Une approche fonctionnelle et abstraite pour mieux enseigner les arbres

Dominique Cansell
Université de Metz
LRIM-UFR MIM-CRDD[1]
Ile du Saulcy
57045 Metz CEDEX 01
France
Tel: 87 31 52 85 email: cansell@ciril.fr

1 Introduction

Cet article présente l'approche que nous utilisons pour enseigner les arbres aux étudiants du DEUG[2] scientifique qui poursuivent leurs études en mathématiques et en informatique. Nous utilisons déjà (depuis 8 années) une approche fonctionnelle et abstraite pour enseigner les listes linéaires aux étudiants de DEUG [Can91], [Can92]. Depuis maintenant 3 années nous nous sommes investi dans l'enseignement des arbres en conservant la même approche. Aujourd'hui nous pouvons dresser un premier bilan tout à fait positif et qui peut s'exprimer par les points suivants:

facilités pour transmettre notre savoir; ce qui n'est pas négligeable quand on connait les difficultés rencontrées par les informaticiens pour former correctement leurs étudiants [SPE88,90a,90b]. Notre approche pour enseigner les listes s'est avérée payante et profitable. En adoptant la même avec les arbres cela a donné une constante et un fil directeur à notre enseignement ce qui a sans nul doute rassuré et conforté nos étudiants.

comme nous donnons à nos étudiants un cadre abstrait dans lequel ils peuvent raisonner proprement, ils ont beaucoup moins de difficultés à analyser et donc à résoudre leurs problèmes. En fait ils n'ont que la difficulté de l'analyse (construction de l'algorithme). Nous considérons que l'analyse est un acte **créatif** [DMa90] et nous insistons beaucoup sur la construction des algorithmes à partir de l'induction [Her93] et sur la détection de type à partir de l'analyse d'un algorithme.

comme nous donnons à nos étudiants des moyens d'expressions simples et efficaces pour transformer leur analyse en une fonction à l'aide des fonctions de base, ils peuvent écrire plus facilement leurs programmes. Nous ne négligons pas le style procédural (actionnel): bien au contraire, nous expliquons à nos étudiants comment, à partir d'une fonction correcte (et testée), ils peuvent obtenir une procédure qui transformera leurs arbres en faisant une économie de place et de temps à l'exécution.

---

[1] **LRIM**: Laboratoire de Recherche en Informatique de Metz
UFR MIM: UFR Mathématique, Informatique, Mécanique et Automatique (Metz)
CRDD: Centre de recherche en Didactique des Discipline (Metz).

[2] **DEUG**: Diplôme d'Etudes Universitaire Général (les deux premières années à l'Université). Avant ce cours sur les arbres (56h), ces étudiants ont eu deux cours en DEUG1 (initiation à l'algorithmique (40h) et à la programmation structurée (60h)) et un cours d'initiation à l'algorithmique récursive (56h) en DEUG2 et plus de 180h de programmation libre.

les étudiants peuvent progresser en douceur. S'ils n'ont pas tout compris (allocation mémoire, style procédural), ils peuvent rester (ou revenir) aux fonctions.
les étudiants abordent d'autres notions de la discipline informatique comme la logique (utilisation de propriétés), la compilation (lecture de termes) ou de réécriture (égalité ou simplification de terme).

Nous montrons notre façon[3] de faire en donnant deux exemples "classiques" d'utilisation des arbres: les arbres binaires (ordonnés) et les expressions arithmétiques. Puis nous donnerons un exemple de détection de type.

