L'informatique, une displine dans les DEUG scientifiques à partir d'une expérience pédagogique à l'Université de Metz

Dominique Cansell

Laboratoire de Recherche en Informatique de Metz
UFR Mathématique, Informatique, Mécanique et Automatique

Thèse

présentée et soutenue le 4 avril 1992

à l'Université de Metz

en vue de l'obtention du grade de

Docteur de l'Université de Metz

spécialité Informatique

par

Dominique Cansell

" L'informatique, une discipline dans les DEUG scientifiques à partir d'une expérience pédagogique à l'Université de Metz"

Composition du jury:

J.-C. Boussard, Professeur à l'université de Nice (rapporteur)
Ch. Duchâteau, Chargé de cours à l'université Notre Dame de la Paix de

Y. Gardan, ...recteur de Thèse)
D. Méry, C... ...ateur)
J.-P. Peyrin... ...de Grenoble (rapporteur)

A Isabelle, Youri et Nastasia

Remerciements

Je tiens d'abord à remercier Patrick Cousot, c'est mon "père" informatique, le peu que je connais c'est lui qui me l'a appris. Il a été le (et mon) premier professeur d'informatique de Metz et c'est dans mes notes de ses cours et du bon souvenir que j'en ai gardé que j'ai trouvé la force et l'inspiration pour les miens.

Jean-Claude Boussard est l'inconditionnel de la première heure. Il m'a fait comprendre l'importance de mon investissement en DEUG. C'est lui, qui en proposant de faire de l'informatique à la "messine" lors des journées SPECIF de Nantes, a fait en sorte que cette Thèse commence; son soutien constant m'a permis de la finir. Le fait qu'il ait accepté de la juger et de participer à ce jury m'honore profondément.

Jean-Pierre Peyrin a été bien plus qu'un rapporteur pour cette Thèse. Nos discussions et ses nombreux conseils m'ont permis de prendre du recul par rapport à mon expérience et donc de valoriser un peu plus mon travail. Je le remercie sincérement et j'espère que notre collaboration ne s'arrêtera pas là.

Je remercie Charles Duchateau d'avoir bien voulu rapporter sur cette Thèse. A ses "trois coeurs" pour enseigner l'informatique il faut ajouter le coeur de l'enseignant, celui qu'il a montré dans ses nombreuses interventions au colloque SPECIF de Lille et en organisant le deuxième colloque francophone de didactique de l'informatique à Namur.

Dominique Méry est mon compagnon de toujours, c'est lui qui a participé à l'élaboration du DEUG rénové tel qu'il existe encore aujourd'hui. Il s'est toujours intéressé à mon travail même lorsqu'il a quitté (momentanément je l'espère) l'Université de Metz pour se consacrer à ses recherches. Je le remercie pour son amitié, pour son aide et pour avoir accepté de participer à ce jury.

Si l'Université de Metz a beaucoup perdu au départ de Patrick Cousot, elle aurait encore plus perdu si Yvon Gardan (deuxième et donc seul professeur d'informatique de Metz) n'avait pas pris les choses en main. Pour qu'il puisse s'occuper en priorité du laboratoire j'ai accepté qu'il me confie des taches administratives et au cours des années une confiance réciproque s'est installée. Après mon intervention de Nantes et la lecture de mes polycopiés, il me pousse, pour régulariser ma situation

d'assistant, à faire une Thèse en didactique, bien que cela ne soit pas son domaine! (professeur d'Université, auteur de plusieurs livres de référence sur la C.A.O dont un prix Roberval). J'accepte ce challenge et le fait que j'écrive ces quelques lignes montre qu'il ne s'est pas trompé. Je ne trouve plus mes mots pour le remercier. J'espère que d'autres professeurs d'informatique viendront à Metz et qu'ils s'investiront au sein de l'Université et du laboratoire, je les y aiderai le plus possible.

Je n'oublie pas mon autre compère Patrick Bellot: nos longues discussions "nocturnes" m'ont fait comprendre les vertus de l'analyse et de la programmation applicative. Je sais qu'il désire revenir dans une Université, quel que soit son choix, cette décision ne peut que me réjouir.

Je veux également remercier Michel Lucas pour m'avoir un jour donné deux feutres et cinq transparents, Pierre-Claude Scholl pour les critiques toujours constructives qu'il a eues sur mon travail. Leur investissement avec celui de Jean-Claude Boussard et bien d'autres dans l'organisation et le déroulement des colloques SPECIF de Besançon, de Nantes et de Lille ont permis de faire avancer la réflexion de la communauté informatique sur l'enseignement de notre discipline en DEUG.

Je tiens à remercier l'ensemble des membres du LRIM, et mes collègues enseignants et/ou chercheurs, IATOS qui s'investissent pour que nos étudiants de DEUG puissent prétendre avoir eu une formation générale qui leur permettra d'évoluer.

Pour finir je veux remercier ma femme Isabelle et mes enfants: c'est eux qui ont du faire le plus de sacrifices lors de ces travaux en didactique (polycopiés, Thèse) car j'étais souvent absent dans mes pensées et devant un clavier. En plus de la charge de ma petite famille Isabelle a du faire face à une tâche presque impossible (pour ceux qui me connaissent): corriger mes fautes d'aurtograsffe.

J'ai enfin compris pourquoi la plupart des auteurs dédient leurs ouvrages à leur femme et à leurs enfants.

L'informatique, une discipline dans les DEUG scientifiques

à partir d'une

expérience pédagogique à l'Université de Metz

TABLE DES MATIERES

DEUXIEME PARTIE: "Réalisation"

Chapitre 4: DECOMPOSITION

4.1) Les outils pour la décomposition
- 4.1.1) L'algorithmique de base
- 4.1.2) Les modules
  - 4.1.2.1) Les fonctions
  - 4.1.2.2) Les procédures
- 4.1.2) Exécution d'un programme
- 4.1.3) Exécution d'une fonction (récursive)

4.2) Une leçon de décomposition
- 4.2.1) Le problème: le jeu mastermind chinois
- 4.2.2) Décomposition du problème
- 4.2.3) Décomposition de jouer_une_fois
- 4.2.4) Décomposition de la procédure initialise
- 4.2.5) Décomposition de la fonction bon
- 4.2.6) Décomposition des procédures et des fonctions non terminées: Celles qui dépendent du type combinaison
- 4.2.7) Vers le logiciel décompositeur

4.3) Utilisation des propriétés pour concevoir les algorithmes
- 4.3.1) Passage du fonctionnel à l'actionnel
- 4.3.2) Conception d'un algorithme

4.4) La notion de coût

Chapitre 5: SCHEMAS

5.1) Schémas itératifs
- 5.1.1) Théorème de récurrence pour les boucles
- 5.1.2) Les propriétés
  - 5.1.2.1) ∀ et ∃
- 5.1.3) Les algorithmes de parcours
  - 5.1.3.1) Les traitements
  - 5.1.3.2) Les calculs

5.2) Schémas récursifs
- 5.2.1) Les procédures
- 5.2.2) Les fonctions
- 5.2.3) Utilisation des schémas

Chapitre 6: UNE APPROCHE FONCTIONNELLE POUR LES LISTES LINEAIRES

6.1) Définition récursive de L(V)

6.2) Fonctions de base sur L(V)
- 6.2.1) Fonctions de consultation
- 6.2.2) Fonctions de construction

6.3) Utilisation abstraite
- 6.3.1) Analyse récursive et applicative
  - 6.3.1.1) Exemple de fonction sur les listes: l'insertion d'une valeur v dans une liste ordonnée
  - 6.3.1.2) Exécution symbolique
- 6.3.2) Théorème d'induction sur L(V)
- 6.3.3) Les schémas de fonction sur les listes

6.4) La réalisation Pascal à l'aide de variable dynamique

PREMIERE PARTIE: "Que faire?"

De l'informatique "sérieuse"

Chapitre 1

INTRODUCTION

1.1) Présentation

Cette Thèse a pour but de donner nos idées sur la didactique de l'informatique à partir de notre expérience: un enseignement de l'informatique en tant que discipline en DEUG scientifique.

Il ne s'agit pas d'une proposition d'enseignement. Il est clair que ce n'est pas une personne (même si elle s'est beaucoup investie dans ce type d'enseignement et dont certaines réflexions ont été utiles aux étudiants) ni même plusieurs qui doivent décider de ce que doit être ou ne pas être un enseignement de l'informatique mais à l'heure où les informaticiens se posent des questions sur la façon d'enseigner l'informatique à leurs étudiants de premier cycle scientifique, au moment où les étudiants arrivent en masse dans les universités, il nous a semblé intéressant de décrire notre propre expérience, d'expliquer les choix pédagogiques qui ont été les nôtres, pour que notre enseignement de l'informatique en DEUG [Can90b], [Can90c] soit cohérent:

- avec ce qui précède: le lycée.
- avec ce qui suit, la licence et la maîtrise d'informatique, ou un autre second cycle où l'informatique prend une part importante.
- et surtout à lui seul car une majorité de nos étudiants de DEUG ne poursuivent pas leurs études en informatique voire ne poursuivent pas de second cycle.

Nous espérons, grâce à cette présentation, intéresser la communauté informatique à nos travaux et faire réagir ceux qui se sont également investis ou qui ont l'intention de s'investir dans ce type d'enseignement pour qu'ils puissent corroborer, modifier ou critiquer nos choix qui sont essentiellement:

- la mise à disposition de nos étudiants d'un cadre où ils peuvent raisonner proprement pour construire des algorithmes corrects.

- l'utilisation du style fonctionnel et abstrait qui forcera nos étudiants à plus de rigueur.

Actuellement les informaticiens réfléchissent beaucoup sur la didactique de leur discipline comme l'ont démontré

- les colloques de Besançon [SPE88] et de Lille (90) ainsi que les journées de Nantes [SPE90] pour l'enseignement de l'informatique en tant que discipline dans les DEUG scientifiques
- les colloques francophones sur la didactique de l'informatique de Paris [CFD88] et de Namur [CFD90]
- les journées "langages applicatifs dans l'enseignement de l'informatique" de Paris [LAE91] et Rennes où le groupe de réflexion Abalone est créé [Aba91].

De nos différentes réflexions naîtra l'enseignement de demain. Il faut évoluer avec son temps, j'espère que plus tard d'autres réfléchiront pour nous, pour ne jamais cesser de progresser. L'Utopie dont vous parliez lors des journées de Nantes [SPE90] n'est pas loin, J-C. Boussard!

1.2) Rôle du DEUG

L'Université est un endroit où l'on forme des esprits capables d'évoluer.

