Cours 4 et 5

Les arbres

---

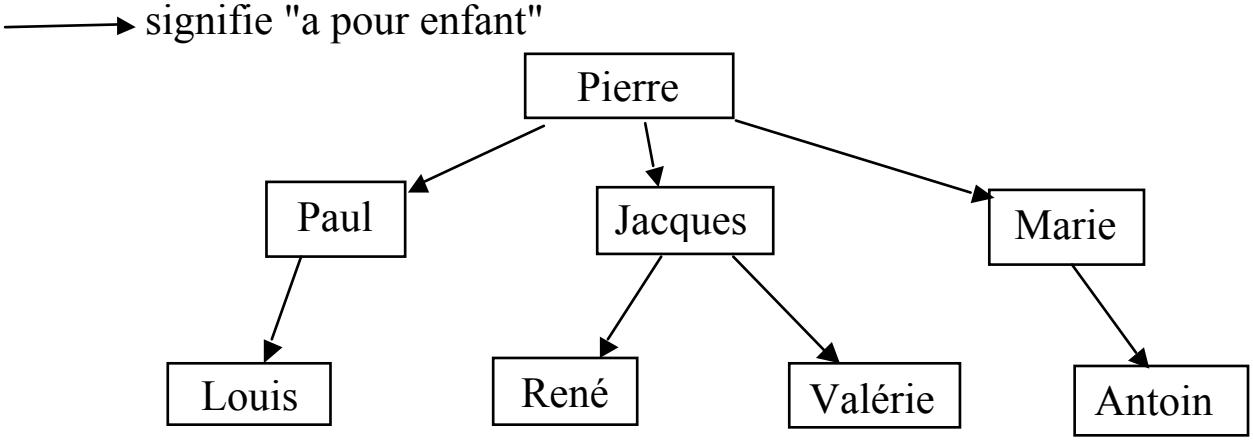
1. Introduction

1.1. Définition

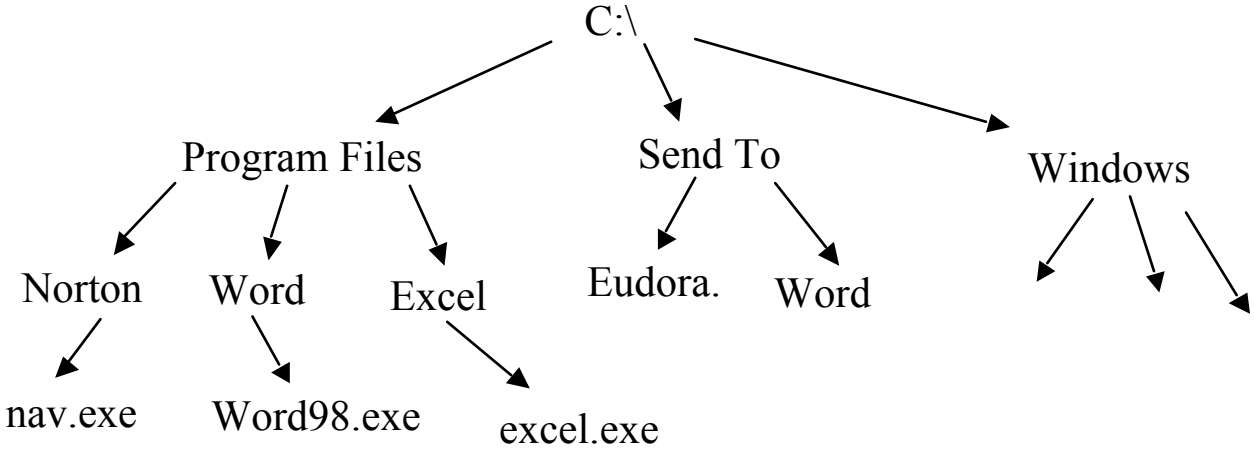
L'arbre est une structure de donnée qui généralise la liste : alors qu'une cellule de liste a un seul successeur (leur suivant), dans un arbre il peut y en avoir plusieurs. On parle alors de **nœud** (au lieu de cellule). Un **nœud père** peut avoir plusieurs **nœud fils**. Un fils n'a qu'un seul père, et tous les nœuds ont un ancêtre commun appelé la **racine** de l'arbre (le seul nœud qui n'a pas de père).