2 Exemple 1: Les arbres binaires

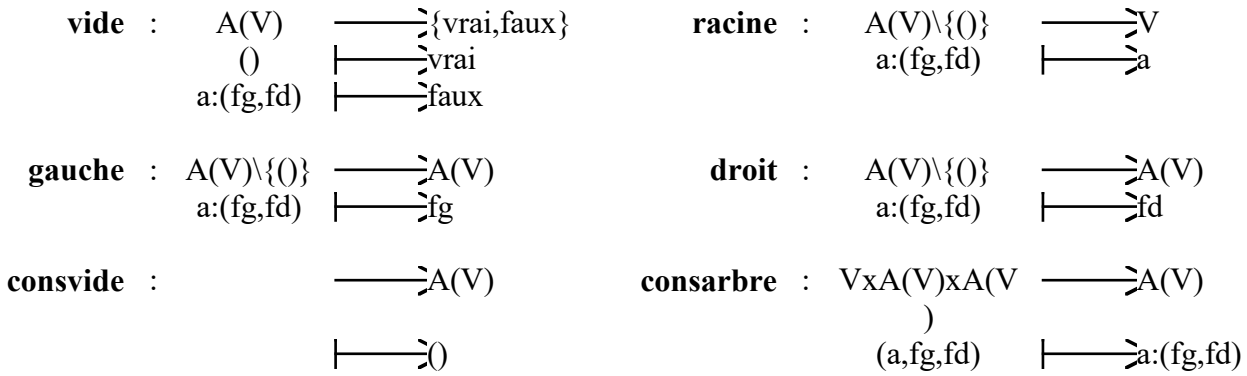
2.1 Définition récursive

Nous définissons l'ensemble des arbres binaires sur V de manière récursive suivante:

$$A(V) = \{()\} \text{ È } \{ a:(fg,fd) \text{ tq } a \hat{I} V \text{ et } fg,fd \hat{I} A(V)\}$$

2.2 Fonctions de base

Sur A(V) nous définissons les fonctions de base suivantes:



la fonction vide est une fonction de sélection, les fonctions racine, gauche et droit sont des fonctions de projection (accès aux informations) et les fonctions consvide et consarbre des fonctions de construction.

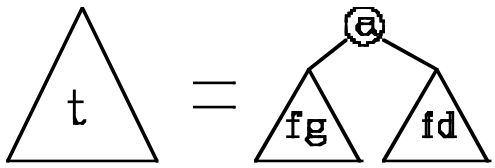
2.3 Approches abstraites

Il est bien clair que () et a:(fg,fd) donnent une idée abstraite d'un arbre. Cette idée peut être renforcée de deux manières: une représentation schématique et une représentation ensembliste.

[3] les analyses ne sont pas toujours complètes, nous laisserons le lecteur informaticien les terminer.

2.3.1 Approche abstraite "schématique"

Nous avons l'abitude de représenter un arbre t=a:(fg,fd) de la manière suivante:



2.3.2 Approche abstraite "ensembliste"

les arbres binaires servent à stoquer des informations (un ensemble de valeurs). L'ensemble des valeurs d'un arbre t peut être défini à l'aide de la fonction VALEUR suivante:

VALEUR : A(V) $\longrightarrow$ P(V)
() $\longmapsto$

a:(fg,fd) $\longmapsto$ VALEUR(fg){a}VALEUR(fd)

On peut donc considérer un arbre comme un ensemble vide ou comme la réunion de trois ensembles dont l'un est un singleton. Ces trois ensembles forment une partition de l'ensemble des valeurs de l'arbre.

2.3.3 Analyse abstraite

L'analyse de l'algorithme de recherche d'une valeur v dans un arbre t peut se faire à l'aide de la propriété suivante: vÎVALEUR(t) que l'on peut exprimer de la manière suivante: vÎt. Cette propriété doit être un guide pour construire correctement l'analyse de l'étudiant.

vÎ()	vÎfaux
vÎa:(fg,fd)	vÎVALEUR(fg){a}VALEUR(fd) (v=a) ou (vÎVALEUR(fg)) ou (vÎVALEUR(fg))

Cet algorithme de recherche est un fil directeur dans la suite du cours. On pourra définir l'ensemble des arbres ordonnés AO(V) avec l'objectif: optimiser l'algorithme de recherche c-à-d éviter les deux appels récursifs.