Cette formation doit commencer dès le DEUG. Le DEUG est le maillon le plus important de l'Université car c'est durant ces deux années que l'étudiant aura reçu une formation scientifique générale qui lui permettra de choisir la poursuite de ses études en connaissance de cause.

1.3) Les informaticiens en DEUG

L'Université joue un rôle fondamental dans la formation des futurs informaticiens car elle est l'un des rares établissements d'enseignement qui forme les étudiants à un haut niveau dans cette discipline. Elle joue également un rôle primordial en dispensant un enseignement d'informatique à la quasi totalité des étudiants des premiers cycles scientifiques.

Nous pensons que ces deux rôles sont liés. Si nous voulons

former des informaticiens il faudra faire découvrir l'informatique à nos étudiants de DEUG. De plus, si nous voulons que la totalité des étudiants puisse plus tard utiliser l'informatique et comprendre ses évolutions, il faudra que nous (informaticiens) leur en donnions les moyens en DEUG car, dans la majorité des seconds cycles, l'informatique n'est considérée que comme un outil. Nous avons donc une tâche difficile car avant ces deux années de DEUG la quasi majorité de nos étudiants n'ont aucun prérequis contrairement aux mathématiques, à la physique ...

Mais voilà, nous croyons sincérement que les informaticiens ont souvent négligé leur cours d'initiation et qu'ils ont, pour la plupart, "enseigné leur discipline à l'envers" c'est-à-dire d'une manière trop descriptive en oubliant l'aspect créatif comme le souligne fort justement P. A. De Marneffe [DMa90]. Il faut dire qu'ils ont des excuses:

* l'informatique est une science jeune, elle est à l'heure actuelle en pleine mutation et un profond malaise habite les informaticiens qui se demandent même si elle est une discipline à part entière.
* le manque d'enseignants en informatique est crucial dans les Universités françaises, où certains cours d'initiation à l'informatique ne sont pas assurés par des informaticiens (mais par des gens qui se disent, souvent, informaticiens).

Il est vrai que tous les informaticiens ont déjà une surcharge de travail importante en deuxième et troisième cycles; de plus, ils doivent constamment recommencer une initiation mal faite précédemment. Mais cela ne doit pas nous pousser à négliger l'enseignement en DEUG, base incontournable des enseignements de deuxième et troisième cycles.

Il est vrai que le temps affecté à l'informatique en DEUG est souvent insuffisant (pas d'enseignement, pas de temps; pas de temps, pas d'enseignant) mais le manque de temps n'excuse pas tout. Nous pensons fortement qu'une bonne initiation doit prendre du temps et que ce temps "perdu" sera profitable à tous pour pouvoir progresser rapidement et en douceur.

Il est surtout vrai qu'enseigner en DEUG n'est pas une chose simple et n'est pas gratifiant pour la carrière d'un informaticien.

Nous, informaticiens, ne savons pas, ou pas bien, enseigner notre discipline à nos étudiants de DEUG. Lorsqu'il ne s'agit pas que d'un enseignement réduit à l'apprentissage d'un langage de programmation, la plupart d'entre nous ont essayé:

- soit de transférer leur expérience du deuxième cycle vers le premier cycle, ce qui semble insuffisant voire incohérent car la finalité n'est pas la même sauf s'il s'agit d'une ré-initiation.

- soit d'appliquer des enseignements de l'informatique qui s'adressent à un public d'étudiants bien différent de celui de DEUG car ils ne poursuivent leurs études qu'en informatique, comme ceux des IUT [CoK86], [ClB84], [Duc84], ou des auditeurs du C.N.A.M [Gre86].

- soit d'appliquer l'enseignement de H. Abelson et G.J. Sussman [AbS89] qui est le premier cours d'informatique au Massachusetts Institute of Technology.

- soit de construire un enseignement spécifique aux DEUG scientifiques comme à Grenoble [BeS90], à Metz [Can90a] et sans doute ailleurs.

Il faut présenter à nos étudiants de DEUG une image "scientifique" de l'informatique. Il s'agit pour nous d'être honnête avec nous-même mais surtout envers nos étudiants. Si nous ne leur enseignons que des techniques de programmation, ils n'auront qu'une image extrêmement réduite et technique de l'informatique, qui perdra ainsi au cours des années son label de discipline aux yeux des étudiants et des collègues des autres disciplines. Cela est dangereux pour notre discipline car nos étudiants n'auront pas tous les éléments en main pour choisir de continuer dans un deuxième cycle et surtout dans un deuxième cycle où l'informatique prend une part non négligeable.

Enseigner l'informatique en DEUG
≠
apprentissage d'un langage de programmation

Les informaticiens ont bien compris cela, mais la plupart de leurs enseignements de DEUG n'ont pas l'écho escompté auprès de leurs étudiants. Au cours des colloques de Besançon [SPE88] et de Lille ainsi que des journées de Nantes [SPE90], certains collègues courageux ont dit qu'ils décourageaient et même dégoûtaient leurs étudiants, d'autres ont même dit qu'aller enseigner en DEUG c'est "aller au charbon". Comment intéresser nos étudiants à notre discipline? Comment, dans le contexte d'un DEUG scientifique où les étudiants sont confrontés à d'autres disciplines, pouvons-nous proposer un enseignement cohérent à lui seul pour participer

pleinement au G(énéral) de DEUG, pour que chaque étudiant d'un DEUG scientifique puisse prétendre avoir reçu une formation informatique?

Ce phénomène de rejet quasi national pour l'informatique n'a pas épargné l'Université de Metz. Nous avons constaté durant ces trois dernières années que les étudiants de DEUG A à dominante Mathématique et Informatique délaissaient de plus en plus la Licence d'Informatique pour un deuxième cycle d'une autre discipline. Moins de 10% des étudiants de la promotion de 1989-90 se sont incrits en Licence d'Informatique. Localement, nous avons réagi en introduisant pour cette année universitaire 1990-91 l'enseignement que nous présentons pour la deuxième année du DEUG. Notre démarche a porté ses fruits: à la fin du premier semestre, plus de 33% des étudiants désirent poursuivre en informatique et plus de 40% hésitent encore entre les mathématiques et l'informatique. A la rentrée 1991-92, environ 50% des diplômés en DEUG ont continué en informatique.

Notre but n'est pas "d'attirer" les étudiants par n'importe quel moyen, nous voulons simplement montrer que si l'enseignement en DEUG est fait correctement d'un point de vue scientifique alors ils choisiront correctement la poursuite de leurs études. Tant mieux s'ils viennent faire de l'informatique, tant mieux s'ils choisissent de faire de l'électronique, tant mieux s'ils continuent à vouloir faire des mathématiques ...

Leur choix ne nous appartient pas.

1.4) Plan de lecture

Nous compléterons cette première partie: "Que faire?" en développant notre parcours personnel, dans le chapitre 2, pour expliquer pourquoi nous nous sommes intéressé à ce type d'enseignement. Nous expliquerons, dans le chapitre 3, ce que doit être pour nous un enseignement d'informatique pour des étudiants d'un DEUG scientifique. L'algorithmique est le fil directeur de notre enseignement dans lequel nous accordons une grande importance à:

- la décomposition des problèmes, analyse arborescente descendante.
- l'utilisation des connaissances scientifiques de nos étudiants (surtout les mathématiques).
- l'analyse applicative et récursive des problèmes.
- l'utilisation du style fonctionnel le plus souvent possible sans négliger le style actionnel.

Nous insisterons également sur le fait que l'écriture simple des

algorithmes doit être la priorité d'un enseignant d'informatique, nous donnerons nos convictions qui sont l'utilisation d'une analyse récursive et l'expression abstraite d'un algorithme pour que sa conception et sa preuve soient simples et convaincantes afin que sa programmation soit la plus sûre et la plus facile possible.

Dans la partie "Réalisation" nous montrons comment notre enseignement est actuellement réalisé. Au niveau du contenu réel de ce cours d'informatique, nous invitons le lecteur à consulter les polycopiés de nos cours de DEUG [Can90b], [Can90c]. Nous présentons les grandes lignes et les points importants et originaux de notre enseignement comme l'importance accordée à:

- une décomposition "créative" des problèmes (chapitre 4)
- l'utilisation de schémas pour en déduire des algorithmes corrects comme le font P.C. Scholl et J.-P. Peyrin [ScP88], [Pey91] ou R.C. Backhouse [Bac89] (chapitre 5).
- l'utilisation d'une approche fonctionnelle et abstraite pour enseigner les listes linéaires [Can91] (chapitre 6).

Dans le chapitre 7, nous donnons l'évaluation d'un certain nombre de contrôles, pour montrer que nos objectifs sont en grande partie atteints. Nous essayerons de donner nos conclusions dans le chapitre 8.

Dans la dernière partie "Ailleurs et demain", nous comparons dans le chapitre 9 notre façon d'enseigner avec d'autres puis, au cours du chapitre 10, nous indiquons nos directions futures (arbre et graphe, optimisation de code, développement de logiciels d'aides pour les étudiants).

Chapitre 2

POURQUOI CETTE THESE ?

Il nous a semblé important de décrire notre parcours d'enseignant d'informatique en DEUG tout particulièrement. Le malaise que nous avions ressenti lors de nos premières années d'enseignement est à la base de nos choix pédagogiques présentés dans cette Thèse. Ce malaise était exclusivement dû au manque de fondement et de "théorie". Nous parlons également quelquefois de notre expérience en second cycle car certaines constatations sur le comportement général des étudiants de licence ou de maîtrise ont guidé nos choix. Comme il s'agit de notre histoire nous prenons volontairement le je pour ce chapitre.

2.1) Nos deux premières années d'enseignement

Le 1er octobre 1984, je suis nommé assistant stagiaire en informatique. Le manque d'enseignants a permis à un jeune assistant d'être chargé de cours, ce qui devait être très rare dans d'autres disciplines. Mes quatre premiers cours étaient:

1 un cours d'initiation à l'algorithmique de base pour les étudiants de DEUG A première année de la filière PIE (Physique-Informatique-Electronique) (PE à l'époque).

* Pour ce cours, commun à tous les étudiants de DEUG A et B première année, nous faisions tous le même cours que D. Méry qui avait choisi d'enseigner "à la nancéenne", en utilisant la méthode dite déductive [Duc84] (seulement pour la composition séquentielle et les boucles). Etant formé par P. Cousot "à la grenobloise" (méthode arborescente descendante), je ne me suis pas trop senti à l'aise (mais mon malaise ne vient pas de là), mais par souci d'harmoniser l'enseignement au niveau du DEUG il fallait que nous nous accordions sur la façon de faire passer notre message.