1.2. Premiers exemples

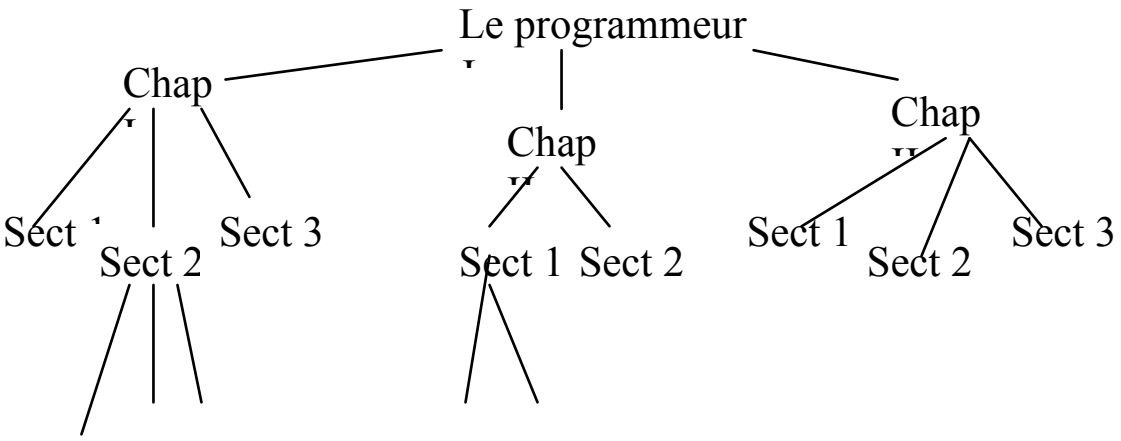
Exemple 1 : arbre généalogique



Exemple 2 : arborescence des répertoires et des fichiers



Exemple 3 : mise en page d'un texte



Une **feuille** est un nœud qui

La profondeur de la racine est 0, celle d'un nœud non racine est …

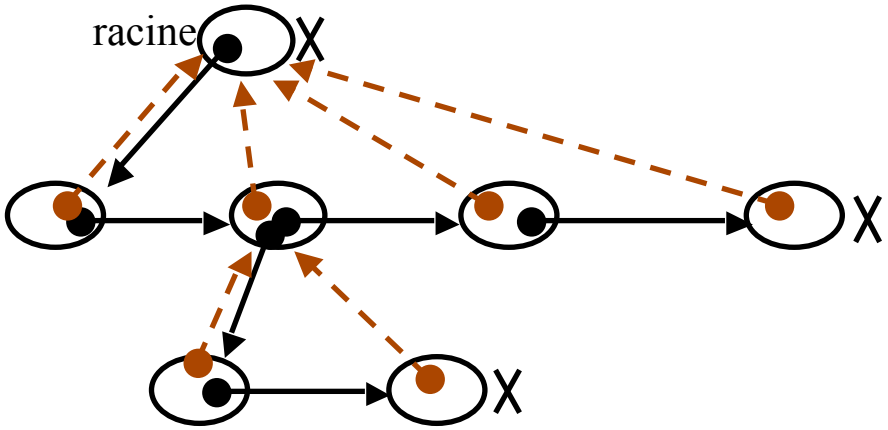
La profondeur de l'arbre est …

1.3. Représentation

On indique ici une représentation par pointeur. Il en existe d'autres, par pointeurs, ou des représentations par tableaux.

```
typedef struct NOEUDELEM {
   Elem valeur ;
   struct NOEUDELEM fils ; // le 1er fils
   struct NOEUDELEM frere ; // le fils suivant du
   même père …
} NoeudElem
```

Un dessin pour aider à comprendre la représentation des arbres par pointeur fils / pointeur frère :



Quand n a deux fils, son fils gauche est n.fils et son fils droit est n.fils.frère
Dans toutes les représentations, on connaît l'arbre quand on connaît sa racine :

