume a le même état qu'au départ, et un carré de sommet p0, de côté t et de cap a0 a été tracé. Le reste de l'écran n'est pas modifié. }

On considère maintenant une action de tracé nommée Tdessin :

```
Tdessin : action (donnée n : entier >0, t, dt : réel >0)
    Tdessin (n, t, dt) :
        Vider ; Baisser ;
        pour i allant de 0 à n-1 : Tcarré (t+i*dt)
```

Quel est le dessin tracé par l'appel Tdessin (3, 2, 1) ? On raisonnera sur les états intermédiaires de la plume, par rapport à la spécification de l'action Tcarré.

Compléter en conséquence la spécification de l'action Tdessin, notamment en ce qui concerne l'état initial et l'état final de la plume.

Q2. On propose la réalisation suivante de l'action Tcarré

```
Tcarré (t) :
    Av (t) ; TrG (90) ; Av (t) ; TrG (90) ; Av (t) ; TrG (90) ; Av (t)
```

Critiquer cette réalisation par rapport à la spécification. Donner la spécification correspondant à cette réalisation de Tcarré. Quel est l'effet, avec cette réalisation de Tcarré, de l'appel Tdessin (3, 2, 1) ?

Q3. Donner une forme correcte de la réalisation de Tcarré par rapport à la spécification donnée initialement.

E2.18 : Compositions de dessins formés d'un ensemble de carrés

Réaliser les actions de tracés suivantes en utilisant une action de tracé d'un carré Tcarré inspirée de l'exercice **E2.17** et toute action intermédiaire déduite d'une structuration du dessin à tracer. Puis les utiliser dans diverses compositions à choisir.

a) Un ensemble de carrés disposés en ligne

On spécifie l'action suivante :

Tligne : action (donnée n : entier > 0, t : réel > 0)
{ Trace une ligne formée de n carrés de côté t comme dans la figure 2.6. A l'état initial, la plume est haute en un point p, cap Ouest-Est ; à l'état final, la figure a été ajoutée sur l'écran et la plume est dans le même état qu'au départ. Pré-condition : les valeurs de n et t et la position p sont telles que le dessin peut être tracé dans la largeur de l'écran. }



Figure 2.6 – Exemple de 4 carrés (n=4)

— Utiliser l'action Tligne pour tracer, à partir du centre de l'écran, une ligne de 5 carrés, avec deux appels de Tligne : l'un traçant les 2 premiers carrés, l'autres les 3 suivants. A l'état final, la plume est haute au centre de l'écran.
— Réaliser l'action Tligne.
— Donner la distance parcourue par la plume pour un appel de Tligne.
— Donner le nombre d'appels des actions Baisser, Lever, Reculer, Avancer et TrG.

b) Un ensemble de carrés disposés en rectangle
Spécifier et réaliser une action nommée Trectangle, de tracé d'un ensemble de n*m carrés de côté t disposés en rectangle comme dans la figure 2.7.

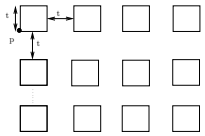


Figure 2.7 – Exemple de 3*4 carrés (n=3, m=4)

c) Un ensemble de carrés disposés en triangle
Spécifier et réaliser une action nommée Ttriangle, de tracé d'un ensemble de carrés disposés en triangle comme dans la figure 2.8.

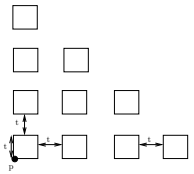


Figure 2.8 – Exemple de carrés en triangle

E2.19 : Les rayons
On considère un dessin formé de n segments répartis régulièrement sur un cercle de rayon r et de centre O, tel que celui illustré dans la figure 2.9 (n=8).

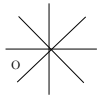


Figure 2.9 – Rayons

On spécifie une action de tracé de la manière suivante :

Trayons : action (donnée t : réel > 0, n : entier > 0)
{ Trace une figure comportant n rayons de taille t, issus du centre. État initial : plume basse au centre, cap Ouest-Est ; état final : la figure a été ajoutée à l'écran ; la plume est dans le même état qu'au départ. }

Réaliser l'action Trayons. Utiliser l'action ainsi définie dans une composition de votre choix.

E2.20 : Tracé de segments



Figure 2.10 – Suite de segments

La figure 2.10 illustre un effet d'optique. Elle est composée d'un ensemble de segments joignant des points espacés régulièrement sur les axes. Une telle figure est caractérisée par trois paramètres :
— le nombre **n** de segments : à chaque segment correspond un point sur chacun des axes ;
— la taille **th** des intervalles séparant deux points de l'axe horizontal ;
— la taille **tv** des intervalles séparant deux points de l'axe vertical.
Pour tracer une telle figure, on spécifie une action nommée TsegmentsGlissants :

TsegmentsGlissants : action (donnée n : entier ≥ 0, th, tv : réel > 0)
*{ Trace la figure formée des deux axes et de n segments glissants, th et tv étant la taille des intervalles séparant deux points, respectivement sur l'axe horizontal et sur l'axe vertical. L'axe vertical est figuré par un segment de taille (n+1)*tv. De même, l'axe horizontal est figuré par un segment de taille (n+1)*th. L'action ne modifie pas le reste de l'écran. Il n'y a aucune contrainte sur les états initial et final de la plume. }*

Pour réaliser cette action, on spécifie une action de tracé d'un segment nommée Tsegment :

Point : type <abscisse : réel, ordonnée : réel> *{ abscisse et ordonnée du point }*
Tsegment : action (donnée P1, P2 : deux Points)
{ Trace le segment défini par les deux points P1 et P2. État initial : indifférent, posons cap=c0 ; état final : segment tracé, pe=haute, pp=P2, cap=c0. }

— Donner une réalisation de l'action Tsegment, en utilisant l'action Placer.
— Donner une réalisation de l'action TsegmentsGlissants, en utilisant l'action Tsegment.
— Donner le nombre d'appels de l'action Tsegment engendrés par un appel de l'action TsegmentsGlissants.

E2.21 : Tricot

On considère un triangle équilatéral pavé de triangles équilatéraux élémentaires (figure 2.11).

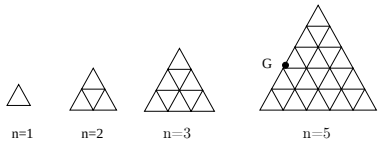


Figure 2.11 – Triangles équilatéraux

On convient qu'un triangle équilatéral est caractérisé par un sommet, une taille de côté et un cap. On convient de plus que l'observateur placé au sommet et regardant dans le sens donné par le cap voit le triangle devant lui et à sa gauche. Un "tricot" est caractérisé par le sommet et le cap du triangle extérieur, la taille du côté des triangles élémentaires qui pavent la figure, et le nombre de triangles élémentaires que l'on place le long d'un côté du triangle extérieur (ordre de la figure).
On spécifie ainsi une action de tracé d'un tricot :

Ttricot : action (donnée n : entier > 0, te : réel > 0)
{ Trace la figure d'ordre n avec te pour taille de trait élémentaire.
État initial : plume basse au sommet du triangle extérieur orientée selon le cap du triangle extérieur selon les conventions ci-dessus.
État final : tricot tracé et plume dans l'état initial. }

a) Observation d'une solution

On voit un "tricot" comme une suite de rangées horizontales, chaque rangée étant formée d'une suite de triangles élémentaires, chaque triangle étant formé de trois segments. L'algorithme est construit en raisonnant sur la suite des rangées formant le tricot. Le tracé d'une rangée est composé du tracé de la base commune aux triangles de la rangée et de la ligne brisée complétant les triangles. Le tracé de la ligne brisée est décomposé en une suite de tracés de couples de côtés consécutifs.
On propose ainsi l'algorithme suivant :

```
Ttricot(N, TE) :
   BC : réel                                   { base de la rangée courante = N*TE }
   BC ⟵ N*TE
   pour NC allant de N à 1, pas -1
      { tracé de la base de la rangée courante. NC est le nombre de triangles de cette rangée. }
      {*1*} Avancer(BC) ; {*2*} TrG(120)
      { tracé des triangles bordant la base }
      {*3*} pour i allant de 1 à NC
         {*4*} Avancer(TE) ; {*5*} TrG(120) ;
         {*6*} Avancer(TE) ; {*7*} TrD(120)
      { préparation du tracé de la rangée suivante }
      {*8*} TrD(60) ; Lever ; {*9*} Avancer(TE) ; {*10*} Baisser ; TrD(60)
      {*11*} BC ⟵ BC - TE
   { mise en place de la plume dans le même état qu'au départ }
   {*12*} TrD(120) ; Lever ; Avancer( N*TE) ; TrG(120) ; Baisser
```


Dans les questions suivantes, on observe la relation entre l'algorithme ci-dessus et un tricot d'ordre 5 (voir figure 2.11). Les trois côtés du triangle extérieur sont nommés *base*, *côté gauche* et *côté droit*. Les rangées sont numérotées à partir de 1 en partant de la base.
Dans ce qui suit, on dit que *la plume arrive en* (respectivement *quitte*) *un point P pour la première* (respectivement *dernière*) *fois*, lorsque c'est la première (respectivement dernière) fois que la position de la plume dans le plan (pp) est ce point P (que la plume soit haute ou basse).

Q1. On considère le point G situé sur le côté **gauche** du triangle extérieur correspondant à la base de la troisième rangée (cf figure 2.11).

— Déterminer le nombre de fois où la plume passe au point G.
— Situer le point du texte de l'algorithme correspondant au moment auquel, lors de son exécution, la plume quitte ce point pour la dernière fois. Indiquer l'état de la plume à ce moment là.
— Situer le point du texte de l'algorithme correspondant au moment auquel, lors de son exécution, la plume arrive en ce point pour la première fois. Indiquer l'état de la plume à ce moment là.

Q2. Montrer sur la figure tous les points auxquels se trouve la plume lorsque l'exécution arrive au point marqué ***10*** dans le texte de l'algorithme.

Q3. Analyse quantitative :

Pour N donné, formuler le nombre d'appels de chacune des actions de la machine tracé : Lever, Baisser, TournerGauche, TournerDroite, Avancer.

b) Tracé du tricot avec contrainte

Q4. Réaliser l'action Ttricot en respectant la contrainte suivante : la figure doit être tracée sans lever la plume et sans passer deux fois sur le même segment. On pourra s'inspirer du chemin illustré dans la figure 2.12.

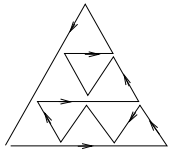


Figure 2.12 – Triangles, tracé avec contrainte

3. Séquences et Tableaux

A. Eléments de cours

a) Tableaux

a.1. Terminologie

Le lexique général associé à la définition d'un tableau est :

```
bi, bs : constante de type entier                    { bornes inférieure et supérieure }
Nom_Tableau : tableau sur [bi...bs] de Type_Element
```

[bi...bs] est l'*intervalle de définition* du tableau : bi etbs sont des constantes fixées au moment de coder le programme. Par convention, si l'intervalle est vide (bi>bs), le tableau est vide.
La *taille* du tableau est : bs-bi+1. Type_Element est le *type* des éléments du tableau.
Il faut distinguer le *rang* d'un élément dans le tableau et son *indice* : rang = indice-bi+1. Dans le cas particulier où bi=1, le rang et l'indice sont alors de même valeur, et la taille est égale à bs.

a.2. Opérations sur les tableaux

Il n'y a pas d'opération globale sur un tableau. Toute opération est réalisée en termes des opérations sur les éléments du tableau. Les éléments d'un tableau T sont dénotés : T_i. Pour le lexique suivant :

```
n : constante de type entier > 0
TA, TB : tableau sur [1...n] d'entier
```

mettre des zéros partout dans le tableau TA est réalisé par : pour i allant de 1 à n : $TA_i \longleftarrow 0$
recopier les valeurs du tableau TA dans le tableau TB par : pour i allant de 1 à n : $TB_i \longleftarrow TA_i$

a.3. Tableaux en paramètres

Pour utiliser un tableau en paramètre d'une action ou d'une fonction, il est nécessaire de fournir à l'action ou à la fonction 3 paramètres : un tableau sur un intervalle "non contraint" (les bornes ne sont pas données) du type des éléments voulu, ainsi que les bornes effectives de ce tableau.
Par exemple, pour spécifier une action de calcul de somme de tous les éléments d'un tableau, il faut paramétrer cette action par le tableau et ses bornes de la façon suivante :

```
AffSomTab : action (donnée a, b : entier, T : tableau sur [*...*] d'entier)
{ Affiche la somme des éléments du tableau T de borne inférieure a et de borne supérieure b. }
```

L'appel à l'action AffSomTab avec le lexique de 3.A.a.2., pour afficher la somme des éléments du tableau TA, serait : AffSomTab(1, n, TA).

a.4. Tableaux à plusieurs dimensions

Une façon de considérer un tableau M à deux dimensions est de le voir comme un ensemble de lignes, chaque ligne étant un tableau à une dimension.
On obtient alors le lexique général suivant :

```
n, m : constante de type entier > 0
M : tableau sur [1...n] de tableau sur [1...n] d'entier { M_{i,j} est le nom associé à l'élément situé à la
                                                          ligne i et à la colonne j }
```

Pour raisonner sur un tel tableau, on compose les schémas usuels de traitement séquentiel : on raisonne sur une suite de lignes et, dans chaque ligne, sur une suite de cases.

b) Représentation contiguë d'une séquence dans un tableau

b.1. Principe de la représentation contiguë

Une séquence peut être représentée de façon contiguë dans un tableau : le tableau est le *contenant*, de taille constante *fixée a priori*; la séquence est le *contenu*, de taille *variable*.
La *contiguïté* est exprimée par le fait que deux éléments consécutifs de la séquence se trouvent dans deux cases d'indices consécutifs (le premier élément est placé en première position du tableau etc...).
Pour caractériser le *dernier élément*, deux méthodes existent : la représentation contiguë *avec longueur explicite* (une variable contient la longueur de la séquence), et la représentation contiguë *avec marque* (une valeur spécifique "marque de fin" est placée après le dernier élément de la séquence). Nous ne parlerons ici que de la représentation avec longueur explicite.

b.2. Schémas d'utilisation de la représentation avec longueur explicite

Les figures 3.1 et 3.2 résument le lexique et les schémas itératifs associés à la représentation d'une séquence avec longueur explicite. Ces schémas sont applicables au traitement de tableaux, en considérant que la longueur de la séquence est égale au nombre d'éléments du tableau.

```
Elément : type ; bi, bs : constante de type entier
TabElt : tableau sur [bi...bs] d'Elément
NbElt : entier sur [0..bs − bi +1] >
{ la séquence est représentée par TabElt le contenant et NbElt la longueur explicite }
P : fonction (E : Elément) ⟶ booléen                { pour caractériser la propriété cherchée }
```

```
{ parcours }
   InitT
   pour IC allant de bi à bi+NbElt-1
      Traiter (TabElt_IC)                           { Traiter ne modifie ni IC, ni bi, ni NbElt }
```

Figure 3.1 – Schémas de traitement d'une séquence représentée dans un tableau avec longueur explicite – 1

```
{ recherche avec arrêt dès que possible }
   IC : entier sur [bi..bs+1]                                  { remarquer bs+1 }
   IC ⟵ bi
   tantque IC ≠ bi + NbElt et puis non P (TabElt_IC) : IC ⟵ IC + 1
   si IC=bi+NbElt alors ...                                    { on n'a pas trouvé }
   sinon ...                                                   { TabElt_IC est l'élément cherché }
```

```
{ parcours partiel }
   IC : entier sur [bi..bs+1]                                  { remarquer bs+1 }
   IC ⟵ bi ; InitT
   tantque IC ≠ bi + NbElt et puis non P (TabElt_IC)
      Traiter (TabElt_IC) ; IC ⟵ IC + 1
   { si IC=bi+NbElt, on a atteint la fin de la séquence }
```

Figure 3.2 – Schémas de traitement d'une séquence représentée dans un tableau avec longueur explicite – 2

b.3. Recherche avec sentinelle

L'idée est de placer, avant d'effectuer la recherche, un élément vérifiant la propriété cherchée à la fin de la séquence. On est alors sûr de le trouver, ce qui permet d'exprimer le contrôle de l'itération plus simplement :

```
IC : entier sur [bi..bs]
sentinelle : Elément { vérifiant P }
TabElt_bi+NbElt ⟵ sentinelle ; IC ⟵ bi
tantque non P (TabElt_IC)
   IC ⟵ IC + 1
si IC = bi+NbElt...                                            { on n'a pas trouvé }
sinon...                                                       { TabElt_IC est l'élément cherché }
```

Remarque : Le fait de placer une sentinelle à la fin de la séquence réduit de 1 la longueur maximale des séquences que l'on peut représenter.

b.4. Existence d'un élément vérifiant une propriété

Pour réaliser un algorithme déterminant l'existence d'un élément vérifiant une propriété **P**, on peut appliquer le schéma de recherche mais aussi le schéma de parcours. Dans le cas d'une représentation avec longueur explicite, on a alors le schéma suivant :

```
{ existence d'un élément vérifiant P }
   Existe : booléen
   Existe ⟵ faux
   pour IC allant de bi à bi+NbElt-1
      Existe ⟵ Existe ou P (TabElt_IC)
   { Existe a la valeur vrai ⟺ il existe un élément vérifiant P }
```

b.5. Propriété vérifiée par tous les éléments

Pour réaliser un algorithme déterminant si tous les éléments d'un tableau vérifient une propriété **P**, on peut appliquer le schéma de recherche mais aussi le schéma de parcours. Dans le cas d'une représentation avec longueur explicite, on a alors le schéma suivant :


```
{ tous les éléments vérifient-ils P ? }
   TousOK : booléen
   TousOK ⟵ vrai
   pour IC allant de bi à bi+NbElt-1
      TousOK ⟵ TousOK et P (TabElt_IC)
   { TousOK a la valeur vrai ⟺ tous les éléments vérifient P }
```

b.6. Insertion d'un élément dans une séquence triée en ordre croissant

```
Elément : type
bi, bs : constante de type entier
Lmax : constante bs-bi+1 de type entier{ longueur maximum des séquences que l'on peut représenter }
TabElt : tableau sur [bi...bs] d'Elément
NbElt : entier sur [0...Lmax]>
```

1. spécification de la position d'insertion k

On examine deux hypothèses selon la manière de placer un nouvel élément E en cas d'égalité.

Hypothèse 1 : en cas d'égalité, le nouvel élément est placé *après* ceux ayant même valeur.
- Cas n° 1 : $NbElt \neq 0$ et $E < TabElt_{bi}$: insertion en tête, k=bi
- Cas n° 2 : nbElt=0 ou $E \geq TabElt_{bi+NbElt-1}$: insertion en queue, k=bi+NbElt
- Cas n° 3 : $NbElt \neq 0$ et $TabElt_{bi} \leq E < TabElt_{bi+NbElt-1}$: k est l'indice vérifiant $TabElt_{k-1} \leq E < TabElt_{k}$

Hypothèse 2 : en cas d'égalité, le nouvel élément est placé *avant* ceux ayant même valeur.
- Cas n° 1 : $NbElt \neq 0$ et $E \leq TabElt_{bi}$: insertion en tête, k=bi
- Cas n° 2 : NbElt=0 ou $E > TabElt_{bi+S.L-1}$: insertion en queue, k=bi+NbElt
- Cas n° 3 : $NbElt \neq 0$ et $TabElt_{bi} < E \leq TabElt_{bi+NbElt-1}$: k est l'indice vérifiant $TabElt_{k-1} < E \leq TabElt_{k}$

2. construction de l'algorithme d'insertion

Une première solution consiste à décomposer le problème en trois étapes :

1. Calcul de la position d'insertion k, par une recherche dans la séquence, éventuellement en sens inverse.
2. Préparation de l'insertion à la position k par un décalage.
3. Placement de l'élément à la position k.

Comme le décalage est un parcours en sens inverse de la séquence, on formule aussi la recherche de la position d'insertion en sens inverse et on fusionne les deux itérations correspondantes.

```
{ Pré-condition S.L<Lmax }
   k : entier sur [bi-1...bs]
   k ⟵ bi + NbElt-1
   tant que k ≠ bi-1 et puis E < TabElt_k
      TabElt_k+1 ⟵ TabElt_k
      k ⟵ k - 1
   { k+1 est la position d'insertion ; elle a été libérée }
   TabElt_k+1 ⟵ E
   NbElt ⟵ NbElt + 1
```

b.7. Recherche dichotomique dans une séquence triée en ordre croissant

Nous fournissons ici un algorithme déterminant la position d'insertion d'un élément X donné (avec l'hypothèse 1 ci-dessus en cas d'égalité).

```
{ Pré-condition NbElt≠0 }
selon X, T
   X < TabElt_bi  : k ⟵ bi
   X ≥ TabElt_bi+NbElt-1  : k ⟵ bi + NbElt
   sinon { TabElt_bi ≤X < TabElt_bi+NbElt-1 et NbElt > 1 }
      i ⟵ bi ; j ⟵ bi + NbElt - 1
      tant que i < j-1
         { Invariant : TabElt_i ≤ X < TabElt_j }
         m ⟵ (i+j) quotient 2                          { i<j-1 ⟹ i < m < j }
         selon X, T :
            X < TabElt_m  : j ⟵ m
            X ≥ TabElt_m  : i ⟵ m
      { TabElt_i ≤ X < TabElt_i+1 }
      k ⟵ i+1
```

B. Exemples d'exercices rédigés

Exemple E3.1 : À propos de somme d'entiers

On considère un tableau **TE** *d'entiers définis sur l'intervalle* **[1. . .n]** *:*

n : constante de type entier > 0 ; TE : tableau sur [1. . .n] d'entier

(i) Somme des éléments du tableau.

Réaliser un algorithme qui affiche la somme des éléments de **TE**.

On parcourt le tableau **TE** de manière à calculer progressivement la somme cherchée à l'aide d'une variable **Somme**.

```
Somme : entier                                   { pour calculer la somme }
Somme ⟵ 0
pour i allant de 1 à n
   { Somme est la somme des i-1 premiers éléments de TE (0 si i =1) }
   Somme ⟵ Somme + TE_i
Écrire(Somme)
```

(ii) Somme des X premiers éléments du tableau.

On spécifie une fonction nommée **SommeDébut** *:*

SommeDébut : fonction (T : tableau sur [*. . .*] d'entier, a, b : entier, R : entier > 0) $\longrightarrow$ entier
{ Somme des R premiers éléments de T ; a et b sont respectivement les bornes inférieure et supérieure du tableau T. Pré-condition : a≤b. Si T contient moins de R éléments, le résultat est la somme de tous les éléments du tableau. }

— *Compléter l'instruction suivante de manière à afficher la somme des 4 premiers éléments du tableau* **TE** *:*

```
Écrire(SommeDébut(●●●●●●●●●))
```

— *Réaliser la fonction* **SommeDébut** *: pour cela modifier l'algorithme obtenu en* **(i)**.

Pour afficher la somme des 4 premiers éléments du tableau **TE**, on utilise la fonction **SommeDébut** de la manière suivante :

Écrire(SommeDébut(**TE, 1, n, 4**))

Pour réaliser la fonction **SommeDébut**, on ne parcourt que les **X** premiers éléments du tableau donné.

```
SommeDébut(T, a, b, R) :
{ pré-condition : R entier strictement positif. Post-condition : Si R ≥ b-a+1, on fait la somme de T }
   Somme : entier                                  { pour calculer la somme }
   nb : entier > 0                  { nombre d'éléments à sommer, pour contrôler le parcours de T }
   nb ⟵ si R < b-a+1 alors R sinon b-a+1
   Somme ⟵ 0
   pour i allant de a à a+nb-1
      { Somme est la somme des i-1 premiers éléments de T (0 si i = 1). }
      Somme ⟵ Somme + T_i
   retour : Somme
```

Exemple E3.2 : Appartenance

Décrire une fonction (spécification, réalisation itérative) d'appartenance d'un entier donné à un tableau d'entiers donné :

Appartient : fonction (T : tableau sur [*. . .*] d'entier, a, b, X : entier) $\longrightarrow$ booléen
{ vrai $\iff$ X ∈ T ; a et b sont respectivement les bornes inférieure et supérieure du tableau T. }

Pour déterminer l'**existence** d'un élément vérifiant une propriété **P** (ici "être égal à X"), on peut appliquer soit un schéma de parcours soit un schéma de recherche.

(i) Version 1 : parcours de tout le tableau

```
{ X ∈ T ⟺ X=T_a ou X=T_a+1 ou . . .X=T_i . . .ou X=T_b }
Appartient(T, a, b, X) :
   Ap : booléen
   Ap ⟵ faux
   pour i allant de a à b
      { Ap a la valeur vrai ⟺ X ∈ T_[a...i−1] }
      Ap ⟵ Ap ou X=T_i
   retour : Ap
```

(ii) Version 2 : recherche avec arrêt dès que possible

Le parcours du tableau est interrompu dès que la valeur X est trouvée.

```
Appartient(T, a, b, X) :
   i : entier sur [a. . .b+1]
   i ⟵ a
   tant que i ≠ b+1 et puis X≠T_i
      { X ∉ T_[a...i−1] }
      i ⟵ i+1
   retour : i ≠ b+1
```

Exemple E3.3 : Intersection (séquence sous forme contiguë avec longueur explicite)

Réaliser un algorithme d'intersection de deux ensembles d'entiers. Les ensembles (données et résultats) sont représentés par des séquences sans répétition sous forme contiguë avec longueur explicite.

(i) Analyse du lexique

Deux variables sont associées à chaque ensemble : un tableau de taille fixe (définie a priori) dans lequel les éléments de l'ensemble seront placés de manière contiguë à partir de la première position ; un entier ≥ 0 associé au cardinal de l'ensemble.

```
Lmax : constante de type entier > 0                    { longueur maximum des séquences considérées }
TabS1, TabS2, TabS3 : tableau sur [1...Lmax] d'entier
NbS1, NbS2, nbS3 : entier sur [0...Lmax]
{ S1, S2 sont les données, S3 le résultat }
```

(ii) Analyse de l'algorithme

On part de la définition de l'intersection :
$x \in E1 \cap E2 \iff x \in E1$ et $x \in E2$
On en déduit un principe d'algorithme : pour chaque élément **x** de **E1** (parcours de **TabS1** entre **1** et **NbS1**), si **x** appartient à **E2** (recherche dans **TabS2**), placer **x** dans **E3** (ajout dans **TabS3**).
Pour réaliser la recherche, on s'appuie sur le principe développé dans l'exemple **E3.2** : la différence est que l'on n'explore le tableau **TabS2** qu'entre les positions **1** et **NbS2** (et non **Lmax**).

(iii) Algorithme

```
NbS3 ⟵ 0                                               { E3 est vide au départ }
pour i allant de 1 à NbS1
    { TabS3[1...NbS3] contient les éléments de l'intersection de E2 et du sous-ensemble de E1 placé dans
    TabS1[1...i−1] }
    { TabS1i appartient-t-il à E2 ? }
    j ⟵ 1 ; tant que j≠NbS2+1 et puis TabS1i ≠ TabSj : j ⟵ j+1
    si j ≠ NbS2+1 alors { placer TabS1i en queue de TabS3 }
        NbS3 ⟵ NbS3+1 ; TabS3NbS3 ⟵ TabS1i
{ E3 intersection de E1 et E2 est représenté par la séquence sous forme contiguë donnée dans TabS3.
avec longueur explicite NbS3 }
```

Exemple E3.4 : Tri d'un tableau par sélection du minimum

Réaliser un algorithme de tri (en ordre croissant) d'un tableau d'entiers T*, selon le principe suivant : on identifie un élément de* T *ayant la valeur minimum ; on échange les places de cet élément et du premier élément de* T *; on répète ce processus avec le sous-tableau commençant en deuxième position. Le processus se termine lorsque le sous-tableau ne comporte qu'un élément. On donnera les invariants permettant de raisonner sur les itérations de l'algorithme.*

Spécification :

```
TabE : type tableau sur [*...*] d'entier
```



```
TriParSelectionDuMinimum : action (donnée a, b : entier ; donnée-résultat T : TabE)
    { Trie le tableau T ; a et b sont ses bornes inférieure et supérieure.
    e.i. : aucune hypothèse sur l'ordre des éléments de T ;
    e.f. : T est trié. }
```

Réalisation : l'énoncé suggère deux itérations emboîtées qui relèvent d'un schéma de parcours.

```
TriParSelectionDuMinimum (a, b, T) :
lexique
    im : entier sur [a...b]                            { indice d'un minimum courant }
    x : entier                                         { pour l'échange }
algorithme
    pour i allant de a à b-1
        { T contient une permutation des valeurs initiales de T. Ta...i-1 est trié et contient les i-a valeurs
        les plus petites du tableau }
        { identification de l'indice im d'un minimum de Ti...b }
        im ⟵ i
        pour j allant de i+1 à b
            { im est l'indice d'un minimum de Ti...j-1 }
            si Tj < Tim alors im ⟵ j
        { échange }
        x ⟵ Tim ; Tim ⟵ Ti ; Ti ⟵ x
```

Exemple E3.5 : Partition d'un ensemble de mots

On considère la partition P *d'un ensemble* E *de mots, selon l'initiale (la première lettre) de ces mots.* P *est un ensemble de classes* **non vides** *; chaque classe de* P *est caractérisée par une lettre de l'alphabet : c'est le sous-ensemble de* E *de tous les mots ayant cette lettre pour initiale. Si une lettre n'est initiale d'aucun mot de* E*, il n'y a pas de classe associée à cette lettre dans* P*.*
Les ensembles sont représentés par des séquences sans répétitions. Une partition est une séquence de classes, chaque classe étant une séquence de mots sans répétitions.
Les séquences sont **sous forme contiguë** *dans des tableaux avec longueur explicite. Aucun ordre n'est imposé sur ces séquences. On utilise le lexique suivant :*

```
{ Les mots : on fait abstraction de leur représentation }
    Lettre : type caractère                            { restreint aux lettres de l'alphabet }
    Mot : type séquence non vide de Lettre
        { M étant de type Mot, premier(M) dénote le 1er caractère de M. Pour affecter une valeur de type
        Mot à une variable de type Mot, on utilise le symbole d'affectation usuel ( ⟵ ). }
{ Les séquences de mots }
    MaxM : constante de type entier > 0                { longueur maximum des séquences de mots }
    TabMot : tableau sur [1...MaxM] de Mot
    NbMot : entier sur [0...MaxM]
        { L'ensemble E est représenté par les mots placés dans le tableau TabMot entre les indices 1 et
        NbMot. }
{ La partition }
    Classe : type < M : tableau sur [1...MaxM] de Mot, NbM : entier sur [0...MaxM] >
    TabClasse : tableau sur [1...26] de Classe
    NbClasse : entier sur [0...26]
        { La partition P est représentée par un tableau de classes, une classe étant représentée par un
        tableau de mots et le nombre de mots de la classe. Aucun ordre n'est imposé (entre les classes,
        dans une classe). Les classes de P sont non vides. }
```

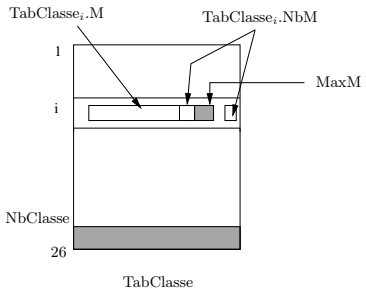


Figure 3.3 – Partition d'un ensemble de mots. TabClasse$_i$ représente une classe constituée de TabClasse$_i$.NbM mots implantés de manière contiguë dans le tableau TabClasse$_i$.M et ayant tous la même initiale, de valeur premier((TabClasse$_i$.M)$_1$)

Q1. Construction de E à partir de P

Algorithme qui construit l'ensemble E *associé à une partition* P *donnée :*

— *donnée : les variables* TabClasse *et* NbClasse *représentant la partition* P.

— *résultat : les variables* TabMot *et* NbMot *représentant l'ensemble* E.

```
{ Chaque classe de P (parcours de TabClasse) est recopiée dans E (parcours d'une classe) en procédant
par ajout en queue de TabMot. }
  { pré-condition : ∑_{i=1...NbClasse} (TabClasse)_i.NbM ≤ MaxM }
    NbMot ⟵ 0
    pour i allant de 1 à NbClasse
      pour j allant de 1 à TabClasse_i.NbM
        NbMot ⟵ NbMot + 1 ; TabMot_NbMot ⟵ TabClasse_i.M_j
```

Q2. Construction de P à partir de E

Algorithme qui construit la partition P *associée à un ensemble* E *donné :*

— *donnée : les variables* TabMot *et* NbMot *eentant l'ensemble laE.*

— *résultat : les variables* TabClasse *et* NbClasse *représentant la partition* P.

```
{ Pour chaque mot de E (parcours), l'insérer dans P. Pour cela, soit I l'initiale du mot courant : s'il
existe dans P (recherche) une classe dont les mots commencent par I, ajouter le mot courant en queue
de cette classe ; sinon créer une nouvelle classe avec ce mot (ajout en queue dans P). }
  MC : Mot                                                    { mot courant dans le parcours de E }
  j : entier sur [1...27]                                     { pour la recherche dans P }
  NbClasse ⟵ 0
  pour i allant de 1 à NbMot
    MC ⟵ TabMot_i
    j ⟵ 1 ; tant que j ≠ NbClasse + 1 et puis premier((TabClasse_j.M)_1) ≠ premier(MC) : j ⟵ j+1
    si j ≠ NbClasse + 1 alors                                 { ajout en queue de la classe j }
      TabClasse_j.NbM ⟵ TabClasse_j.NbM + 1 ; (TabClasse_j.M)_TabClasse_j.NbM ⟵ MC
    sinon                                                     { ajout en queue de P d'une classe singleton }
      NbClasse ⟵ NbClasse + 1
      TabClasse_NbClasse.NbM ⟵ 1 ; (TabClasse_NbClasse.NbM)_1 ⟵ MC
```


C. Exercices

a) Traitement de tableaux

E3.6 : A propos de sommes d'entiers (E3.1 suite)

On veut afficher la somme des éléments d'indices pairs d'un tableau d'entiers **T**. S'il n'en existe pas, 0 doit être affiché. On propose l'algorithme suivant :

```
n : constante de type entier > 0
T : tableau sur [1...n] d'entier
S : entier                                                  { somme courante }
ip : entier ≥ 0                                             { pour énumérer les indices pairs }
ip ⟵ 0 ; S ⟵ 0
{ *P1* }
tant que ip ≠ n-1
   { *P2* }
   ip ⟵ ip + 2 ; S ⟵ S + T_ip
   { *P3* }
{ *P4* }
Ecrire(S)
```

L'objet de l'exercice est de valider cet algorithme, c'est-à-dire sa terminaison et sa correction par rapport à l'effet attendu.

Lors de la rédaction des réponses, une attention particulière doit être apportée à la précision et la pertinence des justifications et démonstrations.

Q1. Cas de non terminaison

— Donner un exemple simple de données (valeurs de **n** et de **T**) pour lesquelles l'exécution de l'algorithme :
  — se termine et donne le résultat correct.
  — ne se termine pas.

— Caractériser les cas de terminaison et de non terminaison de manière générale. Justifier la réponse.

Q2. Raisonnement sur les états

Il est conseillé de relire, au chapitre 2, la définition d'un *invariant d'itération*. Par ailleurs, on prendra les conventions suivantes permettant d'abréger l'expression des assertions : *0 est pair* ; *la somme d'une séquence vide vaut 0.*

— Pour chacune des assertions suivantes indiquer si c'est un invariant de l'itération. Justifier la réponse.
  A1 *: ip est pair* **A2** *: ip est impair* **A3** *: $S \geq 0$*

— Soit l'assertion **A4** : *S est la somme des éléments d'indices pairs de T[1...ip]* :
  — Démontrer que **A4** n'est pas un invariant, en donnant un contre-exemple simple.
  — En utilisant les arguments de cette démonstration, donner un invariant **A5** obtenu en complétant **A4**.

— Parmi les 5 assertions précédentes :
  — lesquelles sont vraies en **P1** ?
  — lesquelles sont toujours vraies au passage de l'exécution en **P2** ? Justifier la réponse.

— Dans l'hypothèse où l'itération se termine (voir **Q1**), déduire de ce qui précède une assertion **A6**, vraie en **P4** et montrer qu'elle correspond au résultat attendu de l'algorithme.

Q3. Correction de l'algorithme
Modifier l'algorithme proposé pour qu'il soit correct, montrer sa terminaison et donner, en fonction de **n**, le nombre d'exécutions du corps de l'itération.

E3.7 : Recherche dans un tableau
Expérimenter l'utilisation des schémas de parcours et de recherche avec arrêt dès que possible (cf 3.A.b.2.) en étudiant chacune des deux fonctions suivantes :

```
n : constante de type entier > 0
QueDesPos : fonction (T : tableau sur [1...n] d'entier) ⟶ booléen
    { vrai si et seulement si le tableau T ne comporte que des entiers strictement positifs. }
EstEnOrdreCroissant : fonction (T : tableau sur [1...n] d'entier) ⟶ booléen
    { vrai si et seulement si le tableau T est trié en ordre croissant. }
```

— pour chaque fonction, donner des jeux d'essais significatifs permettant d'en tester les réalisations
— pour chaque fonction, donner deux réalisations, l'une basée sur le schéma de parcours, l'autre sur le schéma de recherche
— pour chaque réalisation, donner un algorithme utilisant la fonction.

E3.8 : A propos de valeur maximum
Pour chacune des questions suivantes : spécifier une action, en donner la réalisation ; puis donner un exemple d'utilisation de l'action, en précisant le contexte d'appel (lexique).

Q1. Affichage de la valeur maximum des éléments d'un tableau d'entiers.

Q2. Affichage du plus petit indice de la valeur maximum des éléments d'un tableau d'entiers.

Q3. Modifier l'algorithme donné en **Q2** pour afficher le plus grand indice de la valeur maximum.

E3.9 : Algorithme mystère
On propose l'algorithme suivant :

```
n : constante de type entier >0 ; T : tableau sur [1...n] d'entier
i ⟵ 1 ; j ⟵ n
tant que i < j
    si T_i ≤ T_j alors i ⟵ i+1 sinon j ⟵ j-1
Écrire (T_i)
```

Cet algorithme a pour effet d'afficher la valeur d'un élément du tableau T. Lequel ? Justifier la réponse à l'aide d'un invariant.

E3.10 : Matrices carrées symétriques
Une matrice carrée d'entiers d'ordre n est un tableau d'entiers de n lignes et de n colonnes. Une matrice carrée d'ordre n est dite symétrique si et seulement si pour tout i et pour tout j, l'élément situé à l'intersection de la ligne i et de la colonne j a même valeur que l'élément situé à l'intersection de la ligne j et de la colonne i. Par exemple la matrice carrée d'ordre 5 donnée figure 3.4 est symétrique.

On donne une matrice carrée MatCar :

```
n : constante de type entier > 0
MatCar : tableau sur [1...n] de tableau sur [1...n] d'entier
```

On étudie un algorithme qui affiche un message indiquant si oui ou non la matrice MatCar est symétrique.

1	12	0	87	11
12	2	6	7	42
0	6	1	8	4
87	7	8	0	-1
11	42	4	-1	4

Figure 3.4 – Matrice carrée d'ordre 5 symétrique

Q1. On propose l'ébauche d'algorithme suivante fondée sur un parcours de la matrice ligne par ligne :

```
EstSym : booléen                                   { vrai si la matrice est symétrique }

EstSym ⟵ •••
pour i allant de 1 à n
    pour j allant de 1 à n
        EstSym ⟵ •••
si EstSym alors écrire ("la matrice est symétrique")
sinon écrire ("la matrice n'est pas symétrique")
```

— Compléter cette ébauche de l'algorithme.
— Donner, en fonction de **n**, le nombre d'accès à la matrice **MatCar**.

Q2. On observe qu'il suffit de parcourir l'une des deux "demi-matrices"(en dessous ou en dessus de la diagonale, diagonale exclue).
— Modifier l'algorithme de la question **Q1** pour tenir compte de cette observation.
— Donner, en fonction de **n**, le nombre d'accès à la matrice **MatCar**.

Q3. On veut réaliser un nouvel algorithme, traitant l'une des deux "demi-matrices", fondé sur le schéma de recherche avec "arrêt dès que possible".
Version 1
Donner une première version comportant deux itérations de recherche emboîtées :
— L'itération englobante exprime la recherche dans l'ensemble des lignes de la demi matrice, d'une ligne vérifiant la propriété *comporter un élément différent de son symétrique*.
— L'itération interne exprime la recherche dans une ligne d'un élément différent de son symétrique (elle peut être éventuellement réalisée par une fonction intermédiaire).

Version 2
Donner une deuxième version ne comportant qu'une seule itération de recherche. Elle est fondée sur un principe de "linéarisation de la matrice" : on considère l'ensemble des couples d'indices <i,j> donnant accès aux éléments de la demi-matrice. L'algorithme est une recherche avec arrêt dès que possible dans cet ensemble, du premier couple <i, j> tel que $\text{MatCar}_{i,j} \neq \text{MatCar}_{j,i}$.