2 un cours d'algorithmique sur les tris et les listes pour les étudiants de DEUG A deuxième année de la section PIE et TM (Technologie et Mécanique).

* Pour ce cours je devais faire une (petite) initiation à la récursivité, enseigner des tris itératifs sauf pour le quicksort car nous ne pouvions pas faire autrement, et faire une leçon sur les listes avec des algorithmes itératifs.

3 un cours d'initiation à l'algorithmique de base et à la programmation structurée pour les étudiants de DEUG A première année de la filière TM (Technologie et Mécanique) qui était à l'époque le seul enseignement d'informatique des TM 1ère année.

* Pour ce dernier cours, j'étais libre du fait que j'étais le seul à intervenir durant l'année. Le nombre d'étudiants était peu important, par contre le niveau de ce groupe était hétérogène. Il fallait donc sans cesse expliquer de manière simple ce qu'est et ce que fait un algorithme. C'est dans ce cours, ainsi que dans un cours de remise à niveau, que j'ai mis au point la presque totalité de mon premier polycopié [Can90b].

4 Je donnais également des TD d'Algorithmique en Licence et de compilation en maîtrise.

2.2) Bilan au bout des deux premières années

Au bout de deux années d'enseignement le bilan peut s'exprimer de manière un peu abrupte par les termes suivants:

* Les étudiants ne réfléchissent pas: ils se précipitent sur le clavier dès qu'on leur soumet un problème.
* Pour les étudiants, faire de l'informatique, c'est écrire des programmes qui s'exécutent correctement. Qu'importent la façon et les moyens utilisés, seul le résultat compte.
* Pour eux, faire de l'informatique c'est s'asseoir devant une machine ou en posséder une. C'est devant elle que l'on apprend!!!!.
* Ils sont persuadés qu'un programme c'est ce qui s'affiche à l'écran.
* Il n'y a aucune structure dans leurs algorithmes. Ils ne maîtrisent pas l'utilisation des procédures et des fonctions.
* Ils ne maîtrisent pas du tout les boucles, ils ont donc d'énormes problèmes pour corriger leur programme.
* Ceux qui maîtrisent bien les boucles ont par contre des difficultés à s'adapter à la récursivité. Ils pensent "itératif": (modification d'états) quand ils conçoivent un algorithme, ils exécutent un certain nombre de tours dans leur boucle pour montrer que leur algorithme calcule bien ce qu'ils veulent. Dès que les

problèmes se compliquent la récursivité est nécessaire. Ils ne peuvent donc pas trouver de solution, ils sont bloqués.

* Les étudiants utilisent systématiquement le dernier algorithme écrit en cours et le dernier type appris pour essayer de résoudre leur problème.

* Ils conçoivent d'abord des types qui contiendront leurs données et leurs résultats et à l'aide de ces types, ils essayent d'écrire leur algorithme. Si à un moment donné ils se trouvent bloqués, ils ne veulent pas se remettre en question et abandonnent ou forcent en obtenant un algorithme "faux".

* Ils conçoivent les types en fonction de leur lecture, ils font donc des restrictions qui compliquent leur algorithme.

* On fait des démonstrations en mathématique, on écrit des programmes en informatique.

etc....

Pour être plus précis, en enseignant, je me suis rendu compte qu'à chaque fois qu'un étudiant devait écrire un programme ou une procédure, il commençait systématiquement par l'écriture de ce dernier sans réfléchir (sans analyser son problème). Lorsque le problème avait

- une donnée n entière, il écrivait:

```
procedure p(n:integer);
begin
  if
```

- une donnée l une liste linéaire, il écrivait:

```
procedure p(l:liste);
begin
  while l<>nil do
```

Puis il essayait de réfléchir à son problème.

Sans oublier les "horreurs" que je voyais assez fréquemment (à l'époque), nous citerons les quatre suivantes:

1)
```
function f(n:integer):B;
begin
  readln(n);
```

que l'on peut expliquer par la confusion sur les paramètres mais également par le fait que de nombreux algorithmes commencent toujours par une instruction de lecture qui initialise les données

de l'algorithme. Au moment de passer à l'écriture de fonctions les étudiants recopient l'algorithme (actionnel) qui leur permet de calculer la valeur du résultat de la fonction, comme l'écriture fait partie de l'algorithme, elle se retrouve dans la fonction.

2)
```
procedure p(l:liste);
  var l1,l2,l3:liste;
  begin
    new(l1);new(l2);new(l3);
    l1:=l2;l3:=nil;
```

que l'on peut expliquer par la confusion due à new. Les étudiants confondent l et l^ (le pointeur et la place mémoire pointée ou la variable de la pile d'exécution et la variable sur le tas). De plus, lorsqu'ils déclaraient une liste, ils voulaient une place sur le tas pour cette liste même s'ils n'en avaient pas besoin lors de l'exécution, d'où le new systématique.

3) Ainsi que l'utilisation des fonctions comme des procédures et vice-versa.

4) Ce qui était le plus choquant et qui est à la base de mon choix était aussi la mauvaise utilisation de la boucle **while**. A chaque fois qu'un étudiant veut écrire un algorithme, il veut en modifier un qu'il connaît; lorsqu'il y a une boucle il essaye de changer le corps de celle-ci pour atteindre son but, sans toujours parvenir à l'arrêter correctement.

J'anticipe sur la suite pour dire que l'enseignement que j'ai choisi d'effectuer en DEUG a partiellement limité ces "horreurs".

- la première a disparu (1 ou 2 cas en 4 ans).
- la deuxième a disparu grâce à l'utilisation abstraite et fonctionnelle des listes et surtout à la fonction **cons** dont la suite d'actions demande une place sur le tas et met une valeur dans chaque information de cette place.
- la troisième n'a pas totalement disparu; cela est sans doute un héritage de Pascal (fonction trop actionnelle). Rappelons quand même que cette "horreur" est détectée par les compilateurs.
- quand à la quatrième, elle est liée à la difficulté de l'apprentissage des boucles et à l'ambiguité due à l'écriture non récursive d'une boucle alors que son exécution a une définition récursive [Dij76]. Quant à vouloir modifier des algorithmes existants je pense fortement que l'on ne doit pas focaliser l'attention de l'étudiant sur l'écriture d'un algorithme (utilisation d'exemple type) mais plutôt sur la relation de récurrence similaire qui permet d'obtenir

l'écriture de l'algorithme. Je n'ai pas la prétention de dire que mes étudiants ne font plus du tout d'erreurs dans l'écriture de leurs algorithmes utilisant des boucles, par contre j'ai essayé de situer la réflexion de l'étudiant pour la conception de ces algorithmes à un autre niveau:

* en utilisant des propriétés et des schémas itératifs démontrés corrects en DEUG première année. Il en déduit assez facilement des algorithmes corrects, donc des programmes. La programmation renforce les notions apprises durant les cours, au lieu d'installer un doute souvent néfaste.
* en utilisant une analyse récursive (relation de récurrence) en DEUG deuxième année, cette analyse lui assure la correction et la terminaison de l'algorithme

Durant l'année universitaire 1985-86, j'ai pris la responsabilité du cours et des TD de compilation de maîtrise. Cette expérience nouvelle a fait ressortir les points suivants: les étudiants de maîtrise

- étaient réticents à faire des démonstrations mathématiques rigoureuses.
- écrivaient directement leurs programmes sans faire d'analyse.

De plus, lorsque j'ai corrigé leurs projets, leurs programmes n'avaient aucune structure, il y avait même un programme avec uniquement des procédures et des fonctions sans aucun paramètre donnée et résultat!!!. Il fallait réagir ...

2.3) L'introduction de l'approche fonctionnelle pour les listes

En juin 86, avec D. Méry, nous prenons en charge le cours et les TD d'algorithmique de licence d'informatique, malgré une charge déjà importante d'enseignement. Pour ce cours, seule la récursivité (au début nous pensions surtout à la formulation récursive) doit servir pour concevoir ou comprendre des algorithmes. Il faut donc l'enseigner et l'appliquer aux listes et aux arbres. Cela nous savions le faire, nous pensions surtout aux procédures avec passage par variable pour faire des insertions et des suppressions de manière physique dans des listes ou des arbres (formulation récursive actionnelle).

A l'époque, D. Méry, P. Bellot et moi-même réfléchissions beaucoup à la façon d'enseigner l'informatique. P. Bellot nous a parlé de l'approche fonctionnelle des listes linéaires de J.F. Perrot. Nous avons été convaincus du bienfait d'une telle approche qui devait simplifier la compréhention et donc faciliter la résolution de nombreux problèmes des étudiants de licence. Nous

l'avons donc adoptée et adaptée.

Pour moi, qui était chargé d'enseigner les listes, l'approche fonctionnelle me permettait d'avoir une idée abstraite des listes et de concevoir tous les algorithmes sur celles-ci sans aucun problème. Tout s'enchaînait parfaitement, l'approche procédurale devenait encore plus simple. Mes problèmes au niveau de l'enseignement des listes, donc des arbres (même approche) n'existaient plus, il devait en être de même pour les étudiants. De plus nous leur demandions de faire leur projet avec la règle suivante: Pas de boucles, pas de tableau. Pour moi, même si quelques programmes n'étaient pas toujours parfaits, même si parfois la récursivité était mal utilisée, cela n'avait pas trop d'importance: mon objectif était atteint, un choc psychologique avait eu lieu. L'acquis serait profitable plus tard.

Cette expérience concluante m'a poussé à procéder de même en DEUG. Pour cela, il fallait faire plus de récursivité pour pouvoir l'utiliser de façon naturelle avec les listes. Il fallait également, pour répondre aux demandes des responsables des filières PIE et TM, pouvoir éliminer la récursivité pour obtenir des algorithmes itératifs (Basic, Fortran) nécessaires aux physiciens et aux mécaniciens et technologues. Ces derniers voulaient que je fasse du Basic. Il en était hors de question: à l'époque les PIE et les TM étaient pris de plein droit en licence d'informatique, il fallait donc que ce cours puisse leur montrer ce que pouvait être l'informatique scientifique. De plus, l'année suivante, les mécaniciens ont demandé l'habilitation d'une MST CFMAO dans laquelle l'informatique prenait une part importante: ce cours sous cet aspect leur était donc plus que nécessaire. Les étudiants de DEUG TM2 qui voulaient poursuivre en MST ont été très heureux d'avoir suivi un tel cours. J'ai eu moi-même des échos positifs provenant d'étudiants et même de collègues non informaticiens. Cela m'a conforté dans mes choix, je les ai affinés d'année en année, pour me sentir plus qu'à l'aise dans un amphithéatre rempli d'étudiants. Pour quelqu'un qui ne pensait pas être un enseignant, je crois que je me suis trompé. En tout cas j'ai tout fait pour que cela soit plus facile pour moi et les étudiants en ont également profité.