AO(V)={ tÎA(V) tq ord(t)} avec ord défini récursivement de la manière suivante:

ord(())	=	vrai
ord(a:(fg,fd))	=	ord(fg) et ord(fd) et (vÎVALEUR(fg) va) et (vÎVALEUR(fd) va)

On pourra introduire les deux autres algorithmes "classiques" sur les arbres ordonnés qui sont

l'insertion: car il faudra que vÎinsert(v,t)ÎAO(V). La fonction insert devra respecter la propriété VALEUR(insert(v,t))={v}VALEUR(t) et (ord(t)ord(insert(v,t))).

la suppression car il faudra que vsupp(v,t)ÎAO(V). La fonction supp devra respecter la propriété: VALEUR(supp(v,t))=VALEUR(t)\{v} et (ord(t)ord(supp(v,t)))

Analyse de la fonction de suppression[4]

t=()			®	supp(v,()) = ()	\{v}=
t=a:(fg,fd)	et	(va)	®	supp(a:(fg,fd))=a:(supp(v,fg),fd)	t\{v}=(fg\{v}){a}fd[5]
	et	(va)	®	supp(a:(fg,fd))=a:(fg,supp(v,fd))	t\{v}=fg{a}(fd\{v})[6]
	et	(v=a)	®	?	

ce dernier cas est le plus difficile. On voit que les autres cas (faciles) sont règlés par l'induction. On a:

VALEUR(supp(a,a:(fg,fd)) = (VALEUR(fg){a}VALEUR(fd))\{a}
= VALEUR(fg)VALEUR(fd)

et il faut pouvoir décrire l'ensemble VALEUR(fg)VALEUR(fd) à l'aide de la réunion de deux ensembles et d'un singleton (s'il est non vide). Il faut donc choisir la valeur du singleton (de la racine) dans l'un des deux ensembles VALEUR(fg) ou VALEUR(fd). Choisissons VALEUR(fg) et continuons l'analyse:

t=a:(fg,fd)	et (v=a) et fg=()	®	supp(a:(fg,fd))=fd	({a}fd)\{a}=fd
	et (v=a) et fg1()	®	supp(a:(fg,fd))=v':(supp(v',fg),fd)	avec v'VALEUR(fg)

pour que v':(supp(v',fg),fd) soit un arbre ordonné il faut que v' soit la plus grande valeur de l'arbre fg (VALEUR(fg)) donc il faudra analyser les deux problèmes suivants:

plus_grande_valeur(t) et supp_la_plus_grande(t) qui sont simples et qui ont les mêmes cas d'analyse.

2.4 Représentation en PASCAL

Nous utilisons toujours PASCAL comme langage de programmation en DEUG. Nous ne défendons en aucun cas ce langage de programmation. Par contre ce n'est pas le langage qui est important (apprentissage d'un langage de programmation) mais bien l'analyse abstraite et applicative des problèmes. Nous pouvons (ou le lecteur) utiliser un autre langage comme C, SCHEME ou ML.

---

[4] ici nous ne "dessinons" pas les arbres résultat. Par contre nous le faisons avec nos étudiants car ces deux approches sont complémentaires.

[5] toutes les valeurs de fg\{v} sont strictement inférieures à a et fg\{v} est ordonné par induction car fg l'est.

[6] toutes les valeurs de fd\{v} sont strictement supérieures à a et fd\{v} est ordonné par induction car fd l'est.

2.4.1 Le type

```
type      arbre=–noeud;
  noeud=record
          racine:V;
          gauche,droit:arbre
        end;
```

Le noms des champs sont les mêmes que ceux des fonctions. C'est intensionnel. Les étudiants connaîssent bien leurs fonctions, ils connaîtront donc bien les variables qui contiennent le résultat des fonction de base (pour un passage par variable).