E3.11 : Point de selle
Dans un tableau à deux dimensions (une matrice), un élément est un point de selle si sa valeur est la valeur minimum de sa ligne et la valeur maximum de sa colonne. On spécifie la fonction suivante :

L, C : constante de type entier > 0
UnPointSelle : (M : tableau sur [1...L] de tableau sur [1...C] d'entier)
⟶ booléen, entier sur [1...L], entier sur [1...C]
{ Posons <b, i, j> = UnPointSelle(M) ; si M comporte au moins un point de selle, i et j sont les numéros de ligne et de colonne d'un point de selle de M et b a la valeur vrai ; sinon les valeurs de i et j ne sont pas significatives et b a la valeur faux. }

— Donner la réalisation de la fonction en supposant que les éléments de la matrice sont distincts deux à deux, mais en s'imposant de ne pas utiliser de tableau supplémentaire.

b) Représentation contiguë d'une séquence dans un tableau

E3.12 : Représentation contiguë avec longueur explicite
Une séquence d'entiers **S** est donnée dans un tableau **TabElt** de taille **n**, sous forme contiguë avec longueur explicite **NbElt**.

Q1. Étude du lexique
— Donner le lexique correspondant à cette représentation de la séquence **S**.
— Choisir les valeurs d'une séquence d'entiers de longueur 5 et dessiner le tableau **TabElt** associé. Faire figurer **NbElt**.
— Soit l'expression $\textbf{TabElt}_{NbElt+1}$. Que peut-on en dire (correction, valeur) ?

Q2. Existence d'un zéro
Pour déterminer s'il existe un élément de valeur **0** dans **S**, on propose l'algorithme suivant :

```
i : entier
i ⟵ 1
tant que TabElt_i ≠ 0 et i ≠ NbElt + 1
    i ⟵ i+1
Écrire(si TabElt_i = 0 alors "oui" sinon "non")
```

Cet algorithme est incorrect.
— pour n=5 : donner un exemple de données pour lequel l'exécution de l'algorithme est correcte.
— pour n=3, NbElt=3 : donner un exemple de données pour lequel l'exécution de l'algorithme est incorrecte. Justifier la réponse.
— pour n=5, et NbElt=3 : donner un exemple de données pour lequel l'exécution de l'algorithme est incorrecte. Justifier la réponse.
— **En appliquant un schéma du cours**, modifier l'algorithme donné pour que son exécution soit correcte dans tous les cas.

E3.13 : Suppression des espaces d'un texte
Étant donné un texte (séquence de caractères), on étudie divers algorithmes de suppression de tous les espaces présents dans ce texte.
Exemple (les points figurent ici les espaces) :
texte donné : "•••ceci ••••••est •un ••exemple •de •texte •••"
texte résultat : "ceciestunexempledetexte"
Dans ce qui suit, les textes sont représentés sous forme contiguë avec longueur explicite. Les tableaux sont définis sur l'intervalle **[1...n]**, où **n** est une constante de type entier > 0.

n : constante de type entier > 0
espace : constante ' ' de type caractère
TabEnt : type tableau sur [1...n] de caractère
Ent : type entier sur [0...n]

Q1. Construction d'un nouveau texte
On spécifie l'action suivante :

CopierSansEspace : action (donnée T : TabEnt ; N : Ent, résultat T' : TabEnt ; N' : Ent)
{ Recopie dans T' entre les indices 1 et N', le texte donné dans T entre les indices 1 et N sans les espaces. }

— Donner une réalisation de l'action **CopierSansEspace**.
— Analyse quantitative : donner le nombre de "recopies" de caractères qu'engendre votre algorithme. De même donner le nombre de comparaisons de caractères. Formuler les réponses en termes de **N** et du nombre **E** d'espaces du texte donné.

Q2. Modification du texte donné
On spécifie l'action suivante :

SupprimerEspaces : action (donnée-résultat T : TabEnt, N : Ent)
{ Supprime les espaces du texte donné dans T entre les indices 1 et N. À l'état final, T contient le texte donné sans ses espaces entre les indices 1 et N. }

— Donner une réalisation de l'action **SupprimerEspaces** : l'algorithme doit construire le résultat en réarrangeant les éléments du tableau **T**, sans utiliser de tableau supplémentaire.

Version 1
Appliquer le principe suivant : lors d'un parcours du texte donné, les espaces sont ignorés et les autres caractères sont déplacés au début du tableau (remarquer l'analogie avec la solution obtenue en **Q1**). La figure 3.5 schématise l'invariant devant être respecté par l'itération de l'algorithme.

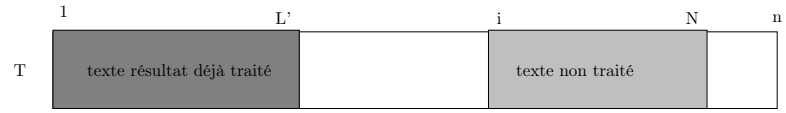


Figure 3.5 – Invariant de l'itération

Version 2
Les éléments du texte donné sont examinés successivement, les espaces rencontrés étant supprimés par décalage. On propose la réalisation suivante de l'action :

```
SupprimerEspaces(T, N) :
    i : entier sur [1...n+1]
    i ⟵ 1
    tant que i ≠ N+1
        si T_i = espace alors
            { décalage à gauche pour supprimer l'espace }
            pour j allant de i à N-1
                T_j ⟵ T_{j+1}
            N ⟵ N-1
        i ⟵ i+1
```

Cet algorithme est incorrect :
- pour n=5, N=4, donner un exemple de données pour lequel l'exécution de l'algorithme est incorrecte ; donner aussi un exemple pour lequel cette exécution est correcte.
- Modifier l'algorithme donné pour que son exécution soit correcte dans tous les cas.

E3.14 : Afficher en ordre sans trier
On étudie un algorithme affichant les éléments d'une séquence d'entiers, en ordre croissant et sans la trier. Dans les premières questions, on se place dans l'hypothèse où les éléments de la séquence traitée sont distincts deux à deux. Cette hypothèse est levée dans la dernière question.

Remarque : un objectif important de cet exercice est d'inciter à expérimenter toute idée d'algorithme sur des exemples simples et ceci avant même de l'exprimer formellement.

Q1. Afficher en ordre sans tableau supplémentaire
(i) Principe et expérimentation préliminaire
Question : ayant déjà affiché un à un les **i** plus petits éléments de la séquence **S** donnée, quel est le prochain élément à afficher ?
Réponse : le i+1ème plus petit, c'est-à-dire, eureka (!) le plus petit élément de **S** supérieur au dernier élément affiché.
On adapte ainsi la structure de l'algorithme de tri par sélection (relire l'exemple **E3.4**) en modifiant le corps de l'itération principale : au lieu de calculer le minimum *des éléments non traités de la séquence*, on calcule le minimum *des éléments de la séquence supérieurs au dernier élément affiché.*
- Expérimenter ce principe pour la séquence **S = [70, 45, 80, 50, 30, 40, 72, 32]** de la manière suivante (sans écrire d'algorithme) :
  - tracer à la main le calcul du minimum de **S**.
  - tracer à la main le calcul du minimum des éléments de **S** qui sont strictement supérieurs à 30.
  - Expliquer brièvement la différence entre le calcul du minimum de **S** et le calcul du minimum des éléments de **S** qui sont supérieurs à un entier **X** donné.
  - tracer à la main, l'affichage en ordre croissant de **S**, en termes de ces opérations.

Spécifications
Dans ce qui suit, les séquences d'entiers sont représentées sous forme contiguë dans un tableau avec longueur explicite. On utilise le lexique suivant :


```
Lmax : constante de type entier > 0
TabEnt : type tableau sur [1...Lmax] d'entier
Ent : type entier sur [0...Lmax]>
PlusInfini : constante de type entier                    { valeur maximum du type entier }
MoinsInfini : constante de type entier                   { valeur minimum du type entier }

IndicePlusPetitSup : fonction (T : TabEnt non vide, N :Ent, X : entier) ⟶ Ent
    { Posons im = IndicePlusPetitSup(T, N, X) :
        — im ≠ 0 : il existe au moins un élément de S strictement supérieur à X et im est l'indice du
          minimum des éléments de T qui sont strictement supérieurs à X.
        — im = 0 : aucun élément de T n'est strictement supérieur à X.
      Pré-condition : les éléments de T sont distincts deux à deux }

AfficherEnOrdre : action (donnée T : TabEnt, N : Ent)
    { Affiche les éléments de S en ordre croissant.
      Pré-condition : les éléments de T sont distincts deux à deux }
```

(ii) Etude de la fonction IndicePlusPetitSup
- En utilisant la fonction IndicePlusPetitSup, compléter l'instruction Ecrire (...), pour que son effet soit d'afficher la valeur minimum d'une séquence d'entiers non vide S.
- Donner une réalisation de la fonction IndicePlusPetitSup.
- Donner des jeux d'essais significatifs pour tester cette réalisation.
- En particulier, donner le résultat de cette fonction, lorsque la séquence traitée contient un élément de valeur PlusInfini. Conclure.

(iii) Etude de l'action AfficherEnOrdre
- Donner une réalisation de l'action AfficherEnOrdre, en utilisant la fonction IndicePlusPetitSup et sans tableau supplémentaire. Commencer par donner l'invariant de l'itération principale (on peut s'inspirer de l'invariant donné dans l'exemple **E3.4**).
- Donner des jeux d'essais significatifs pour tester cette réalisation.

(iv) Produire une séquence triée au lieu d'afficher
- Que faut-il changer à l'étude ci-dessus, si au lieu d'afficher **S** en ordre croissant sans la trier, on veut produire une nouvelle séquence **S'** comportant les éléments de **S** en ordre croissant ?

Q2. Afficher en ordre par le biais d'une permutation ordonnant S.
Une alternative pour disposer des éléments d'une séquence S en ordre croissant, sans trier S, est de construire une permutation P des entiers de l'intervalle [1...N] (c'est-à-dire des indices de T) ordonnant les éléments de T : $\forall$ i, j $\in$ [1...N], i > j $\Rightarrow$ $T_{P[i]} \geq T_{P[j]}$. Autrement dit, $T_{P[i]}$ est la i^{ème} plus petite valeur de S.
On spécifie les actions suivantes :

```
AfficherEnOrdreSelonP : action (donnée T : TabEnt, N : Ent, P : TabEnt)
    { Affiche les éléments de T en ordre croissant sachant que P_[1...N] est une permutation ordonnant T.
      Pré-condition : les éléments de T sont distincts deux à deux. }
CréerPerm : action (donnée T : TabEnt, N :Ent ; résultat P : TabEnt)
    { Construit dans P_[1...N] la permutation ordonnant la séquence T. Pré-condition : les éléments de T
      sont distincts deux à deux. }
```

(i) Etude de l'action AfficherEnOrdreSelonP
- Donner la valeur de P, pour S = [40, 70, 80, 50, 30, 45, 72, 60].
- Donner une réalisation de l'action AfficherEnOrdreSelonP(T,8,P).

(ii) Etude de l'action CréerPerm

— Donner les modifications à apporter à l'algorithme obtenu en **Q1iii** pour obtenir la réalisation de l'action CréerPerm.

Q3. Pour continuer

— Reprendre la question **Q1** et proposer une nouvelle réalisation de l'action **AfficherEnOrdre(S)** en levant la pré-condition *"les éléments de S sont distincts deux à deux"*. On doit aussi reprendre l'étude de la fonction IndicePlusPetitSup.

E3.15 : Partition d'une séquence d'entiers

On considère une séquence non vide d'entiers S, et deux entiers X et Y tels que X≤Y. On veut répartir S en deux séquences S1 et S2 : S1 est en ordre croissant et contient tous les éléments de S appartenant à l'intervalle [X...Y] ; S2 contient tous les éléments de S n'appartenant pas à l'intervalle [X...Y] ; aucun ordre n'est imposé sur S2.
La séquence S est représentée de manière contiguë dans un tableau avec longueur explicite.
On étudie une action nommée Répartir spécifiée de la manière suivante :

```
Lmax : constante de type entier > 0                    { longueur maximum des séquences considérées }
Répartir : action
      ( donnée X, Y : entier ;
      donnée-résultat T : tableau sur [1...Lmax] d'entier,
                  N : entier sur [1...Lmax] ;
      résultat L1, L2 : entier sur [0...Lmax])
   { - à l'état initial, la séquence S de longueur N est implantée dans T entre les indices 1 et N ;
   - à l'état final, les séquences S1 de longueur L1 et S2 de longueur L2 sont implantées dans T : S1
   entre les indices 1 et L1 ; S2 entre les indices L1+1 et N. Pré-condition X ≤ Y. }
```

L'action Répartir doit être réalisée **sans utiliser de tableau supplémentaire**. Pour le faire, on construit une itération qui examine successivement les valeurs de S de manière à réarranger progressivement le tableau T.

```
•••••••••                    { Initialisations : aucun élément n'a encore été traité. S1 et S2 sont vides }
tant que ••••
   si Ti ∈ S2 alors                          { ajout dans S2 : placer Ti dans la zone associée à S2 }
      •••••••••
   sinon                                     { Ti ∈ S1, ajout dans S1 : placer Ti dans la zone associée à S1 }
      •••••••••
•••••••••                                    { calcul des valeurs de L1 et L2 }
```

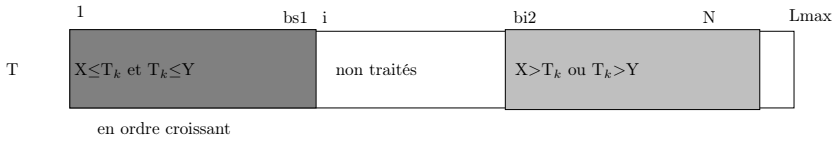


Figure 3.6 – Invariant de l'itération

La figure 3.6 schématise l'invariant proposé pour cette itération.

Au moment du traitement de l'élément courant $\mathbf{T}_i$, $\mathbf{T}_{[1...N]}$ comporte trois zones délimitées par deux indices nommés **bs1** (borne supérieure de **S1**) et **bi2** (borne inférieure de **S2**) de sorte que :

— **bs1 = i-1** et $\mathbf{T}_{[1...bs1]}$ contient les éléments déjà traités appartenant à **S1** ;
— $\mathbf{T}_{[bi2...N]}$ contient les éléments déjà traités appartenant à **S2** ;
— $\mathbf{T}_{[i...bi2-1]}$ contient les éléments qui n'ont pas encore été traités.

Q1. Exploitation de l'invariant.

— par des dessins analogues à celui de la figure 3.6, illustrer l'état :
  — à l'issue de l'exécution de l'action Répartir ;
  — avant et après le traitement d'un élément courant (2 dessins selon le cas) ;
  — au moment de traiter le dernier élément.
— On observe les variables **bs1**, **i** et **bi2** :
  — Donner leurs valeurs d'initialisation de manière cohérente avec l'invariant.
  — Donner les valeurs minimales et maximales qu'elles peuvent prendre.
  — Indiquer, selon la valeur de $\mathbf{T}_i$, les variables dont la valeur doit être modifiée par le corps de l'itération.
— Déduire de ce qui précède la condition d'arrêt de l'itération.

Q2. Réalisation de l'action.

— Réaliser l'action Répartir en ignorant la contrainte *"S1 doit être triée"*.
— Modifier la solution obtenue pour tenir compte de cette contrainte (exploiter l'algorithme d'insertion dans une séquence triée donné au paragraphe 3.A.b.2.).
— Si l'on supprime la pré-condition X ≤ Y, que faut-il modifier dans la spécification et la réalisation de l'action Répartir ?

Q3. Autres principes de solution.

Donner d'autres principes de solution (sans écrire les algorithmes) : les formuler en français et donner le dessin correspondant à l'invariant d'itération sur le modèle de la figure 3.6.

E3.16 : Suppression des éléments redondants

On dit qu'un élément d'une séquence est redondant s'il existe dans la séquence un autre élément de même valeur.
On étudie la suppression des éléments redondants d'une séquence d'entiers S. Par exemple, si à l'état initial S = [15 , 4 , 19 , 4 , 8 , 11 , 11 , 3 , 4 , 19] alors à l'état final, S = [15 , 4 , 19 , 8 , 11 , 3]. Les séquences d'entiers sont représentées sous forme contiguë avec longueur explicite.

Q1. Donner une réalisation de l'action SupRed spécifiée comme suit :

```
SupRed : action (donnée-résultat T : tableau sur [1...Lmax] d'entier, N : entier sur [0...Lmax])
{ Supprime les éléments redondants de T. On conserve le premier uniquement. }
```

Contrainte de réalisation : l'algorithme ne doit pas utiliser de tableau supplémentaire.

Q2. Donner une autre réalisation de l'action SupRed avec la précondition suivante : les éléments de S sont en ordre croissant.

E3.17 : Sous-séquences connexes

On dit qu'une séquence S1 est une sous-séquence connexe d'une séquence S2 si tous les éléments de S1 apparaissent dans S2, dans le même ordre et dans des positions adjacentes.
Par exemple, la séquence "emp" est une sous-séquence connexe de la séquence "un exemple". Par contre, la séquence "uex" n'est pas une sous-séquence connexe de la séquence "un exemple".
On considère des textes non vides, séquences de caractères représentées de manière contiguë selon le type suivant :

Lmax : constante de type entier >0 *{ longueur maximum des séquences considérés }*
TabCar : type tableau sur [1...Lmax] de caractère
Ent : entier sur [0...Lmax]
{ pour représenter une séquence S *on utilise un tableau* T *de type* TabCar *}*
{ entre les indices 1 *et* N *de type* Ent *}*

— Donner une réalisation de la fonction EstSSConnexe spécifiée comme suit :
EstSSConnexe : fonction (T1, T2 : TabCar non vide ; N1, N2 : Ent) ⟶ booléen
{ vrai ⟺ S1 est sous-séquence connexe de S2 }

E3.18 : Tri par insertion

Étant donné un tableau T d'entiers défini sur l'intervalle [1...n], n > 0, réaliser un algorithme de tri en ordre croissant de T, selon le principe suivant : après les i-1 premières étapes de l'itération réalisant le tri, le sous-tableau défini par les positions 1 et i-1 a été trié ; la $i^{ème}$ étape consiste à mettre l'élément de position i à sa place.
Estimer en fonction de n le nombre de comparaisons d'éléments du tableau qu'engendre cet algorithme, dans les deux cas particuliers suivants : considérer le cas favorable où le tableau est en ordre croissant au départ, et le cas défavorable où le tableau est en ordre décroissant au départ.

c) Représentation d'un ensemble par un vecteur de booléens

E3.19 : Opérations ensemblistes

On considère un ensemble P de n personnes numérotées de 1 à n. Le numéro d'une personne sert à l'identifier.
Tout sous-ensemble E de P peut être représenté par un tableau de booléens défini sur [1...n] avec la convention suivante : si la personne de numéro i appartient au sous-ensemble, l'élément du tableau situé à l'indice i a la valeur vrai ; et si l'élément du tableau situé à l'indice i a la valeur faux.
On définit le type suivant :

n : constante de type entier > 0
SousEnsembleP : type tableau sur [1...n] de booléen
{ E étant de type SousEnsembleP, la personne i appartient à E ⟺ E_i a la valeur vrai. }

En utilisant lexique ci-dessus, étudier chacune des primitives suivantes : pour chacune d'elle, donner un principe d'algorithme (en termes de schéma du cours), donner la réalisation correspondante et les jeux d'essais associés.

Cardinal : fonction (A : SousEnsembleP) ⟶ entier sur [0...n]
{ Nombre d'éléments de l'ensemble A }
EstVide : fonction (A : SousEnsembleP) ⟶ booléen
{ vrai si et seulement si A est l'ensemble vide }

CréerIntersection : action (donnée A, B : SousEnsembleP, résultat C : SousEnsembleP)
{ à l'état final, C est l'intersection des ensembles A et B. }
CréerUnion : action (donnée A, B : SousEnsembleP, résultat C : SousEnsembleP)
{ à l'état final, U est l'union des ensembles A et B. }
CréerDifférence : action (donnée A, B : SousEnsembleP, résultat C : SousEnsembleP)
{ à l'état final, DA est la différence des ensembles A et B. }

E3.20 : A propos d'ensembles de sous-ensembles

On considère un ensemble P de n personnes numérotées de 1 à n, adhérentes d'un club de loisirs. Ce club propose à ses adhérents un ensemble A de k activités, numérotées de 1 à k pour les identifier. Une personne peut être inscrite à une ou plusieurs activités.
On s'intéresse ici aussi bien :
— aux activités auxquelles est inscrite une personne : c'est un sous-ensemble de A,
— qu'aux personnes inscrites à une activité donnée : c'est un sous-ensemble de P.
Dans ce qui suit, on considère un tableau de booléens à deux dimensions, nommé Z et formé de k lignes et de n colonnes :

Z : tableau sur [1...k] de tableau sur [1...n] de booléen
{ $Z_{a,p}$ a la valeur vrai si et seulement si la personne de numéro p est inscrite à l'activité de numéro a. }

Le tableau Z peut être interprété de la manière suivante :
— Chaque ligne est un vecteur de n booléens représentant le sous-ensemble de personnes inscrites à l'activité identifiée par le numéro de cette ligne (comme dans l'exercice **E3.19**).
— De même, chaque colonne est un vecteur de k booléens représentant le sous-ensemble d'activités auxquelles participe la personne identifiée par le numéro de cette colonne.
Le tableau Z a été initialisé et contient l'information concernant les inscriptions des n adhérents aux k activités.

Réaliser des algorithmes correspondants à chacune des questions suivantes, en précisant éventuellement la spécification :
— Affichage des activités auxquelles est inscrite la personne de numéro p.
— Les personnes de numéros p1 et p2 sont-elles inscrites aux mêmes activités (exactement) ?
— Construction du sous-ensemble des activités communes aux deux personnes de numéros p1 et p2.
— Construction du sous-ensemble des personnes inscrites à toutes les activités du club.

E3.21 : Les permutations des N premiers entiers

Nous parlons d'une *permutation* P *d'ordre* N (N>0) pour désigner une séquence de N entiers distincts pris dans l'intervalle [1...N] : tout élément de [1...N] se trouve une fois et une seule dans P. Par exemple, la séquence [5, 9, 1, 8, 2, 6, 4, 7, 3] est une permutation d'ordre 9.
On veut déterminer si une séquence de N entiers est une permutation d'ordre N. On définit pour cela le lexique suivant :

MaxE : constante de type entier > 0 *{ longueur maximum des séquences d'entiers }*
{ soit S une séquence représentée dans le tableau TabE entre les indices 1 et NbE. }
EstPerm : fonction (TabE : tableau sur [1...MaxE] d'entier sur [1...MaxE] ;
NbE : entier sur [0...MaxE]) ⟶ booléen
{ vrai si et seulement si S est une permutation d'ordre NbE }

— Donner une réalisation de la fonction EstPerm selon chacun des principes suivants :

— Le résultat a la valeur vrai si et seulement si tout élément de l'intervalle [1...NbE] appartient à TabE : pour chaque entier de cet intervalle, vérifier qu'il appartient à TabE (combinaison de deux schémas d'existence).
— Le résultat a la valeur vrai si et seulement si tout élément de TabE appartient à l'intervalle [1...NbE] et si les éléments de TabE sont distincts deux à deux : pour chaque élément de TabE vérifier qu'il appartient à l'intervalle et qu'aucun élément le précédant dans TabE n'a la même valeur.

D. Problèmes dirigés

E3.22 : A propos de permutations
La notion de permutation est définie exercice **E3.21**.
Une permutation P d'ordre N peut être utilisée pour définir un ordre entre les éléments d'une séquence S de N éléments : l'élément de rang P_i dans S est le $i^{ème}$ plus petit élément de S selon cet ordre. Par exemple pour S = ["jean","alain","marcelle","aline","zoé","hubert"] :
— P1 = [2,4,6,1,3,5] classe les éléments de S selon l'ordre alphabétique.
— P2 = [5,1,2,4,6,3] classe les éléments de S selon les longueurs des noms.
— P3 = [6,5,4,3,2,1] classe les éléments de S dans l'ordre inverse de celui défini par S.
— P4 = [1,2,3,4,5,6] classe les éléments de S dans l'ordre défini par S.
Nous étudions la construction de permutations associées à une séquence de personnes.

a) Permutation classant les personnes selon une relation d'ordre

Q1. Etude préliminaire à la programmation
On considère une séquence S de N personnes dont on connaît le nom et l'année de naissance, par exemple :
N = 10
S = [<"jean", 1974>, <"alain", 1972>, <"marie", 1976>, <"hubert", 1970>, <"zoé", 1960>, <"aline", 1970>, <"anne", 1973>, <"thomas", 1977>, <"fernand", 1971>, <"marcelle", 1965>, ...]
Soit PNom et PNais, les permutations d'ordre N qui classent les personnes respectivement dans l'ordre alphabétique des noms et dans l'ordre chronologique des années de naissance (deux personnes nées la même année sont classées dans l'ordre alphabétique des noms).
Pour l'exemple ci dessus :
PNom = [2, 6, 7, 9, 4, 1, 10, 3, 8, 5, ...], PNais = [5, 10, 6, 4, 9, 2, 7, 1, 3, 8, ...]
On fixe le lexique suivant :

{ le type Personne }
{ les noms }
Nom : type texte *{ restreint aux lettres de l'alphabet }*
{ Pour comparer des lettres ou des noms selon l'ordre alphabétique, on utilise les symboles habituels sur les nombres. Pour afficher un nom, on utiliser l'instruction Écrire. *}*

{ les dates (précision : année) }
DateMin : constante 1900 de type entier ; DateMax : constante 2012 de type entier
Date : type entier sur [DateMin...DateMax]

{ les personnes }
Personne : type <Nm : Nom, Dnais : Date>
{ nom et année de naissance de la personne. }

{ relations d'ordre sur les personnes }
InfNom : fonction (P1, P2 : Personne) ⟶ booléen
{ vrai ⟺ P1 précède strictement P2 selon l'ordre alphabétique }
InfNais : fonction (P1, P2 : Personne) ⟶ booléen
{ vrai ⟺ P1 précède strictement P2 selon les dates de naissances }

{ les séquences de personnes et les permutations : représentation contiguë avec longueur explicite }
MaxP : constante de type entier > 0 *{ longueur maximum des séquences de personnes }*
TabP : tableau sur [1...MaxP] de Personne
NbP : entier sur [0...MaxP]>
{ une séquence de personnes est représentée dans le tableau TabP entre les indices 1 et NbP. }
TabNom : tableau sur [1...MaxP] d'entier sur [1...MaxP]
TabNais : tableau sur [1...MaxP] d'entier sur [1...MaxP]
{ TabNom et TabNais sont les permutations d'ordre N classant S respectivement selon les noms (ordre alphabétique) et les dates de naissances (ordre chronologique). }

(i) Relations d'ordre sur les personnes
Réaliser les fonctions InfNom et InfNais.

(ii) Construction des permutations : adaptation d'un algorithme de tri
On étudie un algorithme de construction de la permutation TabNom classant une séquence de personnes S selon l'ordre alphabétique des noms. Pour obtenir la permutation, on modifie l'algorithme de tri par sélection du minimum donné en **E3.4** : au lieu d'échanger des éléments de TabP, on échange les éléments de TabNom ; **le tableau** TabP **n'est pas modifié par l'algorithme**. On obtient l'ébauche suivante :

{ état initial de TabNom indifférent. L'ordre considéré est l'ordre alphabétique sur les noms (<) }
•••••••••••••••••••• *{ initialisation de TabNom }*
{ $\forall k \in [1 \dots N], TabNom_k = k$ *}*
pour i allant de 1 à N-1
{ (TabNom)$_{[1...N]}$ *est une permutation d'ordre N. Pour i > 1, (TabNom)*$_{[1...i-1]}$ *contient les indices dans TabP des i-1 plus petites valeurs de TabP et la suite (TabP)*$_{TabNom_1}$*, ..., (TabP)*$_{TabNom_{i-1}}$ *est triée. }*
•••••••••••••••••••• *{ recherche de l'indice dans TabP de la* $i^{ème}$ *plus petite valeur de TabP }*
•••••••••••••••••••• *{ échange de deux valeurs de TabNom }*
{ état final : TabNom est la permutation ordonnant S selon l'ordre alphabétique des noms }

— Compléter l'ébauche ci-dessus pour construire la permutation TabNom (en utilisant InfNom).
— Que faut-il modifier pour construire la permutation TabNais au lieu de TabNom ?

(iii) Utilisation des permutations
— Pour S et PNom données, réaliser un algorithme d'affichage du "plus petit nom", (selon l'ordre alphabétique), commençant par une lettre supérieure ou égale (selon l'ordre alphabétique) à une lettre donnée. Un message particulier sera affiché si un tel nom n'existe pas.
— Pour S et PNais données, réaliser un algorithme qui affiche les noms des personnes dans l'ordre chronologique des années de naissance.

b) Pour continuer : répartition d'un ensemble de personnes en classes d'âges

On étudie un algorithme qui répartit les personnes d'une séquence S en une séquence C de classes d'âges définies par une séquence D de K dates en ordre croissant :

— une classe d'âge est représentée par une séquence d'entiers identifiant les personnes de la classe par leur rang dans S ; une classe d'âge peut être vide.
— la première classe d'âge contient toutes les personnes dont l'année de naissance est strictement inférieure à la première date de D (D_1).
— la K+1ème classe contient toutes les personnes dont l'année de naissance est supérieure ou égale à la dernière date de D (D_K).
— pour i appartenant à [2...K], la ième classe contient toutes les personnes dont l'année de naissance est supérieure ou égale à la i-1ème date (D_{i-1}) **et** strictement inférieure à la ième date (D_i).

Exemple :
S = [<"jean", 1974>, <"alain", 1972>, <"marie", 1976>, <"hubert", 1970>, <"zoé", 1960>, <"aline", 1970>, <"anne", 1973>, <"thomas", 1977>, <"fernand", 1971>, <"marcelle", 1965>]
D = [1963, 1973, 1975, 1976], K = 4, C = [[5], [2, 4, 6, 9, 10], [1, 7], [], [3, 8]]

Q2. Etude préliminaire à la programmation
Pour représenter la partition en classes d'âge, on utilise le lexique suivant :

```
{ Séquence de dates en ordre croissant sous forme contiguë }
   MaxD : constante de type entier > 0                    { nombre maximum de dates }
   TabD : tableau sur [1...MaxD] de Date
   NbD : entier sur [0...MaxD]
      { Soit une séquence de NbD dates ; ses éléments sont placés dans le tableau TabD entre les indices
      1 et NbD. }
{ Séquences de classes : représentation contiguë avec longueur explicite ; chaque classe est une séquence
d'entiers. }
   Classe : type < T : tableau sur [1...MaxP] d'entier sur [1 ...MaxP], N : entier sur [0...MaxP]>
   TabC : tableau sur [1...MaxD+1] de Classe
   NbC : entier sur [0...MaxD+1]
```

(i) Construction des classes d'âges
Réaliser un algorithme qui construit TabC à partir des données S et D. Appliquer le principe suivant : pour chaque élément de la séquence S (*parcours*), déterminer à quelle classe il appartient (*recherche* dans TabD) et placer son numéro dans cette classe (*ajout en queue* de la classe).

(ii) Affichage des noms par classes d'âges
En utilisant directement la permutation PNais de la question **Q1**, réaliser un algorithme qui affiche les noms des personnes par classe d'âge. Pour chaque classe d'âge, on doit préciser ce qui la caractérise ("avant ...", "entre ...et ...", "après ..."), la liste des noms correspondante (en ordre chronologique) ou, le cas échéant, la mention "aucune personne dans cette classe".

E3.23 : Gestion d'une bibliothèque
1. Contexte

Une association, constituée d'un ensemble d'*adhérents*, gère une bibliothèque de prêts : elle possède un certain nombre d'exemplaires (au moins un) de chaque *livre* de son *catalogue*. Un adhérent ne peut emprunter un livre que si les règles suivantes sont toutes respectées :
— La personne doit être adhérente de l'association.

— Le livre doit apparaître dans le catalogue et au moins un exemplaire doit être *disponible* en bibliothèque. Un livre est disponible s'il est dans le catalogue et si au moins un de ses exemplaires n'est pas en cours de prêt.
— Une personne ne peut emprunter qu'un seul exemplaire d'un même titre.
— Le nombre de livres empruntés par une personne à un instant donné ne peut pas dépasser une limite fixée qui est la même pour tous les adhérents.

Le logiciel que l'on considère ici gère le catalogue des livres et le *répertoire* des adhérents :
— Le *catalogue* décrit l'ensemble des livres de la bibliothèque. Pour chaque livre, on dispose des informations suivantes : son titre, le nom de son auteur, le nombre total d'exemplaires acquis par la bibliothèque et le nombre d'exemplaires prêtés à un adhérent. **Un livre est identifié par son titre**.
— Le *répertoire* décrit l'ensemble des adhérents de l'association. Pour chaque adhérent, on dispose des informations suivantes : le nom de l'adhérent et l'ensemble (éventuellement vide) des livres qu'il a en prêt actuellement. **Un adhérent est identifié par son nom**.

Les services attendus du logiciel sont les suivants : *adhésion* d'une nouvelle personne et *départ* d'un adhérent ; *acquisition* de livres par la bibliothèque et *suppression* d'un titre du catalogue ; *emprunt* et *restitution* d'un livre par un adhérent ; affichage d'états divers : du catalogue, du répertoire, de *titres disponibles*, de *titres empruntés*, d'un *hit parade* des auteurs.

2. Représentation des informations
Le catalogue, le répertoire et les séquences d'emprunts sont tous représentés par des séquences implantées dans des tableaux sous forme contiguë avec longueur explicite. **Aucun ordre n'est imposé** entre les éléments de ces séquences.

On fixe le lexique suivant :

a) ***{ Divers }***

```
   texte : type séquence de caractère        { on fait ici abstraction de la représentation des textes }
   { Pour comparer des textes, on utilise les symboles habituels sur les nombres. Pour modifier la valeur
   d'une variable de type texte, on utilise le symbole usuel d'affectation. }
```

b) ***{ Le catalogue : séquence de livres }***

```
{ Informations associées à un livre }
   Livre : type
      < Titre : texte                                          { titre identifiant le livre }
        Auteur : texte                                         { nom identifiant l'auteur }
        NbEx : entier > 0                                      { nombre d'exemplaires acquis par la bibliothèque }
        NbPrêts : entier ≥ 0                                   { nombre d'exemplaires du livre prêtés }
      >
{ Séquence de livres }
   MaxLivres : constante de type entier > 0                    { nombre maximum de livres catalogués }
   AdCat : type entier sur [0...MaxLivres+1]    { 0 et MaxLivres+1 sont prévues pour servir de marque }
   CatLivres : tableau sur [0...MaxLivres+1] de Livre          { le catalogue de la bibliothèque }
   NbLivres : entier sur [0...MaxLivres]                       { nombre de livres dans le catalogue }
   { Les éléments de la séquence sont placés entre les indices 1 et NbLivres de CatLivres. CatLivres0 et
   CatLivresMaxLivres+1 peuvent être utilisées pour placer une sentinelle. }
```

c) ***{ Les emprunts : séquence d'adresses dans le catalogue }***

```
   MaxEmprunts : constante de type entier > 0                  { nombre maximum d'emprunts }
   AdEmp : type entier sur [0...MaxEmprunts+1] { 0 et MaxEmprunts+1 sont prévues pour servir de marque }
```

d) *{ Le répertoire : séquence d'adhérents }*

```
{ Informations associées à un adhérent }
  Adhérent : type
    < Nom : texte                                          { nom identifiant l'adhérent }
      TabEmp : tableau sur [0...MaxEmprunts+1] de AdCat    { emprunts de l'adhérent }
      NbEmp : entier sur [0...MaxEmprunts]                 { nombre d'emprunts de l'adhérent }
      { Les éléments de la séquence sont placés entre les indices 1 et NbEmp de TabEmp. TabEmp0 et
        TabEmpMaxEmprunts+1 peuvent être utilisées pour placer une sentinelle. }
    >

{ Séquence d'adhérents }
  MaxAdh : constante de type entier > 0                    { nombre maximum d'adhérents référencés }
  AdRep : type entier sur [0...MaxAdh+1]      { 0 et MaxAdh+1 sont prévues pour servir de marque }
  RepAdh : tableau sur [0...MaxAdh+1] d'Adhérent           { le répertoire de la bibliothèque }
  NbAdh : entier sur [0...MaxAdh]                          { nombre d'adhérents dans le répertoire }
  { Les éléments de la séquence sont placés entre les indices 1 et NbAdh de RepAdh. RepAdh0 et
    RepAdhMaxAdh+1 peuvent être utilisées pour placer une sentinelle. }
```

3. Compréhension de la représentation et du lexique

Q1. Illustration de la représentation

Illustrer par un dessin les données suivantes où P1,P2,P3 ...dénotent des noms d'adhérents, T1, T2, T3 ...des titres de livres et A1, A2, A3 ...des noms d'auteurs :

- Adhérents (et leurs emprunts) : P1 (T1), P2 (T2, T3), P3 ( ), P4 (T2)
- Livres : T3 par A1, 3 ex. ; T1 par A2, 10 ex. ; T4 par A3, 5 ex. ; T2 par A2, 2 ex.

Q2. Expressions de chemin

Pour accéder au titre du premier ouvrage dans le catalogue, on écrit $\mathsf{CatLivres}_1.\mathsf{Titre}$.
De même, donner les phrases de la notation algorithmique permettant l'accès aux informations suivantes :

- Nombre d'emprunts du k^{ème} adhérent dans le répertoire.
- Auteur du premier ouvrage emprunté par le k^{ème} adhérent.

4. Primitives de gestion de la bibliothèque

Dans ce qui suit, on étudie quelques exemples de primitives de gestion de la bibliothèque. On prend pour hypothèse que les paramètres MaxLivres et MaxAdh servant à dimensionner les tableaux représentant le catalogue et le répertoire sont choisis de telle manière qu'il y ait toujours suffisamment de place lors d'un ajout.

Le type nommé Condition permet de rendre compte du bon fonctionnement de l'exécution d'une opération ou de la détection d'une incorrection des paramètres fournis.

```
Condition : type [Normal, NomAdhérentErroné, NomAuteurErroné,
                  TitreNonCatalogué, LivreDéjàEmprunté, LivreNonEmprunté,
                  TitreNonDisponible, TropDEmprunts, SuppressionImpossible,
                  DépartImpossible, AuteurAbsent, AucunDisponible]
```

Q3. Recherche d'un élément

On spécifie les trois actions suivantes :

```
Fournir_AdRep : action(donnée N : texte ; résultat AR : AdRep, Présent : booléen)
  { Fournit dans AR l'adresse associée à l'adhérent de nom N, s'il est dans le répertoire. Dans ce cas,
    Présent a la valeur vrai ; sinon Présent a la valeur faux et la valeur de AR n'est pas pertinente. }
```



```
Fournir_AdCat : action(donnée T : texte ; résultat AC : AdCat, Présent : booléen)
  { Fournit dans AC l'adresse du livre de titre T, si ce livre est catalogué dans Cat. Dans ce cas, Présent
    a la valeur vrai ; sinon Présent a la valeur faux et la valeur de AC n'est pas pertinente. }
Fournir_AdEmp : action(donnée AC : AdCat, TE : tableau sur [0...MaxEmprunts+1] de AdCat,
                                  NE : entier sur [0...MaxEmprunts] ;
                       résultat AE : AdEmp, Présent : booléen)
  { Fournit dans AE l'adresse de l'emprunt AC, si cet emprunt appartient à la séquence TE de longueur
    NE. Dans ce cas, Présent a la valeur vrai ; sinon Présent a la valeur faux et la valeur de AE n'est pas
    pertinente. }
```

— Donner une réalisation de l'action Fournir_AdRep. Utiliser une technique de sentinelle.

Q4. Adhésion d'une nouvelle personne

Lors de l'adhésion d'un adhérent, celui-ci est enregistré dans le répertoire de manière à indiquer qu'il n'y a encore aucun emprunt à son nom. On spécifie le lexique suivant :

```
EnregistrerAdhésion : action (donnée N : texte, résultat C : Condition)
  { Enregistre un nouvel adhérent de nom N dans le répertoire.
    — Si à l'état initial le répertoire comporte déjà un élément de nom N, alors à l'état final C a la
      valeur NomAdhérentErroné et le répertoire est inchangé.
    — Sinon le répertoire est à jour (nombre de prêts nul) et C a la valeur Normal.
  }
```

— Donner une réalisation de l'action EnregistrerAdhésion en utilisant Fournir_AdRep.

Q5. Acquisition de livres par la bibliothèque

La bibliothèque peut acquérir des exemplaires d'un nouveau titre ou d'un titre déjà catalogué. S'il s'agit d'un nouveau titre, il est enregistré de manière à indiquer qu'aucun de ses exemplaires n'a encore été prêté. On spécifie l'action suivante :

```
EnregistrerAcquisition :
      action (donnée T, A : texte, NE : entier > 0 ; résultat C : Condition)
  { Enregistre l'acquisition de NE exemplaires du livre de titre T et d'auteur A.
    — Si à l'état initial, le catalogue comporte un livre de titre T mais de nom d'auteur différent de
      A, alors à l'état final le catalogue est inchangé et C a la valeur NomAuteurErroné.
    — Sinon, à l'état final, C a la valeur Normal et le catalogue est à jour : ajout de NE exemplaires
      pour un titre existant, ou ajout d'un nouveau titre avec NE exemplaires.
  }
```

— Donner une réalisation de l'action EnregistrerAcquisition en utilisant Fournir_AdCat.