2.4) Les colloques "informatique discipline en DEUG"

Arrive le colloque de Besançon. Comme j'ai pris des responsabilités administratives au niveau de l'enseignement, telles que: responsable pédagogique en DEUG (pour l'informatique), directeur des études (pour l'informatique), directeur adjoint du Département de mathématique et d'informatique (pour

l'informatique) ainsi que représentant informaticien au conseil de l'UFR MIM (c'est beaucoup pour un assistant!), c'est à moi de représenter l'informatique des DEUG scientifiques de l'Université de Metz à ce colloque. J'y apprends que c'est à Metz, que l'on fait le plus d'informatique en DEUG, surtout pour les MIE (Mathématique -Informatique-Electronique). J'y apprends également que les informaticiens de France ont d'énormes problèmes pour être reconnus au niveau des DEUG, et pour intéresser leurs étudiants, en un mot ils ne savent pas trop quelle informatique enseigner et surtout de quelle façon enseigner cette discipline à leurs étudiants.

Je reviens à Metz un peu déçu car je pensais que l'on m'apprendrait plein de choses. Je commence donc à réfléchir plus sérieusement à ce que doit être un enseignement d'informatique en DEU; mes objectifs étant déjà à l'époque:

- la décomposition pour structurer le plus rapidement les programmes des étudiants de DEUG première année.
- apprendre aux étudiants de DEUG deuxième année la rigueur (récurrence et récursivité) et à écrire des algorithmes généraux (abstraits) puis définir des types qui leur permettront de programmer au mieux leurs algorithmes

J'ai ainsi découvert mon intérêt pour la didactique de ma discipine et cet intérêt ne m'a plus quitté.

Puis fin mars 1990, SPECIF réunit les informaticiens à Nantes pour débattre de l'enseignement de l'informatique en tant que discipline dans les premiers cycles scientifiques. Durant ces journées, il était question d'écrire un livre blanc pour l'enseignement de l'informatique en DEUG; pour moi, il s'agissait d'écrire un livre d'informatique. Je ne voulais surtout pas que mon expérience reste dans ma tête sans être connue. Il fallait mettre l'imprimante en marche. Je décide donc d'écrire un cours d'algorithmique pour les DEUG scientifiques, il aura l'avantage de servir aux étudiants et pour moi de faire un bilan [Can90b], [Can90c].

A Nantes, j'expose, dans la commission "contenu", ce qui est fait à Metz en informatique dans les DEUG des UFR scientifiques. J.-C. Boussard propose dans le compte-rendu de cette commission de faire de l'informatique à la "messine". A la demande de M. Lucas je fais un exposé [Can90a].

Qualité et quantité vont-elles de pair?

Chapitre 3

NOS CONVICTIONS

3.1) Vers une définition d'un enseignement d'informatique en DEUG

Si nous ne devions garder qu'un unique objectif en DEUG ce serait celui-ci:

Il faut apprendre à nos étudiants à décomposer leurs problèmes

pour réaliser cet objectif notre enseignement doit donc contenir un enseignement d'algorithmique. Nous pouvons donc affiner notre objectif:

> **Il faut apprendre à nos étudiants à décomposer leurs problèmes**
> **pour**
> **qu'ils structurent le plus rapidement leurs algorithmes**
> **dans un souci de lisibilité et de correction**

Il faut, dès le premier cycle, expliquer aux étudiants comment l'on construit un algorithme correct, lisible donc compréhensible par simple lecture. Ce ne sont pas des exécutions même nombreuses d'un algorithme qui vont montrer sa correction [Dij76]. Il faut donc toujours expliquer le pourquoi et le comment et justifier tout ce que nous faisons le plus simplement du monde. L'informatique est une école de rigueur, ne l'oublions pas. Vouloir l'algorithme qui va le plus vite et qui utilise le moins de place est dépassé, en tout cas dans une première phase d'obtention d'un algorithme correct. Si nous donnons ces objectifs à l'étudiant, il voudra les réaliser à chaque fois, il aura donc trois choses à faire en même temps en dehors de la compréhension de son problème, qui sont:

* concevoir un algorithme
* gagner en efficacité
* gagner en place

ces objectifs passeront donc avant la correction de l'algorithme. Même si nos étudiants ont de bonnes idées au départ, l'algorithme obtenu sera faux par endroits. Nous ne formons plus des esprits mais des machines chaotiques qui essayent de construire des programmes.

Par contre, dans une deuxième phase, nous pourrons expliquer à nos étudiants comment on peut optimiser certains algorithmes en les transformant. Avec un algorithme correct et des transformations correctes nous serons ainsi sur d'obtenir des algorithmes équivalents, donc corrects.

Nous connaissons un moyen d'écrire des algorithmes clairs et corrects: c'est utiliser la démarche arborescente descendante, c'est-à-dire décomposer le problème en un ou plusieurs sous problèmes plus simples et ce jusqu'à n'avoir à résoudre que des problèmes simples (Descartes, Discours de la méthode + décomposition récursive). Nous sommes en présence, ne l'oublions pas, d'étudiants scientifiques qui ont tous fait peu d'informatique mais qui ont suivi tout au long de leur scolarité des cours de mathématiques alors profitons de leurs connaissances, comme le conseillent les enseignants du C.N.A.M [Gre86]. Nos étudiants sont capables de comprendre, le fait de les faire réfléchir, de leur faire démontrer des choses avant l'écriture d'un algorithme, ne peut que les rassurer sur la correction de celui-ci. Les plus scientifiques d'entre eux apprécieront davantage et cela pourra les pousser plus volontiers à s'intéresser à l'informatique. Les moins rigoureux seront obligés de le devenir un peu plus et pourront progresser avec l'effet de groupe et la bonne utilisation des machines. En tout cas nos étudiants auront appris un savoir et un savoir faire (faire) correct [Pey91] dans cette discipline pour pouvoir, dans un premier temps, l'utiliser et, dans un deuxième temps, évoluer en ayant la possibilité de comprendre les concepts nouveaux et les techniques futures.

L'écriture simple des algorithmes pour une meilleure compréhension de ceux-ci doit être la priorité d'un enseignant d'informatique.

Nous serons donc en accord avec les "trois coeurs de l'informatique" de C. Duchâteau [Duch90]:
1) **Enfermer le "sens" dans la "forme"**: car avant d'enfermer le sens il faut le comprendre.

2) **Faire faire en différé**: car les algorithmes (textes) sont des formes qui vont construire (**faire en différé**) des solutions et concevoir des algorithmes c'est savoir construire (**faire**) des algorithmes.
3) **Travailler avec les "boites" et non les "contenus"**: c'est la base même de la décomposition des problèmes.

Nous sommes donc amené à poser l'équation un peu brutale qui sert de titre au paragraphe qui suit et que nous tenterons d'affiner:

3.1.1) Informatique=Algorithmique

L'étude et la construction d'algorithmes est la matière fondamentale de notre discipline comme le soulignent fort justement:

- L. Goldschlager et A. Lister [GoL86] car sans eux il n'y aurait pas de programmes et de plus ils sont indépendants des langages de programmation et des ordinateurs.
- R. Sedgewick [Sed91] "Le terme algorithme est employé en informatique pour décrire une méthode de résolution de problème programmable sur machine. Les algorithmes sont la "matière" de l'informatique et sont l'un des centres d'intérêt de la plupart, sinon de la totalité, des domaines de cette science."
  " il faut enseigner l'algorithmique avant l'étude de domaines spécialisés de l'informatique"

L'indépendance de l'algorithme par rapport au langage de programmation et de la machine est importante pour un enseignant car il pourra avoir les caractéristiques qu'il désire, entre autres les styles actionnel, fonctionnel ...

Le fait que la conception fasse partie intégrante de l'algorithme nous plait particulièrement lorsque l'on sait comme le signale J. Arsac [Ars90] que la plupart des ouvrages d'initiation "algorithmique et programmation" ne sont en fait que des ouvrages de programmation PASCAL.

Comme tous les domaines de l'informatique utilisent ou définissent des algorithmes il est donc tout à fait normal de commencer par apprendre cette matière à nos étudiants de DEUG.

Mais pour changer l'idée préconçue que les étudiants ont de l'informatique, pour réorienter ou (re)créer leur motivation, nous

avons choisi d'enseigner l'algorithmique et surtout de montrer à nos étudiants comment "naissent" les algorithmes corrects en leur expliquant le besoin et le principe de ceux-ci et en leur démontrant le plus souvent possible ce que l'on affirme. Nous comptons donc par son biais initier nos étudiants à analyser et décomposer leurs problèmes pour en déduire des algorithmes corrects. Nous rejoignons donc l'aspect "créatif" d'un enseignement de l'algorithmique cher à P.A. De Marneffe [DMa90].

Comme pour la conception d'un algorithme, nous faisons souvent appel aux mathématiques comme la logique (propriétés démonstrations, ...) ou les exemples (connus des étudiants scientifiques) nous pouvons schématiser l'équation Informatique =Algorithmique de la manière suivante:

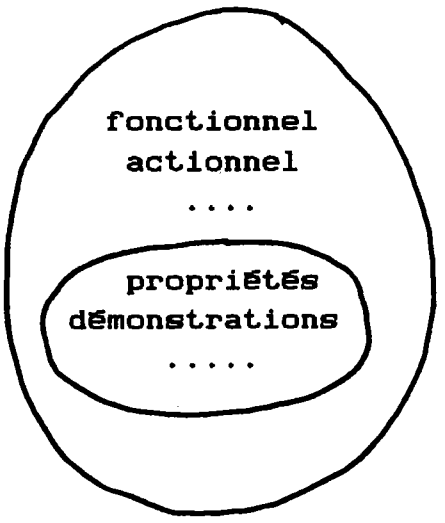


Schéma: Informatique=Algorithmique

L'enseignement de la logique n'est pas assumé par les mathématiciens, nous devons prendre en charge l'initiation des rudiments de logique [SPE90], [Aba91].

3.1.2) L'algorithmique: fil directeur de l'enseignement

L'algorithmique étant partout en informatique à Grenoble [BeS90] comme à Metz [Can90a] nous profitons des exemples pour compléter la formation des étudiants de DEUG. A Metz ces compléments sont en:

- Architecture et système:
  - * codage et algorithmes de codage des entiers à l'aide de tableau de bits.
  - * en décrivant les mécanisme d'allocation mémoire (notion de

pile d'exécution, notion de tas).