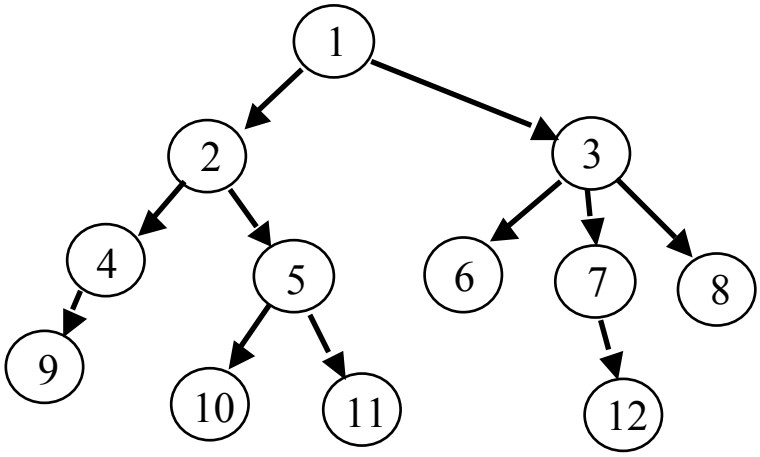
```
typedef struct {
   NoeudElem racine ;
} ArbreElem
```

Les méthodes sont à ajouter.

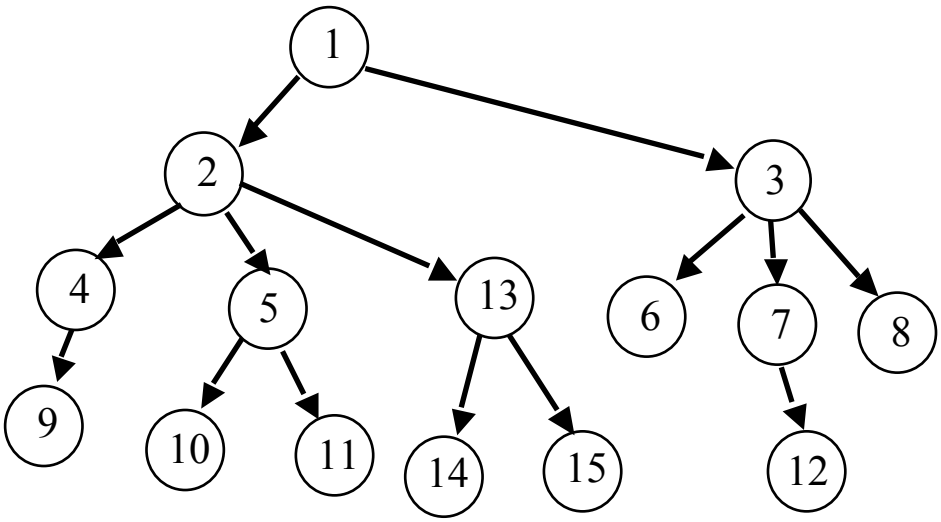
2. Opérations

2.1. Ajout d'un sous arbre

exemple :

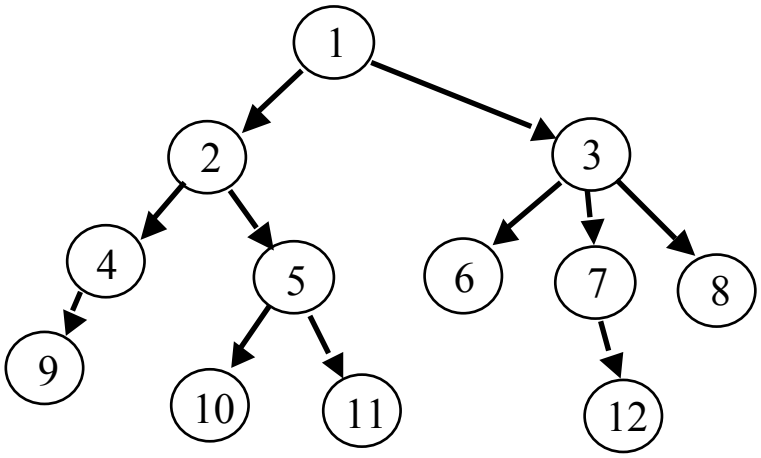


devient :