2.4.2 Réalisation PASCAL des fonctions de base

La réalisation est aussi simple que celle des listes [Can91]. Nous procédons de la manière suivante:

vide:=(t=nil) pour vide(t:arbre):boolean
racine:=t–.racine pour racine(t:arbre):V;
gauche:=t–.gauche pour gauche(t)
consvide:=nil pour consvide
new(vcons);vcons–.racine:=a;vcons–.gauche:=fg;vcons–.droit:=fd;consarbre:=vcons pour consarbre(a:V;fg,fd:arbre):arbre; (vcons est une variable locale à consarbre et de type arbre).

Il est clair que l'on peut facilement, à partir de l'analyse, écrire une fonction qui nous donnera ou construira la solution de notre problème car si t=a:(fg,fd) nous avons a=racine(t), fg=gauche(t), fd=droit(t) et a:(fg,fd)=consarbre(a,fg,fd).

2.4.3 Passage du fonctionnel à l'actionnel

Comme pour les listes [Can92], [Can91] nous montrons aux étudiants

que l'utilisation des fonctions leur assure un vrai passage par valeur au sens arbre et non au sens pointeur.
que l'exécution de leurs fonctions permet de tester leurs analyses.
que les fonctions sont parfois gourmandes en temps et en place mémoire sur le tas. On pourra donc faire des transformations physiques sur les données qui seront donc résultats. Il faudra donc en PASCAL utiliser une procédure .

Exemple: la procédure proc_supp(v:V;var t:arbre), qui supprime de manière physique une valeur dans un arbre sans appeller la fonction consarbre à l'exécution, peut être écrite à l'aide de la propriété suivante:

proc_supp(v,t)	t:=supp(v,t)
0 consarbre à l'exécution	avec des consarbres à l'exécution

La procédure fait donc les mêmes tests que la fonction (mêmes cas d'analyse) et de plus la fonction nous donne la valeur du résultat. Il faut donc dans chaque cas savoir où mettre le résultat. Etudions le cas t=a:(fg,fd) et (va) le résultat de l'analyse est supp(a:(fg,fd))=a:(supp(v,fg),fd) donc il faudrait écrire l'action suivante: t:=consarbre(racine(t),supp(v,gauche(t)),droit(t)). Nous n'avons pas le droit d'utiliser de consarbre. La variable t (résultat) contient déjà l'adresse d'un noeud qui contient racine(t) et droit(t). Il faut donc que la variable t–.gauche (qui contient gauche(t)) contienne supp(v,gauche(t)) et par hypothèse de récurrence on peut réaliser t–.gauche:= supp(v,gauche(t)) à l'aide de l'appel proc_supp(v,t–.gauche). Le dernier cas peut être réglé par les instructions suivantes:

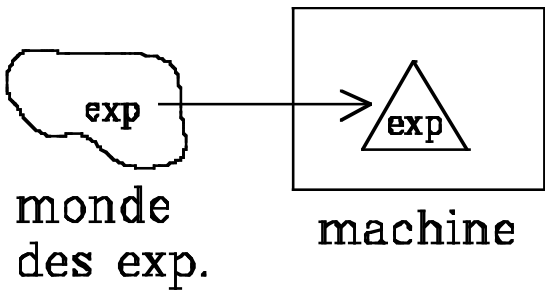
```
t–.racine:=plus_grande_valeur(t);
supp_la_plus_grande(t–.gauche)
```

or comme plus_grande et supp_plus_grande ont les mêmes cas d'analyse on pourra écrire une procédure unique plus_grande_et_supp(var plus_grande:V;var t:arbre) qui calcule et met la plus grande valeur de t dans la variable plus_grande et qui supprime de manière physique la plus grande valeur de t. On obtient donc en réglant tous les autres cas la procédure suivante:

```
procedure supp(a:V; var t:arbre);
begin
  if vide(t) then  t:=consvide
  else if racine(t)<a then  supp(a,t–.gauche)
  else if racine(t)>a then  supp(a,t–.droit)
  else if vide(gauche(t)) then  t:=droit(t)
  else  plus_grande_et_supp(t–.racine;t–.gauche)
end;
```

3 Les expressions

Cette partie montre bien l'intérêt de l'approche abstraite et de l'utilisation des fonctions de base pour définir un type. On retrouve les expressions dans de nombreux ouvrages [BoM83], [PMS88], [Cou92] ... Nous expliquons à nos étudiants qu'il existe un monde des expressions qu'il connaissent bien et qu'ils utilisent soit en mathématique (équation, ...) soit en informatique (dans langage de programmation) et qu'ils doivent faire entrer ce monde dans la machine (trouver un type).