Q6. Consultation du catalogue

On spécifie les primitives suivantes :

```
EstAuteur : fonction (A : texte)  ⟶ booléen                    { sera réutilisée en Q8 }
  { vrai ⟺ A est l'auteur d'au moins un livre du catalogue. }
AfficherDisponibles : action (donnée A : texte, résultat C : Condition)
  { Affiche les titres des livres de l'auteur de nom A qui sont disponibles. Un livre est disponible s'il est
    dans le catalogue et si au moins un de ses exemplaires n'est pas en prêt. À l'état final, C a la valeur :
    — NomAuteurErroné si le catalogue ne comporte pas de titre d'auteur A,
    — AucunDisponible si aucun livre de l'auteur n'est disponible,
    — Normal sinon.
    Il n'y a aucun affichage dans les deux premiers cas. }
```

— Donner une réalisation de la fonction EstAuteur.
— Donner une réalisation de l'action AfficherDisponibles, en utilisant EstAuteur.

Q7. Emprunt d'un livre par un adhérent
L'enregistrement de l'emprunt d'un livre consiste à ajouter un élément dans la liste d'emprunts de l'adhérent et à noter un prêt de plus pour ce livre dans le catalogue. On spécifie l'action suivante :

EnregistrerEmprunt : action (donnée N, T : texte, résultat C : Condition)
{ Enregistre l'emprunt par l'adhérent de nom N d'un exemplaire du livre de titre T.
— Si les règles d'emprunt sont respectées (voir au début de l'énoncé), alors à l'état final, répertoire et catalogue sont à jour et C a la valeur Normal.
— Sinon, à l'état final, catalogue et répertoire sont inchangés et C a la valeur correspondant à la situation rendant l'emprunt impossible (voir type Condition).
}

— Donner une réalisation de l'action EnregistrerEmprunt en utilisant Fournir_AdRep et Fournir_AdCat.

Q8. Consultation du répertoire
— Donner une réalisation de l'action AfficherEmpAuteur spécifiée comme suit, en utilisant Fournir_AdRep et EstAuteur :

AfficherEmpAuteur : action (donnée A, N : texte, résultat C : Condition)
{ Affiche les titres de livres de l'auteur de nom A et prêtés à l'adhérent de nom N. À l'état final C a la valeur :
— NomAuteurErroné si le catalogue ne comporte pas de titre d'auteur A,
— NomAdhérentErroné si le répertoire ne comporte pas d'adhérent de nom N,
— AuteurAbsent si aucun livre écrit par l'auteur A n'a été prêté à l'adhérent de nom N,
— Normal sinon.
Il n'y a aucun affichage dans les trois premiers cas. }

Q9. Restitution d'un livre par un adhérent
Pour enregistrer la restitution d'un livre on supprime l'emprunt de la liste d'emprunts de l'adhérent dans le répertoire. On spécifie à cet effet l'action suivante :

EnregistrerRestitution : action (donnée N, T : texte, résultat C : Condition)
{ Enregistre la restitution par l'adhérent de nom N du livre de titre T.
— Si, à l'état initial, le répertoire ne comporte pas d'élément de nom N, ou si le catalogue ne comporte pas de titre de nom T ou si l'adhérent de nom N n'a pas emprunté de livre de titre T, alors à l'état final, répertoire et catalogue sont inchangés et C a la valeur rendant compte de la situation (voir type Condition).
— Sinon, répertoire et catalogue sont à jour et C a la valeur Normal.
}

— Donner une réalisation de l'action EnregistrerRestitution en utilisant Fournir_AdRep, Fournir_AdCat et Fourni_AdEmp.

Q10. Suppression d'un titre du catalogue
Un titre du catalogue ne peut être supprimé que si aucun de ses exemplaires n'est en prêt. On spécifie l'action suivante :

EnregistrerSuppression : action (donnée T : texte, résultat C : Condition)
{ Enregistre la suppression du catalogue du livre de titre T.
— Si à l'état initial, le catalogue ne comporte pas de titre de nom T, ou si au moins un des exemplaires de ce titre est emprunté, alors à l'état final, le catalogue est inchangé et C a selon le cas l'une des valeurs TitreNonCatalogué ou SuppressionImpossible.
— Sinon, à l'état final, le catalogue est à jour et C a la valeur Normal.
}

— Donner une réalisation de l'action EnregistrerSuppression en utilisant Fournir_AdRep.

Q11. Départ d'un adhérent
Un adhérent ne peut quitter l'association que lorsqu'il a restitué tous les livres empruntés. On spécifie l'action suivante :

EnregistrerDépart : action (donnée N : texte, résultat C : Condition)
{ Enregistre le départ de l'adhérent de nom N.
— Si, à l'état initial, le répertoire ne comporte pas d'élément de nom N, ou si l'adhérent a au moins un livre en prêt, alors à l'état final, répertoire et catalogue sont inchangés et C a selon le cas l'une des valeurs NomAdhérentErroné ou DépartImpossible.
— Sinon, le répertoire est à jour et C a la valeur Normal.
}

— Donner une réalisation de l'action EnregistrerDépart en utilisant Fournir_AdRep.

Q12. Hit parade des auteurs
On spécifie l'action suivante :

AfficherHitParade : action
{ Affiche la liste des auteurs dont au moins un livre est actuellement en prêt. Pour chaque auteur, on trouve son nom et le nombre total d'exemplaires de livres écrits par lui et actuellement en prêt. La liste des auteurs est affichée en ordre décroissant du nombre de livres en prêt. }

Indication pour la réalisation
Pour élaborer le hit parade, on parcourt le catalogue et on construit progressivement un tableau intermédiaire, nommé Hit. Chaque élément du tableau est un score, couple <*auteur, nombre de lecteurs*>. L'itération construisant Hit doit respecter l'invariant suivant : *"Hit est classé en ordre décroissant du nombre de livres en prêt"*.

On suppose ici que l'on connaît le nombre maximum Nmax d'auteurs de livres du catalogue.

On complète donc le lexique comme suit :

Nmax : constante de type entier > 0 *{ nombre maximum d'auteurs de livres dans le catalogue }*
Score : type <NA : texte, NbE : entier ≥ 0 > *{ <Nom d'Auteur, Nombre d'Emprunts> }*
Hit : tableau sur [0. . .Nmax+1] de Score
NbScores : entier sur [0. . .Nmax]
{ Les scores sont placés dans le tableau Hit entre les positions 1 et NbScores. Lorsque la table est vide, NbScores=0. Dans Hit, les noms d'auteurs sont distincts 2 à 2 et les scores sont en ordre décroissant selon le nombre de lecteurs. Les positions 0 et Nmax+1 sont prévues pour utiliser, le cas échéant, une technique de sentinelle. }

4. Séquences et Chaînage

A. Eléments de cours

a) Terminologie

Cellule : case d'un tableau, champ d'un produit, mot mémoire.

Adresse : indice d'un tableau, adresse en mémoire.

Accès direct : accès au contenu d'une cellule par son adresse.

Identificateur de variable : un compilateur associe une adresse à chaque identificateur de variable. Utiliser l'identificateur pour dénoter la valeur de la variable implique un accès direct au travers de l'adresse associée.

Indirection, accès indirect : une cellule d'adresse A contient l'adresse B d'une autre cellule. L'adresse A permet d'accéder indirectement au contenu de la cellule d'adresse B.

Liens explicites : représentent des associations entre informations et sont représentés par des adresses. Association d'une même information et de plusieurs autres informations. Association d'un élément d'une séquence et de son successeur.

b) Représentation chaînée d'une séquence

b.1. Principes

On utilise un *espace mémoire* formé d'un ensemble de *cellules* accessibles par leur *adresse*. Chaque cellule est un *doublet* pouvant contenir un élément et une adresse de cellule. Par convention, il existe une adresse particulière appelée **Nil** qui représente "nulle part".

Une séquence est représentée sous forme chaînée à l'aide d'un ensemble de doublets : chaque élément de la séquence se trouve dans un doublet qui contient l'adresse de son successeur. Pour distinguer le dernier élément de la séquence, on convient que l'adresse de son successeur vaut **Nil**.

L'accès à une liste chaînée non vide se fait par l'intermédiaire de l'adresse de la cellule comportant le premier élément de la liste. Cette adresse est appelée *adresse de tête* de la liste (plus brièvement *tête de liste*). Une liste vide est caractérisée par une tête de valeur **Nil**. L'accès à un élément ou à l'adresse de son successeur se fait toujours indirectement par l'adresse de la cellule qui contient ces valeurs. Le dessin 4.1 figure une liste chaînée d'adresse de tête **T** :

Un tel dessin permet de faire abstraction des valeurs d'adresse.

Les conventions graphiques sont les suivantes :

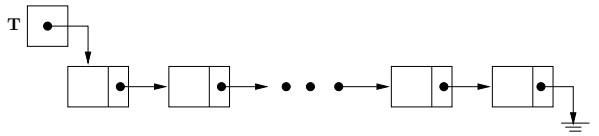
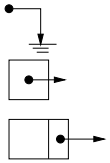


Figure 4.1 – Représentation d'une liste chaînée

Une flèche fournit deux informations : le point d'arrivée (à n'importe quel endroit du contour d'une cellule) désigne une adresse de doublet ; le point de départ (à l'intérieur d'une cellule) désigne l'endroit où cette adresse est mémorisée, que ce soit dans un doublet ou dans une variable de type adresse.



Représentation d'un lien vers "nulle part".

Une variable de type adresse.

Une cellule structurée en doublet comportant un champ de type adresse.

b.2. Lexique général pour la description d'une liste chaînée

De manière générale, on utilise un lexique de la forme suivante :

```
Élément : type                                   { type des éléments que l'on veut chaîner }
adCel : type                                     { adresses de cellules de type Cel }
Cel : type <Elt : Élément, Suc : adCel>          { élément et lien vers son successeur }
Nil : constante de type adCel                    { lien vers "nulle part" }
T : adCel                                        { une adresse de tête par liste considérée }
    { Une liste chaînée non vide est accessible par son adresse de tête T. Le dernier élément d'une liste
    non vide a Nil pour successeur. T = Nil caractérise la liste vide. }
```

La définition du type **adCel** et la valeur de la constante **Nil** doivent être précisées selon que le chaînage est géré à l'aide de pointeurs (cf ci-dessous 4.A.c)) ou dans un tableau (cf ci-dessous 4.A.d)).

b.3. Interface pour la gestion de l'espace mémoire

Une cellule de l'espace mémoire est dans l'état *occupé* si elle est utilisée pour représenter un élément d'une séquence. Dans le cas contraire, la cellule est dans l'état *libre*. L'ensemble des cellules libres constitue la *mémoire libre*.
L'espace mémoire est dans l'état *mémoire saturée* lorsque toutes ses cellules sont occupées (la mémoire libre est vide). Il est dans l'état *mémoire vide* lorsque toutes ses cellules sont libres.
La valeur d'un doublet ne permet en aucun cas de déterminer si la cellule qui le contient est libre ou occupée. Un système de gestion de la mémoire doit maintenir des informations spécifiques permettant d'identifier les cellules libres, de manière à pouvoir répondre à deux types de requêtes :

— requête d'*allocation* : l'utilisateur demande une adresse de cellule libre pour pouvoir l'utiliser à son gré. Le système choisit l'une des cellules libres, en fournit l'adresse à l'utilisateur et la marque comme étant occupée.
— requête de *libération* : l'utilisateur informe le système qu'il n'utilisera plus une cellule qui lui a été allouée auparavant. Le système marque cette cellule comme étant à nouveau libre.

Pour faire abstraction des modalités de cette gestion mémoire, nous utilisons les primitives suivantes :

MémoireSaturée : fonction ⟶ booléen *{ vrai ⟺ la mémoire est saturée }*
Allouer : action (résultat A : adCel)
{ à l'état final, A contient l'adresse d'une cellule qui était dans l'état "libre". Cette cellule est dans l'état "occupé". Pré-condition : la mémoire n'est pas saturée }
Libérer : action (donnée A : adCel)
{ à l'état final, la cellule d'adresse A est libre et la mémoire n'est pas saturée. Pré-condition : A ≠ Nil }

b.4. Modifications du chaînage

Pour schématiser les modifications des liens de chaînage d'une liste, nous représentons sur le même dessin l'état initial et l'état final avec les conventions suivantes :
— Une flèche pleine représente l'état initial et une flèche en pointillés représente l'état final.
— Les flèches en pointillés sont étiquetées par des entiers qui définissent l'ordre dans lequel les modifications de chaînage doivent être faites.

Sur un tel dessin, certains doublets ou variables de type adresse ont deux flèches qui les quittent, l'une en plein, l'autre en pointillés : ce changement d'état se traduit au niveau de l'algorithme par une affectation portant sur un nom (de variable ou de champ de doublet) de type adresse.
A titre d'exemple, nous schématisons l'insertion d'un élément **E** dans une liste, après l'élément situé dans la cellule d'adresse **a**.

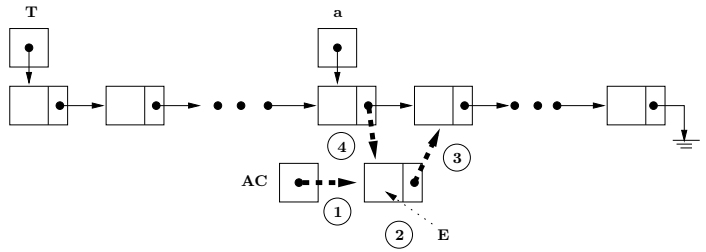


Figure 4.2 – Modifications de chaînage dans une liste chaînée

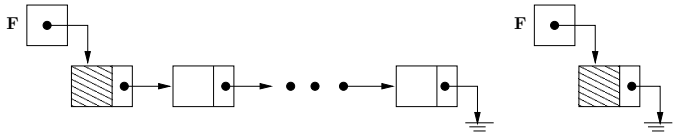
Le dessin 4.2 illustre l'analyse de l'insertion. Il faut :
— (1) Allouer une cellule de doublet libre : son adresse est rangée dans la variable de nom **AC**, paramètre de l'action **Allouer**.
— (2) Ranger **E** dans ce doublet.
— (3,4) Connecter la cellule d'adresse **AC** après celle d'adresse **a**, par modification de deux liens de chaînage.

Lorsque l'insertion se fait en tête, **a** n'est pas défini : c'est alors la valeur de **T** qui doit être modifiée par l'opération 4.

b.5. Représentation chaînée avec *élément fictif de tête*

Dans la représentation *chaînée avec élément fictif de tête*, l'adresse donnant accès à la liste est l'adresse d'une cellule dont :
— le champ destiné à contenir un élément de la séquence n'a pas de valeur spécifiée : on l'appelle *élément fictif* pour indiquer qu'il n'appartient pas à la séquence représentée.
— le champ destiné à contenir l'adresse du successeur
  — contient l'adresse de la cellule contenant le premier élément de la séquence représentée, si elle n'est pas vide,
  — contient Nil si la séquence représentée est vide.

Ceci est illustré par la figure 4.3. Cette représentation est utilisée pour simplifier la réalisation de certains algorithmes (voir l'exemple **E4.6** et l'exercice **E4.14**).



Liste non vide
Le successeur du fictif est le 1er élément de la séquence

Liste vide
Le fictif n'a pas de successeur

Figure 4.3 – Représentation chaînée avec fictif

c) Gestion du chaînage à l'aide de pointeurs

c.1. Valeurs et variables de type Pointeur

Le langage fournit un constructeur de type **pointeur de ...**. Pour manipuler un ensemble d'adresses de cellules comportant des éléments d'un type nommé **X**, on introduit un nouveau type, nommé **adX** par exemple, de la manière suivante :

adX : type pointeur de X *{ une valeur de type adX est l'adresse d'une cellule de type X }*

Le langage fournit de plus une constante nommée **Nil** qui appartient à tout type construit avec le constructeur **pointeur de ...**.
A étant un pointeur (une adresse de type pointeur), **A↑** dénote le contenu de la cellule dont **A** est l'adresse : si **A** est de type **pointeur de X** alors **A↑** est de type **X**.
L'écriture **Nil↑** est incorrecte (**Nil** représentant "nulle part", il n'y a pas lieu de chercher à accéder à son contenu ou de le modifier).
Les primitives de gestion de l'espace mémoire **Allouer**, **Libérer**, et **MémoireSaturée** sont des primitives du langage. L'initialisation de cet espace est réalisée par le système à l'exécution.

c.2. Lexique général

Élément : type *{ type des éléments que l'on veut chaîner }*
adCel : type pointeur de Cel
{ Nil est pré-défini dans le langage et n'apparaît donc pas dans le lexique }
Cel : type <Elt : Élément, Suc : adCel> *{ élément et lien vers son successeur }*
T : adCel *{ une adresse de tête par liste considérée }*
{ Une liste chaînée non vide est accessible par une adresse de tête T. Le dernier élément d'une liste non vide a Nil pour successeur. T = Nil caractérise la liste vide. }

Un élément de liste chaînée est accessible indirectement via l'adresse **A** de la cellule qui le comporte : **A↑** dénote le doublet correspondant, **A↑.Elt** dénote l'élément et **A↑.Suc** dénote le lien vers son successeur.

c.3. Schémas de traitement séquentiel

Les schémas de parcours et de recherche sont construits en raisonnant sur la suite des adresses des doublets marquée par **Nil**.

Parcours

AC : adCel
Initialiser le traitement ; AC ⟵ T
tant que AC ≠ Nil
  Traiter AC↑.Elt ; AC ⟵ AC↑.Suc

Recherche du premier élément vérifiant une propriété P

AC : adCel
P : fonction (E : Elément) ⟶ booléen
AC ⟵ T ; tant que AC ≠ Nil et puis non P(AC↑.Elt) : AC ⟵ AC↑.Suc
si AC = Nil alors *{ non trouvé }* ●●●
sinon *{ trouvé en AC }* ●●●

d) Gestion du chaînage dans un tableau mémoire

L'espace mémoire est décrit par un **tableau de doublets**. Une adresse de cellule est un **indice** dans ce tableau.

d.1. Lexique général

Le lexique donné en 4.A.b)(ii) ci-dessus est précisé de la manière suivante :

Tmem : constante de type entier > 0 *{ taille de la mémoire }*
adCel : type entier sur [0...Tmem] *{ adresses de cellules de type Cel }*
Mem : tableau sur [1...Tmem] de Cel
Nil : constante 0 de type adCel *{ lien vers "nulle part" }*
Élément : type *{ type des éléments que l'on veut chaîner }*
Cel : type <Elt : Élément, Suc : adCel> *{ élément et lien vers son successeur }*
T : adCel *{ une adresse de tête par liste considérée }*
  { Une liste chaînée non vide est accessible par une adresse de tête T. Le dernier élément d'une liste non vide a Nil pour successeur. T = Nil caractérise la liste vide. }

Un élément de liste chaînée est accessible indirectement via l'adresse **A** de la cellule qui le comporte : $\mathbf{Mem}_A$ dénote le doublet correspondant, $\mathbf{Mem}_A$**.Elt** dénote l'élément et $\mathbf{Mem}_A$**.Suc** dénote le lien vers son successeur.

d.2. Schémas de traitement séquentiel

Les schémas de parcours et de recherche sont construits en raisonnant sur la suite des adresses des doublets marquée par **Nil**.

Parcours

AC : adCel
Initialiser le traitement ; AC ⟵ T
tant que AC ≠ Nil
  Traiter Mem_{AC}.Elt ; AC ⟵ Mem_{AC}.Suc

Recherche du premier élément vérifiant une propriété P

AC : adCel
AC ⟵ T ; tant que AC ≠ Nil et puis non P(Mem_{AC}.Elt) : AC ⟵ Mem_{AC}.Suc
si AC = Nil alors *{ non trouvé }* ●●●
sinon *{ trouvé en AC }* ●●●

d.3. Réalisation des primitives de gestion de l'espace mémoire

Nous illustrons ici une technique classique (parmi d'autres) utilisant les liens de chaînage des doublets. Les cellules libres sont chaînées entre elles dans un ordre quelconque, formant ainsi une liste chaînée appelée *liste libre*. Cette liste évolue dans le temps comme une pile :

- Pour initialiser la liste libre avec toutes les cellules, elles sont chaînées entre elles dans l'ordre de leurs adresses.
- Lors d'une *allocation* (demande d'une adresse de cellule libre), l'adresse de cellule fournie est celle du premier élément de la liste libre : elle est alors déconnectée de la liste libre.
- Lors de la *libération* d'une cellule (restitution d'une cellule qui n'est plus occupée), elle est connectée en tête de la liste libre.

Les primitives de gestion sont alors réalisées comme suit :

Tlibre : adCel *{ adresse de tête de la liste libre }*
InitialiserMémoire : action
  { e.i. indifférent ; e.f. : la mémoire est vide }
MémoireSaturée :
  retour : Tlibre = Nil
Allouer(A) :
  A ⟵ Tlibre ; Tlibre ⟵ Mem_{Tlibre}.Suc *{ suppression en tête de liste libre }*
Libérer(A) :
  Mem_A.Suc ⟵ Tlibre ; Tlibre ⟵ A *{ insertion en tête de liste libre }*
InitialiserMémoire :
  pour AC allant de 1 à Tmem-1 : Mem_{AC}.Suc ⟵ AC + 1
  Mem_{Tmem}.Suc ⟵ Nil
  Tlibre ⟵ 1

B. Exemples d'exercices rédigés

a) Gestion du chaînage à l'aide de pointeurs

Dans les exemples qui suivent, les liens de chaînage sont représentés par des **pointeurs** (cf 4.A.c)) et on traite des listes d'entiers.

adCelEnt : type pointeur de CelEnt *{ adresses de cellules }*
CelEnt : type <E : entier, Suc : adCelEnt> *{ élément et lien vers son successeur }*
{ Une liste chaînée est caractérisée par son adresse de tête. }

Exemple E4.1 : Valeur du dernier élément

Étant donnée une liste chaînée d'entiers, réaliser un algorithme qui affiche la valeur du dernier élément de la liste (ou un message si la liste est vide). La liste est donnée par son adresse de tête **T**.

Pour une liste non vide, le dernier élément est situé dans la cellule dont le lien successeur vaut **Nil**. On adapte un schéma de recherche, sachant que cet élément existe toujours dans ce cas.

```
T : adCelEnt                                   { adresse de tête de la liste donnée }
AC : adCelEnt                                  { adresse de l'élément courant }
si T = Nil alors Écrire("liste vide")
sinon      AC ⟵ T
           tant que AC↑.Suc ≠ Nil              { le dernier existe }
                AC ⟵ AC↑.Suc
           Écrire (AC↑.E)
```

Exemple E4.2 : Construction d'une liste chaînée

Réaliser un algorithme qui construit une liste chaînée des entiers de 1 à n (n ≥ 0 donné), en ordre croissant. Si n est nul, la liste construite est vide. On suppose que l'espace mémoire comporte au moins n cellules libres.

```
n : constante de type entier ≥ 0
T : adCelEnt                                   { adresse de tête de la liste créée }
```

(i) Version 1, par ajouts en queue : parcours de l'intervalle [1...n].
Au moment de traiter l'entier i, la liste chaînée des entiers de l'intervalle **[1...i-1]** a été construite : il s'agit d'un ajout en queue. Ceci nécessite de maintenir une adresse de queue, soit **Q**. La liste est initialisée avec l'élément de valeur 1. Le dessin 4.4 figure l'invariant de l'itération et les modifications de chaînage qui doivent être effectuées dans le corps de l'itération (flèches en pointillés).

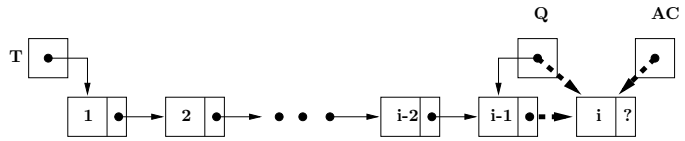


Figure 4.4 – Construction d'une liste chaînée par ajouts en queue

```
Q : adCelEnt                                   { adresse courante de queue }
AC : adCelEnt                                  { adresse de cellule pour l'élément courant }
si n = 0 alors T ⟵ Nil                         { construction de la liste vide }
sinon { création d'une liste singleton avec adresse de queue }
   Allouer (T) ; T↑.E ⟵ 1 ; Q ⟵ T
   pour i allant de 2 à n
      Allouer (AC) ; AC↑.E ⟵ i                 { création nouveau doublet }
      Q↑.Suc ⟵ AC ; Q ⟵ AC                     { connexion en queue }
   Q↑.Suc ⟵ Nil
```

(ii) Version 2, par ajouts en tête : parcours en sens inverse de l'intervalle [1...n].
Au moment de traiter l'entier i, la liste chaînée des entiers de **i+1** à **n** a déjà été construite : il s'agit d'un ajout en tête.

```
AC : adCelEnt                                  { adresse de doublet pour l'élément courant }
T ⟵ Nil                                        { construction de la liste vide }
pour i allant de n à 1, pas -1
   Allouer (AC) ; AC↑.E ⟵ i                    { création nouveau doublet }
   AC↑.Suc ⟵ T ; T ⟵ AC                        { connexion en tête }
```

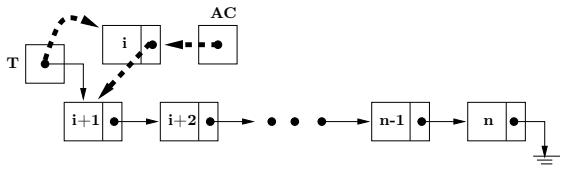



Figure 4.5 – Construction d'une liste chaînée par ajouts en tête

Exemple E4.3 : Une liste est-elle en ordre croissant ?

On veut déterminer si une séquence non vide d'entiers est en ordre croissant. La séquence est donnée sous forme d'une liste chaînée. Réaliser une fonction nommée **EstCrois** *spécifiée comme suit :*

```
EstCrois : fonction (T : adCelEnt) ⟶ booléen
   { vrai ⟺ la séquence d'entiers représentée par la liste chaînée d'adresse de tête T est en ordre
   croissant. Pré-condition : T ≠ Nil }
```

On raisonne sur la séquence des couples d'éléments consécutifs de la séquence donnée : recherche d'un couple <**e,f**> tel que **e** > **f**. On applique un schéma de parcours (version 1) ou de recherche avec arrêt dès que possible (version 2).

(i) Version 1, schéma de parcours

```
EstCrois(T) : { T ≠ Nil }
   AC, AP : adCelEnt                           { adresses de l'élément courant et de son prédécesseur }
   Existe : booléen
   Existe ⟵ faux
   AP ⟵ T ; AC ⟵ T↑.Suc                        { premier couple }
   tant que AC ≠ Nil
      Existe ⟵ Existe ou AP↑.E > AC↑.E
      AP ⟵ AC ; AC ⟵ AP↑.Suc                   { couple suivant }
   retour : non Existe
```

(ii) Version 2, schéma de recherche

```
EstCrois(T) : { T ≠ Nil }
   AC, AP : adCelEnt                           { adresses de l'élément courant et de son prédécesseur }
   AP ⟵ T ; AC ⟵ T↑.Suc                        { premier couple }
   tant que AC ≠ Nil et puis AP↑.E ≤ AC↑.E
      AP ⟵ AC ; AC ⟵ AP↑.Suc                   { couple suivant }
   retour : AC=Nil
```

Exemple E4.4 : Inversion d'une liste chaînée

On donne une liste chaînée d'éléments de type quelconque. Décrire (spécification, réalisation) une action qui construit la liste chaînée inverse de la liste donnée, par modification des liens de chaînage.

Spécification

```
Élément : type
Cel : type <Elt : Élément, Suc : adCel> ; adCel : type pointeur de Cel
```

```
Inverser : action (donnée-résultat T : adCel)
    { Inverse la liste donnée par son adresse de tête T, par modification des liens de chaînage. À l'état
    final, T est l'adresse de tête de la liste inverse. }
```

Réalisation

On raisonne sur deux listes chaînées nommées **L** et **LI** : **L**, de tête **T**, comporte les éléments qui n'ont pas encore été traités, dans l'ordre initial ; **LI**, de tête **TI**, comporte les éléments déjà traités, en ordre inverse par rapport à l'ordre initial. A chaque étape, la cellule de tête de **L** est déconnectée, pour être connectée en tête de **LI**.

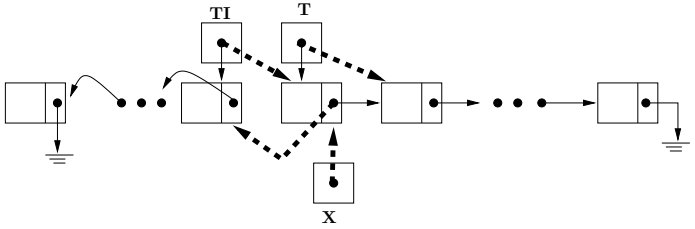


Figure 4.6 – Inversion d'une liste chaînée

```
Inverser(T) :
    TI : adCel                                              { adresse de tête de LI }
    X : adCel                                               { pour déconnexion, connexion }
    TI ⟵ Nil                                                { construction de la liste vide }
    tant que T ≠ Nil
        X ⟵ T ; T ⟵ T↑.Suc                                  { déconnexion en tête de L }
        X↑.Suc ⟵ TI ; TI ⟵ X                                { connexion en tête de LI }
    T ⟵ TI
```

Exemple E4.5 : Tri par insertion

Réaliser un algorithme de tri selon le principe du "tri par insertion" : dans un parcours de la séquence, l'élément courant est inséré en bonne place parmi ceux qui le précèdent (qui sont en ordre croissant). L'algorithme doit procéder par modification des liens de chaînage de cette liste, les valeurs des éléments n'étant pas déplacées. La donnée est une liste chaînée d'entiers.

On raisonne sur deux listes chaînées nommées **L1** et **L2** (voir figure ci-dessous) : **L1**, de tête **T** et de queue **Q**, est la liste des éléments déjà traités, en ordre croissant ; **L2**, dont le premier élément est l'élément courant, est la liste des éléments qui restent à traiter.
AC est l'adresse de l'élément courant, tête de **L2**. Traiter l'élément courant consiste à le connecter en bonne place dans **L1** : dans le cas général, il faut trouver les deux cellules consécutives, d'adresses **ap** et **a**, entre lesquelles doit avoir lieu l'insertion. Il y a deux cas particuliers : insertion en tête (**ap** n'est pas défini) ou insertion en queue (**AC** est en place).
Le point d'insertion dans **L1** de la cellule d'adresse **AC** peut être défini de deux manières :

— **a** est l'adresse du premier doublet (dans le sens du chaînage) vérifiant **a↑.E** > **AC↑.E** ;
— ou bien **a** est l'adresse du premier doublet vérifiant **a↑.E** ≥ **AC↑.E**.

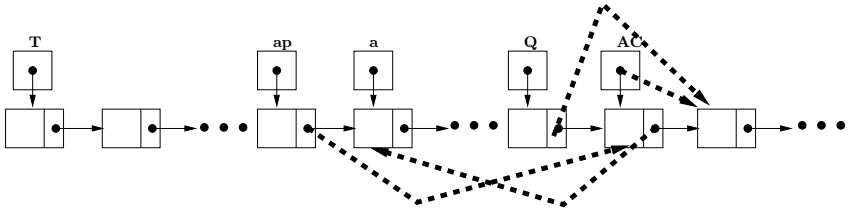


Figure 4.7 – Tri par insertion, cas général

Nous traitons la première solution qui conserve l'ordre entre les éléments égaux (tri *stable*).

```
adCelEnt : type pointeur de CelEnt ; CelEnt : type <E : entier ; Suc : adCelEnt>
T : adCelEnt                                    { adresse de tête de la liste donnée et de la liste résultat }
Q : adCelEnt                                    { adresse de queue de L1 }
AC : adCelEnt                                   { adresse du doublet courant : tête de L2. Q↑.Suc = AC }
EC : entier                                     { Élément courant : EC = AC↑.E }
ap, a : adCelEnt                                { adresses des doublets entre lesquels il faut insérer EC }
si T ≠ Nil alors
    Q ⟵ T                                       { L1 est initialisée par une liste singleton }
    AC ⟵ Q↑.Suc                                 { premier élément à traiter }
    tant que AC ≠ Nil
        { L1 est triée. L'élément courant (premier de L2) a pour adresse AC = Q↑.Suc }
        EC ⟵ AC↑.E
        si Q↑.E ≤ EC alors Q ⟵ AC                                   { AC est déjà en place }
        sinon { Q↑.E > EC : insérer EC quelque part avant Q }
            Q↑.Suc ⟵ AC↑.Suc                                        { déconnexion de AC }
            si EC < T↑.E alors { insérer EC en tête de L1 }
                AC↑.Suc ⟵ T ; T ⟵ AC
            sinon { rechercher ap et a entre lesquels insérer EC. Q est sentinelle }
                ap ⟵ T ; a ⟵ ap↑.Suc
                tant que a↑.E ≤ EC                                  { on est sûr d'arrêter au plus loin avec a = Q }
                    ap ⟵ a ; a ⟵ ap↑.Suc
                ap↑.Suc ⟵ AC ; AC↑.Suc ⟵ a                          { insertion entre ap et a }
        AC ⟵ Q↑.Suc                                                 { obtention du prochain élément à traiter }
```

Exemple E4.6 : Interclassement de deux listes triées (utilisation d'un élément fictif)

On étudie l'interclassement de deux séquences d'entiers triées. Les séquences sont données sous forme de deux listes chaînées **L1** *et* **L2** *de têtes* **T1** *et* **T2**. *Réaliser une action nommée* **Interclasser** *spécifiée comme suit :*

```
CelEnt : type <E : entier, Suc : adCelEnt> ; adCelEnt : type pointeur de CelEnt
Interclasser : action (donnée-résultat T1, T2 : adCelEnt)
    { Interclasse deux listes chaînées d'entiers en ordre croissant L1 et L2, de têtes T1 et T2. À l'état
    final, le résultat est fourni par la liste L1 ; la liste L2 est vide. L'algorithme procède par modification
    des liens de chaînage de L1 et L2, sans déplacement des éléments. }
```

Le principe choisi est de modifier **L1** par insertion en bonne place de chacune des cellules de **L2**. On construit une itération de parcours simultané des deux listes : **AC1** et **AC2** étant les

adresses des éléments courants, on détermine s'il y a lieu de placer la cellule d'adresse **AC2** avant la cellule d'adresse **AC1**. Pour réaliser les insertions, on maintient l'adresse **AP1** de la cellule précédant celle d'adresse **AC1**. **AP1** n'est pas définie lorsque **AC1** est la tête de **L1**. Pour éliminer ce cas particulier, un élément fictif, d'adresse **F**, est placé en tête de **L1**. Il est supprimé à la fin de l'interclassement.

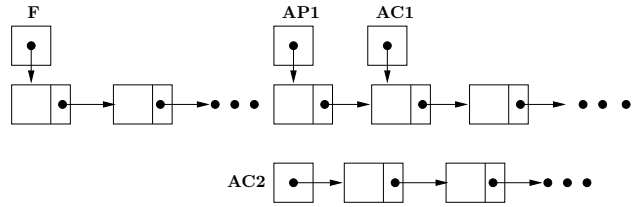


Figure 4.8 – Interclassement de deux listes chaînées

```
Interclasser(T1, T2)
    F : adCelEnt                                          { pour l'élément fictif en tête de L1 }
    AC1, AC2 : adCelEnt                                   { adresses des éléments courants }
    AP1 : adCel                                           { adresse du prédécesseur de l'élément d'adresse AC1 }
    X : adCel                                             { auxiliaire pour déconnexion, connexion }
    Allouer(F) ; F↑.Suc ⟵ T1                              { placement d'un élément fictif en tête de L1 }
    AP1 ⟵ F ; AC1 ⟵ T1 ; AC2 ⟵ T2
    tant que AC1 ≠ Nil et AC2 ≠ Nil
    { Tous les éléments précédant les éléments d'adresses AC1 et AC2 dans les listes données initialement
    se trouvent entre les adresses F↑.Suc et AP1, et ceci en ordre croissant. De plus, si AP1 ≠ F, AP1↑.E ≤
    AC1↑.E et, si AC2 ≠ nil, alors AP1↑.E ≤ AC2↑.E. }
        si AC1↑.E ≤ AC2↑.E alors
            AP1 ⟵ AC1 ; AC1 ⟵ AP1↑.Suc
        sinon { déconnecter AC2 pour le connecter entre AP1 et AC1 }
            X ⟵ AC2 ; AC2 ⟵ AC2↑.Suc                      { déconnexion }
            AP1↑.Suc ⟵ X ; X↑.Suc ⟵ AC1 ; AP1 ⟵ X         { connexion }
    { Le cas échéant, compléter par ce qui reste dans L2 }
    si AC2 ≠ Nil alors AP1↑.Suc ⟵ AC2
    T1 ⟵ F↑.Suc ; Libérer(F)                              { suppression du fictif de tête }
    T2 ⟵ Nil
```

Remarque : la variable **AC2** peut être omise, en la remplaçant par la variable **T2**.

b) Séquences de séquences

Exemple E4.7 : Partition d'un ensemble de mots

On considère un ensemble E *de mots et la partition* P *des mots de* E *selon leur initiale. Chaque classe de* P *est caractérisée par une lettre de l'alphabet : c'est le sous-ensemble de* E *de tous les mots ayant cette lettre pour initiale.*
Les ensembles sont représentés par des séquences sans répétition : E *est une séquence de mots ;* P *est une séquence de classes ; chaque classe est une séquence* **non vide** *de mots.*
Toutes ces séquences *sont sous forme de* **listes chaînées** *:* E *est représenté par une liste de tête* TE *;* P *est représentée par une liste de tête* TP *dont les éléments sont les têtes des listes chaînées représentant les classes. La figure 4.9 schématise cette représentation.*

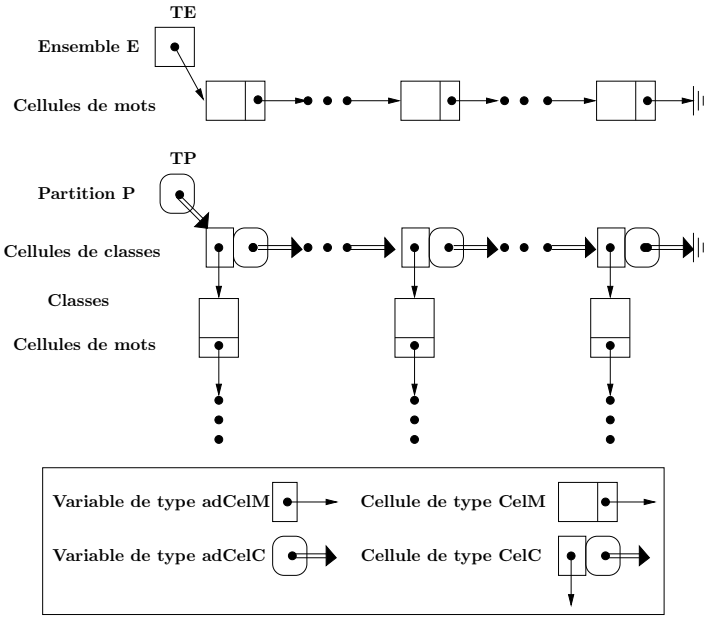


Figure 4.9 – Partition d'un ensemble de mots

On fait abstraction de la représentation des mots à l'aide d'un type nommé Mot.
Aucun ordre *n'est imposé sur les listes (de classes ou de mots). On utilise le lexique suivant :*

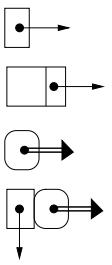
```
{ Mots }
    Lettre : type caractère                               { restreint aux lettres de l'alphabet }
    Mot : type séquence non vide de Lettre
        { M étant de type Mot, premier(M) est l'initiale de M ; pour affecter une valeur de mot à une
        variable de type Mot, on utilise le symbole d'affectation usuel ( ⟵ ). }

{ Listes de mots }
    adCelM : type pointeur de CelM

    CelM : type <M : Mot, Msuiv : adCelM>

{ Listes de Classes (ensembles de mots de même initiale) }
    adCelC : type pointeur de CelC

    CelC : type <T : adCelM, Csuiv : adCelC>
```



```
{ L'ensemble E et la partition P }
   TE : adCelM                          { tête de la liste de mots représentant E }
   TP : adCelC                          { tête de la liste de classes représentant P }
```

Q1. Une expression de chemins dans la structure

On donne l'adresse de tête TP *de la liste de classes représentant la partition d'un ensemble de mots : compléter l'instruction* Écrire(••••••) *pour afficher la lettre initiale des mots de la deuxième classe de la partition ; donner la pré-condition qui doit être satisfaite avant l'exécution de cette instruction.*

- TP↑.Csuiv est l'adresse du deuxième élément de la liste représentant P ou Nil s'il n'y en a pas (précondition : TP≠Nil) ;
- (TP↑.Csuiv)↑.T est l'adresse de tête de la liste non vide représentant la deuxième classe de P (précondition : TP≠Nil et puis TP↑.Csuiv ≠Nil) ;
- ((TP↑.Csuiv)↑.T)↑.M est le premier mot de la deuxième de classe de P (précondition : TP≠Nil et puis TP↑.Csuiv ≠Nil)
- la réponse est donc :

```
   { précondition : TP≠Nil et puis TP↑.Csuiv ≠Nil }
   Écrire (premier(((TP↑.Csuiv)↑.T)↑.M) )
```

Q2. Construction de la partition

Donner la réalisation d'un algorithme qui pour E *donné construit la partition* P *associée. Pour créer les cellules des listes représentant* P*, on utilisera la primitive* Allouer*. La liste représentant* E *n'est pas modifiée.*

(i) Principe général de réalisation

Pour chaque mot de E (*parcours*), recopier le mot dans une nouvelle cellule de mot (*allocation*) ; s'il existe dans P (*recherche*) une classe correspondant à l'initiale du mot, *ajouter* le mot *en tête* de cette classe (aucun ordre imposé) ; sinon créer une nouvelle cellule de classe (*allocation*) avec ce mot, par *ajout en tête* de la liste de classes (aucun ordre imposé).