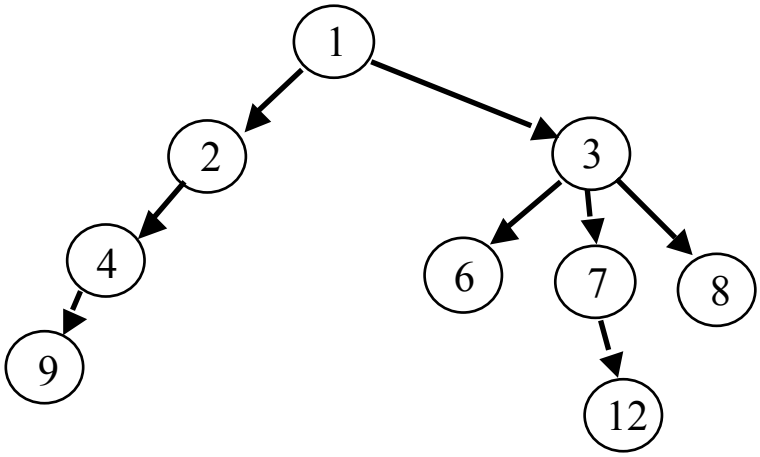


2.2. Suppression d'un sous-arbre

exemple



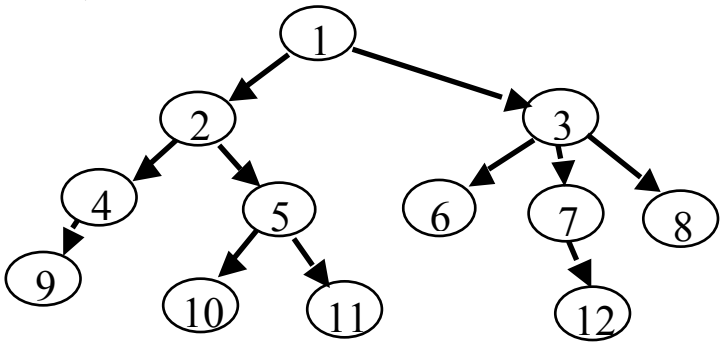
devient :



2.3. Parcours

2.3.1. Parcours en largeur d'abord

(figure 1)



On commence par la racine, puis tous ses fils, puis les fils des fils…

Ici on parcourra les nœuds dans l'ordre :

……

2.3.2. Parcours en profondeur d'abord

On commence par la racine, puis son 1er fils, puis le 1er fils du 1er fils, … Quand on arrive à une feuille, il faut revenir en arrière jusqu'à trouver un fils non encore parcouru.
Ici le parcours en profondeur d'abord donne :

…….

On passe plusieurs fois par le même nœud. Si l'on veut énumérer les nœuds (ne les écrire, ou les traiter, qu'une seule fois), dans le cas général on écrit (ou traite) un nœud **avant** de visiter ses fils, soit ici :

…..

ou après les avoir visité, soit :

…..

2.4. Algorithmes

1.1.1. Algorithme de parcours en largeur

```
/* Parcours en largeur d'un arbre a non vide */
```

paramètres :
NœudElem racine ;

variables :
file f ;
NœudElem courant ;

debut :
stocker(f, racine) ;
courant = NULL;
tant que (non vide(f) OU courant ≠ NULL) faire
si(courant ≠ NULL) faire
stocker(f, courant)
courant = courant.frere
sinon
courant = prélever(f) ;
traiter courant
courant = courant.fils ;
finsi
fin tant que
fin

1.1.2. Algorithme de parcours en profondeur

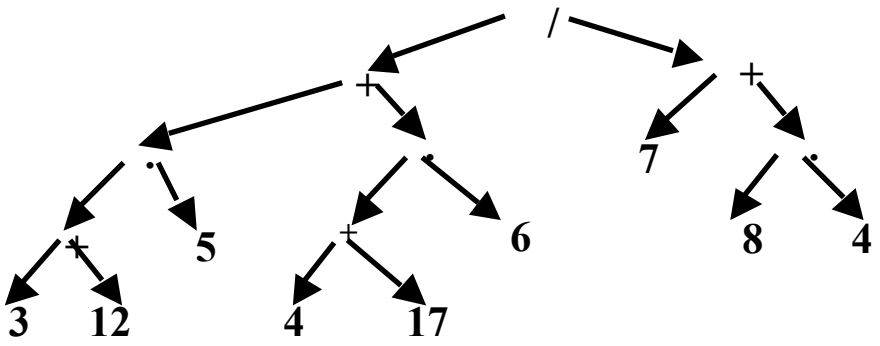
/* Parcours en profondeur d'un arbre a non vide */
paramètres : nœud racine ;
variables : pile p ; nœudElem courant, unFils ;
debut :
courant = racine ;
faire
si (courant ≠ NULL) faire
pousser(p,courant)
/* énumération préfixée : traiter(courant) */
courant = courant.fils
sinon
courant = prendre(p)
/* énumération postfixée : traiter(courant) */
courant = courant.frere
fin si
tant que (non vide(p))
fin

3. Le calcul arithmétique

3.1. Arbres de calcul arithmétique

Définition : un arbre **binaire** est un arbre dont les nœuds ont au plus deux fils

Un arbre de calcul arithmétique est un arbre dont tous les nœuds non feuille ont exactement deux fils, dont les données sur les feuilles sont des nombres et les données sur les nœuds non feuille sont des signes d'opération.