- Informatique de gestion:
  * algorithmes sur les enregistrements et les fichiers.
- Base (ou plutôt Banque) de données:
  * algorithmes de recherche, d'insertion de suppression dans des arbres (listes par défaut).
- Compilation:
  * construction des algorithmes de lecture d'un entier, d'un identificateur (liste de caractères) à l'aide de la définition récursive du langage qui les définit.

Cette liste non exhaustive peut être complétée par des interventions ponctuelles dans d'autres domaines si le cours d'algorithmique ne permet pas de l'aborder correctement. Depuis cette année universitaire 1991-92 nous avons mis en place pour les étudiants de deuxième année de la filière Mathématique Informatique Electronique une intervention en:

- Téléinformatique (présentation des différentes couches d'un réseau et utilisation d'un réseau)
- Conception assistée par ordinateur (présentation des concepts et utilisation du logiciel SACADO développé par le LRIM [Gar87], [Gal188].)

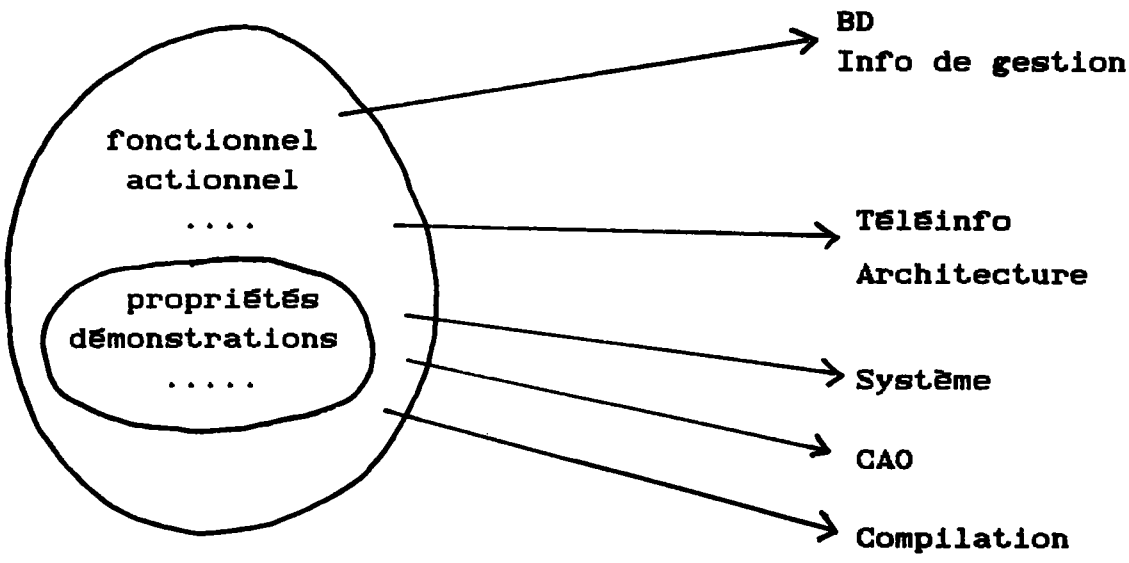


Schéma: Informatique=Algorithmique: fil directeur

3.1.3) Informatique=Algorithmique+Programmation

Comme un algorithme peut donner naissance à un programme et que celui-ci peut s'exécuter sur une machine, il faudra que l'étudiant

puisse se servir de celle-ci au moins pour éditer, compiler et exécuter ses programmes et gérer ses fichiers. Il est impensable de ne pas se servir d'une machine. Nous devons profiter du cours d'algorithmique pour apprendre à nos étudiants un langage de programmation pour:

- qu'ils puissent "essayer" leurs algorithmes, cela les rassurera sur la correction des algorithmes corrects
- qu'ils sachent se servir d'une machine et connaître ses périphériques.

De plus si l'utilisation de la machine est bien gérée par l'enseignant les erreurs des étudiants pourront être source de réflexion et de correction (seul ou par dialogue étudiant-étudiant ou étudiant-enseignant). L'apprentissage par la correction en cours sera complétée par l'apprentissage par l'échec mais à des niveaux différents ce qui tendra à banaliser le rôle de la machine sans l'écarter et minimisera l'effet néfaste de la "sanction immédiate" de la machine [SPE88].

Nous abordons, grâce à l'utilisation de la machine, des notions:

- de système.
- de machine (il faudra la décrire) cela complétera l'enseignement de l'architecture
- (utilisation) de logiciel (éditeur, compilateur ...)
- de programmation applicative (avec le style fonctionnel)
- de programmation impérative (avec le style actionnel)
- etc ...

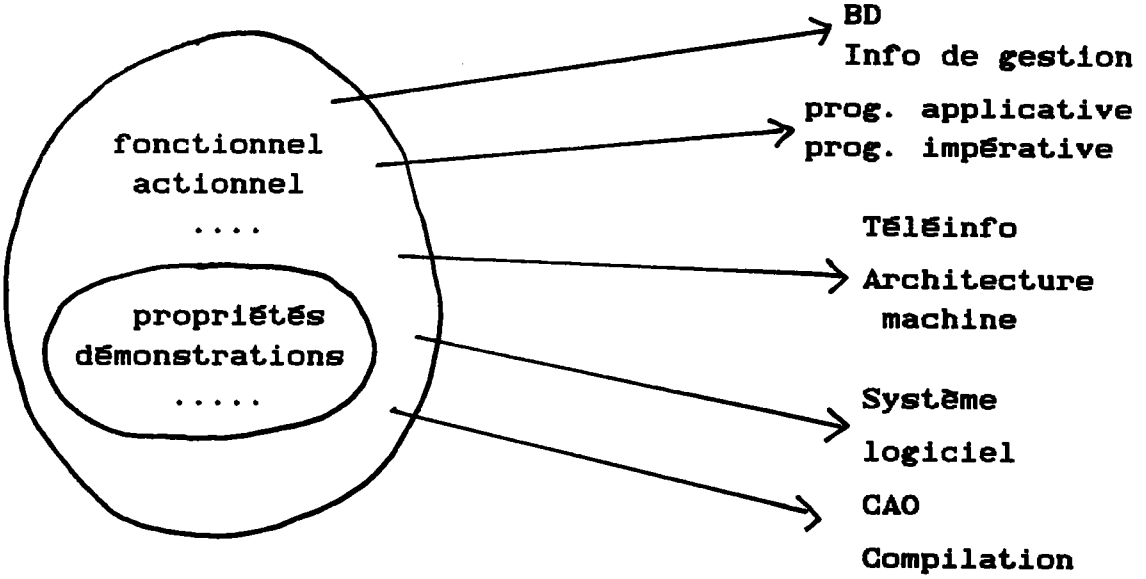


Schéma: Informatique=Algorithmique+programmation

3.2) Quelle algorithmique? Quel style?

C'est actuellement le coeur du débat [CFD90], [LEA91], [Aba91], faut-il choisir le style actionnel ou le style fonctionnel? ou plutôt un langage impératif ou un langage applicatif? voire PASCAL ou SCHEME?

Nous pensons que le style actionnel est une abstraction du langage machine dont on se sert pour résoudre ses problèmes, alors que le style fonctionnel est une réalisation d'un mode de pensée qui peut être interprété par la machine. Cette interprétation permettra une communication entre l'homme et la machine sans trop de contraintes liées à celle-ci.

Avant de répondre à la première question, nous allons d'abord tenter de définir ce que nous appelons analyse "itérative".

3.2.1) Analyse "itérative"

Une analyse "itérative" est une analyse d'un problème qui nous permettra d'en déduire (ou de construire) un algorithme itératif.

3.2.1.1) Recherche de relation de récurrence

Nous sommes de plus en plus nombreux à penser que l'analyse itérative ou plutôt la recherche d'un invariant est la recherche de relation de récurrence [ScP88], [Pey91], [Gre86] ou structure récursive [Can91c]. La recherche de relation de récurrence (resp la structure récursive) nous permettra à l'aide de schémas ou de règles de construire un algorithme. L'aspect "créatif" n'est pas exempt de cette vision de l'analyse itérative. De plus, comme à partir de relation de récurrence, on peut déduire une expression récursive on peut également qualifier cette analyse d'"analyse récursive". Nous pouvons donc dire qu'une même analyse pourra aboutir à deux écritures algorithmiques.

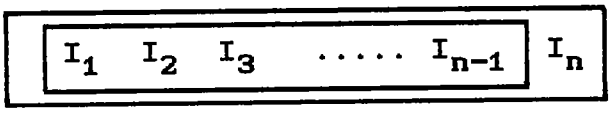
D'ailleurs, si l'on regarde ce qui se passe dans la suite des n instructions exécutées pour n tours dans une boucle:

$$I_1 \quad I_2 \quad I_3 \quad \ldots\ldots \quad I_{n-1} \quad I_n$$

un invariant de la boucle est une propriété vraie avant et après chaque étape. Comme l'invariant de la boucle avant l'exécution de I_n exprime l'invariant après l'exécution des n-1 premières instructions

$$I_1 \quad I_2 \quad I_3 \quad \ldots\ldots \quad I_{n-1}$$

nous avons donc la structure récursive gauche suivante



Un invariant de la boucle doit normalement permettre de trouver la solution que l'on cherche à calculer à l'aide de notre boucle. Si la relation de récurrence est du type:

$p(x_0)=f_0$ pour x_0 vérifiant une condition (d'arrêt)
$p(x)=f(x,p(g(x)))$ sinon

nous avons deux possibilités de résoudre le problème:

1) Soit on connaît x_0 et g^{-1} donc $x=g^{-n}(x_0)$ et à ce moment l'invariant est bien adapté aux relations de récurrence et il n'y a aucun problème pour trouver l'algorithme "itératif"

2) Soit on ne les connaît pas et donc on ne peut construire que la suite des différentes valeurs de la boucle mais à l'envers c'est-à-dire:

$$x,\ g(x),\ g^2(x),\ \ldots\ ,\ g^n(x)$$

où $g^n(x)$ vérifie la conditions d'arrêt,
dans ce cas celà est extrémement dur de trouver le bon invariant alors que l'algorithme récursif est beaucoup plus simple à écrire.