(ii) Schémas de modification des liens

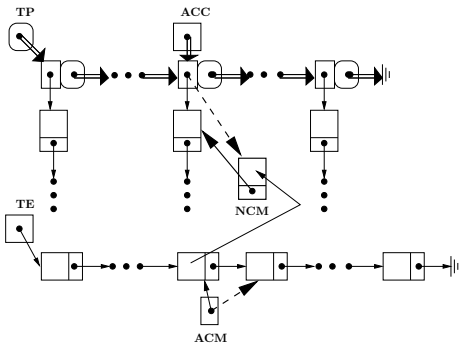


Figure 4.10 – Cas n°1 : ajout dans une classe existante

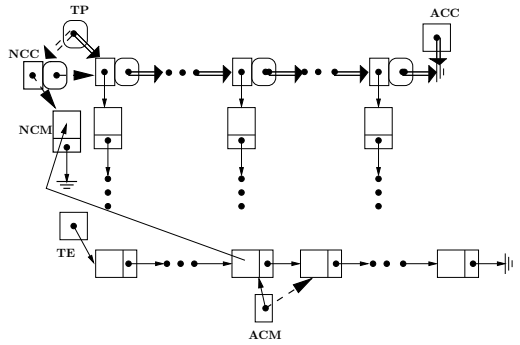


Figure 4.11 – Cas n°2 : création d'une nouvelle classe

(iii) Algorithme

```
ACM : adCelM                                  { pour le parcours de E. }
ACC : adCelC                                  { pour la recherche dans P. }
NCM : adCelM                                  { pour la création de cellules de mots. }
NCC : adCelC                                  { pour la création de cellules de classes. }

TP ⟵ Nil                                      { créer la partition vide }
ACM ⟵ TE
tant que ACM ≠ Nil
   Allouer(NCM) ; NCM↑ ⟵ <ACM↑.M, Nil>        { créer une cellule de mot }
   { recherche de la classe de l'élément courant }
   ACC ⟵ TP
   tant que ACC ≠ Nil et puis premier(ACM↑.M) ≠ premier((ACC↑.T)↑.M)
      ACC ⟵ ACC↑.Csuiv
   si ACC ≠ Nil alors { insertion en tête de classe }
      NCM↑.Msuiv ⟵ ACC↑.T ; ACC↑.T ⟵ NCM
   sinon { création d'une nouvelle classe }
      Allouer(NCC) ; NCC↑ ⟵ <NCM, TP> ; TP ⟵ NCC
   ACM ⟵ ACM↑.Msuiv
```

c) Gestion du chaînage dans un tableau mémoire

Les liens de chaînage sont représentés par des indices dans un tableau mémoire (cf 4.A.d)). Pour des listes d'entiers, on utilise le lexique suivant :

```
TmemEnt : constante de type entier > 0        { taille de la mémoire }
adCelEnt : type entier sur [0...TmemEnt]      { adresses de cellules }
Nil : constante 0 de type adCelEnt            { lien vers "nulle part" }
MemEnt : tableau sur [1...TmemEnt] de CelEnt
CelEnt : type <E : entier, Suc : adCelEnt>    { élément et lien vers son successeur }
```

Exemple E4.8 : Valeur du dernier élément (E4.1 suite)

L'algorithme donné dans l'exemple **E4.1** *est modifié comme suit :*

```
T : adCelEnt                                          { adresse de tête de la liste donnée }
AC : adCelEnt                                         { adresse de l'élément courant }
si T = Nil alors Écrire("liste vide")
sinon       AC ⟵ T
            tant que MemEnt_AC.Suc ≠ Nil              { le dernier existe }
                 AC ⟵ MemEnt_AC.Suc
            Écrire (MemEnt_AC.E)
```

Exemple E4.9 : Construction d'une liste chaînée (E4.2 suite)

La version 1 de l'algorithme donnée dans l'exemple **E4.2** *est modifiée comme suit :*

```
Q : adCelEnt                                          { adresse courante de queue }
AC : adCelEnt                                         { adresse de cellule pour l'élément courant }
si n = 0 alors T ⟵ Nil                                { construction de la liste vide }
sinon { création d'une liste singleton avec adresse de queue }
   Allouer (T) ; MemEnt_T.E ⟵ 1 ; Q ⟵ T
   pour i allant de 2 à n
      Allouer (AC) ; MemEnt_AC.E ⟵ i                  { création nouveau doublet }
      MemEnt_Q.Suc ⟵ AC ; Q ⟵ AC                      { connexion en queue }
   MemEnt_Q.Suc ⟵ Nil
```

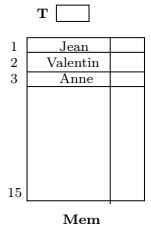
C. Exercices

a) Compréhension de la représentation chaînée

E4.10 : Compréhension de la représentation chaînée

(i) Une liste de prénoms

Pour concrétiser la notion de chaînage, on représente la mémoire par un tableau : les adresses sont des indices dans ce tableau, et un lien de chaînage est représenté par une telle adresse.
On étudie une liste chaînée de prénoms en ordre alphabétique. La mémoire est un tableau de 15 doublets <prénom, adresse>, nommé **Mem**. L'adresse de tête se trouve dans la variable **T**.

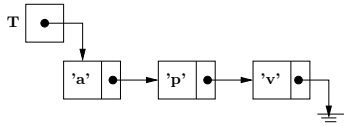


- Au départ, la liste est vide et on y ajoute successivement les trois prénoms suivants : *Jean, Valentin, Anne*. A l'issue de ces opérations, on observe l'état de la mémoire. Complétez le dessin ci-dessus pour que le chaînage décrive l'ordre alphabétique entre les prénoms.
- On ajoute le prénom *Laure* en position 4. Modifiez en conséquence le dessin précédent pour que le chaînage décrive l'ordre alphabétique. Combien de liens ont-ils été modifiés ?
- De même, observer les modifications de l'état de la mémoire au fur et à mesure de l'ajout des prénoms *Amélie, Victor, Agnès, Léa*, en positions 5 à 8.

- Quel est le nombre de modifications de liens engendré par l'insertion d'un prénom ?
- Observer à nouveau les modifications de l'état de la mémoire au fur et à mesure de la suppression des prénoms *Agnès, Victor, Laure*. Quel est le nombre de modifications de liens engendrés par la suppression d'un prénom ?

(ii) Une liste chaînée de trois caractères

On donne la séquence de caractères ['a', 'p', 'v'] sous forme d'une liste chaînée d'adresse de tête **T**, comme l'illustre la figure suivante.



- Illustrer sur le dessin ci-dessus le principe d'insertion du caractère 'd' dans la liste, à la bonne place pour conserver l'ordre alphabétique : faire apparaître les liens qui doivent être modifiés par des flèches en pointillé.
- Donner le lexique correspondant à cette représentation (voir 4.A.c)(ii)).
- Pour chaque composant de chaque cellule de la liste, donner l'expression permettant d'accéder à sa valeur.
- Donner la suite d'instructions réalisant l'insertion du caractère 'd'.

b) Gestion du chaînage à l'aide de pointeurs

Dans les exercices qui suivent, le chaînage est représenté à l'aide de pointeurs (cf 4.A.c)).

E4.11 : Itérations simples sur les listes chaînées

On donne une séquence **S** d'entiers sous forme d'une liste chaînée.

- Donner le lexique correspondant.
- Pour chacun des algorithmes suivants, spécifier une primitive (action ou fonction) et en donner une réalisation en appliquant les schémas du cours :
  - Longueur de la séquence.
  - Rang du premier élément de valeur **X** donnée, s'il existe (0 sinon). Le rang du premier élément d'une liste est 1.
  - Nombre d'éléments situés après le premier élément de valeur **X** donnée.
  - Prédécesseur d'un élément de valeur **X** donnée : le résultat est composé d'un booléen et d'une adresse. Le booléen a la valeur **vrai** si et seulement si la valeur **X** appartient à **S** ; dans ce cas, l'adresse renvoyée est l'adresse du prédécesseur de la cellule contenant le premier exemplaire de **X** dans **S**. Si le premier élément de **S** vaut **X**, l'adresse renvoyée est **Nil**.
  - I^{ème} élément de la séquence, pour **I** donné.

E4.12 : Construction d'une liste chaînée (E4.2 suite)

Étant donnée une séquence marquée d'éléments de type quelconque, spécifier et réaliser une action qui construit une liste chaînée avec ces éléments. Aucun ordre n'est imposé sur les éléments de la liste créée.

E4.13 : Partition d'une séquence
Étant donnés deux entiers **X** et **Y** (0 < X < Y) et une séquence d'entiers strictement positifs **S**, on étudie un algorithme qui construit trois listes chaînées **L1**, **L2** et **L3** : **L1** comporte tous les éléments de **S** strictement inférieurs à **X**, **L2** tous ceux qui appartiennent à l'intervalle **[X...Y[** et **L3** tous ceux qui sont supérieurs ou égaux à **Y**. La séquence **S**, de longueur **n** (donnée), est saisie interactivement. On suppose que l'espace mémoire comporte au moins **n** cellules libres. Les listes **L1**, **L2** et **L3** sont de même type : elles sont gérées à l'aide d'un seul type de cellule.

```
adCelEnt : type pointeur de CelEnt                    { adresses de cellules }
CelEnt : type <E : entier, Suc : adCelEnt>            { élément et lien vers son successeur }
n : constante de type entier ≥ 0                      { taille de la séquence. }
```

Dans ce qui suit, deux versions d'algorithme sont étudiées. On s'inspirera de l'exemple **E4.2** pour les réaliser. Dans la version 1 aucun ordre n'est imposé sur les éléments des listes créées. Dans la version 2, les éléments des listes créées y sont dans le même ordre que dans la séquence donnée.

(i) Version 1 : aucun ordre n'est imposé sur les éléments des listes créées.
On spécifie l'action **Répartir** :

```
Répartir : action(donnée n : entier > 0, X, Y : entier ;
                  résultat T1, T2, T3 : adCelEnt)
{ Répartit n entiers lus au clavier dans les trois listes d'adresse de tête T1, T2 et T3, la première
comportant les éléments strictement inférieurs à X, la seconde ceux appartenant à l'intervalle [X...Y[ et
la dernière ceux supérieurs ou égaux à Y. Pré-condition : X ≤ Y. }
```

Dans un parcours de la séquence donnée, l'entier courant est placé dans la liste correspondant à sa valeur. Aucun ordre n'étant imposé, il est ajouté en tête de liste. On introduit ainsi une action **AjouterEnTête**.

```
AjouterEnTête : action
   (donnée V : entier ; donnée-résultat T : adCelEnt)
   { e.i. : T = t0 et t0 est l'adresse de tête d'une liste (Nil si la liste est vide) ; e.f. : une cellule a été
   allouée et placée dans la liste avant la cellule d'adresse t0 ; elle contient l'élément V ; la valeur de T
   est l'adresse de cette cellule. }
```

On propose l'ébauche suivante pour la réalisation de l'action **Répartir** :

```
Répartir (n, X, Y, Tavant, Tdans, Taprès) :
   EC : entier                                        { pour la saisie : entier courant }
   ●●●●●●●●●●●●●●●●●●●                                { Création de trois listes vides }
   répéter n fois
      Lire(EC)
      selon EC
         EC < X  : ●●●●
         EC ≥ Y : ●●●●
         sinon : ●●●●
```

Q1
- Compléter la réalisation ci-dessus en utilisant l'action **AjouterEnTête**.
- On considère les données suivantes : **n=6, S = [10, 40, 20, 20, 50, 70], X = 30, Y = 60**. Dessiner les états successifs des listes **L1**, **L2** et **L3** (sous forme de schémas faits de cellules et de flèches) observés au début du corps de l'itération, lors de l'exécution de l'action.

- Donner une réalisation de l'action **AjouterEnTête**. Faire un dessin mettant en évidence les modifications de chaînage.

(ii) Version 2 : dans chaque liste créée, les éléments sont dans l'ordre de la séquence donnée.
Pour respecter la contrainte d'ordre, l'ajout d'un élément dans une liste doit être effectué en queue. Ceci nécessite de maintenir l'adresse de queue de chacune des listes créées. Les adresses de tête et de queue sont gérées dans deux tableaux de trois éléments.
On spécifie une nouvelle action **Répartir** :

```
TabadCelEnt : type tableau sur [1...3] d'adCelEnt
Répartir : action(donnée n : entier > 0, X, Y : entier ;
                  résultat T, Q : TabadCelEnt)
{ Répartit n entiers lus au clavier dans les trois listes d'adresse de tête Ti (1≤i≤3) et d'adresse de queue
Qi (1≤i≤3) : la première comporte les éléments strictement inférieurs à X, la seconde ceux appartenant
à l'intervalle [X...Y[ et la dernière ceux supérieurs ou égaux à Y. Si une liste est vide, son adresse de
tête vaut Nil et son adresse de queue n'est pas pertinente. Pré-condition : X ≤ Y. }
```

L'ajout d'un élément dans une liste vide doit être traité de manière spécifique. On introduit ainsi deux actions **AjouterEnQueue** et **CréerListeSingleton** et on complète le lexique de la version 1 de la manière suivante :

```
CréerListeSingleton : action (donnée X : entier ; résultat T, Q : adCelEnt)
   { e.i. : indifférent ; e.f. : T et Q sont les adresses de tête et de queue d'une liste singleton formée
   avec l'élément X. Le champ successeur de cette cellule vaut Nil. }

AjouterEnQueue : action (donnée X : entier ; donnée-résultat Q : adCelEnt)
   { e.i. : Q = q0 et q0 est l'adresse de queue d'une liste non vide ; e.f. : une cellule contenant X a été
   ajoutée après q0. Q contient l'adresse de cette cellule. Le champ successeur de cette cellule vaut Nil.
   }
```

Q2
- Illustrer par des dessins les modifications de l'état dans le cas où l'élément courant appartient à **[X...Y]**.
- Réaliser l'action **Répartir** en utilisant les actions **CréerListeSingleton** et **AjouterEnQueue**.
- Donner une réalisation des actions **CréerListeSingleton** et **AjouterEnQueue**. Faire des dessins mettant en évidence les modifications de chaînage.

(iii) Généralisation
On veut maintenant répartir la séquence donnée en **k+1** listes chaînées selon le critère suivant : **k** entiers distincts sont donnés en ordre croissant dans un tableau nommé **Tab** (généralisation des données **X** et **Y**). La première liste contient tous les éléments strictement inférieurs à $\mathbf{Tab}_1$, la $\mathbf{k+1^{ème}}$ liste contient tous ceux supérieurs ou égaux à $\mathbf{Tab}_k$ et pour **i** appartenant à **[2...k]**, la $\mathbf{i^{ème}}$ liste contient tous les éléments appartenant à l'intervalle $\mathbf{[Tab}_{i-1}\mathbf{...Tab}_i\mathbf{[}$.
Q3 Réaliser l'action de partition dans l'hypothèse où il n'y a pas de contrainte d'ordre sur les éléments des listes créées (généralisation de la version 1).

E4.14 : Suppression du premier élément de valeur donnée (utilisation d'éléments fictifs)
On donne une séquence d'entiers sous forme d'une liste chaînée. On étudie la suppression du premier élément de valeur donnée (dans le sens du chaînage), selon deux représentations de la liste. On utilise le lexique suivant :

Élément : type
adCel : type pointeur de Cel ; Cel : type <Elt : Entier, Suc : adCel>

(i) Liste chaînée sous forme standard
On spécifie une action de suppression :

SupprimerP : action (donnée X : Entier ; donnée-résultat T : adCel)
{ Supprime le premier élément de valeur X dans la liste chaînée de tête T. S'il n'en existe pas, la liste est inchangée }

— Dans cette spécification, le deuxième paramètre est défini comme étant de statut "donnée-résultat". Expliquer pourquoi il ne peut pas être défini comme étant de statut "donnée".
— Le principe choisi pour la réalisation est d'appliquer un schéma de recherche. On maintient le prédécesseur de l'élément courant pour réaliser la déconnexion de la cellule à supprimer.
Compléter l'ébauche suivante de cette réalisation :

```
SupprimerP(X, T) :
   AC, AP : adCel                          { adresse courante et adresse du prédécesseur }
   si T ≠ Nil alors
      si T↑.Elt = X alors
         ●●●●●●●●●●●●                                  { suppression en tête }
      sinon
         AP ⟵ T ; AC ⟵ AP↑.Suc
         tant que AC ≠ Nil et puis AC↑.Elt ≠ X
            ●●●●●●●●●
         si ●●● alors
            ●●●●●●●●●                    { suppression de la cellule d'adresse AC }
```

(ii) Liste chaînée munie d'un élément fictif de tête
On suppose maintenant que la liste traitée est sous forme *chaînée avec élément fictif de tête* comme indiqué au paragraphe 4.A.b)(v).
— On conserve la spécification de l'action **SupprimerP**. La seule différence est que le deuxième paramètre est défini avec le statut "donnée" (et non "donnée-résultat"). Expliquer cette différence.
— Simplifier la réalisation de l'action **SupprimerP** en tenant compte de l'hypothèse d'une représentation avec élément fictif de tête.

E4.15 : Concaténation de deux séquences
On étudie la construction d'une liste chaînée **L3** de tête **T3**, concaténation de deux listes chaînées **L1** et **L2** données, selon deux hypothèses : création d'une nouvelle liste ou modification des liens des listes existantes. On utilise le lexique suivant :

Élément : type
adCel : type pointeur de Cel ; Cel : type <Elt : Élément, Suc : adCel>

(i) Création d'une nouvelle liste
On spécifie l'action suivante :

CréerConcat : action (donnée T1, T2 : adCel, résultat T3 : adCel)
{ Construit une liste chaînée d'adresse de tête T3 représentant la concaténation des deux séquences représentées par les listes données de tête T1 et T2 }

La liste résultat est construite en créant les copies des listes **L1** et **L2** et en connectant le dernier élément de la première au premier de la seconde. Pour réaliser ces copies, on spécifie une action nommée **CopierListeAprès** :

CopierListeAprès : action (donnée T : adCel, donnée-résultat Q : adCel)
{ à l'état initial, Q est l'adresse de queue d'une liste non vide (Q ≠ Nil) ; à l'état final, une copie de la liste d'adresse de tête T a été construite et connectée après la cellule d'adresse Q. Q est l'adresse du dernier élément de la liste ainsi créée (en particulier si T = Nil, Q est inchangé). }

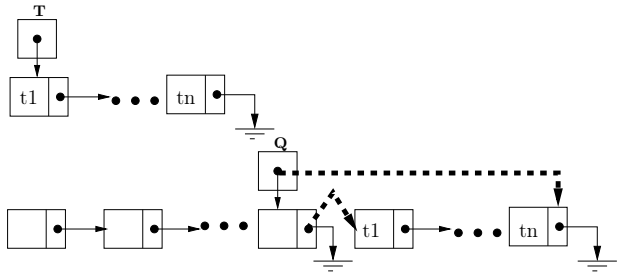


Figure 4.12 – Effet de l'appel à CopierListeApres(T,Q)

— Réaliser l'action **CréerConcat** en utilisant l'action **CopierListeAprès**.
Indication : initialiser la liste résultat par un singleton "fictif" de manière à copier **L1** après cet élément fictif. L'élément fictif sera supprimé après avoir copié **L2**.
— Réaliser l'action **CopierListeAprès**.

(ii) Modification des liens existants
L3 est formée des cellules constituant initialement **L1** et **L2**. A l'état final, **L1** et **L2** sont vides. On spécifie l'action suivante :

Concaténer : action (donnée-résultat T1, T2 : adCel, résultat T3 : adCel)
{ e.i. T1 et T2 sont les têtes de deux listes chaînées représentant deux séquences S1 et S2.
e.f. : T1=Nil et T2=Nil et T3 est l'adresse de tête d'une liste chaînée formée des cellules des deux listes données et représentant la concaténation de S1 et S2 }

— Donner un dessin illustrant les modifications de chaînage nécessitées par la réalisation de l'action.
— Donner une réalisation de l'action **Concaténer**.

E4.16 : Intersection de deux ensembles
On considère des ensembles représentés par des listes chaînées, **sans répétition d'éléments, et sans ordre imposé**, selon le lexique suivant :

Élément : type
adCel : type pointeur de Cel ; Cel : type <Elt : Élément, Suc : adCel>

(i) Créer une liste contenant l'intersection
On spécifie l'action suivante :

CréerIntersection : action (donnée T1, T2 : adCel, résultat T3 : adCel)
{ Construction d'une liste chaînée d'adresse de tête T3 représentant l'intersection des deux ensembles représentées par les deux listes données par leurs adresses de tête T1 et T2 }

— Donner une réalisation de l'action **CréerIntersection** fondée sur le principe suivant : dans un parcours de la première liste, on recopie tout élément appartenant aussi à la deuxième liste.

(ii) Séparer l'intersection et la différence
On rappelle qu'étant donné deux ensembles **E1** et **E2**, **E1** = **(E1** $\cap$ **E2)** $\cup$ **(E1\E2)**.
On spécifie l'action suivante :

SéparerIntersection : action (donnée-résultat T1 : adCel, donnée T2 : adCel,
résultat T3 : adCel)
{ Sà l'état initial, les listes de tête T1 et T2 représentent deux ensembles E1 et E2. A l'état final, la liste de tête T1 représente l'intersection de E1 et E2, la liste de tête T3 représente la différence entre E1 et E2 et la liste de tête T2 inchangée représente l'ensemble E2. }

— Donner une réalisation de l'action **SéparerIntersection**. Appliquer le principe suivant : pour chaque cellule de la liste T1, si son élément n'appartient pas à la liste T2 (il appartient alors à la différence), la déconnecter de la liste T1 puis la connecter à la liste T3.

E4.17 : Union de deux ensembles
Comme pour l'intersection (**E4.16**), étudier deux versions pour l'opération d'union : création d'une nouvelle liste ou modification d'une liste donnée.

E4.18 : Suppression des éléments redondants
Un élément est dit *redondant*, s'il existe dans la séquence un autre élément de même valeur.
Étudier la suppression des éléments redondants d'une liste chaînée, selon les diverses hypothèses suivantes, que l'on pourra combiner (4 versions possibles) :
— Le résultat est une nouvelle liste chaînée ou, au contraire, la liste initiale est modifiée de manière à ne plus comporter d'éléments redondants (les cellules non utilisées sont libérées).
— Dans la liste donnée les éléments sont dans un ordre quelconque, ou au contraire ils sont en ordre croissant (cas d'un type **Élément** muni d'une relation d'ordre).

c) Gestion du chaînage dans un tableau mémoire

E4.19 : Gestion du chaînage dans un tableau
Donner les modifications qu'il faut apporter aux lexiques et aux algorithmes des exemples **E4.3** "Une liste est-elle en ordre croissant ?" et **E4.4** "Inversion d'une liste chaînée", en utilisant un tableau pour gérer le chaînage (cf 4.A.d)).

D. Problèmes dirigés

E4.20 : Gestion d'un répertoire téléphonique
On désire gérer un répertoire téléphonique. Chaque élément du répertoire comporte l'*identification de la personne* (par son nom) et un *numéro de téléphone*. On suppose qu'un seul numéro est associé à une personne donnée, et qu'il n'y a pas deux personnes ayant le même nom.

(i) Représentation par une liste circulaire avec élément fictif
Le répertoire est représenté par une *liste chaînée, circulaire, et avec un élément fictif entre le dernier et le premier élément*. Cette liste est accessible par l'adresse de son élément fictif. Le premier élément de la séquence représentée se trouve dans la cellule d'adresse **F↑.Suc** où la variable **F** contient l'adresse de l'élément fictif de tête. Lorsque la séquence est vide, la liste ne comporte que l'élément fictif. La figure 4.13 illustre cette représentation.

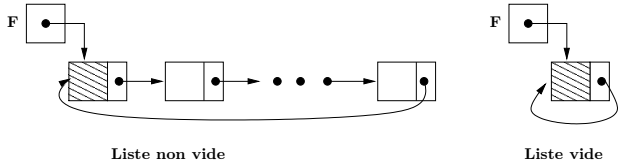


Figure 4.13 – Représentation chaînée avec fictif, circulaire

On fixe le lexique suivant :

Nom : type séquence de caractère *{ identifiant de personne }*
{ La comparaison de valeurs de type Nom est notée avec les symboles habituels sur les nombres ; l'affectation d'une variable de type Nom est notée avec le symbole habituel. }
Tél : type entier > 0 *{ numéro de téléphone }*
Élément : type <Pers : Nom ; Num : Tél>
adRep : type pointeur d'ElRep ; ElRep : type <Elt : Élément ; Suc : adRep>
{ La gestion mémoire se fait à l'aide des actions **Allouer** *et* **Libérer**. *On supposera que l'espace mémoire est suffisamment grand pour ne jamais être saturé. }*

Pour s'approprier la représentation, on étudie la fonction suivante :

LePred : fonction (N : Nom, F : adRep) $\longrightarrow$ adRep
{ Adresse du prédécesseur du doublet de nom N dans la liste F, s'il existe. Sinon, c'est l'adresse du prédécesseur du fictif : et donc le dernier élément du répertoire s'il n'est pas vide ou le fictif lui-même si le répertoire est vide. }

Cette fonction sera utilisée par la suite pour réaliser les primitives de gestion du répertoire.

Q1. Donner une réalisation de la fonction **LePred**.

(ii) Primitives de gestion du répertoire
L'utilisateur du répertoire doit pouvoir **installer** le répertoire en l'initialisant par un ensemble vide, **ajouter** un nouvel élément au répertoire, **modifier** le numéro associé à une personne, **supprimer** un élément du répertoire, **consulter** le répertoire pour obtenir le numéro de téléphone d'une personne et **vider** le répertoire en le ré-initialisant par un ensemble vide.
"Auto-organisation" de la liste :
À chaque fois que l'utilisateur consulte le répertoire, l'élément de répertoire associé à la personne considérée est placé *en tête de la liste*, de telle sorte que les éléments les plus fréquemment consultés se trouvent automatiquement au début de la liste.
Lorsqu'une nouvelle personne est ajoutée au répertoire, l'élément correspondant est placé *en fin de liste*. Lorsqu'un élément est modifié, il n'est pas déplacé dans la liste.

Cohérence des requêtes
Lors de la réalisation des primitives, on doit vérifier la cohérence des données : lors d'un ajout, vérifier que la personne correspondante n'est pas déjà dans le répertoire ; lors d'une modification ou d'une suppression vérifier que la personne correspondante est dans le répertoire.
Pour spécifier les cas d'erreurs, on introduit le type énuméré suivant :

Cond : type [Normal, NomErroné] *{ pour valider les données }*

Consultation du répertoire
Pour permettre la consultation du répertoire, on spécifie l'action suivante :

Consulter : action (donnée N : Nom, F : adRep ; résultat T : Tél, C : Cond)
{ Consulte le répertoire accessible par F pour connaître le numéro de téléphone associé au nom N. Si le nom N n'est pas présent dans le répertoire, à l'état final C a la valeur NomErroné et le répertoire est inchangé. Sinon, C a la valeur Normal, T est le numéro de téléphone associé à N et l'élément de nom N a été placé en tête du répertoire. }

Q2. Donner une réalisation de l'action **Consulter**.

Autres primitives de gestion

Installer : action (résultat F : adRep)
{ Construit un répertoire vide et fournit l'adresse du fictif de tête dans F }

Vider : action (donnée F : adRep)
{ Libère toutes les cellules du répertoire donné sauf celle comportant l'élément fictif de tête. A l'état final, la liste accessible par F représente un répertoire vide. }

Ajouter : action (donnée N : Nom, T : Tél, F : adRep ; résultat C : Cond)
{ Ajoute au répertoire accessible par F, un nouvel élément formé du nom N auquel est associé le numéro T. À l'état final, si le nom N est déjà présent dans le répertoire, C a la valeur NomErroné et le répertoire est inchangé ; sinon, C a la valeur Normal, et le nouvel élément a été ajouté en queue du répertoire. }

Supprimer : action (donnée N : Nom, F : adRep ; résultat C : Cond)
{ Supprime l'élément de nom N dans le répertoire accessible par F. À l'état final, si le nom N n'est pas présent dans le répertoire, C a la valeur NomErroné et le répertoire est inchangé ; sinon, C a la valeur Normal, et l'élément de nom N a été supprimé du répertoire. }

Modifier : action (donnée N : Nom, T : Tél, F : adRep ; résultat C : Cond)
{ Modifie dans le répertoire accessible par F, l'élément de nom N en lui associant le numéro T. À l'état final, si le nom N n'est pas présent dans le répertoire, C a la valeur NomErroné et le répertoire est inchangé ; sinon, C a la valeur Normal, et le numéro de téléphone T a été associé au nom N dans le répertoire. }

Q3. Donner une réalisation de chacune des primitives ci-dessus.

E4.21 : A propos de vecteurs creux
On considère des vecteurs de **n** entiers définis sur l'intervalle **[1...n]** (n > 0). On dit qu'un vecteur est *creux* lorsqu'il comporte un très grand nombre d'éléments nuls. Lorsque **n** est très grand il est intéressant de représenter le vecteur non pas comme un tableau de **n** entiers, mais sous une *forme compacte* à l'aide d'une séquence ne comportant que les valeurs *non nulles* du vecteur : chaque élément de la séquence comporte une valeur non nulle et son indice dans le vecteur considéré. De plus, la séquence est en ordre croissant des indices.

Par exemple, le vecteur défini sur [1...10] de valeur [0, 25, 0, 0, 13, 0, 0, 0, 0, -12] a pour forme compacte la séquence comportant les trois couples correspondant aux trois valeurs non nulles 25, 13 et -12 d'indices 2, 5 et 10 : [<25, 2>, <13,5>, <-12, 10>].
La forme compacte d'un vecteur dont tous les éléments sont nuls est la séquence vide.

(i) Représentation chaînée de la forme compacte d'un vecteur creux
On représente la forme compacte d'un vecteur creux par une *liste doublement chaînée, circulaire, avec un élément fictif* : chaque cellule est chaînée à son successeur et à son prédécesseur ; le prédécesseur de l'élément fictif est le dernier élément de la forme compacte (ou l'élément fictif si la forme compacte est la séquence vide) ; le successeur de l'élément fictif est le premier élément de la forme compacte (ou l'élément fictif si la forme compacte est la séquence vide). La liste est accessible par l'adresse de son élément fictif. Dans ce qui suit, cette adresse est appelée l'adresse de la forme compacte. Chaque cellule de cette liste comporte ainsi une valeur **non nulle**, son indice dans le vecteur et deux liens. La liste est en ordre croissant des indices. Le dessin 4.14 schématise cette représentation dans le cas de l'exemple précédent :

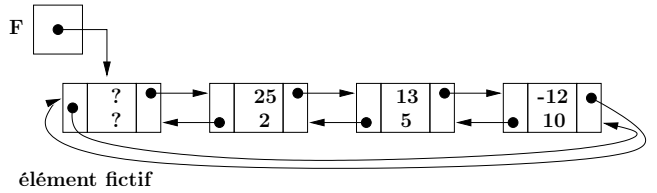


Figure 4.14 – Représentation doublement chaînée avec fictif, circulaire

(ii) Questions
On étudie ici la réalisation de quelques opérations sur les vecteurs. On exploitera au mieux la représentation donnée ci-dessus (ordre entre les éléments, double chaînage, présence d'un fictif, possibilité de sentinelle, etc.).
On utilisera le lexique suivant :

n : constante de type entier > 0 ; Indice : type entier sur [1...n]
adFC : type pointeur de FC
FC : type <LaValeur : entier non nul, Lindice : Indice, LeSuc, LePred : adFC>

Q1. Primitives de traitement d'une forme compacte
Réaliser les primitives suivantes :

Ladresse : fonction (F : adFC, i : Indice) ⟶ adFC
{ Adresse dans la forme compacte d'adresse F de la première cellule d'indice supérieur ou égal à i, ou F s'il n'en existe pas. }
CréerFCVide : action (résultat F : adFC)
{ Crée une forme compacte vide et fournit son adresse dans F. }
ConnecterAvant : action (donnée C, A : adFC)
{ A étant l'adresse d'une cellule appartenant à une forme compacte, et C l'adresse d'une cellule, modifie les liens de la forme compacte de manière à insérer C avant A. }

Déconnecter : action (donnée A : adFC)
{ A étant l'adresse d'une cellule appartenant à une forme compacte, modifie les liens dans cette forme compacte de manière à déconnecter A. }
AjouterAvant : action (donnée E : entier non nul, i : Indice, A : adFC)
{ A étant l'adresse d'une cellule appartenant à une forme compacte, crée un nouvel élément de valeur E et d'indice i et le connecte avant A. }
Supprimer : action (donnée A : adFC)
{ A étant l'adresse d'une cellule appartenant à une forme compacte, déconnecte A et le restitue à la mémoire. Pré-condition : A n'est pas l'élément fictif. }

Q2. Valeur d'un élément d'indice donné
Réaliser la fonction suivante :

Val : fonction (F : adFC, i : Indice) $\longrightarrow$ entier
{ Valeur de l'élément d'indice i du vecteur dont la forme compacte est donnée par l'adresse F }

Par exemple, soit n = 10, et soit **FV** l'adresse de la forme compacte [<25, 2>, <13, 5>, <-12,10>]. On a : Val(FV,1) = 0, Val(FV,2) = 25, Val(FV,3) = 0, ...,Val(FV,5) = 13, ..., Val(FV,10) = -12

Q3. Changement de représentation
On étudie les actions permettant de passer d'un vecteur à une forme compacte et réciproquement.

TabVersFC : action (donnée T : tableau sur [1...n] d'entier, résultat F : adFC)
{ Construit la forme compacte d'un vecteur donné en extension sous forme d'un tableau }
FCVersTab : action (donnée F : adFC, résultat T : tableau sur [1...n] d'entier)
{ Construit le vecteur correspondant à la forme compacte donnée }

Par exemple, si **T** est un tableau sur [1...10] de valeur [0, 25, 0, 0, 13, 0, 0, 0, 0, -12], **TabVersFC(T, F)** crée la forme compacte correspondante et fournit son adresse dans **F** ; **FCVersTab(F, T)** fait l'opération inverse.
— Réaliser les deux actions **TabVersFC** et **FCVersTab**.

Q4. Changement de la valeur d'un élément d'indice donné
Réaliser l'action suivante :

ChangerVal : action (donnée F : adFC, i : Indice, E : entier)
{ Modifie la forme compacte d'adresse F.
état initial : posons pour $i \in [1...n]$, Val(F,i) = α_i (quelconque, éventuellement 0)
état final : $\forall\ j \in [1...n], j \neq i \implies$ Val(F,j) = α_j
Val(F,i) = E (E peut être nul) }

Remarque : une cellule qui n'est plus utilisée doit être restituée à la mémoire libre.

Q5. Somme de deux vecteurs
La somme de deux vecteurs **V1** et **V2** est un vecteur **VS** défini de la manière suivante :

$$\forall\ i \in [1...n],\ VS_i = V1_i + V2_i$$

— Réaliser l'action suivante :
CréerSomme : action (donnée F1, F2 : adFC, résultat F3 : adFC)
{ Construit la forme compacte d'adresse FS du vecteur somme des vecteurs dont les formes compactes sont données par F1 et F2 }

— Réaliser l'action suivante :
Accumuler : action (donnée F1, F2 : adFC)
{ état initial : F1 et F2 sont les adresses des formes compactes de deux vecteurs V1 et V2.
état final : $\bar{F}1$ est l'adresse de la forme compacte du vecteur somme de V1 et V2. La forme compacte d'adresse F2 a été détruite. }

Remarque : une cellule qui n'est plus utilisée doit être restituée à la mémoire libre. On devra exploiter au mieux la représentation pour traiter les parcours et recherches sur les listes.

5. Définitions récursives

A. Eléments de cours

a) Notion de définition récursive

La définition d'une entité est *récursive* lorsqu'elle fait référence à une entité de même nature. Par exemple, dans le dessin d'une boîte de *Vache qui rit*, il y a deux boîtes de *Vache qui rit*.

Pour définir un ensemble de manière constructive récursive, on procède de la manière suivante :
- on définit (en extension, en compréhension) un ensemble d'*objets de base* ;
- on définit un ensemble de *constructeurs* (règles de construction) permettant de construire un objet de l'ensemble à l'aide d'autres objets de l'ensemble.

L'ensemble contient (par définition récursive) les objets de base et tout objet obtenu à l'aide des constructeurs. Et l'ensemble ne contient que ces objets.

Pour un ensemble donné, il peut y avoir plusieurs manières de le définir récursivement.
Par exemple, l'ensemble **N** des entiers naturels peuvent être défini récursivement comme suit.
version 1 :
- base : $\mathbf{0} \in \mathbf{N}$
- constructeur : si $\mathbf{x} \in \mathbf{N}$ alors $\mathbf{P1(x)} \in \mathbf{N}$ (où $\mathbf{P1(x)}$ a la même signification que $\mathbf{x+1}$)

version 2 :
- base : $\mathbf{0} \in \mathbf{N}$
- constructeurs :
  - si $\mathbf{x} \in \mathbf{N}$ alors $\mathbf{D0(x)} \in \mathbf{N}$ (où $\mathbf{D0(x)}$ a la même signification que $\mathbf{2x}$)
  - si $\mathbf{x} \in \mathbf{N}$ alors $\mathbf{D1(x)} \in \mathbf{N}$ (où $\mathbf{D1(x)}$ a la même signification que $\mathbf{2x+1}$)

b) Le constructeur séquence

Constructeurs, testeur et sélecteurs

- **[]** désigne la séquence vide. Le testeur associé est le prédicat **EstVide ?**.
- **S•e** désigne la séquence construite en ajoutant l'élément **e** à droite de la séquence **S**. Les sélecteurs associés sont **début** et **dernier**.
- **e∘S** désigne la séquence construite en ajoutant l'élément **e** à gauche de la séquence **S**. Les sélecteurs associés sont **premier** et **fin**.
- Une séquence peut être décrite en extension, en énumérant ses éléments séparés par des virgules, le tout étant entouré de crochets : $\mathbf{[e_1, e_2, \ldots, e_n]}$. En particulier, **[e]** dénote un singleton, séquence formée du seul élément e : $\mathbf{[e] = []\bullet e = e\circ[]}$.