3.2. Enumérations

On utilise dans ce cas plusieurs énumérations en profondeur :

- Énumération **préfixée** : on écrit un nœud **avant** de visiter ses fils. Dans l'exemple on obtient
  / + . + 3 12 5 . + 4 17 6 + 7 . 8 4

On peut reconstituer l'arbre à partir de l'écriture (voir algorithme plus loin).

- Énumération **postfixée** : on écrit ou traite un nœud **après** avoir visité ses fils. Dans l'exemple 4 on obtient
  3 12 + 5 . 4 17 + 6 . + 7 8 4 . + /

On peut reconstituer l'arbre à partir de l'écriture (voir algorithme plus loin).

- Énumération **infixée** : on écrit ou traite un nœud **entre** son 1[er] et son 2[ème] fils. Dans l'exemple 4 on obtient
  3 + 12 . 5 + 4 + 17 . 6 / 7 + 8 . 4

C'est l'écriture classique, mais sous cette forme elle est ambiguë. On ne peut se passer de parenthèses. Quand on ne donne pas de priorité

entre opérations, on met une parenthèse ouvrante en entrant dans un sous-arbre et une parenthèse fermante quand on a fini d'écrire le sous-arbre.

((((3 + 12) . 5) + ((4 + 17) . 6)) / (7 + (8 . 4)))

- Il n'y a qu'une seule énumération en largeur, celle du cas général. Dans l'exemple on obtient

  / + + . . 7. + 5 + 6 8 4 3 12 4 17

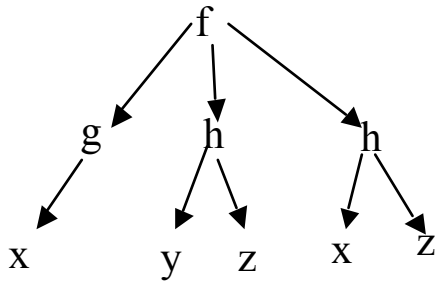
Dans ce cas aussi on peut reconstituer l'arbre…

3.3. Arbres de calcul généraux

Une opération arithmétique est un cas particulier de fonction. Dans le cas général, une fonction mathématique peut avoir un nombre ≥1 quelconque d'arguments. Pour retrouver l'arbre, la fonction est le nœud père, ses arguments sont les nœuds fils. Ex :

f(g(x), h(y,z), h(x,z))

se traduit par l'arbre :



L'écriture est obtenue par une énumération préfixée. Comme on ne sait pas le nombre de fils, il faut soit

(a) l'indiquer, soit (b) utiliser des parenthèses. (b1) Dans l'écriture mathématique, on ouvre la parenthèse quand on sort du nœud père la première fois, et on ferme la parenthèse quand on y revient la dernière fois. (b2) Une autre solution consiste à ouvrir la parenthèse quand on entre dans le nœud père la première fois, et à la fermer quand on en ressort la dernière fois (on ne met pas de parenthèse aux feuilles)

(a) $f_3 g_1 x_0 h_2 y_0 z_0 h_2 x_0 z_0$

(b1) f(g(x) h(y z) h(x z))

(b2) (f (g x) (h y z) (h x z))