3.2.1.2) Recherche d'un invariant

D'autres pensent que trouver l'invariant suffit pour construire l'algorithme [Bac89]. Il est vrai qu'il existe une relation forte entre algorithme et invariant. De toute façon trouver un invariant pour en déduire un algorithme est un acte créatif aussi difficile, voire plus, que d'arriver à trouver une (bonne) idée pour écrire l'algorithme. C'est tellement vrai que certains [Lan90] donnent l'algorithme (à trouver) et montrent que l'invariant est bon donc que l'algorithme est bien celui que l'on voulait. La notion même de conception d'algorithme est perdue au profit d'une méthode de preuve de programme. Il faut éviter le travers suivant: comme on donne l'algorithme et l'invariant aux étudiants on ne leur expliquera jamais comment on trouve cet invariant. Plus tard ils procéderont de même ce qui peut être extrémement dangereux pour notre discipline. De plus, l'expérience montre que cela déroute

les étudiants car le langage algorithmique et celui des assertions se mélangent ou se superposent. Pour prouver la correction des algorithmes, on utilise le théorème de récurrence (pour montrer les relations de récurrence!!) lorsque l'étudiant passera au style récursif il peut être déroutè, une fois de plus, par la simplicité de l'écriture des algorithmes et de leurs corrections.

3.2.1.3) Analyse opératoire

Vu la façon dont les étudiants analysaient leurs problèmes, il doit exister une troisième manière d'obtenir des boucles donc une troisième "analyse itérative" qui consiste à raisonner en terme de modification d'état des variables. Les étudiants ont donc une vision trop opératoire (actionnelle) de ce qu'est une boucle et en général ils font faire "quelques tours" à leur algorithme pour montrer sa correction. Nous ne savons pas d'où pouvait provenir une telle "analyse" en tout cas elle n'est pas l'oeuvre d'un informaticien ou alors une conséquence d'un enseignement réduit à un apprentissage d'un langage de programmation.

3.2.2) Nos devoirs d'enseignant d'informatique

Il est de notre devoir d'enseignant:

- de ne pas écarter le style actionnel pour l'écriture d'algorithme (variable, conditionnelles, itérations) car c'est celui utilisé par la plupart des langages de programmation dans le monde du travail et c'est aussi celui de la machine.
- d'enseigner l'analyse applicative et récursive des problèmes et le style fonctionnel d'expression des algorithmes, qui est à notre avis le plus fiable pour l'écriture d'algorithmes corrects.

Par contre, au niveau de la mise en oeuvre d'un tel enseignement, à en croire les livres d'initiations pré-cités, ces deux styles sont presque incompatibles:

Le premier type d'enseignement est celui adopté par la majorité des enseignants informaticiens mais, comme nous l'avons signalé, il n'est pas toujours bien ressenti par les étudiants.

Le deuxième type d'enseignement est possible si l'on utilise un langage applicatif. L'ouvrage de H. Abelson et de G.J. Sussman [AbS86] peut servir de support à cet enseignement [LAE90], [Aba91]. Les auteurs n'ont pas écarté le style actionnel, ils apprennent à leurs étudiants comment écrire des fonctions

récursives terminales pour que l'interprète du langage applicatif qu'ils utilisent optimise l'exécution de ces fonctions. Dans le même but, après avoir assimilé les fonctions, ils donnent aux étudiants les moyens de modifier les chaînages des listes (pointeur + effet de bord) pour optimiser l'allocation mémoire et les temps d'exécution de leurs algorithmes.

Comme nous le laissions entendre lors de la définition d'un cours d'informatique de DEUG et comme le conseille J.P. Peyrin [Pey91], nous ne devons pas écarter un style au dépend d'un autre. Et cela peut commencer dès l'initiation à l'algorithmique de base.

3.2.2.1) Dès l'algorithmique de base

Notre objectif en DEUG première année est que nos étudiants structurent le plus rapidement possible leurs algorithmes, il faut pour cela qu'ils puissent décomposer des problèmes, ils doivent donc connaître les moyens qui permettent de le faire, c'est-à-dire:

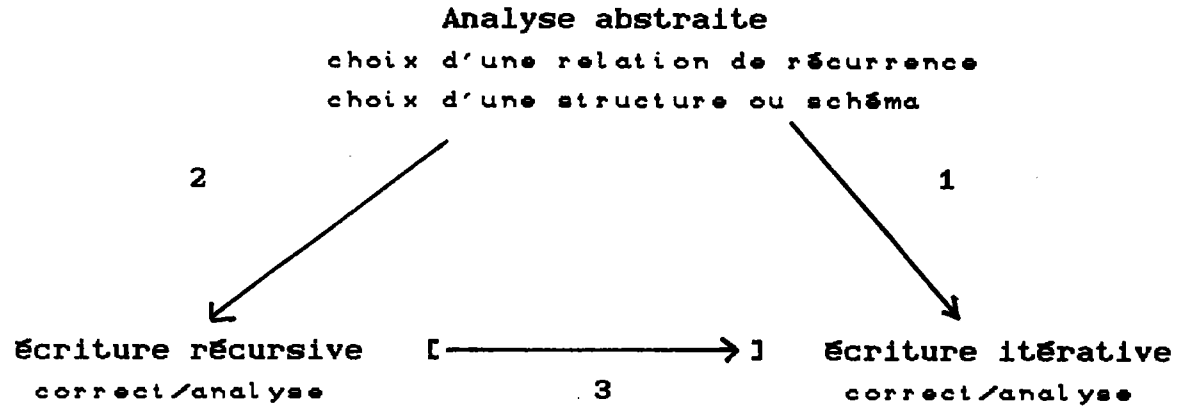
- identifier correctement les données et les résultats du problème à résoudre.
- typer ces données et résultats.
- trouver les relations entre les données et les résultats.
- trouver la suite d'opérations ou fonction qui permet de calculer les résultats en fonction des données (réalisation des relations).
- réaliser les opérations dans un langage algorithmique impératif (affectation, conditionnelle, itération, composition séquentielle).
- représenter au niveau de ce langage algorithmique le type des données et des résultats.

Si on utilise le style actionnel il faudra préserver le plus possible les expressions qui permettent de définir les relations il est nécessaire de ne pas trop découper les expressions comme celà est fait par A. Ducrin [Duc84] car on perd tout l'aspect fonctionnel des relations.

Il faudra également que nos étudiants connaissent l'équivalent, dans un langage de programmation (si possible algorithmique) des opérations citées ci-dessus ainsi que les moyens de construire leur type.

3.2.2.2) Importance de l'analyse

Pour la conception d'algorithmes, le style n'est pas le point capital. Nous avons voulu situer la réflexion des étudiants à un autre niveau d'abstraction, celui de l'analyse, pour qu'ils puissent réduire et exprimer les complexités de leur problème. Nous avons essayé de leur donner un cadre où ils peuvent raisonner proprement (propriété, analyse abstraite ...). Il fallait donc que nous leur fournissions des moyens d'expression, pour qu'ils obtiennent des algorithmes et donc des programmes corrects à partir de leur analyse. Comme le conseillent H. Abelson et G.J. Sussman [AbS86], nous donnerons à nos étudiants deux expressions possibles qui sont l'écriture de texte itératif et l'écriture de texte récursif. Les moyens d'expressions sont représentés par le schéma suivant:



1) Pour les étudiants de DEUG première année, nous avons privilégié l'écriture itérative. Par contre pour l'analyse de ses problèmes nous avons situé la réflexion de l'étudiant au niveau de

- définition d'une propriété que doit respecter l'algorithme ou une de ses données ou résultats [ScP88], [Bac89].
- l'identification ou l'utilisation d'un schéma de parcours (traitement et calcul) [ScP88].

Une fois cette réflexion terminée, l'étudiant sait qu'il obtiendra un algorithme correct avec les moyens que nous lui avons donnés.

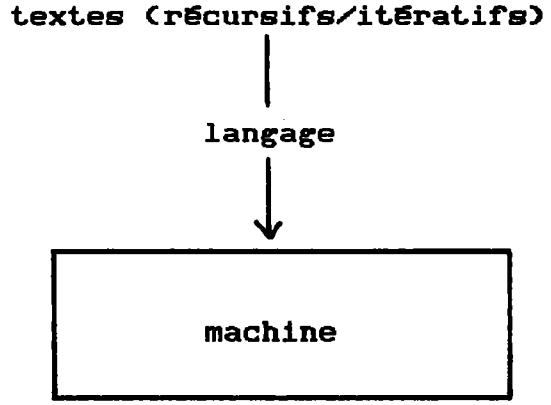
2) Nous avons, tout au moins pour les étudiants de DEUG deuxième année, privilégié l'écriture récursive et cela pour plusieurs raisons:

- Pour bien initier les étudiants à la récursivité, il faut absolument que ceux-ci fassent confiance à la récursivité
- Le style récursif est beaucoup plus fonctionnel et le passage de l'analyse à l'écriture récursive est beaucoup plus facile pour les étudiants. Comme les langages de programmation actuellement utilisés pour l'enseignement de l'informatique en DEUG permettent la récursivité, l'analyse pourra donner naissance à un algorithme puis donner lieu à l'écriture d'un programme, cela permettra donc de tester à l'aide de la machine plus facilement l'analyse du problème. Le langage de programmation confirme l'analyse. Ce sera un garant supplémentaire, ce qui accentuera la confiance des étudiants dans ce style d'écriture des algorithmes.

3) Pour les relations de récurrence qui ne permettent pas une écriture itérative simple, le mode d'expression qui permet d'obtenir l'écriture récursive et des règles d'élimination de la récursivité (règle d'équivalence de schéma récursif et schéma itératif) pourront servir de moyen d'expression entre une analyse et l'écriture itérative.

3.2.3) Choix d'un langage

A partir des textes (algorithmes) obtenus nous devons choisir un langage de programmation qui permettra d'exécuter les algorithmes de nos étudiants.