Elément : type ; Seq : type séquence d'Elément

EstVide ? : (X : Seq) ⟶ booléen	*{ EstVide ?(X) est vrai ⟺ : (X=[]) }*
début : (X : Seq non vide) ⟶ Seq	*{ début(S•e) = S }*
dernier : (X : Seq non vide) ⟶ Elément	*{ dernier(S•e) = e }*
premier : (X : Seq non vide) ⟶ Elément	*{ premier(e∘S) = e }*
fin : (X : Seq non vide) ⟶ Seq	*{ fin(e∘S) = S }*
EstSingleton ? : (X : Seq) ⟶ booléen	*{ vrai ⟺ non EstVide ?(X) et puis EstVide ?(début(X)) ⟺ non EstVide ?(X) et puis EstVide ?(fin(X)). }*

Exemples de définitions de types

texte : type séquence de caractère
relevé : type séquence de réel
date : type <j, m, a : entier> ; événement : type <d : date, e : texte>
histoire : type séquence d'événement
lettre : type caractère sur ['a'...'z'] ; mot : type séquence de lettre
phrase : type séquence de mot ; paragraphe : type séquence de phrase

Exemples d'expressions de constantes

un texte : " ceci est 1 exemple de texte. Il comporte des lettres, des chiffres, des espaces, et des séparateurs. "
un relevé : [12.5, 13.2, -2.3]
un mot : "exemple"
une phrase : ["ceci", "est", "une", "phrase"]
un paragraphe : [["voici", "la", "première", "phrase"], ["et", "voici", "la", "seconde"], ["et", "ainsi", "de", "suite"]]
une histoire : [< <20, 9, 1999>, "rentrée à 9 heures" >, < <1, 11, 1999>, "jour férié" >, < <31, 12, 1999>, "demain l'an 2000" >]

c) Modèles de découpage d'une séquence selon les extrémités

Découpage isolant ...	**Forme des équations de récurrence**	**Forme des réalisations récursives de f(X)**	**Variante : séquence non vide**
Le premier élément	f([]) = f(e∘S) =	si EstVide ?(X) alors ••• sinon soit e = premier(X), S = fin(X) dans •••	f([e]) = f(e ∘S) = *{ S non vide }*
Le dernier élément	f([]) = f(S•e) =	si EstVide ?(X) alors ••• sinon soit e = dernier(X), S = début(X) dans •••	f([e]) = f(S•e) = *{ S non vide }*
Le premier et le dernier élément	f([]) = f([e]) = f(e1∘S•e2) =	si EstVide ?(X) alors ••• sinon si EstVide ?(fin(X)) alors ••• sinon soit e1 = premier(X), e2 = dernier(X), S = début(fin(X)) dans •••	f([e]) = f([e1,e2]) = f(e1∘S•e2) = *{ S non vide }*
Les deux premiers éléments	f([]) = f([e]) = f(e1∘e2∘S) =	si EstVide(X) alors ••• sinon si EstVide ?(fin(X)) alors ••• sinon soit e1 = premier(X) e2 = premier(fin(X)) S = fin(fin(X)) dans •••	f([e]) = f([e1, e2]) = f(e1∘e2∘S) = *{ S non vide }*
Les deux derniers éléments	f([]) = f([e]) = f(S•e1•e2) =	si EstVide ?(X) alors ••• sinon si EstVide(début(X)) alors ••• sinon soit e2 = dernier(X) e1 = dernier(début(X)) S = début(début(X)) dans •••	f([e]) = f([e1, e2]) = f(S•e1•e2) = *{ S non vide }*

d) Modèles de découpage d'une séquence selon une propriété

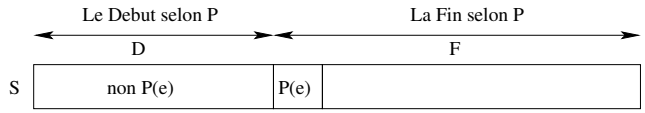


Figure 5.1 – découpage d'une séquence selon une propriété

Découper une séquence donnée **S** selon une propriété **P** (Figure 5.1) consiste à identifier le premier élément de la séquence (de gauche à droite) vérifiant **P**, nous parlons de l'*élément de séparation* (selon **P**) : **S** est découpée selon deux sous-séquences **D** et **F** (S=D&F), respectivement appelées *le début de* **S** et *la fin de* **S** *selon* **P**. Par convention, l'élément de séparation, s'il existe est le premier élément de la séquence de fin **F**. Si aucun élément de la séquence donnée ne vérifie **P**, F est vide. Si le premier élément de **S** vérifie **P**, **D** est vide. Si **S** est vide, **D** et **F** le sont aussi.
Pour modéliser un tel découpage, le lexique suivant est utilisé (aux noms près) :

Elément : type
P : fonction (E : Elément) ⟶ booléen *{ propriété caractérisant le découpage }*
LeDébutSelonP : fonction (X : séquence d'Elément, ●●●) ⟶ séquence d'Elément
{ séquence de début de la séquence donnée jusqu'au premier élément (exclu) vérifiant P }
(1) LeDébutSelonP([], ●●●) = []
(2) LeDébutSelonP(e∘S, ●●●) = si P(e) alors [] sinon e ∘LeDébutSelonP(S, ●●●)

LaFinSelonP : fonction (X : séquence d'Elément, ●●●) ⟶ séquence d'Elément
{ séquence de fin de la séquence donnée à partir du premier élément (inclus) vérifiant P }
(1) LaFinSelonP([], ●●●) = []
(2) LaFinSelonP(e∘S, ●●●) = si P(e) alors e∘S sinon LaFinSelonP(S, ●●●)

DebEtFinSelonP : fonction (X : séquence d'Elément, ●●●) ⟶ séquence d'Elément, séquence d'Elément
{ DebEtFinSelonP(S, ●●●) = <LeDébutSelonP(S, ●●●), LaFinSelonP(S, ●●●)> }
(1) DebEtFinSelonP([], ●●●) = <[], []>
(2) DebEtFinSelonP(e∘S, ●●●) =
si P(e) alors <[], e ∘S>
sinon soit <D, F> = DebEtFinSelonP(S, ●●●) dans < e∘D, F >

B. Exemples d'exercices rédigés

a) Relations de récurrence, réalisation récursive

Exemple E5.1 : Factorielle

Décrire une fonction qui à un entier strictement positif **n** *associe le produit des* **n** *premiers entiers (noté habituellement* **n !***).*

fac : fonction (m : entier > 0) ⟶ entier > 0 *{ fac(m) = m ! }*

(i) Relations de récurrence

(1) fac(1) = 1
(2) fac(n+1) = fac(n) * (n+1) *{ n > 0 }*

(ii) Réalisation récursive

fac(n) :
retour : si n=1 alors 1 sinon fac(n-1) * n

Liste des appels récursifs engendrés
Pour visualiser l'évaluation d'une application de la fonction, on dessine une liste des appels engendrés comme l'illustre la Figure 5.2.

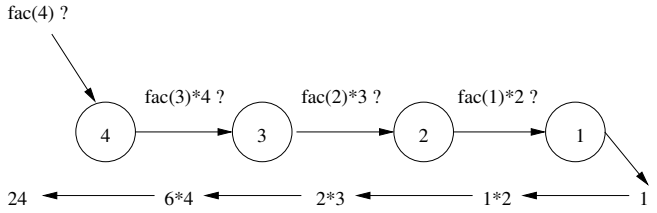


Figure 5.2 – liste des appels récursifs engendrés par l'appel fac(4)

Exemple E5.2 : Nombre de 'a'

Décrire une fonction qui dénombre les 'a' d'un texte.

texte : type séquence de caractère
nba : fonction (X : texte) ⟶ entier ≥ 0 *{ nombre de 'a' du texte donné }*

(i) Relations de récurrence

(1) nba([]) = 0 (2) nba(c ∘t) = (si c='a' alors 1 sinon 0) + nba(t)

(ii) Réalisation récursive

nba(X) :
retour : si EstVide ?(X) alors 0
sinon (si premier(X)='a' alors 1 sinon 0) + nba(fin(X))

(iii) Variante sur l'exemple

On reprend l'exemple en excluant le cas du texte vide dans la spécification.

nba : fonction (X : texte non vide) ⟶ entier ≥ 0 *{ nba(X) : nombre de 'a' de X }*
(1') nba([c]) = (si c = 'a' alors 1 sinon 0)
(2') nba(c∘t) = (si c = 'a' alors 1 sinon 0) + nba(t) **{ t non vide }**

L'équation (2') diffère de l'équation (2) précédente uniquement par le commentaire qui précise que le nom local **t** est un texte non vide. Ceci peut aussi être exprimé de la manière suivante :

(2") nba(c1∘c2∘t) = (si c1='a' alors 1 sinon 0) + nba(c2∘t)

Dans cette équation, **c1** et **c2** désignent des caractères et **t** un texte éventuellement vide. L'appel récursif **nba(c2∘t)** porte sur un texte non vide, comme exigé par le profil de la fonction.
La réalisation récursive peut s'écrire :

nba(X) :
    retour : (si premier(X)='a' alors 1 sinon 0) + (si EstVide ?(fin(X)) alors 0 sinon nba(fin(X)))

b) Découpage d'une séquence selon les extrémités

Exemple E5.3 : Inverse d'une séquence

Décrire une fonction nommée inv qui à une séquence associe son inverse (image miroir). Par exemple, inv("abcd")="dcba". On convient que l'inverse d'une séquence vide est la séquence vide.

Elément : type
inv : (X : séquence d'Elément) ⟶ séquence d'Elément

(i) Découpage à droite

(1) inv([]) = [] (2) inv(S•e) = e∘inv(S)

(ii) Découpage à gauche

(1) inv([]) = [] (2) inv(e∘S) = inv(S)•e

(iii) Découpage aux deux extrémités

(1) inv([]) = [] (2) inv([e]) = [e]
(3) inv(e1∘S•e2) = e2 ∘inv(S) •e1

Exemple E5.4 : Egalité de deux mots

Décrire une fonction à valeur booléenne déterminant si deux mots sont égaux.

lettre : type caractère sur ['a'...'z']
mot : type séquence de lettre
EgMot : fonction (M1, M2 : mot) ⟶ booléen *{ vrai ⟺ les mots sont égaux }*
(1) EgMot([], []) = vrai
(2) EgMot([], pc ∘MF) = faux
(3) EgMot(pc ∘MF, []) = faux
(4) EgMot(pc1∘MF1, pc2∘MF2) = (pc1=pc2) et puis EgMot(MF1, MF2)

Exemple E5.5 : Une séquence d'entiers est-elle en ordre croissant ?

Décrire une fonction à valeur booléenne déterminant si une séquence non vide d'entiers est en ordre croissant.

EstCrois : fonction (X : séquence non vide d'entiers) ⟶ booléen
    { vrai ⟺ X est en ordre croissant. EstCrois([e]) = vrai }
(1) EstCrois([e]) = vrai
(2) EstCrois(e1∘e2∘S) = (e1 ≤ e2) et puis EstCrois(e2 ∘S)

Exemple E5.6 : N^{ème} élément d'une séquence

Décrire une fonction d'extraction du Nème élément d'une séquence.

Nième : (X : séquence non vide d'Elément, m : entier > 0) ⟶ booléen, Elément
    { Posons <T, E> = Nième(X, m) : s'il existe un élément de rang m dans X (m ≤ longueur(X)), T a la valeur vrai et E est l'élément de rang m dans X. Sinon, T a la valeur faux et la valeur de E n'est pas spécifiée (parce que non significative). }
(1) Nième([e], 1) = <vrai, e>
(2) Nième(e∘S, 1) = <vrai, e> *{ S non vide }*
(3) Nième([e], n+1) = <faux, e>
(4) Nième(eoS, n+1) = Nième(S, n)

Remarque : les deux équations (1) et (2) peuvent être regroupées en une seule :

Nième(e∘S, 1) = <vrai, e> *{ S peut être vide }*

c) Regroupement de fonctions intermédiaires

Exemple E5.7 : Nombre d'exemplaires de la valeur maximum d'une séquence d'entiers

Décrire une fonction qui détermine le nombre d'exemplaires de la valeur maximum d'une séquence d'entiers.

Nbvmax : fonction (X : séquence d'entier non vide) ⟶ entier > 0

(i) Version 1 : En première analyse, on introduit deux fonctions, **Max**, valeur maximum d'une séquence et **NbEx**, nombre d'exemplaires d'un élément dans une séquence, ce qui permet d'écrire :

Nbvmax(X) :
    retour : NbEx(Max(X), X)

La fonction **NbEx** est une généralisation de l'exemple **E5.2** ; l'étude de la fonction **Max** est traitée dans l'exercice **E5.11**.

(ii) Regroupement de fonctions intermédiaires
On observe que le processus d'évaluation de la fonction comporte successivement deux phases, dont la taille dépend de celle de la séquence considérée : l'une pour l'évaluation de **Max** et l'autre pour l'évaluation de **NbEx**. On peut construire une solution dans laquelle les valeurs des fonctions intermédiaires peuvent être calculées simultanément : on définit une fonction à valeur couple nommée **MaxEtNb**, chaque élément du couple correspondant à la valeur d'une des fonctions intermédiaires.

MaxEtNb : fonction (X : séquence non vide d'entiers) ⟶ entier, entier >0
    { MaxEtNb(X) = <Max(X), NbEx(Max(X), X)> }

On a alors :

Nbvmax(X) :
    retour : soit <m, n> = MaxEtNb(X) dans n

et la fonction **MaxEtNb** peut être définie par les équations de récurrence suivantes :

(1) MaxEtNb([e]) = <e, 1>
(2) MaxEtNb(S•e) =
        soit <m, n> = MaxEtNb(S)
        dans selon m, e
                m<e : <e, 1>
                m=e : <e, 1+n>
                m>e : <m, n>

d) Découpage d'une séquence selon une propriété

On considère des textes formés de lettres et de séparateurs (espaces, signes de ponctuation, ...). Dans un texte, les mots sont des suites de lettres délimitées par des séparateurs ou par les extrémités du texte. Deux mots sont séparés par un ou plusieurs séparateurs. Le texte peut débuter par une lettre ou par un ou plusieurs séparateurs. De même il peut se terminer par une lettre ou par un ou plusieurs séparateurs.

lettre, séparateur : type — *{ non précisé ici }*
texte : type séquence de caractère — *{ restreint aux lettres et aux séparateurs }*
EstLettre : fonction (c : caractère) ⟶ booléen — *{ vrai ⟺ le caractère est une lettre }*
EstSéparateur : fonction (c : caractère) ⟶ booléen — *{ vrai ⟺ le caractère est un séparateur }*
mot : type séquence de lettre

Exemple E5.8 : Premier mot d'un texte

Décrire une fonction qui isole le premier mot d'un texte.

LePremierMot : fonction (X : texte) ⟶ mot

Pour isoler le premier mot du texte, on compose deux fonctions de découpage définies respectivement selon les propriétés "être une lettre" et "être un séparateur" (cf paragraphe 5.A.d)) :

SansSépAuDébut : fonction (X : texte) ⟶ texte *{ la fin du texte selon la propriété "être une lettre" }*
LesLettresDuDébut : fonction (X : texte) ⟶ mot *{ le début selon la propriété "être un séparateur" }*

On en déduit :

```
LePremierMot(X) :
    retour : LesLettresDuDébut(SansSépAuDébut(X))
```

Les définitions récurrentes des deux fonctions introduites ci-dessus sont déduites des formes générales des fonctions de découpage en particularisant les propriétés caractérisant les découpages :

(1) SansSépAuDébut([]) = []
(2) SansSépAuDébut(c∘T) = si EstLettre(c) alors c∘T sinon SansSépAuDébut(T)
(3) LesLettresDuDébut([]) = []
(4) LesLettresDuDébut(c∘T) = si EstSéparateur(c) alors [] sinon c∘LesLettresDuDébut(T)

Exemple E5.9 : Les mots d'un texte

Décrire une fonction qui construit la séquence des mots d'un texte.

LesMots : (X : texte) ⟶ séquence de mot

L'idée est d'isoler le premier mot du texte puis d'appliquer récursivement la fonction **LesMots** à ce qui suit le premier mot dans le texte donné.
On réutilise la fonction **SansSépAuDébut** de l'exemple **E5.8** et on introduit une fonction qui découpe un texte en deux parties, ses lettres du début et ce qui reste :

```
LettresDébutEtReste : fonction (X : texte) ⟶ mot, texte
    { Posons <LD, R> = LettresDébutEtReste(X) : LD est le début du texte selon la propriété "être un
    séparateur" et R est la fin du texte selon la même propriété. }

LesMots(X) :
    retour : soit <LD, R> = LettresDébutEtReste(SansSépAuDébut(X))
          dans si EstVide?(LD) alors[] sinon LD∘LesMots(R)
```


(1) LettresDébutEtReste([]) = <[], []>
(2) LettresDébutEtReste(c∘T) =
    si EstSéparateur(c) alors <[], c∘T>
    sinon soit <LD, R> = LettresDébutEtReste(T) dans < c∘LD, R>

e) Fonctions dont l'évaluation engendre plus d'un appel récursif

Exemple E5.10 : Marelle

On considère un chemin pavé. Les pavés y sont numérotés à partir de 0, séquentiellement dans le sens du chemin en commençant par le premier pavé. Pour se déplacer sur le chemin, on avance d'un pavé à l'autre selon l'une des deux règles suivantes : (1) passer du pavé courant au suivant ou (2) sauter le prochain pavé.
On étudie ici deux questions : dénombrer les chemins partant du pavé 0 et arrivant au pavé $\boldsymbol{n}$ *; énumérer tous ces chemins. Dans les deux cas, on s'appuie l'étude suivante de l'ensemble des chemins : pour atteindre le pavé* $\boldsymbol{n+2}$*, on ne peut que provenir du pavé* $\boldsymbol{n+1}$ *en appliquant la règle (1), ou du pavé* $\boldsymbol{n}$ *en appliquant la règle (2). On divise ainsi l'ensemble des chemins arrivant au pavé* $\boldsymbol{n+2}$ *en deux sous-ensembles disjoints : les chemins passant par le pavé* $\boldsymbol{n+1}$ *et les chemins ne passant pas par le pavé* $\boldsymbol{n+1}$*. Tout chemin passant par le pavé* $\boldsymbol{n+1}$ *est construit à partir d'un chemin arrivant au pavé* $\boldsymbol{n+1}$ *et du déplacement élémentaire* $\boldsymbol{n+1} \longrightarrow \boldsymbol{n+2}$*. Tout chemin ne passant pas par le pavé* $\boldsymbol{n+1}$ *est construit à partir d'un chemin arrivant en* $\boldsymbol{n}$ *et du déplacement élémentaire* $\boldsymbol{n} \longrightarrow \boldsymbol{n+2}$*.*

(i) Nombre de chemins

NbC : fonction (m : entier > 0) ⟶ entier > 0
{ Nombre de chemins conduisant au pavé m }

Une première solution
Le principe d'analyse présenté ci-dessus permet d'affirmer que pour n>0, $\mathbf{NbC(n+2) = NbC(n+1) + NbC(n)}$. Par ailleurs, il n'y a qu'un chemin pour atteindre le pavé 1 et il y a deux chemins pour atteindre le pavé 2. La définition récurrente de la fonction **NbC** est ainsi :

(1) NbC(1) = 1 (2) NbC(2) = 2 (3) NbC(n+2) = NbC(n+1) + NbC(n)

et sa réalisation récursive est :

```
NbC(m) :
    retour : si m ≤ 2 alors m sinon Nbc(m-1) + NbC(m-2)
```

Arbre des appels récursifs engendrés
Pour visualiser l'évaluation d'un appel de la fonction, par exemple **NbC(6)**, on dessine maintenant un arbre des appels de **NbC** (Figure 5.3) : chaque noeud est étiqueté par une valeur d'argument à laquelle **NbC** est appliquée. La racine de l'arbre correspond à l'appel externe initial. Les autres noeuds correspondent aux appels récursifs engendrés par l'appel initial.

Une transformation
Le schéma ci-dessus met en évidence la redondance de calcul impliquée par cette forme de la fonction. Pour l'éviter, on introduit une fonction intermédiaire, soit **g**, calculant simultanément deux valeurs successives de la fonction **NbC**. La fonction **g** est définie en termes de **NbC** par :

g : fonction (x : entier > 0) ⟶ entier > 0, entier > 0 — *{ g(x) = < NbC(x), NbC(x+1) > }*

Ceci permet de réaliser **NbC** :

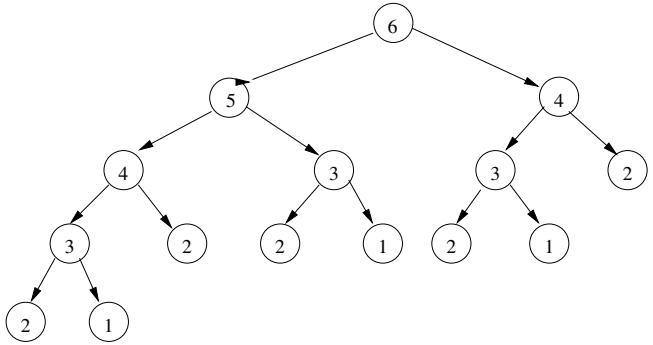


Figure 5.3 – Arbre des appels de NbC(6)

NbC(x) :
retour : soit <a, b>=g(x) dans a

La définition récurrente de **g** découle de celle de **NbC** :

(1) g(1) = <NbC(1), NbC(2)> = <1, 2>
(2) g(n+1) = soit <x, y>= g(n) dans <y, x + y> *{ n>0 }*

En effet :

g(n+1) = <NbC(n+1), NbC(n+2)>
= <NbC(n+1), NbC(n)+NbC(n+1)>
= soit x= NbC(n), y=NbC(n+1) dans <y, x+y>
= soit <x, y>= g(n) dans <y, x + y>

(ii) Enumération des chemins possibles.

Un chemin est caractérisé par une suite de numéros de case sur la marelle. L'ensemble des chemins est représenté par une séquence de chemins.
Les chemins conduisant au pavé **n+2** (pour **n** >0) sont obtenus en ajoutant le pavé **n+2** d'une part à tous les chemins conduisant au pavé **n** et d'autre part à tous les chemins conduisant au pavé **n+1**. Pour exprimer ce principe, on introduit une fonction **PlusD** qui ajoute un numéro de case à droite de tous les chemins d'un ensemble donné.

Chemin : type séquence non vide d'entier > 0
LesCh : fonction (m : entier > 0) ⟶ séquence de Chemin
{ Ensemble des chemins conduisant au pavé m }
PlusD : fonction (SC : séquence non vide de Chemin, n : entier > 0) ⟶ séquence non vide de Chemin
{ Ajoute n à droite de tous les chemins de SC }

(1) PlusD([Ch], n) = [Ch•n]
(2) PlusD(Ch∘SC, n) =(Ch•n)∘PlusD(SC, n) *{ SC non vide }*
(3) LesCh(1) = [[1]]
(4) LesCh(2) = [[1,2], [2]]
(5) LesCh(n+2) = PlusD(LesCh(n+1)&LesCh(n),n+2)

C. Exercices

a) Analyse récurrente, réalisation récursive d'une fonction

E5.11 : Maximum de n valeurs

Soit les fonctions **Max2V** et **MaxNv** spécifiées comme suit :

Max2v : fonction (n1, n2 : entier) ⟶ entier *{ maximum des deux entiers }*
MaxNv : fonction (X : séquence non vide d'entier) ⟶ entier *{ maximum des entiers donnés }*

(i) Recherche de propriétés

Compléter les relations suivantes en utilisant **Max2v** et en respectant les indications :

(1) MaxNv([e]) = •••
(2) MaxNv(e∘S) = Max2v(•••) *{ S non vide. En se ramenant au maximum de S }*
(3) MaxNv(S•e) = Max2v(•••) *{ S non vide. En se ramenant au maximum de S }*
(4) MaxNv(e1∘e2∘S) = MaxNv(•••) *{ En utilisant le maximum de e1 et e2 }*

(ii) Définition récurrente, réalisation récursive, observation de l'évaluation

En combinant l'équation 1 avec chacune des équations 2 à 4, on obtient trois formes de définition récurrente de la fonction **MaxNv**.

- A chacune de ces formes, correspond une réalisation récursive de la fonction : donner ces trois réalisations en les nommant respectivement **MaxNv12**, **MaxNv13**, **MaxNv14** (les suffixes correspondant aux numéros d'équations composant la définition récurrente).
- Choisir une séquence de 5 entiers nommée X : en raisonnant sur la liste des appels récursifs, comparer le processus d'évaluation du maximum de X selon la version utilisée.

E5.12 : Une séquence d'entiers est-elle de longueur impaire ?

On spécifie la fonction suivante :

LgImp : fonction (X : séquence d'entier) ⟶ booléen
{ vrai ⟺ X est de longueur impaire }

(i) Solutions directes

Compléter les équations de récurrence suivantes puis en déduire plusieurs définitions récurrentes de la fonction LgImp :

(1) LgImp([]) = ••• (2) LgImp([e]) = •••
(3) LgImp(e∘S) = ••• (4) LgImp(e1∘e2∘S) = •••

(ii) Fonctions mutuellement récursives

Une séquence est de longueur impaire si et seulement si la séquence obtenue en enlevant le premier élément est de longueur paire. Pour appliquer ce principe on introduit la fonction LgP :

LgP : fonction (X : séquence d'entier) ⟶ booléen
{ vrai ⟺ X est de longueur paire }

En appliquant le principe ci-dessus, donner des équations de récurrence définissant chacune des deux fonctions LgImp et LgP.

E5.13 : Appartenance

Soit la fonction ApE spécifiée comme suit :

texte : type séquence de caractère
ApE : fonction (X : texte) ⟶ booléen *{ vrai ⟺ X comporte au moins un 'E' }*

— Compléter la définition récurrente suivante de la fonction ApE, avec une expression conditionnelle puis en utilisant des opérateurs logiques :

(1) ApE ([]) = faux (2) ApE (c∘T) = •••

— Donner la réalisation récursive correspondante, sans utiliser d'expression conditionnelle.
— Généraliser à un caractère quelconque, puis à une séquence de type quelconque.

E5.14 : Suppression des espaces
(i) Étudier une fonction SansEspace qui à un texte donné associe le texte qui en est issu en enlevant tous les espaces, les autres caractères restant dans le même ordre.

(ii) Étudier une fonction SansEspaceAuDébut qui à un texte donné associe le texte qui en est issu en enlevant les espaces du début (ceux situés avant le premier caractère autre qu'un espace).

(iii) Généraliser à la suppression d'un caractère quelconque.

E5.15 : Plus de 'A' que de 'E'
On étudie la fonction suivante :

PlusDeAE : fonction (t : texte) ⟶ booléen
{ vrai ⟺ le texte t comporte plus de 'A' que de 'E' (au sens strict). PlusDeAE([])=faux. }

(i) Étudier chacune des trois solutions suivantes : réaliser la fonction PlusDeAE en termes de la fonction supplémentaire suggérée, puis donner des équations de récurrence définissant cette fonction et la réalisation récursive correspondante.
— version 1 : avec une fonction nommée NbEx, déterminant le nombre d'exemplaires d'une lettre dans un texte.
— version 2 : avec une fonction nommée NbAE, spécifiée comme suit :
NbAE : fonction (t : texte) ⟶ entier ≥ 0, entier ≥ 0
{ posons <na, ne> = NbAE(t) : na est le nombre de 'A' et ne est le nombre de 'E' de t. }
— version 3 : avec une fonction nommée DifAE, spécifiée comme suit :
DifAE : fonction (t : texte) ⟶ entier *{ différence entre le nombre de 'A' et le nombre de 'E' du texte }*
(ii) Reprendre la version 3 précédente, en généralisant le problème à deux lettres quelconques :

PlusDe : fonction (c1, c2 : caractère, t : texte) ⟶ booléen
{ vrai ⟺ le texte t comporte plus d'exemplaires du caractère c1 que d'exemplaires du caractère c2. }

E5.16 : Palindromes
Une séquence est un *palindrome* si et seulement si elle est égale à son inverse. Étudier une fonction booléenne qui caractérise un texte palindrome, en supposant qu'il est formé de lettres et d'espaces : tout d'abord en tenant compte des espaces, puis sans tenir compte des espaces.

E5.17 : Valeur la plus proche
Décrire une fonction qui étant donné un entier X, détermine la valeur la plus proche de X dans une séquence d'entiers. Par exemple, dans la séquence [3, -1, 5, 5, 12], la valeur la plus proche de 6 est 5, la valeur la plus proche de 10 est 12, la valeur la plus proche de 3 est 3.

E5.18 : Valeurs cumulées
On étudie une fonction nommée ValCum construisant la séquence des valeurs cumulées d'une séquence d'entiers : le $i^{ème}$ élément de la séquence résultat est la somme des i premiers éléments de la séquence donnée. Par exemple, la séquence des valeurs cumulées de la séquence [1, 2, 3, 4, 5] est [1, 3, 6, 10, 15]. Traiter cette étude selon les indications suivantes :

(i) Étudier une première version selon le principe suivant : fonder l'analyse récurrente sur un découpage à droite et utiliser une fonction Somme (calcul de la somme d'une séquence d'entiers).

(ii) On considère une application de la fonction ValCum à une séquence S de longueur n : exprimer en fonction de n le nombre d'applications de l'opération binaire + engendrées par ValCum(S).

(iii) Étudier une deuxième version, toujours fondée sur un découpage à droite, mais n'utilisant pas la fonction Somme. Comme en **(ii)**, évaluer le nombre d'additions.

(iv) Étudier d'autres versions en exploitant les découpages suivants : premier élément et fin de la séquence ; deux premiers éléments et reste de la séquence.

E5.19 : Concaténation de deux séquences
On étudie ici la concaténation de deux séquences (opérateur dénoté par le symbole &) en cherchant à la décrire en termes des constructeurs de base sur les séquences ([], ∘ et •). On veut donner plusieurs solutions, en faisant varier le mode de découpage choisi des séquences données pour faire apparaître la récurrence.

Elément : type ; Concat : fonction (X1, X2 : séquence d'Elément) ⟶ séquence d'Elément

(i) Propriétés de la fonction Concat
On étudie les propriétés de la fonction, par exemple Concat(S1,e∘S2) = Concat(S1•e,S2). Compléter les équations suivantes en les examinant indépendamment les unes des autres :

(1) Concat([], []) = •••••
(2) Concat([], S) = •••••
(3) Concat(S, []) = •••••
(4) Concat(S1, e∘S2) = •••••
(5) Concat(S1•e, S2) = •••••
(6) Concat(e∘S1, S2) = •••••
(7) Concat(S1, S2•e) = •••••
(8) Concat(e1∘S1, S2•e2) = •••••

(ii) Equations de récurrence définissant la fonction Concat
Observer que si l'on combine les équations 3 et 4 on peut obtenir une première solution de définition récurrente de la fonction Concat.
— Expliquer pourquoi la combinaison des équations 3 et 5 fournit une solution incorrecte.
— Expliquer pourquoi la combinaison des équations 3 et 8 fournit une solution incorrecte.
— Donner diverses définitions récurrentes de la fonction en combinant les propriétés ci-dessus.

E5.20 : Interclassement de deux séquences triées
L'*interclassement* de deux séquences triées X1 et X2 est une séquence triée X3 comportant tous les éléments de X1 et de X2. Par exemple, si X1= [1, 1, 3, 11, 11, 25] et X2 = [7, 9, 9, 11, 14, 25, 38] alors X3 = [1, 1, 3, 7, 9, 9, 11, 11, 11, 14, 25, 25, 38].

InterClas : fonction (X1, X2 : séquence d'entier en ordre croissant) ⟶ séquence d'entier en ordre croissant

— Compléter l'équation suivante traduisant une propriété générale de la fonction InterClas dans le cas de deux séquences triées non vides :

InterClas(e1∘S1, e2∘S2) = selon e1, e2
e1 < e2 : ••••••
e1 = e2 : ••••••
e1 > e2 : ••••••

— En déduire une définition récurrente de la fonction InterClas.

E5.21 : A propos de préfixes
Une séquence P est un préfixe d'une séquence S si et seulement si il existe S1 telle que S=P&S1. La séquence vide est préfixe de **toute** séquence ; toute séquence est préfixe d'elle-même.
(i) Étudier une fonction booléenne nommée EstPréf portant sur deux séquences d'entiers et ayant la valeur vrai lorsque la première est un préfixe de la seconde.
(ii) Étudier une fonction LesPréfixes qui construit une séquence formée de tous les préfixes d'une séquence d'entiers. **Indication** : raisonner sur un **découpage à droite** de la séquence donnée.

E5.22 : Relation d'ordre alphabétique sur des mots.
On étudie une fonction booléenne nommée InfMot correspondant à la relation "inférieur (strict)" selon l'ordre alphabétique sur des mots. Si un mot M1 est le préfixe d'un mot M2, InfMot(M1, M2) a la valeur vrai. En s'appuyant sur l'analyse de l'exemple **E5.4** donner des équations de récurrence définissant InfMot. On supposera que l'on dispose d'opérations primitives permettant de comparer deux caractères selon l'ordre alphabétique, dénotées par les symboles utilisés pour les nombres.

E5.23 : Plus grand commun diviseur
On étudie une fonction pgcd déterminant le plus grand commun diviseur de deux entiers naturels. Cette fonction vérifie les relations suivantes :

(1) pgcd(a, a) = a
(2) pour a > b : pgcd(a, b) = pgcd(a - b, b)
(3) pour a < b : pgcd(a, b) = pgcd(a, b - a)

— Donner une réalisation récursive de la fonction en exploitant ces relations.
— Le pgcd de deux entiers positifs vérifie aussi la propriété suivante :
(4) pgcd(a, b) = pgcd(b, r) avec a = b*q + r et $0 \leq r < b$
Donner une autre réalisation de la fonction exploitant cette propriété.

E5.24 : Arithmétique sur les entiers naturels

— Donner des relations de récurrence définissant les opérations somme et produit. Pour chaque opération, donner deux versions en utilisant successivement les deux modèles d'analyse récurrente définis sur le type entier naturel.
— Pour chacune des deux versions de la fonction produit, dessiner la liste des appels récursifs engendrés par l'évaluation de l'expression produit(3, 17). Dénombrer ces appels et conclure en comparant les versions.
— Étudier d'autres opérations sur les entiers, par exemple l'élévation à la puissance.

E5.25 : Anagrammes
Deux mots sont des *anagrammes* l'un de l'autre, s'ils comportent exactement les mêmes lettres, éventuellement dans un ordre différent. Par exemple, SEL et LES sont des anagrammes, de

même que ANIS et SAIN, ARBRE et BARRE, ECRAN et NACRE, ARSENELUPIN et PAULSERNINE et LUISPERENNA...Par contre LASSE et SALLE ne sont pas des anagrammes. Tout mot est anagramme de lui-même.
On étudie une fonction à valeur booléenne, nommée EstAna, qui détermine s'il est vrai que deux mots donnés sont des anagrammes. Une solution est fondée sur le principe suivant : chaque lettre du premier mot doit appartenir au second et le second mot ne doit pas comporter d'autres lettres que celles du premier.

(i) Une première version
— Spécifier deux fonctions intermédiaires, l'une nommée Ap d'appartenance d'une lettre à un mot et l'autre nommée Moins de suppression d'une lettre dans un mot.
— Donner des équations de récurrence définissant la fonction EstAna en termes de ces fonctions, puis donner des équations définissant chacune d'entre elles.

(ii) Regroupement des fonctions intermédiaires
En s'inspirant de l'exemple **E5.7**, donner une deuxième version de la fonction EstAna, obtenue en regroupant les deux fonctions Ap et Moins en une seule nommée ApMoins.

E5.26 : Nombre de "le"
Étudier une fonction qui à un texte associe le nombre d'exemplaires du sous-texte "le" qu'il comporte. Par exemple,le texte "elle appelle le vieil employé de l'hôtel" comporte trois exemplaires de "le". Généraliser à n'importe quel couple de lettres.

E5.27 : A propos de mots dans un texte
On considère des textes formés de lettres et d'espaces. Dans un tel texte, un mot est une sous-séquence de lettres délimitée par des espaces ou par les extrémités du texte. Le texte peut commencer ou se terminer un ou plusieurs espaces. Mais il peut aussi commencer ou se terminer par une lettre. Deux mots sont séparés par un ou plusieurs espaces.
Étudier les fonctions suivantes : nombre de mots ; longueur moyenne des mots du texte ; nombre de mots terminés par la lettre 's' ; séquence des premières lettres des mots du texte.

b) Opérations génériques sur les séquences

Les exercices qui suivent traitent de fonctions dont la définition est liée à la structure de séquence et ne dépend pas du type des éléments des séquence considérée.
E5.28 : Découpage selon le rang d'un élément (E5.6 suite)
Étudier les fonctions suivantes :
— Extraction des X premiers éléments.
— Rang d'un élément donné dans une séquence
— Milieu : L désignant la longueur d'une séquence non vide, son milieu est défini comme étant l'élément de rang (L+1) quotient 2 (en numérotant les éléments de gauche à droite à partir de 1). Donner une solution "directe" (sans utiliser de fonction intermédiaire).

E5.29 : Opérations ensemblistes
On étudie l'opération d'appartenance d'un élément à un ensemble et les opérations d'intersection, d'union et de différence de deux ensembles.

(i) cas général
Les ensembles considérés (données et résultats) sont représentés par des séquences sans répétitions. Décrire les quatre opérations.
(ii) cas d'un type ordonné
Les ensembles considérés sont munis d'une relation d'ordre. Les opérations de comparaison associées à cet ordre sont notées par les symboles habituels utilisés pour les nombres.
Données et résultats sont représentés par des séquences (sans répétitions) en ordre croissant. Décrire les quatre opérations ensemblistes. Pour les opérations binaires, travailler par analogie avec l'exercice **E5.20**.

E5.30 : Sous-séquences d'une séquence

Étant donnée une séquence S, une *sous-séquence* de S est une séquence obtenue à partir de S en enlevant un nombre quelconque d'éléments de S et en conservant l'ordre initial dans S pour les éléments restants. Nous dirons qu'une sous-séquence de S est *connexe*, si tous ses éléments sont adjacents dans S. Par exemple, soit S= [23, 41, -5, 18, 23, 0] : [23, -5, 23] est une sous-séquence de S (non connexe) ; [41] et [18, 23] sont des sous-séquences de S (connexes) ; [-5, 41] n'est pas une sous-séquence de S (l'ordre initial n'est pas respecté). Toute séquence S est une sous-séquence d'elle-même. On peut considérer que la séquence vide est sous-séquence de n'importe quelle séquence.

— Étudier une fonction booléenne nommée EstSS, qui détermine si une séquence donnée X est sous-séquence d'une séquence donnée Y.
— Étudier une fonction booléenne nommée EstSSC, qui détermine si une séquence donnée X est sous-séquence connexe d'une séquence donnée Y.

E5.31 : Partition des éléments d'une séquence

Étudier une fonction nommée PartitionP qui à une séquence X associe une séquence comportant tous les éléments de X répartis selon un critère donné P, en regroupant au début les éléments ne vérifiant pas P et à la fin ceux vérifiant P.
On s'appuie sur le lexique suivant :

```
Elément : type
P : fonction (E : Elément) ⟶ booléen                    { critère de partition }
PartitionP : fonction (X : séquence d'Elément) ⟶ séquence d'Elément
```

c) Application des schémas de découpage selon une propriété

E5.32 : Plus petite valeur supérieure à un entier donné

Étant donnée une séquence d'entiers en ordre croissant, on veut déterminer la plus petite valeur de cette séquence strictement supérieure à un entier donné. Étudier une fonction correspondant à ce problème en l'analysant en termes de schémas de découpage selon une propriété.

E5.33 : Eléments d'un intervalle d'entiers

On considère une séquence d'entiers S en ordre croissant et deux entiers X et Y (X ≤ Y). Décrire une fonction qui construit la sous-séquence des éléments de S appartenant à l'intervalle [X...Y].

```
SéqEntCrois : type séquence d'entier en ordre croissant
EIInt : fonction (S : SéqEntCrois, X, Y : entier) ⟶ SéqEntCrois
    { Séquence des éléments de S appartenant à [X, Y]. X ≤ Y }
```

Étudier une solution basée sur la composition de fonctions de découpage selon une propriété.

d) Séquences de séquences

E5.34 : Plan de ville

On considère un plan quadrillé d'une ville (Figure 5.4). L'origine des axes figure l'entrée Sud-Ouest de la ville. Un piéton situé à l'origine cherche à atteindre un point P par le plus court chemin. Il se donne pour cela deux règles de déplacement : avancer d'un bloc au Nord ou d'un bloc à l'Est.

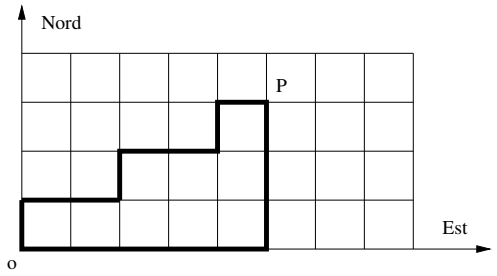


Figure 5.4 – Plan de ville

— Étudier une fonction nommée NbC, dénombrant les chemins permettant d'atteindre un point P en appliquant les règles ci-dessus : donner une solution fondée sur une analyse récurrente.
— Étudier une fonction nommée LesCh énumérant tous les chemins possibles.

Indications : décrire un chemin par une séquence de mouvements, un mouvement étant décrit par un type énuméré à deux valeurs notées Nord et Est. Introduire une fonction nommée PlusD qui étant donnés une séquence de chemins et un mouvement, ajoute ce mouvement à droite de tous les chemins de la séquence donnée. On peut s'inspirer de l'exemple **E5.10**.

E5.35 : Le compte est bon

On étudie une forme simplifiée du jeu "le compte est bon". Étant donné un entier X strictement positif et une séquence non vide E d'entiers strictement positifs, on cherche à exprimer X comme une somme d'éléments de E. Par exemple, pour X=20 et E=[18, 3, 1, 4, 20], on a une solution, X=20 ; pour X =25 et E =[21, 3, 2, 4, 2], on a deux solutions X = 21 + 4 et X = 21 + 2 + 2. Pour X et E donnés, il peut ne pas y avoir de solution. Pour X=23 et E = [21, 3, 2, 4, 2], on considère qu'il y a deux solutions, une pour chaque occurrence de 2 dans E.
Pour chacune des quatre fonctions suivantes, fournir une réalisation (n'utilisant pas les autres) :

1. ExisteSol détermine s'il existe une solution.
2. NbSol détermine le nombre de solutions.
3. UneSol produit une séquence d'éléments de E dont la somme est X. Le résultat est une séquence vide s'il n'y a pas de solution.
4. LesSol produit toutes les solutions sous forme d'une séquence. Chaque solution est elle-même une séquence d'entiers de E de somme X. Le résultat est la séquence vide s'il n'y a pas de solution.

D. Problèmes dirigés

E5.36 : Quelques principes de tri
On étudie une fonction PO (Permutation ordonnée) :

ElemOrd : type ordonné *{ muni des opérations de comparaison habituelles }*
PermOrd : fonction (S : séquence d'ElemOrd) ⟶ séquence triée d'ElemOrd

Étudier différentes réalisations de la fonction PermOrd selon les principes suivants :
- Tri par *insertion* : la fin de S est triée, puis son premier élément est inséré dans le résultat à la bonne place.
- Tri par *sélection du minimum* : M étant la valeur minimum de S, la séquence obtenue en enlevant M de S est triée, puis M est placé en tête du résultat.
- Tri par *interclassement* : S est découpée en deux parties de taille égale (à 1 près), chacune de ces deux parties est triée et enfin les séquences obtenues sont interclassées (voir **E5.20**).
- Tri par *partition* ("quick sort") : S est découpée en trois parties p, D et F : p est le premier élément de S ; D est une séquence constituée des autres éléments de S inférieurs ou égaux à p ; F est constituée des éléments de S strictement supérieurs à p. Par exemple, pour S= [7, 5, 25, 3, 7, 15, 4, 20], on peut considérer le découpage suivant : p=7, D= [5, 3, 7, 4], F= [25, 15, 20].
  La permutation ordonnée de S est alors obtenue en concaténant la permutation ordonnée de D, le singleton formé de l'élément p et la permutation ordonnée de F.