La solution (b2) est employée par certains langages de programmation comme Lisp

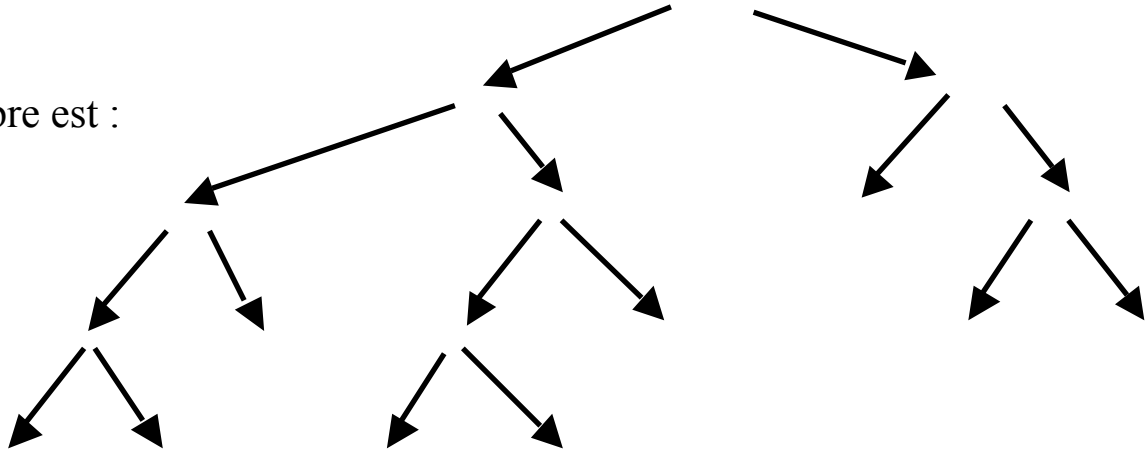
4. Ecrire les expressions arithmétiques

4.1. Bilan et point de départ

4.1.1. Quelle que soit la façon dont on l'écrit, une expression est un arbre

((3 + 12) . 5 + (4 + 17) . 6) / (7 + (8 . 4))

L'arbre est :



Chaque signe d'opération a deux arguments (donc deux fils). Un nombre n'a pas de fils (il est dans une feuille).

4.1.2. Il y a 3 règles d'écriture possibles

Ecriture infixée

Le signe d'opération est entre ses arguments :

((3 + 12) . 5 + (4 + 17) . 6) / (7 + (8 . 4))

Ecriture postfixée

Le signe d'opération est après ses arguments

3 12 + 5 . 4 17 + 6 . + 7 8 4 . + /

Ecriture préfixée :

Le signe d'opération est avant ses arguments

/ + . + 3 12 5 . + 4 17 6 + 7 . 8 4

4.2. L'arbre étant connu, générer ces écritures.

Utiliser l'algorithme de parcours en profondeur d'abord :

4.2.1. énumération préfixée

```
debut :          // Pour une énumération préfixée
   courant = racine ;
   faire
        si (courant ≠ NULL) faire
            pousser(p,courant)
            /* On écrit le symbole première fois qu'on y passe, donc quand on le
            pousse */
            écrire (courant.valeur)
            courant = courant.fils
        sinon
            courant = prendre(p)
            courant = courant.frere
        fin si
   tant que (non vide(p))
fin
```

1.1.2. énumération postfixée

```
debut :   // Pour une énumération postfixée
   courant = racine ;
   faire
        si (courant ≠ NULL) faire
            pousser(p,courant)
            courant = courant.fils
        sinon
            courant = prendre(p)
            /* Quand on dépile le noeud c'est la derniere fois qu'on y passe */
            écrire(courant.valeur)
            courant = courant.frere
        fin si
   tant que (non vide(p))
fin
```

1.1.3. énumération infixée

```
debut :   // Pour une énumération infixée
   courant = racine ;
   faire
        si (courant ≠ NULL) faire
            pousser(p,courant)
            courant = courant.fils
        sinon
            courant = prendre(p)
            /* S'il n'y a pas de fils, c'est une feuille, on l'écrit */
            si (courant.fils = NULL) ecrire(courant.valeur)
            courant = courant.frere
            /* Si on est passé du fils gauche au fils droit, c'est le moment
            d'écrire leur père */
            si (courant ≠ NULL)
              aux = prendre(p)
              écrire(aux.valeur)
              pousser(p,aux)
            fin si
        fin si
   tant que (non vide(p))
fin
```

5. Question inverse : l'écriture étant connue, générer l'arbre.

Problème général de l'analyse syntaxique. On étudie seulement un cas particulier, sans recourir aux méthodes plus élaborées (définition d'une grammaire)

Spécification : on doit lire l'expression de gauche à droite sans retour en arrière. On suppose que l'écriture est correcte (erreurs traitées dans la section suivante)

5.1. Un exemple de programme

C'est un programme de calcul, mais il aide à comprendre la lecture des expressions postfixées.