Nous leur expliquons qu'une expression est soit une valeur entière, soit une variable (muette ou informatique) ou soit une opération binaire (définition récursive syntaxique avec les

priorités). Nous leur donnons ensuite la définition récursive suivante de l'ensemble EXP des expressions:

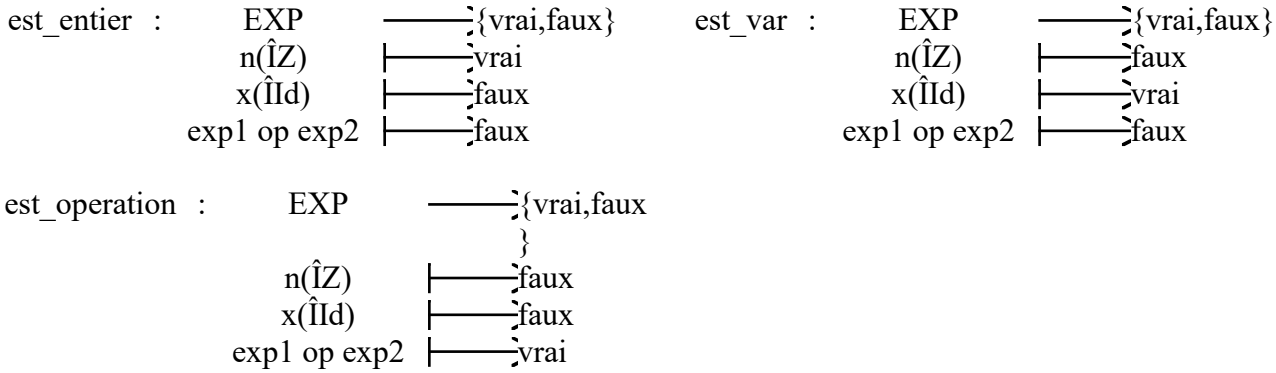
EXP = Z È Id È {exp1 op exp2 tq exp1,exp2ÎEXP et opÎ{+,-,*,/} }

3.1 Les fonctions de base

et nous leur définissons les fonctions de base sur l'ensemble EXP qui sont les suivantes:

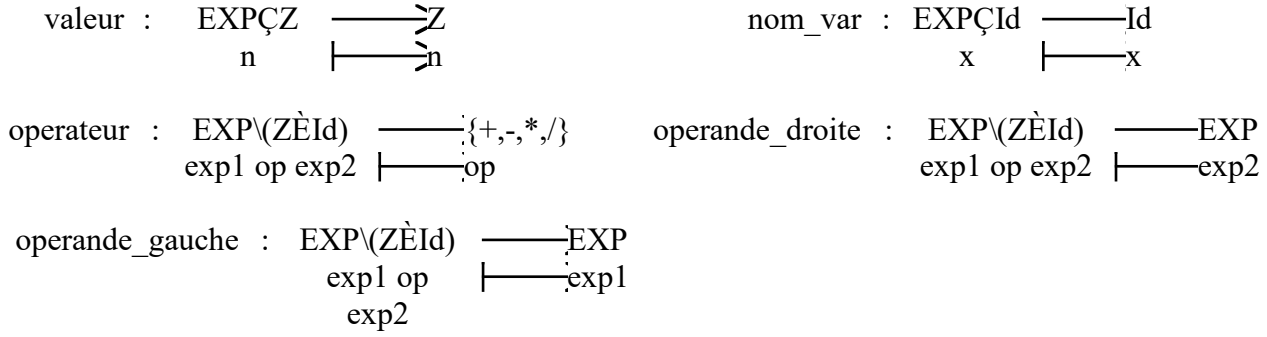
3.1.1 Fonctions de selection

L'ensemble EXP est la réunion de trois ensembles disjoints. Il faudra donc savoir à quel ensemble appartient une expression exp. Nous aurons donc les trois fonctions suivantes (deux suffisent):



3.1.2 Fonctions de projections

Ceux sont les fonctions qui nous permettent d'accèder aux différentes informations d'une expression qui est soit un entier, soit une variable soit une opération.