E5.37 : Polynômes
On considère des polynômes en x, par exemple -3x⁴ + 2x² + 10x -12. Un polynôme est une suite de monômes. Un monôme, par exemple -34x⁴, est caractérisé par un coefficient, -34, et la puissance de x, 4. Un polynôme est représenté par une séquence de monômes à coefficients non nuls en ordre décroissant des puissances ; un monôme est représenté par un couple <coefficient, puissance> :

monôme : type <c : entier ≠ 0, p : entier ≥ 0 >
polynôme : type séquence de monôme *{ en ordre décroissant des puissances de x }*

Par exemple, au polynôme -34x⁴ + 2x² + 10x -12 correspond la séquence [<-34, 4>, <2, 2>, <10, 1>, <-12, 0>]
Dans cette représentation, les monômes à coefficient nul ne sont pas représentés. C'est un invariant de la représentation, qui doit constamment être respecté. En particulier, le polynôme 0 est représenté par une séquence vide.
Q1. Étudier une fonction, nommée DérivP, qui étant donné un polynôme, lui associe le polynôme dérivée par rapport à x. Par exemple, la séquence représentant le polynôme dérivée du polynôme ci-dessus est [<-136, 3>, <4, 1>, <10, 0>].

Q2. Étudier une fonction, nommée SommeP, qui étant donnés deux polynômes, leur associe le polynôme somme. Par exemple, la somme des polynômes -34x⁴ + 2x² + 10x -12 et 3x⁵ + 4x⁴ - 2x² est 3x⁵ - 30x⁴ + 10x -12. On tiendra compte du fait que les séquences représentant les polynômes sont en ordre décroissant des puissances.

E5.38 : Gestion d'un ensemble d'événements historiques
On considère des événements historiques. La description d'un tel événement comporte :
- Un nom, séquence de caractères, qui caractérise l'événement : deux événements différents ont des noms différents.

- La date à laquelle l'événement a eu lieu, sous forme d'un triplet d'entiers correspondant au jour, au mois, et à l'année.

(i) Lexique
Dans ce qui suit, un ensemble d'événements historiques est représenté sous forme d'une séquence d'événements en ordre chronologique. Lorsque deux (ou plus) événements ont lieu à la même date, aucun ordre particulier n'est imposé entre eux dans la séquence. On fixe ainsi le lexique suivant :

jm : type entier sur [1...31] *{ quantième d'un jour dans le mois }*
ma : type entier sur [1...12] *{ quantième d'un mois dans l'année }*
an : type entier > 0 *{ quantième de l'année dans le calendrier }*
Date : type <jour : jm, mois : ma, année : an>
NomEv : type séquence non vide de caractère
EvHist : type <Nom : NomEv, Moment : Date>
EnsEvHist : type séquence d'EvHist
{ séquence d'événements historiques en ordre chronologique. En cas d'égalité de dates, aucun ordre particulier n'est spécifié }

(ii) Questions

Q1. Étudier une fonction qui étant donné un ensemble d'événements, permet d'obtenir la date à laquelle a eu lieu un événement de nom donné. Le résultat de la fonction est un couple formé d'une part d'un booléen qui indique si l'événement considéré fait bien partie de l'ensemble d'événements considéré et d'autre part de la date cherchée, le cas échéant.

Q2. Étudier une fonction d'ajout d'un nouvel événement à un ensemble d'événements. On sait a priori que l'ensemble considéré ne comporte pas déjà cet événement.

Q3. Étudier une fonction de suppression d'un événement d'un ensemble d'événements. On sait a priori que l'ensemble considéré comporte cet événement.

Q4. Étudier une fonction d'extraction de tous les noms d'événements ayant eu lieu une année donnée parmi un ensemble d'événements.

E5.39 : Somme d'une suite de nombres
On considère des textes constitués de chiffres et d'espaces. Un nombre est une suite non vide de chiffres décimaux ('0', '1', ..., '9') délimitée par des espaces ou par les extrémités du texte. Le texte peut commencer ou se terminer par un chiffre. Mais il peut aussi commencer ou se terminer par un ou plusieurs espaces. Deux nombres sont séparés par un ou plusieurs espaces. Chaque nombre est la représentation d'un entier en base 10 et le texte représente une suite d'entiers.
Étant donné un tel texte, on veut déterminer la somme des entiers qu'il représente. Par exemple, pour le texte **" 343 47 50 36 "**, l'entier résultat est **476**.

(i) Principe de résolution
On décide de procéder en plusieurs étapes :
- étape 1 : obtention d'une séquence de nombres SN à partir des caractères du texte source S ;
- étape 2 : obtention d'une séquence d'entiers SE à partir des nombres de SN ;
- étape 3 : obtention de la somme des entiers de SE.

Pour l'exemple précédent, les étapes sont les suivantes : **" 343 47 50 36 " ⟶ ["343", "47", "50", "36"] ⟶ [343, 47, 50, 36] ⟶ 476**

(ii) Fonctions intermédiaires
SommeNb étant le nom associé au problème posé, on associe une fonction à chacune des étapes : LesNombres (étape 1), LesEntiers (étape 2) et Somme (étape 3). Plus précisément, on définit le lexique suivant :

espace : caractère ' '
chiffre : type caractère sur ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
texte : type séquence de caractère *{ chiffres ou espaces }*
Nombre : type séquence non vide de chiffre
SéqNb : type séquence de Nombre
SéqEnt : type séquence d'entier ≥ 0
SommeNb : fonction (t : texte) $\longrightarrow$ entier ≥ 0 *{ somme des entiers représentés par le texte donné }*
LesNombres : fonction (t : texte) $\longrightarrow$ SéqNb *{ séquence des nombres du texte }*
{ exemple : LesNombres(" 343 47 50 36 " = ["343", "47", "50", "36"] }
LesEntiers : fonction (S : SéqNb) $\longrightarrow$ SéqEnt *{ séquence des entiers représentés par les nombres }*
{ exemple : LesEntiers(["343", "47", "50", "36"]) = [343, 47, 50, 36] }
Somme : fonction (S : SéqEnt) $\longrightarrow$ entier ≥ 0 *{ somme des entiers de la séquence, 0 si elle est vide }*

(iii) Questions

Q1. Donner une réalisation de la fonction SommeNb, en termes des fonctions du lexique ci-dessus.

Q2. Donner des équations de récurrence définissant la fonction Somme.

Q3. Pour réaliser la fonction LesEntiers, on définit les fonctions suivantes :

DversE : fonction (N : Nombre) $\longrightarrow$ entier ≥ 0 *{ entier dont la représentation en base 10 est donnée }*
{ exemple : DversE("343") = 343 }
CversE : fonction (c : chiffre) $\longrightarrow$ entier ≥ 0 *{ entier correspondant au chiffre donné }*
{ exemple : CversE('3') = 3 }

— Donner des équations de récurrence définissant la fonction LesEntiers, en utilisant DversE.
— Donner des équations de récurrence définissant la fonction DversE, en utilisant CversE.

Q4. Pour réaliser la fonction LesNombres on introduit deux fonctions de découpage selon une propriété (voir l'analogie avec l'exemple **E5.9**) :

SsEspD : fonction (T : texte) $\longrightarrow$ texte *{ le texte sans ses espaces éventuels du début }*
{ Fin de T selon la propriété "être un chiffre". }
{ Exemple : SsEspD(" 343 47 50 36 ") = "343 47 50 36 " }
CDetR : fonction (T : texte) $\longrightarrow$ texte, texte *{ les chiffres du début, s'il y en a, et le reste }*
{ Posons <CD, R> = CDetR(T) : CD est le début de T selon la propriété "être un espace" et R est la fin de T selon cette propriété. }
{ Exemple : CDetR("343 47 50 36 ") = <"343", " 47 50 36 "> }

— Donner une réalisation récursive de la fonction LesNombres en termes de SsEspD et CDetR.
— Donner des équations de récurrence définissant chacune des fonctions SsEspD et CDetR.

E5.40 : Tautogrammes
Une phrase est un *tautogramme* si tous ses mots commencent par la même lettre. Par exemple, la phrase *le lion lape le lait lentement* est un tautogramme. On étudie une fonction booléenne nommée EstTaut qui détermine si une phrase est un tautogramme.

Les phrases considérées sont des séquences formées de lettres et de séparateurs (espaces, signes de ponctuation, etc.). Un mot est une séquence de lettres délimitée par des séparateurs ou par les extrémités de la phrase. Deux mots sont séparés par un ou plusieurs séparateurs. La phrase peut débuter ou se terminer par zéro, un ou plusieurs séparateurs. On dispose de deux fonctions booléennes nommées EstLettre et EstSéparateur, permettant respectivement de savoir si un caractère est une lettre ou si un caractère est un séparateur.
Pour traiter les questions qui suivent, on introduira des séquences intermédiaires et les fonctions adéquates associées, de manière à décrire la solution par composition de ces fonctions.

Q1. Décrire la fonction EstTaut, en raisonnant sur la séquence des premières lettres de mot.

Q2. On assouplit maintenant la notion de tautogramme en ne considérant pas les mots appartenant à une liste LM donnée : une phrase est un tautogramme si tous ses mots n'appartenant pas à LM commencent par la même lettre. Décrire une fonction booléenne EstTaut1 qui indique si une phrase est un tautogramme dans ces nouvelles conditions, en raisonnant sur la séquence des mots.

Q3. On considère un texte composé de phrases séparées par des points ("."). Décrire une fonction qui donne le nombre de phrases du texte qui sont des tautogrammes, en raisonnant sur la séquence des phrases.

E5.41 : Monnaie
On considère une monnaie M constituée de pièces (ou billets) ayant des valeurs nominales distinctes. Étant donné une somme d'argent S, il existe plusieurs manières d'en faire la monnaie, c'est-à-dire d'en donner l'équivalent en pièces de la monnaie M. Par exemple si M comporte des pièces de 1 €, 0,5 €, 0,2 € et 0,1 €, il y a quatre manières de faire la monnaie de 0,5 € : 1 pièce de 0,5 € ; 2 pièces de 0,2 € et 1 pièce de 0,1 € ; 1 pièce de 0,2 € et 3 pièces de 0,1 € ; 5 pièces de de 0,1 €.
Pour une somme donnée S, il s'agit de calculer le nombre de manières d'en faire la monnaie dans une monnaie donnée M, en supposant qu'il n'y a aucune limite sur le nombre de pièces disponibles.

(i) Principe de l'analyse récurrente
On choisit l'une des valeurs de pièces de M, soit p, et l'on distingue toutes les solutions comportant au moins une pièce de p € et toutes les solutions n'en comportant pas. Ainsi on ramène le problème portant sur M et S, au même problème portant d'une part sur M et S-p et d'autre part sur M' et S, où M' est une monnaie comportant toutes les pièces de M sauf celles de valeur p.
On représente une monnaie par une séquence de pièces. Pour choisir une valeur de pièces d'une monnaie, on prend la première pièce de la séquence.

Valeur : type entier > 0
pièce : type Valeur
Monnaie : type séquence de pièce
NbM : fonction (M : Monnaie non vide, S : Valeur) $\longrightarrow$ entier > 0
{ Nombre de manières de faire la monnaie de S dans M }

(ii) Questions

Q1. Compléter la réalisation suivante de la fonction NbM :

```
NbM(M, S) :
    retour : soit p= premier(M), M' = fin(M)
             dans soit avec = ●●●●●●●,                    { nombre de solutions avec p }
                          sans = ●●●●●●●                  { nombre de solutions sans p }
                  dans avec + sans
```

Q2. Examiner le processus d'évaluation d'un appel de la fonction NbM en dessinant un arbre des appels récursifs engendrés (Cf. exemple **E5.10**).

UGA, UFR IM²AG, Algorithmique, langages et programmation Définitions récursives

6. Arbres : une introduction

A. Eléments de cours : arbres binaires

a) Notations

(i) Constructeurs, testeurs et sélecteurs

— /\ dénote l'arbre binaire vide. Le testeur associé est la fonction booléenne nommée EstVide ?.

— /G, r, D\ dénote l'arbre binaire non vide de racine r, de sous-arbre gauche G et de sous-arbre droit D. Les sélecteurs associés sont les fonctions nommées Gauche, Droit et Racine.

Elément : type ; ArbreBin : type arbre binaire Elément
EstVide ? : fonction (A : ArbreBin) ⟶ booléen — *{ vrai ⟺ A est l'arbre vide }*
Gauche : fonction (A : ArbreBin non vide) ⟶ ArbreBin — *{ Gauche (/G, r, D\) = G }*
Droit : fonction (A : ArbreBin non vide) ⟶ ArbreBin — *{ Droit (/G, r, D\) = D }*
Racine : fonction (A : ArbreBin non vide) ⟶ Elément — *{ Racine (/G, r, D\) = r }*

(ii) Notations abrégées pour décrire les arbres non vides

— //r\\ est une forme abrégée de / /\, r, /\ \, et dénote un arbre binaire *singleton*, de racine r. Le testeur associé est la fonction booléenne nommée EstSingleton ?.

— //G, r\\ est une forme abrégée de / G, r, /\ \ où G est non vide : la racine de cet arbre est un noeud *unaire gauche*. Le testeur associé est la fonction booléenne nommée EstUnaireGauche ?.

— //r, D\\ est une forme abrégée de / /\, r, D \ où D est non vide : la racine de cet arbre est un noeud *unaire droit*. Le testeur associé est la fonction booléenne nommée EstUnaireDroit ?.

— //G, r, D\\ est une forme abrégée de / G, r, D \ où G et D sont non vides : la racine de cet arbre est un noeud *binaire*. Le testeur associé est la fonction booléenne nommée EstBinaire ?.

Pour spécifier ces testeurs, on utilise les notations suivantes :

ExG ?(A) ⟺ non EstVide ?(Gauche(A)) ; ExD ?(A) ⟺ non EstVide ?(Droit(A))
EstSingleton ? : fonction (A : ArbreBin non vide) ⟶ booléen
{ vrai ⟺ non ExG ?(A) et non ExD ?(A) }
EstUnaireG ? : fonction (A : ArbreBin non vide) ⟶ booléen
{ vrai ⟺ ExG ?(A) et non ExD ?(A) }
EstUnaireD ? : fonction (A : ArbreBin non vide) ⟶ booléen
{ vrai ⟺ non ExG ?(A) et ExD ?(A) }
EstBinaire ? : fonction (A : ArbreBin non vide) ⟶ booléen
{ vrai ⟺ ExG ?(A) et ExD ?(A) }

b) Modèles d'analyse récurrente

	Forme des équations de récurrence de f	Forme des réalisations récursives de f(X)
Modèle 1 : avec l'arbre vide	f(/\) = f(/G, r, D\) =	si EstVide ?(X) alors ••• sinon soit /G, r, D\ = X dans •••
Modèle 2 : sans l'arbre vide	f(//r\\) = f(//G,r\\) = f(//r,D\\) = f(//G, r, D\\) =	soit /G, r, D\ = X dans selon G, D EstSingleton ?(X) : ••• EstUnaireG ?(X) : ••• EstUnaireD ?(X) : ••• EstBinaire ?(X) : •••

c) Linéarisation d'un arbre binaire : parcours d'arbres

Le parcours d'un arbre binaire peut s'effectuer de plusieurs façons :

Préordre : on parcourt la racine, puis le sous-arbre gauche, puis le sous-arbre droit

Ordre symétrique : on parcourt le sous-arbre gauche, puis la racine, puis le sous-arbre droit

Ordre terminal : on parcourt le sous-arbre gauche, puis le sous-arbre droit, puis la racine

Sur l'arbre de la figure 6.1, le parcours en préodre construit la liste A,B,D,C,E,G,F, le parcours en ordre symétrique construit la liste D,B,A,E,G,C,F et le parcours en ordre terminal construit la liste D,B,G,E,F,C,A.

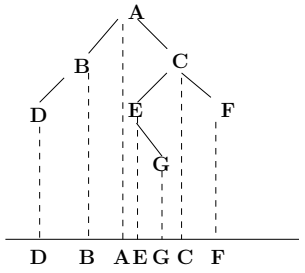


Figure 6.1 – parcours d'un arbre en ordre symétrique

Pour réaliser un tel parcours, on peut spécifier la fonction suivante :

SéqSym : fonction (A : arbre binaire Elément) ⟶ séquence d'Elément
{ Séquence des nœuds de l'arbre en ordre symétrique }

Les équations de récurrence associées sont :

SéqSym(/\) = []
SéqSym(/G,r,D\) = SéqSym(G) & [r] & SéqSym(D)

d) Recherche en arbre

La recherche en arbre peut se décliner sous trois formes. Le lexique général est le suivant :

Elément : type
ArbreBin : type arbre binaire Elément
P : fonction (E : Elément) ⟶ booléen

— Existence d'un élément vérifiant une propriété P :
On spécifie la fonction

ExisteP : fonction (A : ArbreBin, •••) ⟶ booléen

dont les équations de récurrence sont :

ExisteP(/\, •••) = faux
ExisteP(/G,r,D\, •••) = P(r) ou ExisteP(G, •••) ou ExisteP(D, •••)

— Extraction d'un élément vérifiant une propriété P :
On spécifie la fonction

UnP : fonction (A : ArbreBin, •••) ⟶ booléen, Elément

dont les équations de récurrence sont :

UnP(/\, •••) = <faux, ?>
UnP(/G,r,D\, •••) =
si P(r) alors <vrai, r>
sinon soit <trouvé, E> = UnP(G, •••)
dans si trouvé alors <vrai, E> sinon UnP(D, •••)

— Extraction du premier élément vérifiant une propriété P selon un ordre de parcours :
On spécifie la fonction

PremierPSym : fonction (A : ArbreBin, •••) ⟶ booléen, Elément

dont les équations de récurrence sont (pour le cas d'un parcours en ordre symétrique) :

PremierPSym(/\, •••) = <faux, ?>
PremierPSym(/G,r,D\, •••) =
soit <trouvé, E> = PremierPSym(G, •••)
dans si trouvé alors <vrai, E>
sinon si P(r) alors <vrai, r> sinon PremierPSym(D, •••)

B. Exemples d'exercices rédigés

Exemple E6.1 : Somme des entiers d'un arbre binaire

Décrire une fonction qui détermine la somme des entiers d'un arbre binaire donné.

Somme : fonction (A : arbre binaire d'entier) ⟶ entier

(i) Analyse récurrente (modèle 1)

(1) Somme (/\) = 0
(2) Somme(/G, r, D\) = r + Somme(G) +Somme(D)

(ii) Réalisation récursive

Somme(A) :
retour : si EstVide ?(A) alors 0
sinon Racine(A) + Somme(Gauche(A)) + Somme(Droit(A))

On peut aussi écrire

Somme(A) :
retour : si EstVide ?(A) alors 0
sinon soit /G, r, D\ = A dans r + Somme(G) + Somme(D)

Exemple E6.2 : Niveau minimum des feuilles d'un arbre

Décrire une fonction qui détermine le niveau minimum des feuilles d'un arbre binaire non vide.

Le *niveau* d'un noeud est le nombre de noeuds du chemin y conduisant à partir de la racine. On s'intéresse donc aux chemins les plus courts pour atteindre une feuille.

Elément : type ; ArbreBin : type arbre binaire Elément
NivM : fonction (A : ArbreBin non vide) ⟶ entier > 0

(i) Analyse récurrente (modèle 2)

La fonction ne peut pas être définie pour l'arbre vide. On applique le modèle d'analyse sur les arbres non vides et on utilise une fonction Min2V qui détermine le minimum de deux entiers :

(1) NivM (/ /\, r, /\ \) = 1
(2) NivM (/ G, r, /\ \) = 1 + NivM(G) { **G non vide** }
(3) NivM (/ /\, r, D \) = 1 + NivM (D) { **D non vide** }
(4) NivM (/G, r, D\) = 1 + Min2V(NivM (G), NivM (D)) { **G et D non vides** }

On peut simplifier l'écriture en utilisant les **notations abrégées** suivantes (remarquer qu'il n'est plus nécessaire de préciser dans les équations 2 à 4 que G ou D ne sont pas vides) :

(1) NivM (//r\\) = 1
(2) NivM (//G, r\\) = 1 + NivM(G)
(3) NivM (//r, D\\) = 1 + NivM (D)
(4) NivM (//G, r, D\\) = 1 + Min2V(NivM (G), NivM (D))

(ii) Réalisation récursive

NivM(A) :
retour : soit G = Gauche(A), D = Droit(A)
dans selon A
EstSingleton ?(A) : 1
EstUnaireG ?(A) : 1 + NivM(G)
EstUnaireD ?(A) : 1 + NivM(D)
EstBinaire ?(A) : 1 + Min2V(NivM (G), NivM (D))

Exemple E6.3 : Nombre de feuilles d'un arbre

Décrire une fonction qui détermine le nombre de feuilles d'un arbre binaire.

NbF : fonction (A : ArbreBin) ⟶ entier ≥ 0

Pour identifier les feuilles d'un arbre, il faut distinguer le cas du singleton.

(1) NbF(/\) = 0
(2) NbF(/G, r, D\) = si EstVide ?(G) et EstVide ?(D) alors 1 sinon NbF(G) + NbF (D)

On peut aussi proposer une analyse excluant l'arbre vide. Il faut alors le préciser dans la spécification de la fonction :

NbF : fonction (A : ArbreBin non vide) ⟶ entier ≥ 0
(1) NbF(/e\) = 1
(2) NbF(/G, r\) = NbF(G)
(3) NbF(/r, D\) = NbF(D)
(4) NbF(/G, r, D\) = NbF(G) + NbF(D)

Exemple E6.4 : Égalité de deux arbres

Décrire une fonction qui détermine si deux arbres binaires sont égaux, c'est-à-dire s'ils ont la même structure et s'ils comportent les mêmes éléments aux mêmes noeuds.

EgArbre : fonction (A1, A2 : ArbreBin) ⟶ booléen

L'analyse récurrente est fondée sur une combinaison des cas du modèle 1 d'analyse des arbres binaires.

(1) EgArbre(/\, /\) = vrai
(2) EgArbre(/\, /G, r, D\) = faux
(3) EgArbre(/G, r, D\, /\) = faux
(4) EgArbre(/G1, r1, D1\, /G2, r2, D2\) =
r1 = r2 et puis EgArbre(G1, G2) et puis EgArbre(D1, D2)

Exemple E6.5 : Une représentation linéaire d'un arbre binaire

Décrire une fonction qui à un arbre binaire d'entiers associe un texte représentant cet arbre, avec les conventions suivantes :

- *le texte représentant l'arbre vide ou tout sous-arbre vide de l'arbre donné est un singleton formé du caractère '_' (souligné).*
- *le texte représentant un arbre non vide commence par une parenthèse ouvrante ('(') suivie d'un espace et se termine par un espace suivi d'une parenthèse fermante (')'). A l'intérieur de ces parenthèses, on trouve d'abord le texte représentant le sous-arbre gauche, puis une virgule entourée d'espaces, puis le texte représentant la racine, puis une virgule entourée d'espaces, puis le texte représentant le sous-arbre droit.*

TexteABE : arbre binaire d'entier ⟶ texte
TexteABVide : texte "_"
ParOuv : texte "(" ; ParFerm : texte ")" ; Sép : texte " , "
TexteE : fonction (e : entier) ⟶ texte { *texte représentant l'entier donné sous forme décimale* }

(1) TexteABE(/\) = TexteABVide
(2) TexteABE (/G, r, D\) =
ParOuv & TexteABE(G) & Sép & TexteE(r) & Sép & TexteABE(D) & ParFerm

C. Exercices

a) Familiarisation avec la notion d'arbre

E6.6 : Structuration d'un ensemble en arbre binaire
On considère un ensemble quelconque de 7 éléments. Dessiner diverses manières de structurer cet ensemble en arbre binaire. Donner un arbre de profondeur maximum, de profondeur minimum. En déduire la valeur des profondeurs minimum et maximum pour n éléments.

E6.7 : Des relations remarquables
Étant donné un arbre binaire A de n noeuds, on désigne par b, u et f les nombres de noeuds binaires, de noeuds unaires, et de feuilles : n = b+u+f.

- Montrer que n-1 = u + 2b, pour tout arbre binaire non vide.
- Montrer que le nombre de sous-arbres vides est n + 1.
- La relation b=f-1 est vraie pour tout arbre binaire non vide. Démontrer cette relation par récurrence sur l'ensemble des arbres binaires. Puis retrouver cette relation de manière analytique.

E6.8 : Recherche dichotomique
Étant donnée une suite d'éléments *classée* selon une relation d'ordre, la *recherche dichotomique* d'un élément dans cette suite procède de la manière suivante : si l'élément cherché n'est pas l'élément milieu de la suite, on se ramène au même problème en considérant soit la moitié gauche de la suite donnée, soit sa moitié droite, selon que l'élément cherché est inférieur ou supérieur à l'élément milieu.

- Illustrer ce principe sous forme d'un arbre binaire : S étant une suite de 15 entiers en ordre croissant, dessiner un arbre construit de la manière suivante. Chacun des sous-arbres de cet arbre correspond à la recherche dans une sous-séquence SS de S. Les noeuds de l'arbre sont ainsi étiquetés par des sous-séquences de S ; S est l'étiquette la racine ; si SS sous-séquence de S est l'étiquette d'un noeud et si m est l'élément milieu de SS, alors les sous-séquences moitié de SS (sans l'élément milieu m) sont les étiquettes des fils gauche et droit de ce noeud. Les feuilles de l'arbre sont étiquetées par des séquences singleton.
- Estimer le nombre maximum d'essais lorsque l'on pratique une recherche dichotomique dans une suite de 2^n-1 éléments.

b) Analyse récurrente sur les arbres binaires

Spécifier les fonctions suivantes, puis donner des équations de récurrence les définissant.

E6.9 : Profondeur d'un arbre

E6.10 : L'élément E est-il présent dans l'arbre ?

E6.11 : L'élément E fait-il partie de la descendance de l'élément F dans un arbre ?
E et/ou F peuvent ne pas être présents dans l'arbre.

E6.12 : Arbres symétriques
Le *symétrique* d'un arbre binaire A, est un arbre binaire B déduit de A en faisant correspondre un sous-arbre droit à tout sous-arbre gauche et vice versa : tout se passe comme si B était

l'image de A dans un miroir (Figure 6.2)

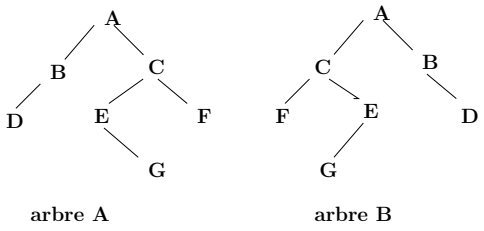


Figure 6.2 – deux arbres symétriques l'un de l'autre

Spécifier les fonctions suivantes et donner des équations de récurrence les définissant :
- Deux arbres sont-ils symétriques l'un de l'autre ?
- Construction de l'arbre symétrique d'un arbre.

E6.13 : Tous égaux ?
Décrire une fonction (spécification, équations de récurrence) déterminant si tous les éléments d'un arbre binaire sont égaux entre eux.
Indication : introduire une fonction déterminant si les éléments d'un arbre binaire sont tous égaux à un élément donné.

E6.14 : A propos du niveau d'un noeud dans un arbre
Le niveau d'un noeud dans un arbre binaire est le nombre de noeuds que l'on trouve sur le chemin conduisant de la racine à ce noeud (inclus). La racine est de niveau 1.
Si un noeud est de niveau n >1 dans un arbre /G, r, D\ alors il se situe soit dans G, soit dans D et il est de niveau n-1 dans le sous-arbre auquel il appartient.
Spécifier les fonctions suivantes, puis donner des équations de récurrence les définissant :
- Nombre de noeuds de niveau k. (k > 0 donné).
- Nombre de feuilles de niveau k. (k > 0 donné).
- Niveau d'un élément donné (0 si l'élément n'est pas présent dans l'arbre)

E6.15 : Les ascendants

- Spécifier une fonction déterminant la liste des ascendants d'un élément dans un arbre dont les éléments sont distincts deux à deux. Préciser le cas où l'élément donné n'est pas présent dans l'arbre.
- Donner des équations de récurrence définissant cette fonction.

D. Problèmes dirigés

E6.16 : Arbre de recherche binaire
Nous traitons ici une forme particulière d'organisation arborescente d'un ensemble d'informations destinée à faciliter la recherche d'une information donnée. Lorsque l'ensemble est organisé en séquence, et dans le cas défavorable où l'élément cherché n'est pas présent, le coût de l'évaluation de la fonction de recherche dépend linéairement du nombre d'éléments de l'ensemble. Il

en est de même lorsque l'ensemble est structuré en arbre binaire. Le lecteur peut le vérifier en examinant la fonction qui fait l'objet de l'exercice **E6.10**. On peut améliorer cette performance en imposant une propriété particulière aux éléments de l'arbre. C'est l'objet de ce problème.

(i) Définition
On considère un arbre binaire dont les éléments appartiennent à un type muni d'une relation d'ordre.
Un tel arbre est dit *arbre de recherche binaire* s'il vérifie la propriété ARB définie comme suit (Max(A) et Min(A) dénotent les valeurs maximum et minimum de l'ensemble des éléments de A) :

(1) ARB(/\) = vrai
(2) ARB(/G, r, D\) =
  (EstVide ?(G) ou alors (ARB(G) et puis r ≥ Max(G)))
  et (EstVide ?(D) ou alors (ARB (D) et puis r < Min(D)))

Cette définition signifie que pour tout noeud de l'arbre, si E désigne l'élément associé à ce noeud, tous les éléments du sous-arbre gauche de ce noeud sont inférieurs ou égaux à E et tous ceux du sous-arbre droit sont strictement supérieurs à E. Ceci est illustré par l'exemple de la Figure 6.3.

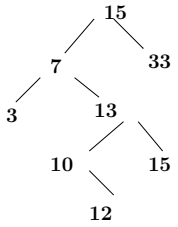


Figure 6.3 – un arbre de recherche binaire

(ii) Compréhension intuitive
A titre d'exemple, nous considérons ici une liste de prénoms avec répétitions éventuelles. L'ordre considéré est l'ordre alphabétique.
- Construire manuellement un arbre binaire de recherche à partir de cette liste : le premier prénom permet de construire un arbre simple. Le deuxième prénom est ajouté à l'arbre de manière à respecter la propriété. On procède ainsi prénom par prénom dans l'ordre de la liste initiale.
- Construire un nouvel arbre selon le même principe mais en partant d'une autre liste des mêmes prénoms. Comparer les arbres obtenus. Calculer la profondeur des arbres obtenus.
- Donner une liste initiale des prénoms qui conduise à un arbre de profondeur maximale.
- Donner une liste initiale des prénoms qui conduise à un arbre de profondeur minimale.
- Formuler en français le principe d'ajout d'un nouveau prénom à un arbre existant.
- Formuler en français le principe de recherche d'un prénom dans l'arbre. Dénombrer les comparaisons de prénoms nécessaires dans le cas où le prénom n'est pas présent. Donner une borne supérieure.
- Formuler en français le principe d'obtention de la liste des prénoms en ordre alphabétique.

(iii) Gestion d'un arbre de recherche binaire

On considère le problème de la gestion d'un répertoire téléphonique, ensemble de couples <*nom d'une personne, numéro de téléphone*>. Le répertoire est représenté par un arbre de recherche binaire, l'ordre considéré étant l'ordre alphabétique sur les noms.

Étudier (spécification, équations de récurrence) les fonctions suivantes :

— Liste du répertoire dans l'ordre alphabétique des personnes.

— Opérations de gestion (sans tenir compte d'erreurs éventuelles de cohérence des données) : initialisation, ajout, modification, suppression, consultation.

E6.17 : Dictionnaire arborescent

(i) Organisation du dictionnaire

On considère un dictionnaire composé d'un ensemble de mots distincts, organisé comme indiqué Figure 6.4.

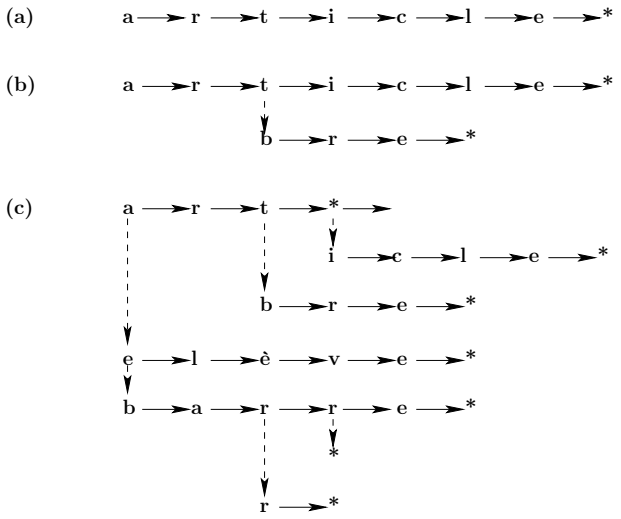


Figure 6.4 – organisation d'un dictionnaire sous forme arborescente avec partage des plus longs préfixes communs : (a) le mot *article* marqué par une étoile ; (b) les mots *article* et *arbre* ont *ar* pour plus long préfixe commun ; (c) les mots *art, arbre, élève, barre, bal, article, bar*.

— A chaque mot correspond la séquence de ses caractères, marquée par le caractère étoile ('*').

— Deux mots peuvent avoir un préfixe commun : par exemple les mots *arbre* et *article* ont *ar* pour plus long préfixe commun. Ce plus long préfixe commun n'est pas dupliqué : les deux séquences représentant les deux mots sont reliées au niveau du premier caractère qui diffère d'une séquence à l'autre.

— Il est possible qu'un mot soit le préfixe d'un autre mot. Par exemple, le mot *art* est le début du mot *article*. Le caractère '*' qui marque la fin des mots permet alors de distinguer le cas où *art* est pas présent dans l'ensemble du cas où il ne l'est pas.

Ainsi, le dictionnaire est organisé comme une collection structurée de caractères dans laquelle chaque caractère peut avoir deux types de liens :

— Un lien *successeur* : dans le mot *art*, la lettre *r* suit la lettre *a*, la lettre *t* suit la lettre *r*. Un tel lien est figuré dans les dessins par une flèche horizontale.

— Un lien *alternan* : après le préfixe *ar*, la lettre *b* est un alternant de la lettre *t* ; après le préfixe *art*, le caractère * est un alternant de la lettre *i*, etc. Un tel lien est figuré dans les dessins par une flèche verticale.

Remarques

— A priori, il n'y a pas d'ordre imposé entre les alternants.

— Le partage des plus longs préfixes communs implique que, dans une liste d'alternants comme par exemple ['é', 'r', 'l'], dans la Figure 6.4 c, on ne trouve jamais deux fois le même caractère.

— Un caractère '*' n'a jamais de successeur mais il peut avoir un alternant.

(ii) Interprétation comme un arbre binaire

Le dictionnaire peut être inteprété comme étant un arbre binaire de caractères : on peut par exemple convenir qu'un lien *successeur* correspond à un lien *gauche* et qu'un lien *alternant* correspond à un lien *droit*. La racine de l'arbre correspond au premier caractère du premier mot : dans l'exemple ci-dessus, il s'agit de la lettre *a* des mots commençant par *a*. Un sous-arbre gauche n'est vide que si sa racine est une étoile. Dans tout arbre de la forme /G, r, D\, si r n'est pas une étoile, la racine de G est le successeur de r et si D n'est pas vide, la racine de D est l'alternant de r.

Dessiner l'arbre binaire correspondant à la Figure 6.4.

(iii) Consultation et gestion du dictionnaire

L'objet de l'exemple est d'étudier quelques fonctions de traitement d'un dictionnaire représenté sous forme arborescente, en l'interprétant comme un arbre binaire de caractères avec la convention énoncée ci-dessus en (ii)

Nous utilisons le lexique suivant :

étoile : caractère '*'
MotMarqué : type séquence non vide de caractère terminée par une étoile
Mot : type séquence de caractère
Dico : type arbre binaire de caractère

Étudier les fonctions suivantes :

Q1. NbM détermine le nombre de mots présents dans un dictionnaire.

Q2. Ap détermine si un mot marqué donné appartient à un dictionnaire.

Q3. NbML détermine le nombre de mots de longueur k dans un dictionnaire (marque incluse).

Q4. LesMots construit la séquence des mots (non marqués) d'un dictionnaire.

Q5. LeDico est une opération de construction d'un dictionnaire formé d'un seul mot.

Q6. Plus est une opération d'ajout d'un mot marqué à un dictionnaire.

Q7. Moins est une opération de suppression d'un mot marqué d'un dictionnaire.

7. Réalisation récursive d'une action

A. Eléments de cours

a) Observation de la réalisation récursive d'une action

Contexte : une machine d'affichage

L'état de la machine d'affichage est caractérisé par l'état de deux composants : l'*écran* et le *curseur*. L'écran est un ensemble de *positions d'affichage* dans chacune desquelles on peut placer un caractère. Un *curseur* nommé Crs, désigne à tout instant la position de l'écran dans laquelle on est susceptible de placer un caractère.
Un état de l'écran est caractérisé par l'ensemble des caractères qu'il comporte. L'écran est *vide* lorsque toutes les positions d'affichage comportent un espace. L'écran est organisé sous forme d'une suite de lignes. Chaque ligne est une suite de positions d'affichage. La première position de la première ligne de l'écran est nommée PremPos (en abrégé PrP). Toute position p de l'écran a un successeur (successeur dans la ligne ou première position de la ligne suivante) noté suc (p). Lorsque le curseur est sur la première position d'une ligne, on dit qu'il est au début de la ligne. Dans les exemples et exercices qui suivent, nous utilisons les primitives suivantes :

ViderEcran *{ en abrégé VE }* : action
{ état initial : indifférent ; état final : écran vide et Crs = PrP }
AfficherCaractère *{ en abrégé AfCar }* : action (donnée c : caractère)
{ état initial : Crs=p0, écran indifférent
état final : Crs=suc (p0), écran modifié en p0 par le caractère donné }
AfficherChaîne *{ en abrégé AfCh }* :
action (donnée S : séquence de caractère)
{ état initial : Crs=p0, écran indifférent
état final : la chaîne donnée a été affichée à partir de la position p0 et le curseur suit le dernier caractère de la chaîne affichée }
EstAuDébut ? *{ en abrégé AuDébut }* : fonction $\longrightarrow$ booléen
{ vrai $\Longleftrightarrow$ le curseur est au début de sa ligne }
ALaLigne : action
{ état initial : Crs=p0, écran indifférent ; état final : Crs = 1ère position de la ligne suivant celle de p0 ; écran inchangé par rapport à l'état initial }

Exemple E7.1 : Forme binaire d'un entier

Étudier l'affichage de la forme binaire d'un entier naturel.

AfBin1 : action (donnée e : entier > 0)
{ état initial : Crs=p0, écran indifférent ; état final : la représentation binaire de l'entier donné a été affichée à partir de p0. Le curseur suit le dernier bit affiché. Le reste de l'écran est inchangé. }

Analyse récurrente (fonctionnelle) :

EntVersBin : fonction (e : entier > 0) $\longrightarrow$ texte *{ formé de '0' et de '1' }*
(1) EntVersBin (1) = "1"
(2) EntVersBin (2*n) = EntVersBin(n) •'0' *{ n > 0 }*
(3) EntVersBin (2*n +1) = EntVersBin(n) •'1' *{ n > 0 }*

On obtient la réalisation récursive suivante :

```
AfBin1 (X) :
   si X = 1 alors AfCar ('1')
   sinon AfBin1 (X quotient 2) ; AfCar (si pair (X) alors '0' sinon '1')
```

Pour comprendre ce texte, on peut procéder par réécriture. Par exemple, l'appel AfBin1 (13) se réécrit successivement de la manière suivante :

AfBin1 (13) $\longrightarrow$ AfBin1 (6) ; AfCar ('1')
$\longrightarrow$ AfBin1 (3) ; AfCar ('0') ; AfCar ('1')
$\longrightarrow$ AfBin1 (1) ; AfCar ('1') ; AfCar ('0') ; AfCar ('1')
$\longrightarrow$ AfCar ('1') ; AfCar ('1') ; AfCar ('0') ; AfCar ('1')

L'exécution de l'appel AfBin1 (13) engendre tout d'abord successivement les appels AfBin1 (6), AfBin1 (3), et AfBin1 (1), laissant en attente à chaque niveau un appel de l'action AfCar. Ces appels sont exécutés en ordre inverse dans un second temps. On peut visualiser ce processus en mettant en évidence sa structure emboîtée par des indentations, chaque niveau de marge correspondant à un niveau d'appel de l'action récursive :

```
AfBin1 (13)
   AfBin1 (6)
      AfBin1 (3)
         AfBin1 (1)
            AfCar ('1')
         AfCar ('1')
      AfCar ('0')
   AfCar ('1')
```

b) Du récursif à l'itératif

b.1. Schéma général considéré

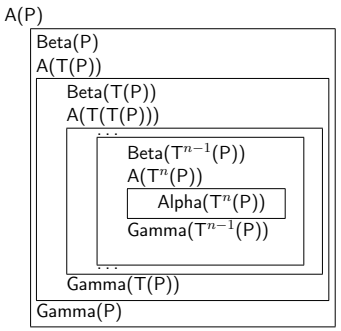
Le schéma général que nous étudions est le suivant :

```
A(P) :
   si EstBase(P) alors
      Alpha(P)
   sinon Beta(P)
         A(T(P))
         Gamma(P)
Param : type des paramètres
EstBase : fonction (P : Param) ⟶ booléen
   { EstBase(P) vrai ⟺ la valeur de P correspond au cas de base }
Alpha, Beta, Gamma : action (donnée : Param)
T : fonction (P : Param) ⟶ Param
```

b.2. Structure des appels récursifs

Pour une telle définition, la liste des paramètres des appels récursifs est : P, T(P), T(T(P)), ..., T^{n-1}(P), T^n(P).