Le programme utilise une pile. On saisit soit des nombres, soit des signes d'opération :

- Quand c'est un nombre, on le met sur la pile
- Quand c'est un signe d'opération, on dépile ses arguments, on fait l'opération et on remet le résultat sur la pile

On visualise la pile

5.2. Transformer une expression postfixée en arbre

Ex : 3 4 5 - * 5 12 7 + * +

Algorithme
/* Construction de l'arbre de calcul à partir d'une expression postfixée correcte */

paramètre en entrée : Expression expr ; /* *Liste de symboles : signes d'opération ou nombres* */
paramètre en sortie : NoeudElem racine ; /* *c.a.d. : on a accès à tout l'arbre dès qu'on a accès à sa racine* */
Variables : pile pi ; NoeudElem courant, aux ; Symbole s ;
Fonctions :
ResteSymbole(Expression expr) /* 2 f*onctions standard …* */
ProchainSymbole(Expression expr) /* *… sur les listes* */
signeOpération(Symbole s) /* *test sur un symbole* */
créer(NoeudElem) /* *Allocation dynamique d'espace de la taille d'un noeud* */

```
Début
    tant que (ResteSymbole(expr)) faire
        courant =créer(NoeudElem) ;
        s = ProchainSymbole(expr) ;
        courant.valeur = s ;
        si (signeOpération(s))
            aux = dépiler(pi) ; /* les 2 arguments ont été empilés
                    dans le mauvais ordre */
            courant.fils = dépiler(pi) ;
            courant.fils.frère = aux;
          fin si
          empiler(pi, courant) ;
    fin tant que
    racine = dépiler ;
    renvoyer (racine) ;
fin
```

5.3. Transformer une expression préfixée en arbre

ex : + * 3 – 4 5 * 5 + 12 7

Algorithme
/* Construction de l'arbre de calcul à partir d'une expression préfixée correcte */

paramètre en entrée : Expression expr ; /* *Liste de symboles : signes d'opération ou nombres* */
paramètres en sortie : NoeudElem racine ; /* *c.a.d. : on a accès à tout l'arbre dès qu'on a accès à sa racine* */
Variables : pile attente ; NoeudElem courant ; Symbole lu ;
Fonctions :
créer(NoeudElem) /* *Allocation dynamique d'espace de la taille d'un noeud* */
ProchainSymbole (Expression expr) /* *Fonction standard sur les listes* */
signeOpération(Symbole s) /* *test sur un symbole* */

```
Début
    si estEnFin(expr) renvoyer null ; /* L'arbre est vide */
    racine = créer(NoeudElem) ;
    empiler(attente, racine) ;
    tant que (non estVide(attente))
        courant = dépiler(attente) ; /* courant est en attente de
        ses arguments */
        lu = ProchainSymbole (expr) ;
        courant.valeur = lu ;
        si (estOpération(lu))
            /* on crée deux fils qui attendent leur valeur. Les nœuds en
            attente sont donc déjà insérés dans l'arbre */
            courant.fils = créer (NoeudElem) ;
            courant.fils.frère = créer (NoeudElem) ;
            empiler(attente, courant.fils.frère) ;
            empiler(attente, courant.fils) ;
        fin si /*
    fin tant que
    renvoyer (racine) ;
fin
```

5.4. Transformer une expression infixée totalement parenthésée en arbre

Ex : ((3 * (4 – 5)) + (5 * (12 + 7)))
Une parenthèse ouvrante marque le début du sous arbre (donc du fils gauche). Une parenthèse fermante marque la fin du sous arbre (donc du fils droit). Le père est entre ses deux fils.

/* Construction de l'arbre de calcul à partir d'une expression infixée correcte */

paramètre en entrée : Expression expr ;
paramètres en sortie : NoeudElem racine
Variables : pile pi ; NoeudElem courant, pere ; Symbole s ;
Fonctions :
créer(NoeudElem) */* *Allocation dynamique d'espace de la taille d'un noeud* */*
ProchainSymbole (Expression expr) */* *Fonction standard sur les listes* */*
signeOpération(Symbole s), ouvrante(Symbole s), fermante(Symbole s) */* *tests sur les symboles* */*