3.1.3 Les constructeurs

cons_entier : Z ⟶ EXP
n ⟼ n

cons_var : Id ⟶ EXP
x ⟼ x

cons_op : EXPx{+,-,*,/}xEXP ⟶ EXP
(exp1,op,exp2) ⟼ exp1 op exp2

3.2 Les propriétés

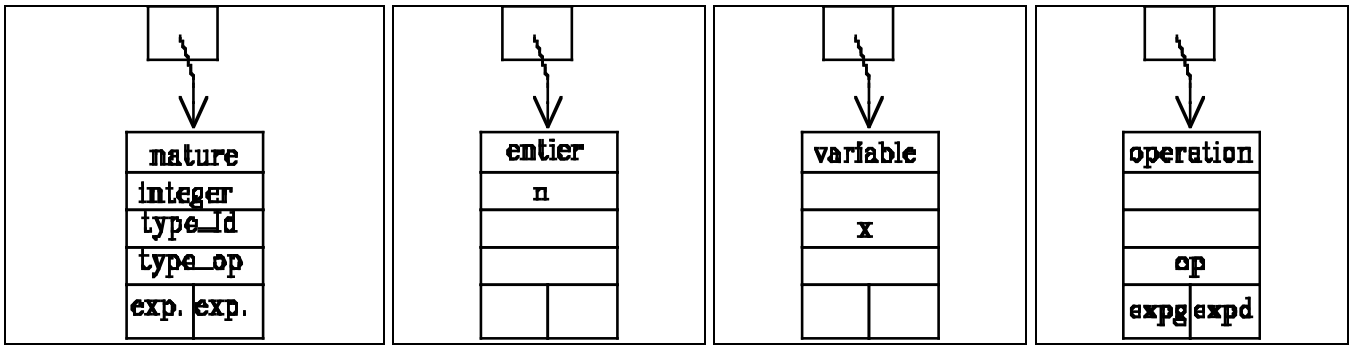
Nous avons les propriétés suivantes de cohérences entre les fonctions de construction et les autres fonctions de base sur les expressions:

est_entier(cons_entier(n))=vrai (les autres faux) et valeur(cons_entier(n))=n
est_var(cons_var(x))=vrai (les autres faux) et nom_var(cons_var(x))=x
est_operation(cons(op,expg,expd))=vrai (les autres faux) et
operateur(cons(op,expg,expd))=op et operande_gauche(cons(op,expg,expd))=expg et
operande_droite(cons(op,expg,expd))=expd

3.3 Représentatin en PASCAL

Il faut qu'à l'aide du type on puisse écrire les fonctions de base en respectant les propriétés. Voici un tableau qui peut permettre de comprendre la construction de ce type:

variable de type expression	cons_entier(n)	cons_var(x)	cons_op(op,expg,expd)



3.4 L'approche fonctionnelle

Tous les algorithmes sur une expression exp seront du style:

```
if est_entier(exp) then
    bricole_entier(valeur(exp))
else if est_var(exp) then
    bricole_Id(nom_var(exp))
else
    bricole_op(operande_gauche(exp),operateur(exp),operande_droite(exp))
```

L'avantage des fonctions de base est que les algorithmes sur les expressions ne sont pas imprégnés du type PASCAL. Cela permet une meilleure lecture et compréhension des algorithmes et surtout une modification plus aisée des programmes:

si on change le type il faut uniquement modifier les fonctions de base

si l'on augmente le type (autres cas) il faudra ajouter des fonctions de base et rajouter ces cas dans les algorithmes.