La structure du processus d'exécution est illustrée comme suit :



On observe les appels récursifs successifs : exécution des Beta, mise en attente des Gamma. Puis, le cas de base : exécution d'Alpha. Puis, l'exécution des Gamma mises en attente, dans l'ordre inverse de la mise en attente.

b.3. Schéma itératif associé (utilisation d'une pile)

Pour traduire un algorithme récursif en un algorithme itératif, on utilise une séquence S pour mémoriser les valeurs de P.
Si l'algorithme récursif a la forme suivante :

```
A(P) :
   si EstBase(P) alors
      Alpha(P)
   sinon
      Beta(P)
      A(T(P))
      Gamma(P)
```

L'algorithme itératif équivalent a la forme suivante :

```
A(P) :
   S : séquence de Param
   Pcour : Param

   S ⟵ [] ; Pcour ⟵ P
   tant que non EstBase(Pcour)
      Beta(Pcour)
      S ⟵ S •Pcour ; Pcour ⟵ T(Pcour)
   Alpha(Pcour)
   tant que non EstVide ?(S)
      Pcour ⟵ dernier(S) ; S ⟵ début(S)
      Gamma(Pcour)
```

Les ajouts et les suppressions dans S se font toujours à la même extrémité : S est une pile.

b.4. Action récursive terminale

Les cas particuliers du schéma général récursif sont les cas où Beta ou Gamma sont vides. Lorsque Gamma est vide, on parle d'une **action récursive terminale**.
Le schéma récursif est alors :

```
A(P) :
   si EstBase(P) alors
      Alpha(P)
   sinon
      Beta(P)
      A(T(P))
```

Le schéma itératif équivalent est :

```
A(P) :
   Pcour : Param

   Pcour ⟵ P
   tant que non EstBase(Pcour)
      Beta(Pcour) ; Pcour ⟵ T(Pcour)
   Alpha(Pcour)
```

B. Exemples d'exercices rédigés

a) Tracé de dessins

On utilise la machine-tracés définie au §2.D.

Exemple E7.2 : Carrés emboîtés

Définir une action de tracé permettant de tracer une figure formée de n *carrés emboîtés (*n=4 *sur la figure 7.1). La figure doit être tracée sans lever la plume et sans tracer deux fois le même trait. On peut toutefois passer plusieurs fois par le même point.*

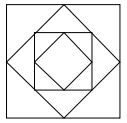


Figure 7.1 – 4 carrés emboîtés à tracer sans lever la plume

Une telle figure est définie par le nombre de carrés emboîtés et son carré extérieur. On convient ici qu'*un carré est défini par l'un de ses sommets, sa taille et son cap* : la taille du carré est la taille de ses côtés ; le cap du carré est le cap du côté passant par le sommet tel qu'un observateur placé sur ce sommet et regardant dans le sens de ce côté, voit tous les autres côtés *à sa gauche et devant lui* .
Ainsi *une figure est donnée par son ordre* (nombre de carrés emboîtés), *son sommet* (sommet du carré extérieur), *sa taille* (taille du carré extérieur) *et son cap* (cap du carré extérieur).

On spécifie l'action Tcaremb, de tracé de carrés emboîtés :

```
Tcaremb : action (donnée n : entier > 0, t : réel > 0)
    { Trace sans lever la plume et sans tracer deux fois le même trait, une figure de n carrés emboîtés et
    de taille t :
    état initial : plume basse au sommet de la figure et au cap de la figure.
    état final : figure tracée, plume dans l'état initial }
```

On définit l'ensemble des figures de manière récurrente :
- Base : la figure d'ordre 1 comporte un seul carré ;
- Construction récurrente : une figure d'ordre n+1 est composée d'un carré extérieur contenant une figure d'ordre n, comme l'illustre la figure 7.2.

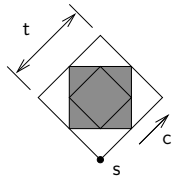


Figure 7.2 – Une figure d'ordre $n+1$ est constituée d'un carré et d'une figure d'ordre n.

Pour tenir compte de la contrainte de tracé, on décompose le carré extérieur en deux parties délimitées par l'un des points communs au carré extérieur et à la figure intérieure. On décompose ainsi le tracé d'un carré en deux actions intermédiaires de tracé : TDébutCar et TFinCar de telle sorte que :

```
Tcarré (t) : TDébutCar (t) ; TFinCar (t)


Tdébutcar : action (donnée t : réel >0)
    { trace le début d'un carré de taille donnée :
    Etat initial : plume basse au sommet du carré, orientée selon le cap du carré
    Etat final : début du carré tracé, plume basse au milieu d'un côté et au cap de ce côté }
Tfincar : action (donnée t : réel >0)
    { trace la fin d'un carré de taille donnée :
    Etat initial : plume basse au milieu du côté d'un carré, orientée selon le cap de ce côté
    Etat final : fin du carré tracée, plume basse au sommet du carré et au cap du carré }
```

En choisissant comme point intermédiaire le milieu du premier côté du carré, on obtient :

```
Tdébutcar (t) : Av (t/2)
Tfincar (t) :
    Av (t/2) ; TrG (90)
    pour i allant de 1 à 3 : Av (t) ; TrG (90)
```


On peut maintenant donner une réalisation récursive de l'action Tcaremb, s'appuyant sur la définition récurrente de la figure :

```
Tcaremb (n, t) :                                                    { réalisation récursive }
{ plume basse au sommet et au cap de la figure }
    si n = 1 alors
        Tdébutcar (t) ; Tfincar (t)
    sinon
        Tdébutcar (t)
        TrG (45)
            { plume basse au milieu du côté, au cap de la figure intérieure }
        Tcaremb (n-1, t/√2)
            { figure intérieure tracée, plume basse à son sommet et à son cap }
        TrD (45)
        Tfincar (t)
{ figure tracée, plume dans le même état qu'au départ }
```

Par exemple l'algorithme

```
Vider ; Baisser ; Tcaremb (10, 4)
```

trace une figure d'ordre 10 de sommet le centre de l'écran, de taille 4 et de cap OuestEst.

Forme itérative de l'action de tracé de carrés emboîtés

Une manière de concrétiser le processus engendré par l'appel d'une action récursive est de réaliser une action itérative qui décrit le même processus. L'équivalence considérée ici est définie en considérant la succession d'appels des actions élémentaires qui sont composées par l'action récursive (dans notre exemple, les actions Tdébutcar, Tfincar, TrD et TrG).
Le tracé de la figure consiste tout d'abord en une succession de tracés de débuts de carrés, en commençant par le carré extérieur et en terminant par le carré le plus interne, puis en une succession de tracés de fins de carrés dans l'ordre inverse.

```
Tcaremb (n, t) :                                                    { réalisation itérative }
{ plume basse au sommet et au cap de la figure }
    tc : réel                                                       { taille de la figure courante }
    nc : entier                                                     { ordre de la figure courante }
    tc ⟵ t ; nc ⟵ n                                                 { figure à tracer }
    { suite des tracés des débuts des carrés extérieurs des figures d'ordre > 1 }
    tant que nc > 1
        { plume basse au sommet et au cap de la figure courante, d'ordre nc et de taille tc }
        Tdébutcar (tc) ; TrG (45) ; tc ⟵ tc/√2 ; nc ⟵ nc -1
    { tracé du carré le plus interne : figure d'ordre 1 }
    Tdébutcar (tc) ; Tfincar (tc)
    { suite de tracés de fins de carrés extérieurs de figures d'ordre > 1 }
    tant que nc < n
        { plume basse au sommet et au cap de la figure intérieure d'ordre nc et de taille tc }
        nc ⟵ nc +1 ; tc ⟵ tc*√2 ;
        TrD (45) ; Tfincar (tc)
{ figure tracée, plume dans le même état qu'au départ }
```

b) Machine d'affichage

On utilise la machine d'affichage définie au paragraphe 7.A.a).

Exemple E7.3 : Affichage d'un texte

On considère l'action AfChaîne1 *spécifiée de la manière suivante :*

```
AfChaîne1 : action (donnée T : texte)
    { Affiche le texte T.
    e.i. : soit p la position initiale du curseur sur l'écran ;
    e.f. : le texte T est affiché à partir de la position p ; le curseur se trouve sur la position suivant celle
    du dernier caractère affiché. Si le texte est vide : action vide. }
```

Donner une réalisation récursive de l'action AfChaîne1.

Version 1 : On découpe le texte donné selon son premier caractère : T = premier(T)∘fin(T), ce qui donne la réalisation correspondante :

```
AfChaîne1(T) :                                                    { version 1 }
    si non EstVide ?(T) alors
        AfCar(premier(T))
        AfChaîne1(fin(T))
```

Version 2 : On découpe le texte donné selon son dernier caractère : T = début(T)•dernier(T), ce qui donne la réalisation correspondante :

```
AfChaîne1(T) :                                                    { version 2 }
    si non EstVide ?(T) alors
        AfChaîne1(début(T))
        AfCar(dernier(T))
```

On modifie maintenant la spécification initiale de l'action et on définit une nouvelle action :

```
AfChaîne2 : action (donnée T : texte)
    { Affiche le texte T.
    e.i. : soit p la position initiale du curseur sur l'écran ;
    e.f. : le texte T est affiché à partir de la position p ; le curseur se trouve au début de la ligne suivant
    celle du dernier caractère affiché. Si le texte est vide, le curseur se trouve au début de la ligne suivant
    celle de l'état initial. }
```

Modifier la réalisation de l'action AfChaîne1 pour en déduire celle de AfChaîne2 entraîne une solution fausse qui serait, pour la version 1 par exemple :

```
AfChaîne2(T) :
    si non EstVide ?(T) alors
        AfCar(premier(T)) ; AfChaîne2(fin(T)) ; ALaLigne
    sinon ALaLigne
```

Cette solution est incorrecte car elle engendre autant de ALaLigne que d'appels récursifs, alors qu'on ne veut qu'un seul ALaLigne à la fin du texte.
La solution correcte est d'utiliser l'action AfChaîne1 :

```
AfChaîne2(T) :
    AfChaîne1(T) ; ALaLigne
```


c) Traitement récursif de listes chaînées

Le traitement récursif d'une séquence est décrit en termes du même traitement portant sur la séquence de fin (la séquence donnée sans son premier élément).
Lorsqu'une séquence représentée sous forme de liste chaînée est en paramètre d'une action, l'adresse d'un élément E d'une séquence S donne accès à la sous-séquence de S commençant en E et se terminant par le dernier élément de S.

Exemple E7.4 : Somme des éléments d'une liste chaînée

Réaliser un algorithme récursif de calcul de la somme des éléments d'une liste chaînée d'entiers.

L'analyse récurrente du problème est :

(1) Somme([]) = 0 ; (2) Somme(e∘S) = e+Somme(S)

La réalisation récursive correspondante est :

```
adCel : type pointeur de Cel
Cel : type <E : entier, Suc : adCel>
Somme : fonction (Z : adCel)  ⟶ entier
T : adCel

Somme(Z) :
    retour :
        si Z = Nil alors 0
        sinon Z↑.E + Somme(Z↑.Suc)

Écrire(Somme(T))
```

Exemple E7.5 : Sélection des éléments positifs

Construire une séquence formée des entiers strictement positifs d'une séquence d'entiers donnée. L'ordre initial doit être maintenu.

L'analyse récurrente (fonctionnelle) est :

```
SélPos : fonction (S : séquence d'entier)  ⟶ séquence d'entier > 0

(1) SélPos([]) = []
(2) SélPos(e∘S) = si e > 0 alors e∘SélPos(S) sinon SélPos(S)
```

Réalisation récursive, version 1 : Construction des éléments du résultat "de gauche à droite" (ordre où ils apparaissent dans la donnée).

```
AdCel : type pointeur de Cel ; Cel : type <E : entier ; Suc : AdCel>

CréerSélPosListe : action (donnée X1 : AdCel ; résultat X2 : AdCel)
    { e.i. : soit s0 la séquence représentée par la liste de tête X1
    e.f. : la liste de tête X2 représente SélPos(s0) }

CréerSélPosListe(X1, X2) :                                  { la liste X1 représente s0 }
    si X1 = Nil alors X2 ⟵ Nil                                          { s0 est vide }
    sinon si X1↑.E ≤ 0 alors
            CréerSélPosListe(X1↑.Suc, X2)
        sinon                                                    { premier(s0) > 0 }
            Allouer(X2) ; X2↑ ⟵ <X1↑.E, Nil>                  { création du singleton }
            CréerSélPosListe(X1↑.Suc, X2↑.Suc)
```

Réalisation récursive, version 2 : Construction des éléments du résultat "de droite à gauche" (inverse de l'ordre dans la donnée)

```
CréerSélPosListe(X1, X2) :                                  { la liste X1 représente s0 }
    Z : AdCel

    si X1 = Nil alors X2 ⟵ Nil                              { s0 est vide }
    sinon CréerSélPosListe(X↑.Suc, X2)
          { la liste de tête X2 représente SélPos(fin(s0)) }
          si X1↑.E > 0 alors                                { premier(s0) > 0 }
              Allouer(Z)                                    { ajout en tête }
              Z↑ ⟵ <X1↑.E, X2> ; X2 ⟵ Z
```

Exemple E7.6 : Suppression des éléments négatifs

Modifier une séquence d'entiers donnée en en supprimant les entiers négatifs.

Version 1 : suppression des éléments "de droite à gauche" (inverse de l'ordre dans la donnée).

```
AdCel : type pointeur de Cel
Cel : type <E : entier ; Suc : AdCel>

SupprimerNégListe : action (donnée-résultat X : AdCel)
    { e.i. : soit s0 la séquence représentée par la liste d'adresse X
    e.f. : la liste d'adresse X représente SélPos(s0) }

SupprimerNégListe(X) :                                      { la liste X représente s0 }
    Z : AdCel                                   { peut aussi être déclarée au niveau global }
    si X ≠ Nil alors
        SupprimerNégListe(X↑.Suc)
        { la liste de tête X↑.Suc représente SélPos(fin(s0)) }
        si X↑.E ≤ 0 alors                                   { premier(s0) ≤ 0 }
            Z ⟵ X ; X ⟵ X↑.Suc ; libérer(Z)                 { suppression en tête }
```

Version 2 : suppression des éléments "de gauche à droite" (ordre où ils apparaissent dans la donnée)

```
SupprimerNégListe(X) :                                      { la liste X représente s0 }
    Z : AdCel                                   { peut aussi être déclarée au niveau global }
    si X ≠Nil alors
        si X↑.E ≤ 0 alors                                   { premier(s0) ≤ 0 }
            Z ⟵ X ; X ⟵ X↑.Suc ; libérer(Z)                 { suppression en tête }
            SupprimerNégListe(X)
        sinon SupprimerNégListe(X↑.Suc)
```

d) Traitement récursif de séquences sous forme contigüe

Dans le cas de séquences représentées sous forme contigüe, une sous-séquence de fin d'une séquence est représentée par le tableau contenant la séquence, une borne et la longueur de la sous-séquence considérée. Une variante possible est la déclaration du tableau et des deux bornes de la sous-séquence. Dans les deux cas, on peut déclarer le tableau comme une variable globale.

Exemple E7.7 : Somme des éléments (E7.4 suite)

On reprend l'exemple **E7.4** *dans le cas d'une séquence sous forme contigüe.*

L'analyse récurrente du problème est :

(1) Somme([]) = 0 ; (2) Somme(S•e) = Somme(S)+e

La réalisation récursive correspondante est :

```
Lmax : constante de type entier > 0
TabE : type tableau sur [1 . . .Lmax] d'entier
Somme : fonction (L : entier sur [0 . . .Lmax], T : TabE) ⟶ entier
LD : entier sur [0 . . .Lmax], TD : TabE


Somme (L, T) :
    retour :
        si L = 0 alors 0
        sinon Somme(L-1, T) + TL

Écrire(Somme(LD, TD))
```

Exemple E7.8 : Sélection des éléments positifs (E7.5 suite)

On reprend l'exemple **E7.5** *dans le cas d'une séquence sous forme contigüe.*

L'analyse récurrente (fonctionnelle) est :

```
SélPos : fonction (S : séquence d'entier) ⟶ séquence d'entier > 0
(1) SélPos([]) = []
(3) SélPos(S•e) = si e > 0 alors SélPos(S)•e sinon SélPos(S)
```

La réalisation récursive est la suivante (construction "de gauche à droite" c'est-à-dire dans l'ordre où ils apparaissent dans la donnée) :

```
Lmax : constante de type entier > 0 ; TabE : type tableau sur [1. . .Lmax] d'entier
CréerSélPosTab : action (donnée T1 : TabE, L1 : entier sur [0. . .Lmax],
                         résultat T2 : TabE, L2 : entier sur [0. . .Lmax])
    {  e.i. : soit s0 la séquence représentée par <T1, L1>
    e.f. : <T2, L2> représente SélPos(s0)  }

CréerSélPosTab(T1, L1, T2, L2) :                    { <T1, L1> représente s0 }
    si L1 = 0 alors L2 ⟵ 0                                  { s0 est vide }
    sinon CréerSélPosTab(T1, L1-1, T2, L2)
          { <T2, L2> représente SélPos(début(s0)) }
          si T1_L1 > 0 alors                                { dernier(s0) > 0 }
              L2 ⟵ L2 + 1 ; T2_L2 ⟵ T1_L1                   { ajout en queue }
```

Exemple E7.9 : Suppression des éléments négatifs (E7.6 suite)

On reprend l'exemple **E7.6** *dans le cas d'une séquence sous forme contigüe.*

On a la réalisation suivante (construction "de gauche à droite") :

```
Lmax : constante de type entier > 0 ; TabE : type tableau sur [1. . .Lmax] d'entier
SupprimerNégTab : action (donnée-résultat T : TabE, L : entier sur [0 . . .Lmax])
    {  e.i. : soit s0 la séquence représentée par <T, L>
    e.f. : <T, L> représente SélPos(s0) }
```

```
SupprimerNégTab(T, L) :                                  { <T, L> représente s0 }
   L1 : entier sur [0...Lmax]
   si L ≠ 0 alors
      si T_L ≤ 0 alors                                   { dernier(s0) ≤ 0 }
         L ⟵ L - 1                                       { suppression en queue }
         SupprimerNégTab(T, L)
      sinon
         L1 ⟵ L-1 ; SupprimerNégTab(T, L1)
         T_L1+1 ⟵ T_L ; L ⟵ L1 + 1                       { ajout en queue }
```

C. Exercices

a) Machine d'affichage

E7.10 : Formation d'un texte

On spécifie une action nommée FormerT :

```
Mot : type séquence de lettres
FormerT : action (donnée SM : séquence de mot ; résultat T : texte)
   { T est le texte formé des mots de SM, dans le même ordre et tel que tout mot sauf le dernier est
   suivi d'un espace. Si SM est vide, T est vide. }
```

On propose la réalisation suivante de cette action :

```
espace : caractère ' '
FormerT(SM, T) :
   si EstVide ?(SM) alors T ⟵ []                                    (1)
   sinon FormerT(fin(SM), T)                                        (2)
         T ⟵ (premier(SM) •espace) & T                              (3)
```

— Quelle est la valeur de R après l'exécution de l'appel

```
FormerT (["voici", "un", "exemple"], R)                  { R : texte }
```

— La solution proposée est-elle correcte ? Justifier votre réponse.

— Si la solution est incorrecte, donner un exemple le plus simple possible mettant l'erreur en évidence, puis proposer une solution récursive correcte.

E7.11 : Un mot et son inverse

Q1. On spécifie une action AfMir :

```
Mot : type séquence de lettre
AfMir : action (donnée M : Mot)
   { Affiche le mot M suivi de son image miroir, à partir de la position courante du curseur. A l'état
   final, le curseur est placé juste après le dernier caractère affiché. Si M est vide, l'action n'a aucun
   effet. }
```

Par exemple : AfMir ("abcde") affiche la chaîne "abcdeedcba" à partir de la position courante du curseur.

Dans ce qui suit, on fera abstraction de la représentation des textes traités en utilisant des variables de type Mot manipulées avec les opérateurs généraux définis sur les séquences.

i) Donner une première réalisation de AfMir, fondée sur l'introduction d'une action intermédiaire nommée AfChEnvers d'affichage d'un texte "à l'envers" : spécifier cette action puis l'utiliser pour réaliser l'action AfMir. Terminer en donnant une réalisation récursive de AfChEnvers.

affichage	commentaires
A	*Le 'A' est en 1ère position de l'écran*
B	*Le 'B' est en 2ème ligne décalé de 3 positions / au 'A'*
C	...
D	*Le 'D' est en quatrième ligne décalé de 3 positions / au 'C'*
C	...
B	
A	
	Le curseur se trouve sur la première position de la première ligne après le 'A'. Le reste de l'écran est vide.

Figure 7.3 – le mot "ABCD" et son inverse "en triangle"

ii) Donner une deuxième réalisation de AfMir, en la décrivant directement sous forme récursive.

iii) Que faut-il modifier pour qu'après l'affichage, le curseur se trouve au début de la ligne suivante ?

Q2. On désire maintenant afficher le mot donné et son inverse à raison d'un caractère par ligne en structurant l'affichage "en triangle" par des décalages adéquats comme l'illustre la figure 7.3 pour le mot "ABCD". L'affichage est effectué sur un écran vide, la première lettre du mot apparaissant en première position de la première ligne de l'écran.
Pour obtenir cet affichage, on spécifie l'action AF de la manière suivante :

```
AF : action (donnée M : Mot)
   { Affiche le mot et son inverse "en triangle" comme illustré par la figure 7.3. État initial : indifférent ;
   État final : l'affichage a été réalisé sur un écran vide et le curseur se trouve au début de la ligne
   suivant le dernier caractère affiché. Si le mot est vide, l'état de l'écran et du curseur sont inchangés.
   Précondition : la longueur du mot donné permet l'affichage attendu (par rapport aux dimensions de
   l'écran). }
```

L'observation de la figure pour un mot non vide m = pc ∘ fm montre que l'on peut la décrire récursivement, en termes de l'affichage du caractère pc et de la figure associée à fm, à condition de caractériser la figure non seulement par le mot donné mais aussi par la marge fixant la position sur leur ligne respective du premier et du dernier caractères affichés. On spécifie ainsi une nouvelle action AF1 de la manière suivante :

```
AF1 : action (donnée n : entier ≥ 0, M : Mot non vide)
   { Affiche le mot M et son inverse "en triangle" à partir de la marge n de la ligne courante. Etat
   initial : curseur au début d'une ligne, soit L0 ; Etat final : le mot M et son inverse "en triangle" ont été
   ajoutés à l'écran à partir de la ligne L0 , le premier et le dernier caractère étant affichés en position n ;
   le curseur se trouve au début de la ligne suivant le dernier caractère affiché. Précondition : la valeur
   de la marge n et la longueur du mot M permettent l'affichage attendu (par rapport aux dimensions
   de l'écran). }
```

i) Réaliser l'action AF en termes de l'action AF1.

ii) Donner une réalisation récursive de l'action AF1.

iii) Analyse quantitative : donner le nombre d'appels de l'action AlaLigne engendrés par un appel de l'action AF.

iv) Prouver par récurrence la correction de la réalisation proposée.

v) Reprendre le problème pour réaliser l'affichage suivant (cas du mot "ABCD") :

```
      A
    B
  C
D
  C
    B
      A
```

b) Dessins récursifs

On utilise la machine-tracés définie au §2.D.

E7.12 : Trace de l'exécution d'une action réalisée récursivement
Observer l'exécution de l'action récursive Tcaremb de tracé de carrés emboîtés :

- Donner la trace de l'exécution de l'algorithme Vider ; Baisser ; Tcaremb (4, 4) sous forme de la liste des appels engendrés, mise en page avec une indentation reflétant la structure emboîtée des appels récursifs (voir la trace de AfBin1(13) au paragraphe 7.A.a)).
- Spécifier une action qui produit cette trace (procéder par analogie avec l'exercice E7.11 Q2). Réaliser cette action sous forme récursive.

E7.13 : Arbres binaires
On veut réaliser une action de tracé d'un arbre binaire. Les données sont la profondeur de l'arbre, le point origine, la taille et le cap du tronc. On dit que la profondeur de l'arbre est le nombre d'arcs sur un chemin de longueur maximale conduisant de la racine à une feuille. La figure 7.4 montre un arbre de profondeur 1 et un arbre de profondeur 4.

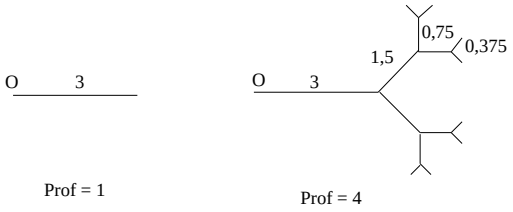


Figure 7.4 – Arbre binaire

Q1. Schématiser par un dessin la définition récurrente de la figure de profondeur P, d'origine O, de taille t et de cap C, en termes des figures correspondant aux sous-arbres. Faire apparaître sur le dessin le point O' origine des sous-arbres et donner les autres caractéristiques des sous-arbres en fonction de P, t et C.

Q2. L'action de tracé est spécifiée comme suit :

TracerAB : action (donnée P : entier ≥ 1, T : réel > 0)
{ Trace l'arbre binaire de profondeur P et de taille T dont l'origine et le cap sont fixés par la position initiale de la plume. Etat initial : plume basse à l'origine de l'arbre et au cap de l'arbre. Etat final : l'arbre binaire a été ajouté à l'écran et la plume est dans l'état initial. }

Donner l'appel de l'action TracerAB permettant de tracer l'arbre de profondeur 4 de la figure 7.4.

Q3. On propose l'ébauche suivante pour la réalisation récursive de l'action TracerAB :

```
TracerAB(P, T) :
  si P = 1 alors
      ••••
  sinon
      ••••
    TracerAB(••••)
      ••••
    TracerAB(••••)
      ••••
  ••••
```

- Compléter cette réalisation récursive.
- Dessiner l'arbre des appels récursifs engendrés par l'appel permettant de tracer l'arbre de profondeur 4 de la figure 7.4.
- Donner le nombre d'appels récursifs engendrés par un appel de TracerAB pour un arbre de profondeur P.
- Modifier la réalisation précédente de manière à produire un dessin dans lequel deux niveaux successifs de l'arbre sont de couleur différentes. On utilisera la primitive Colorier de la machine tracé spécifiée de la façon suivante :
  Colorier : action (donnée c : Couleur)
  { c est une couleur dans Bleu, Vert, Rouge, Jaune, Noir ; e.i. : indifférent ; e.f. : CouleurEcr = c où CouleurEcr est la couleur de tracé de la plume. }

E7.14 : Flocons
On étudie une action de tracé de dessins de la famille donnée par la figure 7.5. Les données sont la taille du trait élémentaire et le nombre de niveaux.
Cette action est définie comme suit :

TracerF : action (donnée n : entier ≥ 0, t : réel > 0)
*{ Trace un flocon de n niveaux et dont le trait élémentaire est de taille t. L'origine du tracé et son cap sont donnés par l'état initial. Le reste de l'écran est inchangé. À l'état initial, la plume est basse, à l'origine et au cap du dessin ; à l'état final, la figure est tracée, la plume est basse, au cap initial et elle est translatée, par rapport au point initial, de t*3^n selon ce cap. }*

Donner une réalisation récursive de cette action.

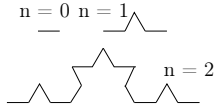


Figure 7.5 – Flocons

E7.15 : Etoiles
L'objet de cet exercice est de construire un algorithme de tracé d'un dessin récursif formé d'étoiles à cinq branches, comme l'illustrent les deux figures 7.6 et 7.7.

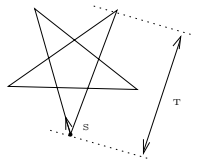


Figure 7.6 – Etoiles à cinq branches

Q1. Tracé d'une étoile à cinq branches
On spécifie une action de tracé d'une étoile à cinq branches comme celle de la figure 7.6. Une telle étoile est caractérisée par :

- son sommet (S sur la figure 7.6) : c'est l'un des sommets (points extrêmes) de l'étoile
- sa taille (T sur la figure 7.6) : c'est la taille commune de tous les segments reliant deux sommets de l'étoile
- son cap (marqué par un vecteur sur la figure 7.6) : c'est le cap de l'un des segments passant par le sommet de l'étoile, et tel qu'un observateur placé au sommet et regardant dans le sens défini par ce cap, voit l'autre extrémité du segment devant lui et le deuxième segment passant par le sommet à sa droite. On propose l'action suivante :

```
TracerUneEtoile : action (donnée T : réel)
    { S et C étant respectivement la position et le cap de la plume à l'appel de l'action, trace une étoile
    de taille T, de sommet S et de cap C. }

TracerUneEtoile (T) :
    A : constante 720/5 de type entier
    Répéter 5 fois
        Avancer (T) ; TournerDroite (A)
```

Compléter la spécification de l'action TracerUneEtoile en ce qui concerne l'état de la plume

Q2. Une explosion d'étoiles
On veut maintenant tracer les dessins récursifs composés d'étoiles comme ceux donnés dans la figure 7.7.
Un tel dessin est caractérisé par :

- son ordre, comme le suggèrent les figures : la figure d'ordre 1 comporte une étoile, la figure d'ordre 2 comporte une étoile de taille T et cinq étoiles de taille $T/3$, placées à chacun de ses sommets, ...
- son sommet, sa taille, son cap : ce sont le sommet, la taille et le cap de l'étoile la plus grande

On spécifie une action de tracé d'un tel dessin comme suit :

```
TracerDEtoiles : action (donnée n : entier > 0, T : réel)
{ à l'état initial, la plume est basse au sommet et au cap de la figure ; à l'état final, la figure de taille T
et d'ordre n est tracée ; la plume est dans l'état initial. }
```

Donner une réalisation récursive de cette action.

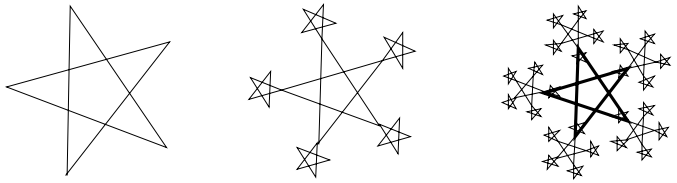


Figure 7.7 – Tracé d'étoiles récursif

E7.16 : Tricot (E2.21 suite)
On considère un triangle équilatéral pavé de triangles équilatéraux élémentaires (comme à l'exercice **E2.21**). Le dessin est caractérisé par un sommet du triangle extérieur, un cap, la taille du côté du triangle extérieur et le nombre n de divisions en traits élémentaires du côté du triangle extérieur (n est une puissance de 2).

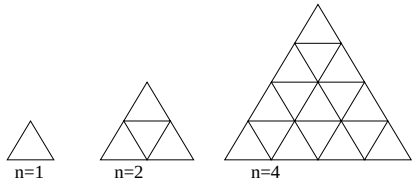


Figure 7.8 – Tricot

Pour tracer un tel dessin, on définit une action Ttriangles. Pour la réaliser, on peut envisager deux solutions :
(i) Solution 1
On décompose la figure en distinguant le triangle extérieur et l'intérieur de la figure. La figure intérieure doit être analysée récursivement. On peut la voir comme suggéré figure 7.9 (le point P et le cap apparaissant sur la figure illustrent l'état initial et l'état final de la plume). Elle n'existe que pour n > 1.

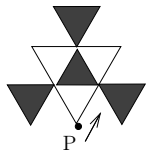


Figure 7.9 – Tricot : analyse de la figure intérieure, solution 1

On introduit une action intermédiaire TFigInt(n, T) qui trace l'intérieur de la figure d'ordre n et de taille T.
L'action Ttriangles est alors réalisée en trois étapes : tracé d'une partie du triangle extérieur délimitée par un point de connexion avec la figure intérieure (par exemple, le point P sur la figure 7.9) ; appel de TFigInt ; tracé de la fin du triangle extérieur.
L'action TFigInt est réalisée de manière récursive, en remarquant qu'elle est composée d'un triangle (apparaissant en blanc sur la figure 7.9) et de 4 exemplaires de figures intérieures (apparaissant en noir). Dans le cas général la réalisation de TFigInt est ainsi formée de 4 appels récursifs séparés par les instructions traçant le triangle.

— Donner la réalisation récursive de TFigInt.
— Donner la réalisation de Ttriangles en termes de TFigInt.

(ii) Solution 2
Comme pour la solution 1, on introduit une action intermédiaire TFig(n, T, sens) qui trace la figure d'ordre n et de taille T, sauf deux cotés extérieurs, comme l'illustre la figure 7.10. Le paramètre sens indique si la figure tracée se trouve à gauche (sens = 1) ou à droite (sens = -1) par rapport au cap initial.

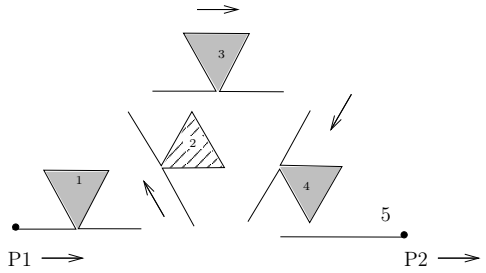


Figure 7.10 – Tricot : analyse de la figure intérieure, solution 2

La réalisation de l'action Ttriangles est alors :

```
Ttriangles(n, T) :
    TFig(n, T, 1)
    TournerGauche(120) ; Avancer(T) ; TournerGauche(120)
    Avancer(T) ; TournerGauche(120)
```

Donner la réalisation récursive de l'action TFig.
Attention : à l'instruction TournerGauche(A) lorsque sens = 1 correspond l'instruction TournerDroite(A) ou l'instruction TournerGauche(-A) lorsque sens = -1.

c) Traitement récursif de listes chaînées

E7.17 : Répartition des éléments d'une liste en deux listes
On étudie un algorithme de répartition des éléments d'une séquence S en deux séquences selon leur signe. Les séquences d'entiers sont représentées par des listes chaînées sous forme standard. On spécifie une action nommée Répartir :


```
adCelE : type pointeur de CelE ; CelE : type <E : entier, Esuiv : adCelE>
Répartir : action (donnée-résultat TS : adCelE ; résultat TP, TN : adCelE)
    { soit S la séquence représentée par la liste d'adresse TS.
    À l'état final :
        — la liste de tête TP représente la séquence des entiers strictement positifs de S ;
        — la liste de tête TN représente la séquence des entiers négatifs ou nuls de S ;
        — TS = Nil.
    L'algorithme procède par modification des liens de chaînage : à l'état final, les cellules des listes de
    têtes TP et TN sont celles données à l'état initial dans la liste de tête TS. }
```

L'algorithme doit être réalisé récursivement. Pour cela, on s'appuie sur l'analyse fonctionnelle suivante :

```
Répartition : fonction (S : séquence d'entier) ⟶ séquence d'entier, séquence d'entier
    { soit <P,N> = Répartition(S) : P est la séquence des entiers strictement positifs de S et N est la
    séquence des entiers négatifs ou nuls de S. }
(1) Répartition([]) = < [], []>
(2) Répartition(e∘S) = soit <P, N> = Répartition(S)
                       dans si e > 0 alors <e∘P, N> sinon <P, e∘N>
```

(i) Version 1
Le principe choisi est le suivant : répartir la fin de la liste donnée (tout sauf son premier élément) en deux listes, puis placer le premier élément dans l'une de ces deux listes selon son signe.

— Compléter l'ébauche suivante de la réalisation récursive de l'action Répartir :

```
Répartir(TS, TP, TN) :
    si TS = Nil alors
        ••••••••••••
    sinon Répartir(••••••••••••)
            si TS↑.E > 0 alors
                ••••••••••••
            sinon ••••••••••••
            ••••••••••••
```

— Dans quel ordre se trouvent les éléments des listes résultant de l'exécution de l'action Répartir (par rapport à l'ordre dans la liste donnée) ? Justifier la réponse par rapport à la réalisation donnée.

(ii) Version 2
Le principe choisi est le suivant : placer le premier élément de S dans la bonne liste selon son signe, puis répartir la liste de fin de S.

— Donner une réalisation récursive de l'action Répartir correspondant à ce principe.
— Dans quel ordre se trouvent les éléments des listes résultant de l'exécution de l'action Répartir (par rapport à l'ordre dans la liste donnée) ? Justifier la réponse par rapport à votre réalisation.

E7.18 : Opérations sur des séquences triées
On considère des séquences d'entiers triées, sous forme chaînée selon le lexique suivant :

```
adCelEnt : type pointeur de CelEnt ; CelEnt : type <E : entier, Suc : adCelEnt>
```

Pour chacune des opérations suivantes :
- proposer une analyse récurrente en termes fonctionnels et en faisant abstraction de la représentation ;
- donner une réalisation récursive ;
- donner une réalisation itérative.

Q1. Appartenance

ApListe : fonction (E : entier, T : adCelEnt) $\longrightarrow$ booléen
{ vrai $\Longleftrightarrow$ E $\in$ la séquence triée donnée par la liste de tête T }

Q2. Insertion

InsListe : action (donnée E : entier, donnée-résultat T : adCelEnt)
{ Insère un élément de valeur E dans la liste triée de tête T. Si à l'état initial la liste contient un ou plusieurs éléments de valeur E, à l'état final le nouvel élément de valeur E se trouve après les autres. A l'état final, la liste de tête T est triée. }

Q3. Suppression

SupListe : action (donnée E : entier, donnée-résultat T : adCelEnt)
{ Supprime **tous** *les éléments de valeur E dans la liste triée de tête T ; si à l'état initial la liste ne contient pas d'élément de valeur E, à l'état final elle est inchangée. A l'état final, la liste de tête T est triée. }*

Q4. Reprendre les questions précédentes dans le cas de séquences sous forme contigüe.

d) Un algorithme de tri

E7.19 : Tri par interclassement
On considère un tableau T d'entiers défini sur l'intervalle [1...n]. Réaliser un algorithme de tri selon le principe suivant : on découpe le tableau en deux sous-tableaux de longueur égale (à 1 près). Chaque sous-tableau est trié indépendamment. On interclasse ensuite les deux sous-tableaux.

On prend en compte la représentation de la séquence sous forme d'un tableau. Les sous-tableaux sont caractérisés par leurs bornes. Le découpage est un calcul d'indice. On fixe le lexique suivant :

n : constante de type entier > 0
indice : type entier sur [1...n]
T : tableau sur [1...n] d'entier
sous-tableau : type <bi, bs : indice>
{ si bi > bs, le sous-tableau considéré est vide }

TrierItc : action (donnée <inf, sup> : sous-tableau)
{ Trie, par interclassement, le sous-tableau $T_{inf...sup}$*. les autres parties du tableau T ne sont pas modifiées }*
Interclasser : action (donnée <i1, s1>, <i2, s2> : sous-tableau)
{ Ordonne le sous-tableau $T_{i1...s2}$ *sachant que les sous-tableaux* $T_{i1...s1}$ *et* $T_{i2...s2}$ *sont triés et qu'ils sont adjacents : s1=i2-1 }*

Le tri du tableau T est réalisé par l'appel : TrierItc (<1, n>).
On propose la réalisation récursive suivante de l'action de tri :


```
TrierItc (<inf, sup>) :                                   { réalisation récursive }
   m : un indice
   si inf < sup alors
      m ⟵ (inf + sup) quotient 2                  { 1≤inf<sup≤n ⟹ 1≤m<sup }
      TrierItc (<inf, m>) ; TrierItc (<m+1, sup>)
      Interclasser (<inf, m>, <m+1, sup>)
```

On suppose que T comporte les 8 éléments suivants :

T= [310, 285, 179, 652, 351, 423, 861, 254] , n = 8

- Dessiner l'arbre des appels récursifs engendrés par l'appel initial TrierItc (<1, 8>). Chaque noeud de l'arbre est étiqueté par les bornes du sous-tableau concerné par l'appel récursif.
- Donner l'arbre des appels successifs de l'action Interclasser engendrés par l'appel TrierItc (<1, 8>) et les états correspondants du tableau T.
- Si, comme dans cet exemple, n est une puissance de 2, évaluer les nombres d'appels de TrierItc et de Interclasser engendrés par l'appel initial.