Début
si estEnFin(expr) renvoyer null ; */* *L'arbre est vide* */*
racine = créer(NoeudElem) ;
courant = racine ;
faire
s = ProchainSymbole (expr) ;
si (ouvrante(s))
/* *on est au début d'une sous-expression, donc à la racine d'un sous-arbre. On crée ses fils, on empile la racine du sous-arbre et on se place dans le fils gauche qui attend sa valeur.* */
courant.fils = créer(NoeudElem) ;
courant.fils.frère = créer(NoeudElem) ;
empiler(pi, courant) ;
courant = courant.fils ;
sinon si (signeOpération(s))
/ *On est dans le fils gauche et il est terminé. On doit noter le signe opération dans la racine du sous-arbre (qui est le père de courant), puis passer au fils droit pour lire sa sous-expression.* */*
pere = dépiler(pi) ; */* *On doit sortir le père de la..*/*
pere.valeur = s ; */*… *pile uniquement pour …* */*

```
            empiler(pi, pere) ;        /*… noter le signe opération */
            courant = courant.frère() ; /* on va dans le fils droit  */
        sinon si (s.fermante())
        /* On a terminé la sous-expression, donc on revient à la racine
        de son sous-arbre. On attend d'avoir lu le symbole suivant (un
        signe d'opération ou une parenthèse fermante) pour décider de
        la suite. */
            courant = dépiler(pi) ;
        sinon /* c'est un nombre, donc le sous arbre est réduit à une
        feuille et on est dans sa racine. On attend d'avoir lu le symbole
        suivant (un signe d'opération ou une parenthèse fermante) pour
        décider de la suite.  */
            courant.valeur = s ;
        fin si
    tant que (non vide(pi))
    renvoyer (racine) ;
fin
```

6. Arbre de calcul arithmétique : correction des écritures

Propriété générale des expressions arithmétiques de ce chapitre : dans une expression correcte il y a un nombre de plus que de signe d'opération. Preuve par récurence sur l'arbre.

6.1. Vérification syntaxique d'une expression postfixée :

Comme tous ses arguments sont avant chaque signe d'opération, à tout moment de la lecture il doit y avoir plus de nombres que de signes d'opération.

Ex : 3 4 + 5 / - .

	3	4	+	5	/	-	3	*
nbres	1	2	2	3	3	3	4	4
sign op	0	0	1	1	2	3	3	4
différence	1	2	1	2	1	**0**	1	**0**
						Erreur		Erreur

```
Algorithme
   /* vérification syntaxique d'une expression postfixée non
   vide */
   paramètre en entrée : Expression expr ;
   paramètres en sortie : booléen correct ;
   /* c.a.d. : renvoie Vrai (correct) ou faux (incorrect) */
   Variables : Symbole s ; entier diffNbOps ;
   Initialisation : diffNbOps = 0 ;

   Début
      faire
         s = expr.nextSymbol() ;
         si (s. signeOpération())
            diffNbOps = diffNbOps - 1 ;
         sinon
            diffNbOps = diffNbOps + 1 ;
         finsi
      tant que (diffNbOps >= 1 ET expr.hasMoreSymbol())

      si (diffOpsNb < 1)
         message("Erreur : il y a des signes d'opération sans
         assez d'arguments") ;
         correct = Faux ;
      sinon si (diffOpsNb > 1)
         message("Erreur : il manque des signes d'opération") ;
         correct = Faux ;
      sinon
         correct = Vrai ;
      fin si
      renvoyer (correct) ;
fin
```

6.2. Vérification d'une expression préfixée :

ex : + * 3 – 4 5 * 5 + 12 7

Comme les signes d'opération sont avant, il y a au moins autant de signes que de nombres. Dés qu'il y a un nombre de plus, c'est fini – on ne peut rien rajouter.

		+	*	3	-	4	5	*	5	+	12	7
nb signes op	0	**1**	**2**	2	**3**	3	3	**4**	4	**5**	5	5
nb nombres	0	0	0	**1**	1	**2**	**3**	3	**4**	4	**5**	**6**
diffOpsNb	0	1	2	1	2	1	0	1	0	1	0	-1

Algorithme
/* vérification syntaxique d'une expression préfixée */
paramètre en entrée : Expression expr
paramètres en sortie : Booléen correct ;
/* c.a.d. : renvoie Vrai (correct) ou faux (incorrect) */
Variables : Symbole s ; entier diffOpsNb ;
Initialisation : diffOpsNb = 0 ;
Début
tant que (diffOpsNb > -1 ET expr.hasMoreSymbol()) faire
s = expr.nextSymbol() ;
si (s. signeOpération())
diffOpsNb = diffOpsNb + 1 ;
sinon
diffOpsNb = diffOpsNb - 1 ;
finsi
fin tant que

si (diffOpsNb > -1)
message("Erreur : il y a des signes d'opération sans argument") ;
correct = Faux ;
sinon si (expr.hasMoreSymbol())
message("Erreur : il reste des symboles après la fin de l'expression") ;
correct = Faux ;
sinon
correct = Vrai
fin si
renvoyer (correct) ;
fin

Correction d'une expression arithmétique infixée :
Essayez de trouver..