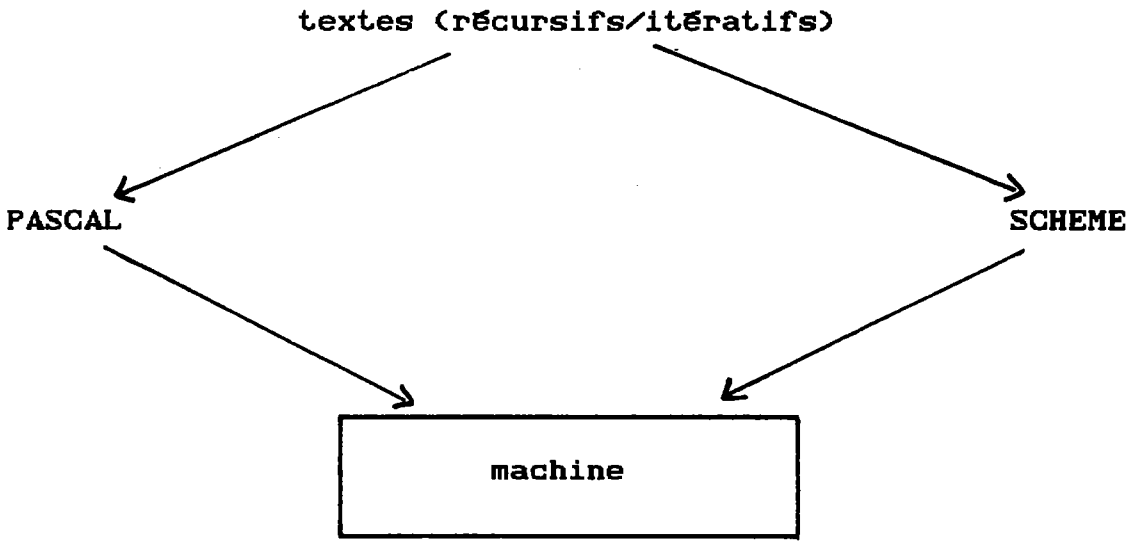
A propos de guerre des langages

Nous donnerons dans le chapitre 8 d'autres raisons sur notre choix du langage. Il nous fallait un langage qui permette aux étudiants d'écrire leurs algorithmes sous forme de programmes. Ce

langage devait avoir les caractéristiques suivantes:

- un langage impératif pour permettre le style actionnel.
- un langage applicatif pour permettre le style fonctionnel.
- un langage typé pour contraindre les étudiants à typer leurs paramètres.
- un langage qui permet de structurer (décomposition) les programmes.
- un langage qui permet d'utiliser la récursivité.
- un langage qui permet de définir des types inductifs.

Alors impératif ou applicatif? PASCAL ou SCHEME?



Nous avons choisi PASCAL, parce qu'à l'époque de ce choix (1984) la question ne se posait pas. Par contre, nous avons assumé ce choix, nous avons fait en sorte que nos étudiants puissent définir des types inductifs. Nous montrons [Can91] à nos étudiants comment

- raisonner proprement sur les listes linéaires (analyse abstraite, applicative).
- déduire de ce raisonnement un algorithme dans un style fonctionnel.
- on peut réaliser au niveau du langage PASCAL les types inductifs à l'aide des pointeurs.

Pour nous, il n'est pas question de guerre des langages mais bien de la façon d'aborder les problèmes et donc de pouvoir les résoudre. Nous avons fait un compromis avec PASCAL: nous nous sommes servi de lui pour apprendre l'algorithmique à nos étudiants, jamais pour faire un cours de programmation PASCAL. Ce n'est pas la peine de prendre SCHEME pour une question de syntaxe [Pey91] la forme des algorithmes n'est pas un aspect fondamental

de l'algorithmique.

3.3) L'utilisation de la récursivité et du théorème de récurrence

Voici la conviction forte qui a fait changer notre enseignement en 1986:

Le théorème de récurrence qui nous assurera la correction et la terminaison des algorithmes récursifs est préférable à une tentative d'exécution d'une boucle pour montrer qu'elle calcule bien ce que l'on voulait. Le principe de récurrence sera également d'une aide précieuse pour concevoir des algorithmes corrects, structurés et lisibles. La récursivité a donc pris sa place, à l'époque, dans l'enseignement de DEUG deuxième année, après que nos étudiants aient appris tous les éléments prérequis qui permettent de l'aborder (décomposition).

Nous savions également que le théorème de récurrence permet de prouver des itérations mais comme nous l'avons déjà signalé l'introduction d'invariants et d'un système de preuves dans un enseignement de DEUG est un chemin difficile (pour nos étudiants) sur lequel nous n'avons pas voulu nous engager.

Lorsque nous abordons la récursivité, les étudiants (surtout les non "matheux") acceptent plus volontiers l'apport scientifique que représente le théorème de récurrence et acceptent donc de faire des démonstrations.

Contrairement aux étudiants de second cycle, les étudiants de DEUG sont plus malléables et donc plus réceptifs à notre message. Ils essayeront d'avoir une analyse (récursive) de leurs problèmes et ils la valideront en utilisant la progammation récursive. Nous les aiderons en cela en leur donnant des schémas d'algorithmes récursifs et surtout en leur démontrant toujours ce qu'on leur affirme. Ils n'auront plus peur de la récursivité.

Aujourd'hui cette conviction est toujours forte

Pour pouvoir faire des analyses correctement il faut que nos étudiants aient bien compris le principe de récurrence. Nous pouvons donc nous servir d'un cours d'initiation à la récursivité pour bien initier nos étudiants à ce concept fondamental en informatique. L'écriture récursive nous forcera à rester le plus proche possible du style fonctionnel.

3.4) Algorithme abstrait

Cette unique conviction en a fait apparaître d'autres de manière tout à fait naturelle. Pour que la conception de l'algorithme soit la plus simple possible, ce dernier doit être le plus abstrait possible, et il ne doit surtout pas dépendre des choix trop hâtifs des types pour stocker les valeurs des données et des résultats. Par contre, l'analyse d'un problème fera apparaître des opérations de base sur les données, et ce sont ces opérations de base qui vont décider du choix du type. La réalisation d'un programme ne sera donc qu'une réalisation de ces opérations de base dans le langage de programmation utilisé. La programmation vue sous cet aspect devient presque un jeu d'enfant.

Rappelons l'idée de base de M. Lucas, J-P. Peyrin et P-C.Scholl: [LPS83] "la découverte d'un algorithme repose toujours sur la mise en évidence des données qu'il doit traiter et sur la recherche d'une formulation adéquate de ces traitements" Il ne faut surtout pas lire "trouver des types pour les données et ensuite trouver des instructions qui réaliseront l'algorithme".

Nous retrouvons également cette idée chez les auteurs d'Anna GRAM dans leur paradigme REPRESENTER [Gra86].

3.4.1) Un abstrait concrétisable

Il n'est pas question pour nous d'enseigner une "informatique bourbakienne"; le fait d'avoir une vision abstraite d'un algorithme permettra aux étudiants de mieux le comprendre, de le construire en étant davantage sûrs de sa correction et donc de mieux programmer.

Comme nous l'avons déjà dit, ce sont les opérations sur les types qui vont décider du choix de leur construction. Nous irons même plus loin: un programme n'est qu'une représentation d'un algorithme. Si l'algorithme est correct et si la représentation l'est également, il en sera de même pour le programme. Que pouvons-nous demander de mieux?

L'équation de Wirth "Program = Data structure + Algorithm" peut être remplacée par celle-ci:

Programme = Représenter(Algorithme ⊢—— Structures de données)

où ⊢—— indique que l'on doit déduire de l'algorithme les

structures de données qui lui sont le mieux adaptées.

Donc nous devons commencer à analyser notre problème, l'analyse montrera l'existence de type, dans un premier temps nous nous contenterons de lui donner un nom (typage faible), la poursuite de l'analyse affinera la "structure" de ce type en identifiant sur ce type des fonctions de base.

De plus, avec cette manière de procéder, nous pouvons obtenir les programmmes les plus rapides (par rapport à l'algorithme), en réalisant le type de telle manière que les opérations de base sur celui-ci soient les plus rapides possibles.

3.4.2) Détection et utilisation abstraite des types

Dans cet objectif, il faudra donc initier les étudiants à la décomposition arborescente des problèmes et commencer à définir et surtout à utiliser les types de manière abstraite. Prenons un exemple du livre de M-C. Gaudel, M. Soria, C. Froidevaux [GSF87]: elles utilisent pour définir tous leurs types listes, arbres, ... la notion de type abstrait avec les fonctions de base sur ces types, mais dans tous leurs algorithmes on retrouve la réalisation Pascal de leur type (capsule). Saluons quand même leur mérite, elles se servent des types abstraits alors que beaucoup d'ouvrages ne se contentent que du type Pascal.

3.4.3) L'exemple du dictionnaire

L'exemple que nous donnons à nos étudiants de licences de mathématique et d'informatique est celui du dictionnaire que voici:

Définitions: un dictionnaire est une suite ordonnée de mots.
un mot est une suite de symboles de l'alphabet.

Un éditeur veut stocker dans un ordinateur son dictionnaire, il veut faire les opérations suivantes:

créer un dictionnaire d
rechercher un mot m dans un dictionnaire d
insérer un mot m dans un dictionnaire d
supprimer un mot m dans un dictionnaire d
imprimer tous les mots d'un dictionnaire d

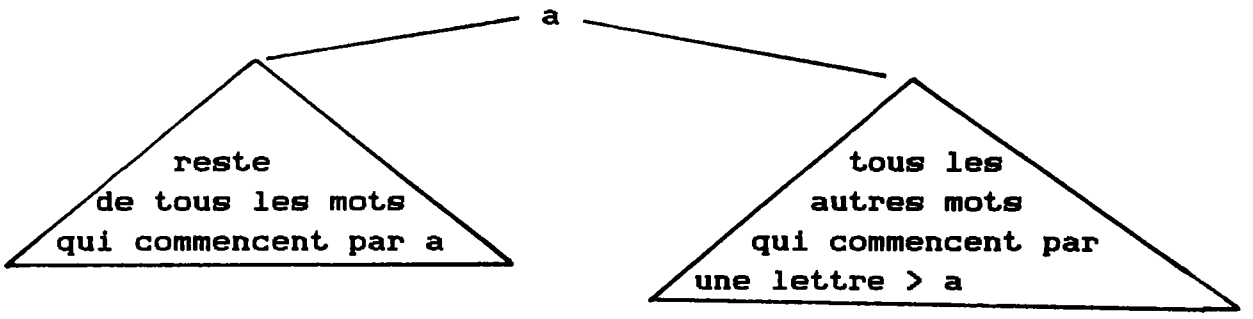
Avec le type suivant:
un dictionnaire étant une liste de mots
un mot étant une liste de caractères

toutes les opérations de base sont figées et classiques sur les

listes. La recherche d'un mot m dans la liste de mots va être très longue surtout si l'on cherche un mot qui commence par z. Pour optimiser cet algorithme nous pouvons avoir l'idée suivante:

Une analyse de recherche, verbale comme dans [LSP83].
si le mot m que l'on cherche ne commence pas par a il ne faut surtout pas chercher notre mot dans les mots qui commencent par a, il faut donc chercher dans les mots qui commencent par une lettre supérieure à a.
si le mot m commence par a il faut chercher le reste du mot dans tous les restes des mots qui commencent par a.