8. Représentation chaînée des arbres

A. Éléments de cours

a) Représentation chaînée des arbres binaires

Comme pour les listes chaînées (voir chapitre 4), l'espace mémoire peut être représenté dans un tableau ou à l'aide de pointeurs. Les lexiques donnés ci-dessous correspondent à cette deuxième possibilité. Pour gérer cet espace, on utilise les primitives **MémoireSaturée**, **Allouer** et **Libérer**.

a.1. Principe de la représentation

À chaque élément de l'arbre est associée une adresse de nœud comportant un triplet <étiquette du nœud, lien vers le nœud racine du sous-arbre gauche, lien vers le nœud racine du sous-arbre droit>. **Nil** caractérise l'arbre vide.

```
Élément : type                                              { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud                            { adresses des cellules }
Noeud : type <R : Élément ; G, D : adNoeud >
    { Étiquette du nœud et adresses (éventuellement Nil) des fils gauche et droit. }
{ Un arbre binaire est accessible par l'adresse de sa racine. L'arbre vide a pour adresse Nil. }
```

Une adresse de type **adNoeud** permet d'accéder aussi bien à l'information portée par ce nœud qu'à l'arbre dont ce nœud est racine.

a.2. Principe d'analyse récurrente

De manière générale, le traitement d'un arbre binaire est ramené au traitement de sa racine et au traitement de chacun de ses sous-arbres (voir chapitre 6).

a.3. Traitement séquentiel d'un arbre binaire : schémas récursifs

Ces schémas correspondent aux problèmes que l'on peut reformuler comme un parcours ou une recherche (au sens donné pour les séquences, voir chapitre 4) **sur une séquence des éléments de l'arbre**. L'ordre entre les éléments est choisi en fonction du problème parmi les suivants (voir chapitre 6) : *préordre* (racine, sous-arbre gauche, sous-arbre droit), *ordre symétrique* (sous-arbre gauche, racine, sous-arbre droit) ou *ordre terminal* (sous-arbre gauche, sous-arbre droit, racine).
Dans le cas d'un *parcours*, le traitement est décrit par un lexique comportant les variables propres à ce traitement et par les actions de traitement élémentaire (nommée **Traiter**) et d'initialisation (nommée **InitTraitement**). Une *recherche* est spécifiée par la propriété caractérisant l'élément cherché. Dans les deux cas, les schémas récursifs engendrent les traitements (visite ou test de la propriété) dans l'ordre adéquat.
La **Figure 8.1** ci-dessous donne trois schémas : parcours en préordre, existence d'un élément vérifiant une propriété donnée, adresse du premier élément vérifiant une propriété en ordre symétrique.

Variante du schéma de parcours :
l'arbre vide est exclu en pré-condition. La ligne "si X ≠ Nil alors" disparaît, mais il faut ajouter le contrôle de cette pré-condition avant les appels récursifs.

Parcours en ordre symétrique ou en ordre terminal
Les réalisations sont analogues à celles données : il suffit de changer l'ordre entre l'appel **Traiter** et les appels récursifs portant sur les sous-arbres.

```
Élément : type                                              { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud
Noeud : type <R : Élément ; G, D : adNoeud >
Parcours en préordre
ParcoursPréordre : action (donnée X : AdNoeud)
    { Application de l'action Traiter à chaque élément de l'arbre binaire, une et une seule fois en
    préordre. Action vide si l'arbre est vide. }
ParcoursPréordre(X) :
    si X ≠ Nil alors
        Traiter(X↑.R) ; ParcoursPréordre(X↑.G) ; ParcoursPréordre(X↑.D)
{ Appel pour traiter un arbre binaire d'adresse A : InitTraitement ; ParcoursPréordre(A) }
Existence d'un élément vérifiant P
P : fonction (E : Élément) ⟶ booléen                        { caractérise la propriété cherchée }
ExisteP : fonction (X : AdNoeud) ⟶ booléen
    { vrai ⟺ il existe dans A au moins un élément vérifiant P }
ExisteP(X) :
    retour : X ≠ Nil et puis (P(X↑.R) ou ExisteP(X↑.G) ou ExisteP(X↑.D))
Adresse du premier élément vérifiant une propriété P (ordre symétrique)
PremierSymSelonP : fonction (X : AdNoeud) ⟶ AdNoeud
    { Posons Y = PremierSymSelonP(X). S'il existe un élément de X vérifiant P, Y est l'adresse
    du premier élément en ordre symétrique de X, qui vérifie P. Sinon Y = Nil. }
PremierSymSelonP(X) :
    retour : si X = Nil alors Nil
        sinon soit Y = PremierSymSelonP(X↑.G)
            dans si Y ≠ Nil alors Y
                sinon si P(X↑.R) alors X sinon PremierSymSelonP(X↑.D)
```

Figure 8.1 – Schémas de traitement séquentiel d'un arbre binaire

b) Arbres n-aires et forêts

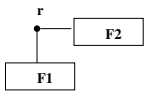
b.1. Définition récursive

Un arbre n-aire non vide est formé de sa racine et de la forêt de ses sous-arbres :
— **<r, F>** dénote un arbre n-aire non vide formé d'une racine **r** et d'une forêt **F** de sous-arbres (éventuellement vide) : **F** est la *forêt des sous-arbres fils* de **r**.



Une forêt est une séquence éventuellement vide d'arbres n-aires **non vides**.

— <**r, F1**> ∘**F2** dénote une forêt formée d'un premier arbre et d'une *forêt d'arbres frères* **F2**, le premier arbre étant formé d'une racine **r** et d'une forêt de sous-arbres fils **F1**.
— **[]** dénote la forêt vide.



b.2. Principe d'analyse récurrente

Le traitement d'un arbre n-aire est ramené au traitement de sa racine et au traitement de chacun de ses sous-arbres. La réalisation d'une fonction ou d'une action portant sur un arbre peut être conçue de deux manières :

— sans intermédiaire : les sous-arbres sont traités dans une itération de parcours de la liste des sous-arbres (un appel récursif par sous-arbre).
— en introduisant une fonction ou une action généralisant le problème posé à une forêt.

Le traitement d'une forêt est ramené au traitement de la racine de son premier arbre et aux traitements de ses forêts de fils et de frères. La réalisation d'une fonction ou d'une action portant sur une forêt comporte ainsi deux appels récursifs, l'un pour la forêt des fils et l'autre pour la forêt des frères.

b.3. Principe de la représentation chaînée des arbres n-aires et des forêts

L'ensemble des fils d'un élément de l'arbre est représenté par une liste chaînée, appelée *liste des frères* : le premier élément de cette liste est le *fils ainé* de l'élément considéré. Le successeur d'un élément dans sa liste de frères (s'il existe) est le *frère cadet* de cet élément. De plus, à tout élément de l'arbre est associée l'adresse de son fils ainé. Ainsi, tout élément d'un arbre appartient à une liste de fils aînés (profondeur) et à une liste de frères (largeur). Une forêt est représentée par une liste chaînée d'arbres formant une liste de frères.
À chaque élément de l'arbre est associée une adresse de nœud comportant un triplet <étiquette du nœud, lien vers le fils ainé ou Nil, lien vers le frère cadet ou Nil> :

```
Élément : type                                              { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud                            { adresses des nœuds }
Noeud : type <R : Élément, Fils : adNoeud, Frère : adNoeud >
    { Étiquette de la racine et adresses (éventuellement Nil) du fils aîné et du frère cadet. }
{ Un arbre n-aire est accessible par l'adresse de sa racine. Une forêt est accessible par l'adresse de la
racine de son premier arbre. Nil caractérise l'arbre vide ou la forêt vide. }
```

b.4. Schémas récursifs de traitement séquentiel d'un arbre n-aire

On raisonne sur une séquence des éléments de l'arbre. L'ordre des éléments dans cette séquence est choisi en fonction du problème parmi l'un des deux ordres suivants (dits *en profondeur d'abord*) : préfixé (racine puis sous-arbres) ou postfixé (sous-arbres puis racine). Dans la Figure 8.2, on ne donne que le schéma de parcours préfixé : pour le parcours postfixé, il suffit de déplacer l'appel Traiter(X↑.R) après l'itération de parcours des sous-arbres.


```
Élément : type                                              { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud                            { adresses des nœuds }
Noeud : type <R : Élément, Fils : adNoeud, Frère : adNoeud >
Parcours préfixé d'un arbre n-aire
ParcoursPréfixéArbre : action (donnée X : AdNoeud)
    { Application de l'action Traiter à chaque élément de l'arbre, une et une seule fois en ordre
    préfixé. Pré-condition : arbre non vide }
ParcoursPréfixéArbre(X) : { précondition : X ≠ Nil }
    AC : adNoeud                                            { pour parcourir la liste des sous-arbres }
    Traiter(X↑.R)
    AC ⟵ X↑.Fils
    tant que AC ≠ Nil
        ParcoursPréfixéArbre(AC) ; AC ⟵ AC↑.Frère
{ Appel pour traiter un arbre d'adresse A : InitTraitement ; si A ≠ Nil alors ParcoursPréfixéArbre(A)
}
Existence dans un arbre n-aire d'un élément vérifiant P
P : fonction (E : Élément) ⟶ booléen                        { caractérise la propriété cherchée }
ExisteParbre : fonction (X : AdNoeud) ⟶ booléen
    { vrai ⟺ l'arbre d'adresse X a un élément vérifiant P. }
ExisteParbre(X) :
    T : booléen ; AC : adNoeud
    si X = Nil alors T ⟵ faux
    sinon si P(X↑.R) alors T ⟵ vrai
            sinon
                AC ⟵ X↑.Fils
                tant que AC ≠ Nil et puis non ExisteParbre(AC) : AC ⟵ AC↑.Frère
                T ⟵ AC ≠ Nil
    retour : T
Adresse du premier élément vérifiant une propriété P (ordre préfixé)
PremierPréfixéArbreSelonP : fonction (X : AdNoeud) ⟶ AdNoeud
    { Posons Y = PremierPréfixéArbreSelonP(X). S'il existe un élément de X vérifiant P, Y est
    l'adresse du premier élément en ordre préfixé de X qui vérifie P. Sinon Y = Nil. Pré-condition :
    arbre non vide. }
PremierPréfixéArbreSelonP(X) : { X ≠ Nil }
    Y : adNoeud ; AC : adNoeud
    si P(X↑.R) alors Y ⟵ X
    sinon Y ⟵ Nil ; AC ⟵ X↑.Fils
            tant que AC ≠ Nil et Y = Nil
                Y ⟵ PremierPréfixéArbreSelonP(AC) ; AC ⟵ AC↑.Frère
    retour : Y
```

Figure 8.2 – Schémas de traitement séquentiel d'un arbre n-aire

b.5. Schémas récursifs de traitement séquentiel d'une forêt

On raisonne sur une séquence des éléments de la forêt en ordre *préfixé* (premier arbre en ordre préfixé puis forêt des arbres frères en ordre préfixé) ou en ordre *postfixé* (premier arbre en ordre postfixé puis forêt des arbres frères en ordre postfixé). Dans la Figure 8.3, on ne donne que le schéma de parcours préfixé : pour le parcours postfixé, il suffit de changer les noms des appels récursifs et de placer le traitement de la racine après le parcours du premier arbre.
On peut décrire le parcours d'un arbre n-aire en termes d'un parcours de forêt. Par exemple le parcours préfixé d'un arbre n-aire non vide d'adresse **A** s'écrit :

```
InitTraitement ; Trater(AC↑.R) ; ParcoursPréfixéForêt(AC↑.Fils)
```

Parcours préfixé d'une forêt

```
ParcoursPréfixéForêt : action (donnée X : AdNoeud)
    { Application de l'action Traiter à chaque élément de la forêt, une et une seule fois en ordre
    préfixé ; action vide si la forêt est vide. }
ParcoursPréfixéForêt(X) :
    si X ≠ Nil alors
        Traiter(X↑.R) ; ParcoursPréfixéForêt(X↑.Fils)                    { 1er arbre }
        ParcoursPréfixéForêt(Frère de X↑)
{ Appel pour traiter une forêt d'adresse A : InitTraitement ; ParcoursPréfixéForêt(A) }
```

Existence dans une forêt d'un élément vérifiant P

```
P : fonction (E : Élément) ⟶ booléen                    { caractérise la propriété cherchée }
ExistePforêt : fonction (X : AdNoeud) ⟶ booléen
    { vrai ⟺ la forêt d'adresse X a un élément vérifiant P. }
ExistePforêt(X) :
    retour : X ≠ Nil et puis
                ( P(X↑.R) ou ExisteP(X↑.Fils) ou ExisteP(X↑.Frère) )
```

Adresse du premier élément d'une forêt vérifiant une propriété P (ordre préfixé)

```
PremierPréfixéForêtSelonP : fonction (X : AdNoeud) ⟶ AdNoeud
    { Posons Y = PremierPréfixéForêtSelonP(X). S'il existe un élément de X vérifiant P, Y est
    l'adresse du premier élément de X (en préfixé) qui vérifie P. Sinon Y = Nil. }
PremierPréfixéForêtSelonP(X) :
    retour : si X = Nil ou alors P(X↑.R) alors X
            sinon soit Y = PremierPréfixéForêtSelonP(X↑.Fils)
                  dans si Y ≠ Nil alors Y sinon PremierPréfixéForêtSelonP(X↑.Frère)
```

Figure 8.3 – Schémas de traitement séquentiel d'une forêt

B. Exemples d'exercices rédigés

a) Arbres binaires sous forme chaînée

```
Élément : type                                          { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud
Noeud : type <R : Élément ; G, D : adNoeud>
```


Exemple E8.1 : Création d'un arbre binaire

Construire une copie d'un arbre binaire. Les arbres sont représentés sous forme chaînée.

```
A : adNoeud                                             { donnée : adresse de l'arbre considéré }
CopieDeA : adNoeud                                      { résultat : adresse de l'arbre construit }
CopierArbreBin : action (donnée X : adNoeud ; résultat Y : adNoeud)
    { Construit une copie de l'arbre d'adresse X et fournit son adresse dans Y. }
{ Recopie de l'arbre donné }
    CopierArbreBin(A, CopieDeA)

{ Réalisation de CopierArbreBin }
    { Recopier un arbre, c'est construire le nœud racine du nouvel arbre, puis recopier les sous-arbres
    (préordre). }
    CopierArbreBin(X, Y) :
        si X = Nil alors Y ⟵ Nil
        sinon Allouer(Y) ; Y↑.R ⟵ X↑.R
              CopierArbreBin(X↑.G, Y↑.G) ; CopierArbreBin(X↑.D, Y↑.D)
```

Exemple E8.2 : Libération d'un arbre binaire

Restituer à la mémoire libre tous les nœuds d'un arbre binaire.

```
A : adNoeud                                             { donnée : adresse de l'arbre considéré }
LibérerArbreBin : action (donnée-résultat X : adNoeud)
    { Restitue à la mémoire libre tous les nœuds de l'arbre d'adresse X. A l'état final, X=Nil. }

{ Libération de l'arbre donné }
    LibérerArbreBin(A)

{ Réalisation de LibérerArbreBin }
    { Libérer un arbre, c'est libérer les sous-arbres de la racine, puis libérer la racine (ordre terminal). }
    LibérerArbreBin(X) :
        si X ≠ Nil alors
            LibérerArbreBin(X↑.G) ; LibérerArbreBin(X↑.D)
            Libérer(X) ; X ⟵ Nil
```

b) Arbres n-aires et forêts

```
Élément : type                                          { type des éléments de l'arbre }
adNoeud : type pointeur de Noeud
Noeud : type <R : Élément ; Fils, Frère : adNoeud>
```

Exemple E8.3 : Nombre d'éléments positifs

Déterminer le nombre d'éléments strictement positifs d'un arbre n-aire d'entiers.

```
Élément : type entier
NbPosArbre : fonction (A : adNoeud) ⟶ entier ≥ 0
    { Nombre d'éléments > 0 de l'arbre d'adresse A. }
```

On illustre dans cet exemple diverses manières d'aborder un problème de traitement d'arbre n-aire.

(i) Approche 1 : analyse récurrente
Le nombre d'éléments positifs d'un arbre non vide est le nombre des éléments positifs de chacun des sous-arbres fils, plus 1 si la racine est positive.

Version 1 : solution directe (itération sur les sous-arbres)

```
NbPosArbre(A) :
   N : entier ≥ 0                                          { pour construire le résultat }
   AC : adNoeud                                            { pour parcourir la liste des sous-arbres }
   si A = Nil alors N ⟵ 0
   sinon N ⟵ (si A↑.R > 0 alors 1 sinon 0) ; AC ⟵ A↑.Fils
         tant que AC ≠ Nil
            N ⟵ N + NbPosArbre(AC)
            AC ⟵ AC↑.Frère
   retour : N
```

Remarque : le cas **A = Nil** n'est pertinent que pour l'appel initial. En effet, dans le corps de l'itération on sait que **AC ≠ Nil**. Une alternative est de spécifier **NbPosArbre(A)** avec **A ≠ Nil** pour pré-condition, le cas **A= Nil** étant traité lors de l'appel initial.
Cette remarque est valable pour tout problème d'arbre traité selon cette approche.

Version 2 : généralisation du problème à une forêt
On spécifie le problème pour une forêt :

```
NbPosForêt : fonction (F : adNoeud) ⟶ entier ≥ 0
   { Nombre d'éléments strictement positifs de la forêt d'adresse F. }
{ Réalisation de NbPosArbre. }
   NbPosArbre(A) :
      retour : si A = Nil alors 0 sinon (si A↑.R > 0 alors 1 sinon 0) + NbPosForêt(A↑.Fils)
{ Réalisation de NbPosForêt }
   NbPosForêt(F) :
      retour : si F = Nil alors 0
            sinon (si F↑.R > 0 alors 1 sinon 0) + NbPosForêt(F↑.Fils)        { 1er arbre }
                  + NbPosForêt(F↑.Frère))
```

Remarques : si l'on sait a priori que $A \neq Nil \Rightarrow A\uparrow.Frère = Nil$, on peut écrire :

```
NbPosArbre(A) :
   retour : NbPosForêt(A)
```

Par ailleurs, **NbPosForêt** peut aussi être réalisée sous forme d'une itération de parcours de (la séquence des arbres de) la forêt dans laquelle chaque arbre est traité par **NbPosArbre** (récursivité "croisée").
Ces remarques sont valables pour tout problème d'arbre traité selon cette approche.

(ii) Approche 2 : traitement séquentiel, application d'un schéma de parcours
Il s'agit d'incrémenter un compteur (initialisé à 0) pour chaque élément positif de l'arbre. Il suffit donc d'appliquer un schéma de parcours d'un arbre ou d'une forêt.

Version 1 : parcours préfixé de l'arbre

```
Compteur : entier ≥ 0
AjouterNbPosArbre : action (donnée X : adNoeud)
   { Ajoute à Compteur le nombre d'éléments strictement positifs de l'arbre d'adresse X. Pré-condition :
   X ≠ Nil. }
```



```
{ Réalisation de NbPosArbre. }
   NbPosArbre(A) :
      Compteur ⟵ 0                                         { InitTraitement }
      si A ≠ Nil alors AjouterNbPosArbre(A)
      retour : Compteur
{ Réalisation de AjouterNbPosArbre : application du schéma de parcours préfixé d'un arbre. }
   AjouterNbPosArbre(X) : { X ≠ Nil }
      AC : adNoeud                                         { pour parcourir la liste des sous-arbres }
      si X↑.R > 0 alors Compteur ⟵ Compteur +1             { Traiter }
      AC ⟵ X↑.Fils
      tant que AC ≠ Nil
         AjouterNbPosArbre(AC) ; AC ⟵ AC↑.Frère
```

Version 2 : parcours préfixé de la forêt des sous-arbres

```
Compteur : entier ≥ 0
AjouterNbPosForêt : action (donnée X : adNoeud)
   { AjouterNbPosForêt(X) ajoute à Compteur le nombre d'éléments strictement positifs de la forêt
   d'adresse X. }
{ Réalisation de NbPosArbre. }
   NbPosArbre(A) :
      Compteur ⟵ 0                                         { InitTraitement }
      si A ≠ Nil alors
         si A↑.R > 0 alors Compteur ⟵ 1
         AjouterNbPosForêt(A↑.Fils)
      retour : Compteur
{ Réalisation de AjouterNbPosForêt : application du schéma de parcours préfixé d'une forêt. }
   AjouterNbPosForêt(X) :
      si X ≠ Nil alors
         si X↑.R > 0 alors Compteur ⟵ Compteur +1          { Traiter }
         AjouterNbPosForêt(X↑.Fils) ; AjouterNbPosForêt(X↑.Frère)
```

Exemple E8.4 : Nombre de nœuds de niveau n dans un arbre n-aire

Déterminer le nombre de nœuds de niveau n dans un arbre n-aire non vide.

```
NbNivArbre : fonction (A : adNoeud, n : entier > 0) ⟶ entier ≥ 0
   { Nombre de nœuds de niveau n dans l'arbre d'adresse A.
   Pré-condition : A ≠ Nil }
```

La racine est le seul nœud de niveau 1. Les nœuds de niveau **n+1** dans un arbre **A** sont ceux de niveau **n** dans les sous-arbres de **A**.
Version 1 : solution directe

```
NbNivArbre(A, n) : { A ≠ Nil }
   Nb : entier ≥ 0                                         { pour construire le résultat }
   AC : adNoeud                                            { pour parcourir la liste des sous-arbres }
   si n = 1 alors Nb ⟵ 1
   sinon Nb ⟵ 0 ; AC ⟵ AC↑.Fils
         tant que AC ≠ Nil
            Nb ⟵ Nb + NbNivArbre(AC, n-1) ; AC ⟵ AC↑.Frère
   retour : Nb
```

Version 2 : généralisation à une forêt

```
NbNivForêt : fonction (F : adNoeud, n : entier > 0)  ⟶ entier ≥ 0
    { Nombre de nœuds de niveau n dans la forêt d'adresse F. }
{ réalisation de NbNivArbre }
    NbNivArbre(A, n) :
        retour : si n = 1 alors 1 sinon NbNivForêt(AC↑.Fils, n-1)
    { réalisation de NbNivForêt }
        NbNivForêt(F, n) :
            retour : si F = Nil alors 0
                  sinon (si n = 1 alors 1 sinon NbNivForêt(F↑.Fils, n-1))   { 1er arbre }
                        + NbNivForêt(F↑.Frère, n)                           { forêt des frères }
```

c) Arbres n-aires, forêts et séquences sous forme chaînée

Exemple E8.5 : Liste préfixée des éléments d'une forêt

Construire la liste préfixée des éléments d'une forêt. La forêt et la liste sont sous forme chaînée.

```
Élément : type
adCel : type pointeur de Cel ; Cel : type <Elt : Élément, Suc : adCel>
adNoeud : type pointeur de Noeud
Noeud : type <R : Élément ; Fils, Frère : adNoeud>
CréerLPréForêt : action (donnée F : adNoeud, résultat L : adCel)
    { Construit la liste préfixée des éléments de la forêt d'adresse F et fournit son adresse de tête dans
    L. }
```

Version 1 : analyse récurrente

Si la forêt est vide, la liste est vide. Sinon elle comporte la racine du premier arbre, puis la liste préfixée de la forêt des sous-arbres du premier arbre, puis la liste préfixée de la forêt des frères.

```
Der : fonction (T : adCel)  ⟶ adCel
    { Adresse du dernier élément de la liste d'adresse T. Pré-condition : T ≠ Nil. }

{ Réalisation de CréerLPforêt }
    CréerLPréForêt(F, L) :
        si F = Nil alors L  ⟵ Nil
        sinon Allouer(L) ; L↑.Elt  ⟵ F↑.R
              CréerLPréForêt(F↑.Fils, L↑.Suc)
              CréerLPréForêt(F↑.Frère, Der(L)↑.Suc)
```

Version 2 : traitement séquentiel

Dans un parcours préfixé de la forêt, on ajoute l'élément courant en queue de la liste en cours de création. La liste est initialisée par un élément fictif de tête pour éviter le cas particulier du premier élément. Ce fictif est libéré en fin de création.

```
ConcaténerLPréForêt : action (donnée X : adNoeud, donnée-résultat Y : adCel)
    { Concatène la liste préfixée des éléments de la forêt d'adresse X, à une liste non vide dont le dernier
    élément a Y pour adresse. A l'état final, Y est l'adresse du dernier élément de la liste ainsi modifiée.
    }

{ Réalisation de CréerLPréForêt }
    CréerLPréForêt(F, L) :
        Q : adCel                          { adresse du dernier élément de la liste en cours de création }
        Fic : adCel                                                  { adresse de l'élément fictif }
```



```
        Allouer(Fic) ; Q  ⟵ Fic                          { InitTraitement : création du fictif de tête }
        ConcaténerLPréForêt(F, Q)
        Q↑.Suc  ⟵ Nil ; L  ⟵ Fic↑.Suc
        Libérer(Fic)                                      { libération du fictif de tête }

{ Réalisation de ConcaténerLPréForêt : parcours préfixé de la forêt }
    ConcaténerLPréForêt(X, Y) :
        Z : adCel                                         { pour construire un nouvel élément }
        si X ≠ Nil alors
            Allouer(Z) ; Z↑.Elt  ⟵ X↑.R ; Y↑.Suc  ⟵ Z ; Y  ⟵ Z          { ajout en queue }
            ConcaténerLPréForêt(X↑.Fils, Y) ; ConcaténerLPréForêt(X↑.Frère, Y)
```

C. Exercices

a) Arbres binaires

E8.6 : Parcours d'arbres binaires

(i) Expérimentation sur un exemple :

Dessiner un arbre binaire de 8 entiers et les listes de ses éléments, en préordre, en ordre symétrique et en ordre terminal.

(ii) Caractérisation d'un arbre binaire

- Deux arbres binaires différents peuvent avoir la même liste en préordre : donner un exemple, le plus simple possible, illustrant cette propriété. Sur cet exemple, vérifier que les listes en ordre symétrique sont différentes.
- Donner un principe récursif de construction d'un arbre binaire à partir des listes de ses éléments en préordre et en ordre symétrique.
- Examiner les deux autres combinaisons : préordre, terminal ; symétrique, terminal.

E8.7 : Arbres symétriques (E6.12 suite)

Le *symétrique* d'un arbre binaire **A** est un arbre binaire déduit de **A** en faisant correspondre un sous-arbre gauche à tout sous-arbre droit et vice versa (**B** est l'image miroir de **A**). Les arbres sont représentés sous forme chaînée.

(i) Construction du symétrique

- Spécifier et réaliser une action nommée **CréerSymétrique** qui construit le symétrique d'un arbre binaire.

(ii) Remplacement par le symétrique

- Spécifier et réaliser une action nommée **RemplacerParSymétrique** qui modifie un arbre binaire de telle sorte que sa valeur à l'état final soit le symétrique de sa valeur à l'état initial.

E8.8 : Suppression des feuilles d'un arbre binaire

On étudie la suppression des feuilles d'un arbre dans diverses conditions de réalisation. Par exemple, dans la **Figure 8.4**, l'arbre **B** est obtenu en supprimant les feuilles de l'arbre **A**.

(i) Etude fonctionnelle abstraite

On fait ici abstraction de la représentation des arbres en utilisant les opérations sur les valeurs de type arbre binaire (/\, /x, y, z\, EstVide ?, Racine, Gauche, Droit).

On spécifie une fonction nommée **SansF** :

```
SansF : fonction (A : arbre binaire d'entier)  ⟶ arbre binaire d'entier
    { Arbre obtenu à partir de A en supprimant ses feuilles. SansF(/\) = /\ }
```

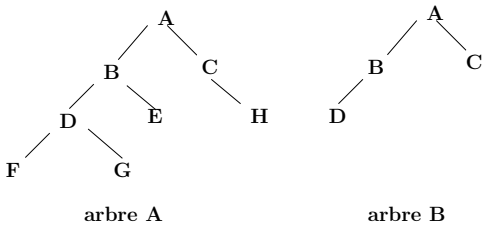


Figure 8.4 – suppression des feuilles d'un arbre

— Donner des équations de récurrence définissant la fonction **SansF**.
— Donner une réalisation récursive de la fonction **SansF**.

(ii) Réalisation pour une représentation chaînée
On spécifie l'action suivante :

adNoeud : type pointeur de Noeud ; Noeud : type <R : Élément ; G, D : adNoeud >
CopierSansF : action (donnée A : AdNoeud, résultat B : AdNoeud)
{ Construit un arbre dont la valeur est celle de A sans les feuilles, et fournit son adresse dans B. }

— Donner une réalisation de l'action **CopierSansF** (sur la base des équations définissant **SansF**).

(iii) Extraction des feuilles
On veut modifier un arbre binaire **A** en supprimant ses feuilles. De plus, on veut construire une liste **LF** formée des éléments situés sur les feuilles enlevées à l'arbre A. (**Figure 8.5**).

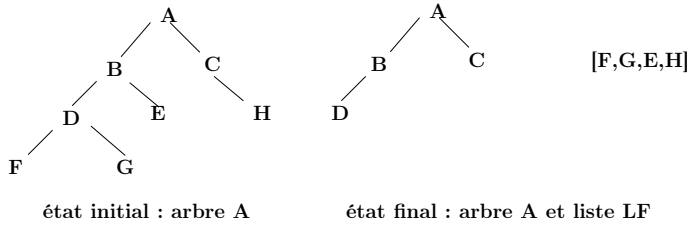


Figure 8.5 – extraction des feuilles d'un arbre

Étude fonctionnelle
On spécifie une fonction nommée **SansFetLF** :

SansFetLF : fonction (A : arbre binaire d'entier) ⟶ <arbre binaire d'entier, séquence d'entier>
{ posons <A,L> = SansFetLF(A) : A= SansF(A) et L est la séquence des feuilles de A. }

— Compléter les équations suivantes :
(1) SansFetLF(/\) = ••••••••••••

(2) SansFetLF(/G,r,D\) =
si EstVide ?(G) et EstVide ?(D) alors ••••••••••••
sinon soit <Ag, Lg> = SansFetLF(G), <Ad, Ld> = SansFetLF(D)
dans ••••••••••••

Version 1
L'arbre **A** et la liste **LF** des feuilles sont sous forme chaînée standard. Aucun ordre n'est imposé entre les éléments de **LF**. Les cellules feuilles de l'arbre initial doivent être restituées à la mémoire.
On spécifie l'action suivante :

AdCel : type pointeur de Cel ; Cel : type <E : entier ; Suc : AdCel>
ExtraireF : action (donnée-résultat A : adNoeud, résultat L : AdCel)
{ Supprime les feuilles de l'arbre donné de racine A et construit la liste des entiers situés sur ces feuilles. L'adresse de la liste résultat est fournie dans L. }

— Donner une réalisation de l'action **ExtraireF** (sur la base des équations définissant **SansFetLF**).

Version 2
On spécifie l'action suivante :

ExtraireEtConcatF : action (donnée-résultat A : AdNoeud, L : AdCel)
{ Supprime les feuilles de l'arbre d'adresse A et complète la liste de tête L par les entiers situés sur ces feuilles. Si l'arbre est vide, la liste L est inchangée. }

— Donner une réalisation de l'action **ExtraireEtConcatF**.
— Donner une réalisation de l'action **ExtraireF** en utilisant **ExtraireEtConcatF**.

b) Arbres n-aires

E8.9 : Parcours d'arbres n-aires

— Dessiner un arbre n-aire **A** de 12 entiers et les listes de ses éléments en ordre préfixé et postfixé.

Une interprétation de la représentation chaînée d'un arbre n-aire
— Dessiner la représentation chaînée **A'** de l'arbre **A** choisi ci-dessus.
— Dessiner un arbre binaire **B** déduit de **A'** de la manière suivante : à tout nœud de **A'** correspond un nœud de **B** ayant même valeur d'élément. A tout lien vers un fils aîné dans **A'** correspond un lien vers un fils gauche dans **B** ; et à tout lien vers un frère cadet dans **A'** correspond un lien vers un fils droit dans **B**.
— Donner les listes en préordre, ordre symétrique et ordre terminal de l'arbre **B** (parcours d'arbre binaire) et les comparer avec les listes obtenues au premier point (parcours d'arbre n-aire).
— Comparer les algorithmes de parcours portant sur les arbres binaires et sur les arbres n-aires.

E8.10 : Degré maximum des nœuds d'un arbre ou d'une forêt
Le *degré* d'un nœud est le nombre de ses fils.
Les arbres n-aires et les forêts sont sous forme chaînée standard.

(i) Degré maximum d'un arbre
On spécifie une fonction nommée **DegréMaxArbre** :

DegréMaxArbre : fonction (X : AdNoeud) ⟶ entier ≥ 0
{ Degré maximum des nœuds de l'arbre d'adresse X. Pré-condition : X ≠ Nil }

— Donner une réalisation récursive directe de la fonction **DegréMaxArbre**.
— On veut afficher le degré maximum du i^{ème} arbre (s'il existe) d'une forêt. Spécifier une action pour cela et la réaliser en utilisant la fonction **DegréMaxArbre**.

(ii) Degré maximum d'une forêt
On spécifie une fonction nommée **DegréMaxForêt** :

DegréMaxForêt : fonction (X : AdNoeud) ⟶ entier
{ Degré maximum des nœuds de la forêt d'adresse X. Si la forêt est vide, le résultat a la valeur 1. }

— Donner deux réalisations récursives de la fonction **DegréMaxForêt** : la première en utilisant la fonction **DegréMaxArbre** et la seconde sans l'utiliser.

E8.11 : Nombre d'exemplaires de la valeur maximum dans une forêt
On considère des forêts d'entiers strictement positifs sous forme chaînée standard :

AdNoeud : type pointeur de Noeud
Noeud : type <R : entier > 0 ; Fils, Frère : AdNoeud>

On spécifie une action nommée **AfficherNbValMaxF** :

AfficherNbValMaxF : action (donnée F : AdNoeud)
{ Affiche le nombre d'exemplaires de la valeur maximum dans la forêt d'adresse F. Si la forêt est vide, la valeur 0 est affichée. }

(i) Version 1 : analyse récurrente portant sur une forêt
On spécifie une fonction nommée **MaxEtNbF** :

MaxEtNbF : fonction (X : AdNoeud) ⟶ entier ≥ 0, entier ≥ 0
{ Posons <M, N> = MaxEtNbF(X). M est la valeur maximum dans la forêt d'adresse X et N est le nombre d'exemplaires de M dans cette forêt. Si X=Nil, M et N ont la valeur 0. }

— Donner une réalisation de l'action **AfficherNbValMaxF**.
— Compléter la réalisation récursive suivante de la fonction **MaxEtNbF** :

```
MaxEtNbForêt(X) :
    retour : si X = Nil alors <0, 0>
             sinon soit V = X↑.R, <M1, N1> = MaxEtNbF (X↑.Fils),
                        <M2, N2> = MaxEtNbF (X↑.Frère)
                    dans ●●●●●●●●●●●●
```

(ii) Version 2 : application d'un traitement séquentiel
On raisonne sur une liste des éléments de la forêt. Pour réaliser l'action **AfficherNbValMaxF**, on définit le lexique global suivant :

ValMax : entier ≥ 0 *{ valeur maximum courante }*
NbValMax : entier ≥ 0 *{ nombre d'exemplaires courant de la valeur maximum courante }*

et on spécifie une action nommée **MajMaxEtNb** :

MajMaxEtNbF : action (donnée X : AdNoeud)
{ Met à jour les valeurs de ValMax et NbValMax en tenant compte de leur valeur initiale et des valeurs de la forêt d'adresse X. Si X=Nil, l'action est vide. }

— Donner une réalisation récursive de l'action **MajMaxEtNbF**.
— Donner une réalisation de l'action **AfficherNbValMaxF**.

D. Problème dirigé : à propos d'arbres n-aires

a) Présentation

On considère un ensemble **non vide** d'éléments **distincts deux à deux** structuré en arbre n-aire. Il s'agit par exemple d'un ensemble de dossiers emboîtés organisé selon la relation "*être un sous-dossier de*".

L'arbre considéré est appelé l'*arbre principal* de manière à le distinguer des sous-arbres qu'il contient. Sa racine est appelée l'*ancêtre*.

Chaque nœud de l'arbre est étiqueté par une valeur d'un type **Élément**. A chaque élément est associée une date (par exemple sa date de création dans le contexte considéré). Cette date fixe deux propriétés de l'arbre :
— tout élément de l'arbre a une date antérieure à celle de ses fils ;
— la liste des fils d'un élément est en ordre chronologique selon les dates : le premier fils a la plus ancienne et le dernier fils a la plus récente. Si deux frères ont la même date, ils apparaissent dans un ordre quelconque dans cette liste.

L'arbre est représenté sous forme chaînée, comme l'illustre la **Figure 8.6** : les lettres figurent des éléments ; les flèches verticales figurent une relation père ⟶ fils aîné ; et les flèches horizontales figurent une relation frère ⟶ frère cadet.

Sur l'exemple de la **figure 8.6**, **A** est l'ancêtre. La liste des fils de **E** est **[L,M,N]**. **L** est le fils aîné de **E**. **F** est le frère cadet de **E**. **F** n'a pas de fils. **D** est le plus récent des fils de **A**.

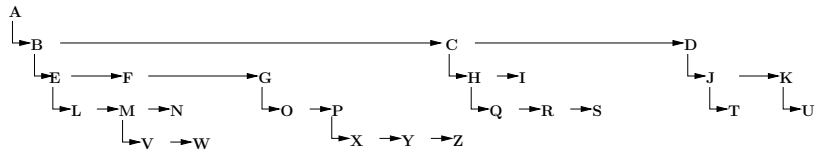


Figure 8.6 – un exemple d'arbre n-aire

On fixe le lexique suivant :

```
{ dates }
    Date : type {  =, ≠, <, >, ≥ dénotent les comparaisons de dates (ordre chronologique). }
{ éléments }
    Élément : type
        { =, ≠ et ⟵ dénotent les comparaisons et l'affectation de variables de type Élément }
DateCréation : fonction (E : Élément) ⟶ Date        { date de création de l'élément donné }
    { Si besoin est, on spécifiera d'autres fonctions sur les éléments, mais sans les réaliser }
{ représentation chaînée des arbres ou des forêts. Leurs éléments sont distincts deux à deux. }
AdNoeud : type pointeur de Noeud
Noeud : type <Elt : Élément ; Fils, Frère : AdNoeud>
AdAncêtre : AdNoeud                                 { adresse de l'arbre principal }
```

b) Questions

Donner une réalisation de chacune des fonctions ou actions suivantes en tenant compte de la représentation. Dans toutes les questions, les arbres ou forêts considérés vérifient les propriétés

énoncées ci-dessus : éléments distincts deux à deux, liste de fils en ordre chronologique.

E8.12 : Adresse d'un élément
(i) Dans un arbre

AdElémentArbre : fonction (E : Élément, X : AdNoeud) $\longrightarrow$ AdNoeud
{ Posons Y = AdÉlémentArbre(E, X) : si E appartient à l'arbre d'adresse X, Y est l'adresse du nœud étiqueté par E, sinon Y = Nil. Pré-conditions : X ≠ Nil ; les éléments de l'arbre sont distincts deux à deux. }

(ii) Dans une forêt

AdÉlémentForêt : fonction (E : Élément, X : AdNoeud) $\longrightarrow$ AdNoeud
{ Posons Y = AdÉlémentForêt(E, X) : si E appartient à la forêt d'adresse X, Y est l'adresse du nœud étiqueté par E, sinon Y = Nil. Pré-condition : les éléments de la forêt sont distincts deux à deux. }

E8.13 : Filiation

EstFils : fonction (E1, E2 : Élément) $\longrightarrow$ booléen
{ vrai $\Longleftrightarrow$ E2 est un fils de E1 dans l'arbre principal. En particulier, EstFils(E1,E2) a la valeur faux si E1 n'appartient pas à l'arbre principal. Dans l'exemple ci-dessus, EstFils(H, R) a la valeur vrai ; EstFils(H, U) a la valeur faux ; EstFils(E, V) a la valeur faux. }

E8.14 : Parenté

EstDesc : fonction (E1, E2 : Élément) $\longrightarrow$ booléen
{ vrai si et seulement si E2 fait partie de la descendance de E1 dans l'arbre principal. En particulier, EstDesc(E1,E2) a la valeur faux si E1 n'appartient pas à l'arbre principal. Dans l'exemple ci-dessus, EstDesc(D, U) a la valeur vrai ; EstDesc(D, J) a la valeur vrai, EstDesc(D, Q) a la valeur faux. }

E8.15 : Dénombrement

NbAvant : fonction (D : Date) $\longrightarrow$ entier ≥ 0
{ Nombre d'éléments de l'arbre principal dont la date de création est strictement antérieure à la date D. }

Étudier deux versions :
— l'une en généralisant le problème à un arbre d'adresse **X**,
— l'autre en le généralisant à une forêt d'adresse **X**. On exploitera le fait que tout élément de l'arbre a une date antérieure à celles de ses descendants.

E8.16 : Ajout d'un élément

Ajouter : action (donnée E1, E2 : Élément)
{ Modifie l'arbre principal de manière à prendre en compte l'ajout d'un élément E2, comme fils d'un élément E1.
Pré-conditions : E1 ≠ E2, E1 appartient déjà à l'arbre principal, E2 ne lui appartient pas et la date de E1 est antérieure à celle de E2. }

On tiendra compte de l'ordre chronologique entre les fils d'un nœud.

E8.17 : Edition de l'arbre
Décrire une action, nommée **ÉditerArbre**, qui imprime l'arbre principal sous forme indentée. Pour l'exemple de la **Figure 8.6**, on obtient l'impression ci-dessous.

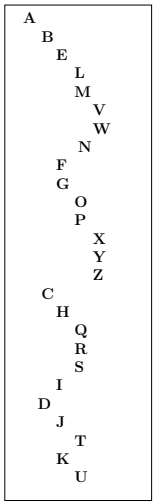


Figure 8.7 – Edition de l'arbre principal

A l'état initial le curseur d'impression est positionné au début de la première ligne d'une nouvelle page. A l'état final, il est positionné au début de la ligne qui suit la dernière ligne imprimée. On supposera que la page est suffisamment grande pour contenir l'arbre (tant en largeur qu'en hauteur).

On dispose des primitives suivantes : **AlaPage** positionne le curseur au début de la première ligne d'une nouvelle page ; **AlaLigne** positionne le curseur au début de la prochaine ligne ; **Marge(n)** écrit **n** espaces à partir de la position courante du curseur ; **EditerÉlément(E)**, a pour effet d'écrire les informations correspondant à l'élément de valeur **E**, à partir de la position courante du curseur.