Cet exemple nous permet d'aborder des thème qui nous sont chers en informatique:

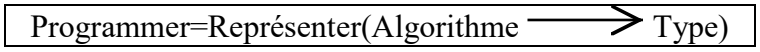
compilation (lecture de terme: préfixée, infixée bien parenthèsée, infixée) et interpréteur (evaluation d'expression avec un état des variables).

calcul formel: dérivé [BoM83]

réécriture: simplification d'expression et égalité d'expression

4 Détection de type

Après avoir donné un certain nombre de type à nos étudiants, nous leur donnons un moyen de détecter des types et de les représenter dans leur langage de programmation pour obtenir des programmes. Nous leur proposons d'utiliser notre équation:



où $\longrightarrow$ veut dire détecter le(s) type(s) à partir de l'algorithme. C'est l'analyse qui va faire ressortir des fonctions de base sur ce type, qui nous permettront de l'identifier correctement. L'exemple que nous donnons aux étudiants est celui, bien connu, du dictionnaire[7]. La première idée de l'étudiant est d'utiliser une liste de mots, un mot étant une liste de caractères. Même si tous les algorithmes (recherche, insertion et suppression d'un mot dans un dico) sur un tel dictionnaire sont simples à écrire, ils sont longs à l'exécution (rechercher un mot qui commence par "z" nécessite de le comparer à presque tous les mots du dictionnaire. Il faut donc une autre idée!: les premiers mots d'un dictionnaire d commencent par "a"

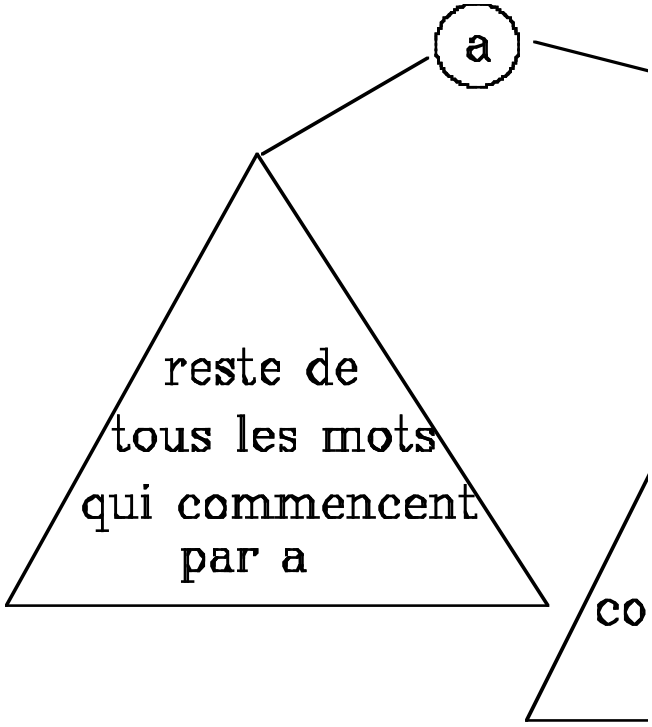
si le mot m que l'on cherche ne commence pas par "a" il faut le chercher parmi les mots qui ne commencent pas par "a"

si le mot m que l'on cherche commence par "a", il faudra chercher le reste du mot dans les restes des mots de d qui commencent par "a"

Et donc nous en déduisons:

1) la forme d'un dictionnaire 2) les fonctions de base sur un dictionnaire d

[7]un dictionnaire est un ensemble de mots. On veut pouvoir rechercher, inserer, supprimer un mot dans un dictionnaire , afficher tous les mots d'un dictionnaire ...



racine(d) (une lettre)
autres_mots(d) (un dictionnaire)
reste_des_mots(d) (un dictionnaire)

D'où l'analyse de md (d est considéré comme un ensemble de mots)
d= md faux
d et m=am'[8] et a=racine(d) md m'reste_des_mots(d)
d et m=am' et aracine(d) md mautres_mots(d)

L'analyse a permis de mettre en évidence ce que doit contenir un dictionnaire. On remarque que l'analyse n'est pas complète car le mot peut-être vide. Lorsque le mot est vide c'est qu'il existe un "chemin" dans le dictionnaire qui nous a "vider" le mot. Il faut absolument qu'il existe dans le dictionnaire, une information qui nous dise si oui ou non le mot vide est dans un dictionnaire (existence de la marque de fin).
Le choix de l'algorithme de recherche est judicieux car les algorithmes de l'insertion et de la suppression en dépendent. Pour afficher l'ensemble des mots il faut le connaître, la propriété suivante peut nous aider:

md m{racine(d).m' tq m'reste_des_mots(d)}{m'autres_mots(d)}

Dans l'analyse on se rend compte que les opérations sur les mots sont
- vide (il n'y a plus de lettre dans le mot)
- la première lettre du mot
- le reste du mot

Ces opérations sont les fonctions de base sur le mots qui permettent de définir le type d'un mot.

5 Conclusions

[8] a est la première lettre du mot m=am' et m' est le reste du mot m.

Notre objectif est que nos étudiants fassent des analyses propres et correctes pour qu'ils puissent programmer plus facilement. L'important c'est aussi l'image que nous donnons de l'informatique à nos étudiants. C'est une image "scientifique". Notre plus grande satisfaction est l'image de notre discipline que nous renvoient nos étudiants mais cela nous est personnel.
.

Références:

[AbS89]: H. Abelson, et G.J. Sussman, *Structure et interprétation des programmes informatiques*, InterEditions, 1989.

[BoM83]: J.C. Boussard, et R. Mahl, *Programmation avancée*, Eyrolles, 1983.

[Can91]: D. Cansell, *PASCAL un langage applicatif,* Journée langage applicatif dans l'enseignement de l'informatique, mars 1991, revue BIGRE 73 juin 1991.

[Can92]: D. Cansell, *Un enseignement de l'informatique en tant que discipline en DEUG scientifique à partir d'une expérience pédagogique à l'Université de Metz*, Thèse de l'Université de Metz, soutenue le 4 avril 1992.

[Cou92]: P. Cousot, *Algorithmique et programmation en PASCAL*, Exercices et corrigés, Ellipses, 1992.

[DMa90]: P. A. De Marneffe, *Enseigner l'algorithmique*, Deuxième colloque francophone sur la didactique de l'informatique, 1990.

GSF87]: M.C. Gaudel, M. Soria, et C. Froidevaux, *Types de données et Algorithmes* (2 tomes), INRIA, 1987.

[Gra86]: A. Gram, *Raisonner pour programmer*, DUNOD informatique, 1987

[Her]: D. Herman, Ph. Ingels, G. Lesventes, R. Vorc'h, *Scheme, Deug et première initiation à l'informatique*, Actes des 2èmes journées Langages applicatifs, Rennes, avril 1993.

[Pey91]: J.-P. Peyrin, *Un environnement logiciel pour assister l'enseignement et l'apprentissage de la programmation*, Habilitation à diriger les recherches, Université Joseph Fourier, Grenoble, février 1991.

[PMS88]: C. Pair, R. Mohr, et R. Schott, *Construire les algorithmes*, Dunod informatique, 1988.

[Sch84]: P.C. Scholl, *Algorithmique et représentation des données, tome 3: récursivité et arbres*, Masson, 1984.

[SPE88]: L'informatique dans les premiers cycles scientifiques, Colloque de Besançon, Rapport SPECIF, novembre 1988.

[SPE90]: Enseigner l'informatique en tant que discipline en DEUG scientifique, Journées de Nantes, Rapport SPECIF, mars 1990.

[Wir76]: N. Wirth, *Algorithms + Data structures = Programs*, Prentice Hall, Englewood Cliffs, 1976.