Nous obtenons donc un dictionnaire qui possède la structure d'arbre suivante:



Pour accéder à la lettre de la racine il faut que le dictionnaire soit non vide et s'il est vide le mot que l'on cherche n'est pas dedans. Nous pouvons donc réaliser ce type à l'aide d'un arbre binaire etnous aurons donc les fonctions de base suivantes pour un dictionnaire d:

- vidico(d) qui nous dira si le dictionnaire d est vide ou pas.
- racine(d) qui donne la valeur du caractère de la racine de d
- reste_des_mots(d) qui donne le reste des mots qui commencent par racine(d)
- autres_mots(d) qui nous donne tous les mots qui commencent par une lettre strictement plus grande que racine(d)

plus les deux fonctions de construction :

- consdicovide qui construit le dictionnaire vide
- consdico(racine,reste,autres) qui construit le dictionnaire non vide

continuons l'analyse (med)

si d est vide alors med ⇔ faux
si d n'est pas vide et si racine(d) est égale à la première lettre du mot m alors med ⇔ le reste du mot m ∈ reste_des_mots(d).

si d n'est pas vide et si racine(d) n'est pas égale à la première lettre du mot m alors med ⇔ m ∈ autres_mots(d).
si d n'est pas vide et s'il n'y a plus de lettres dans le mot m du mot m alors med ⇔ il y a une marque de fin dans racine(d).

On voit bien que les opérations de base sur le type mot sont:

- s'il n'y a plus de lettres dans m
- la première lettre de m
- le reste du mot de m

Ces opérations de base sont les opérations de base des listes linéaires: vide, valeur et suivant .....

Les étudiants, en général, se précipitent sur le type string (chaine de caractère);

l'analyse a permis de mettre en évidence ce que doit contenir un dictionnaire car lorsque le mot est vide c'est qu'il existe un "chemin" dans le dictionnaire qui a permis de "vider" le mot donc il faut absolument qu'il existe dans le dictionnaire une information qui nous dise si oui ou non le mot recherché est dans le dictionnaire. Il faudra donc une marque de fin dans le dictionnaire pour connaitre cette information. Cette marque pourra nous diriger vers la définition du mot

Lors du chapitre 4 nous donnerons dans la leçon de décomposition un autre exemple assez parlant, et où nous ne réalisons le type combinaison qu'une fois que l'on sait exactement comment il est utilisé.

DEUXIEME PARTIE: **"Réalisation"**

Une informatique scientifique

AVANT PROPOS

Fil directeur

Comme annoncé, notre objectif ambitieux mais réaliste est de montrer aux étudiants comment naissent les algorithmes corrects et lisibles, c'est-à-dire trouver le principe de ceux-ci, pour qu'ils procèdent plus tard de même. Nous serons également obligés d'en écrire, ainsi les étudiants disposeront dans leurs cours d'un nombre important d'algorithmes classiques, parmi lesquels ceux qui utilisent des boucles while seront obtenus de manière systématique en utilisant des schémas ou des règles (calcul de propriétés et règles d'équivalence de schémas itératifs et récursifs). Ils connaîtront également un certain nombre de types de base nécessaires à l'écriture des programmes.

Moule et acte pédagogique

Par contre il faut former les étudiants dans un moule rigide, c'est-à-dire leur donner le minimum d'outils pour concevoir leurs algorithmes (voir explication en 9.2 Anna Gram), afin qu'ils soient suffisamment armés pour continuer à évoluer seuls (théorie du cocon?). Ce moule est simple: il faut les faire réfléchir, analyser, démontrer des choses avant d'écrire l'algorithme forcément correct, qui va résoudre leur problème; il faut également leur montrer que ce que nous leur apprenons est réalisable au niveau de la machine, pour qu'ils puissent s'appuyer dessus pour mieux s'en abstraire. Mettre nos étudiants dans ce moule et leur expliquer le pourquoi et le comment est notre seul acte pédagogique.

But d'un cours de première année

Le but d'un cours de première année est d'apprendre à nos étudiants à décomposer leurs problèmes. Il faudra donc leur apprendre l'algorithmique de base et leur montrer le plus souvent

possible la correction des algorithmes que nous leur donnons et ceux que nous leur demandons d'écrire.

[Can90b]

But d'un cours de deuxième année

Pour construire des algorithmes corrects, dont le principe est récursif, et pour avoir une approche fonctionnelle des listes linéaires, il y a plusieurs conditions à remplir:

* Les étudiants doivent connaître et appliquer le principe de récurrence.
* Les étudiants de DEUG doivent maîtriser la récursivité avant les listes. Il faut donc une initiation poussée, qui leur donnera une deuxième chance de comprendre l'algorithmique. Nous pensons surtout aux étudiants "matheux" qui délaissent souvent l'informatique pour ne faire que des mathématiques.
* Nous ne voulons obtenir pour ce cours des algorithmes avec des boucles qu'en "éliminant" la récursivité dans les algorithmes récursifs correctement écrits. Cela pour insister sur l'analyse récursive des problèmes inductifs.
* Nous proposons également des schémas d'algorithmes récursifs, pour que l'étudiant puisse, dans un certain nombre de cas, en déduire de manière systématique des algorithmes récursifs.

Nous voulons que les étudiants de DEUG réfléchissent de manière récursive: en définissant récursivement le résultat d'une fonction ou d'une procédure, en trouvant une structure récursive au comportement d'une procédure, puis qu'ils puissent déduire de façon systématique de cette réflexion un algorithme correct.

Nous voulions également que les étudiants puissent critiquer leurs algorithmes, il fallait donc leur enseigner la notion de coût d'un algorithme à l'exécution.

[Can90c]

Chapitre 4

DECOMPOSITION

"La conception descendante reste utile pour de petits programmes et des algorithmes simples; c'est une technique utile pour enseigner la programmation à des étudiants débutants, pour aborder un problème d'une façon méthodique."

Bertrand Meyer [Mey90]

Il faut que l'étudiant maîtrise parfaitement la décomposition, même s'il est au début tout-à-fait incapable de l'utiliser. Il faudra absolument lui donner au début des guides spécifiques à la décomposition d'un problème. Ce n'est pas en une année voire deux que l'on apprend à un étudiant à décomposer. Par contre, si nous lui préparons la décomposition, il faut qu'il soit capable d'écrire un algorithme qui résoud son problème.

4.1) Les outils pour la décomposition

4.1.1) L'algorithmique de base

Pour pouvoir décomposer un problème il faut pouvoir identifier les données et les résultats, il faut pour cela que les étudiants connaissent la notion de variable et donc quelques types de base qui sont:

- les entier, réel, booléen et caractère avec la notion d'expression associée qui permettra les calculs.
- les tableaux
- les enregistrements
- les fichiers

Pour exprimer les relations entre les données et les résultats sous la forme d'un algorithme il faut que les étudiants

connaissent quelques opérations de base qui sont:

- l'affectation
- la composition séquentielle
- les conditionnelles
- les itérations

Au niveau de l'enseignement, nous avons décidé d'introduire ces opérations et ces types par une méthode que l'informaticien connaît bien, c'est-à-dire:

Lors de la réalisation d'un problème, nous avons des besoins, nous montrons que l'état actuel des connaissances de l'étudiant ne lui permet pas de résoudre le problème, un peu comme le font L. Goldschlager et A. Lister [GoL86]. Il faut lui apprendre que pour résoudre ces cas les informaticiens se sont donnés, au niveau de leur langage algorithmique de programmation, les moyens de les réaliser. Comme le langage de programmation impérative est une abstraction du langage machine, on rassurera l'étudiant en lui montrant que les moyens donnés sont réalisables au niveau de la machine. En fait, on refait l'histoire de l'informatique a posteriori.

4.1.2) Les modules

4.1.2.1) Les fonctions

Nous commençons systématiquement par les fonctions car elles sont plus simples à comprendre et à utiliser (liées aux fonctions mathématiques et expressions).

Nous les enseignons en faisant le lien avec la définition mathématique des fonctions suivante:

$$f : S \longrightarrow B$$
$$x \longrightarrow f(x)$$

donc au niveau informatique (algorithmique) il faudra que toutes ces informations apparaissent dans la définition informatique de la fonction.

f est le nom de la fonction (identificateur)
S et B des ensembles de valeurs (type)
x une variable mathématique (variable informatique)
f(x) une définition de la valeur de f(x) en fonction de x (expression ou algorithme)

Puis, nous donnons la définition Pascal d'une fonction en

distinguant deux cas:

1) une expression fonctionnelle "pure" qui est la plus simple:

```
function f(x:S):B;
begin
  f:=expression simple de f(x) en fonction de x
end;
```

2) une expression fonctionnelle qui permet de cacher une réalisation actionnelle. Cela permettra aux étudiants d'utiliser de manière fonctionnelle des algorithmes écrits au préalable.

```
function f(x:S):B;
var vf:B; + autres déclarations locales à la fonction
            nécessaires à l'algorithme
begin
   Algo qui calcule
   f(x) et qui range
   cette valeur dans
   la variable vf   ;
    f:=vf
end;
```

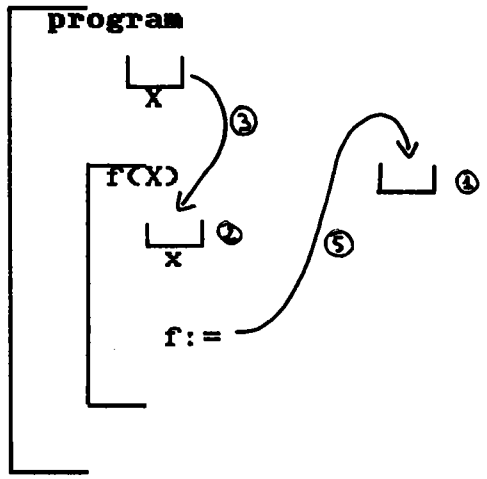
Puis nous donnons l'utilisation de ces fonctions dans les expressions:

f(expS) est une expression de type B si expS est une expression de type S

Et surtout l'évaluation d'un appel à la fonction

1) on évalue expS
2) on évalue f avec la valeur de expS calculée en 1

On donne également le schéma d'activation suivant d'une fonction:



où l'on explique toutes les étapes de l'activation d'une fonction
1) Réservation d'une mémoire pour le résultat.
2) Réservation de mémoire pour les données (+celles pour l'algorithme)
3) Transmission de la valeur de la donnée (x:=X)
4) Evaluation de l'expression ou exécution de l'algorithme qui permettent le calcul de f(x).
5) Transmission du résultat (mémoire réservée en 1 := vf ou valeur évaluée en 4)
6) Retour au programme ou sous-programme appelant avec restitution des mémoires allouées en 2; exploitation du résultat.

Remarque: Nous expliquons l'exécution d'une fonction de cette manière pour être complet mais surtout pour préparer le terrain à l'appel d'une procédure avec les deux modes de passage des paramètres (voir 4.1.2.3) . Ce schéma d'activation nous permettra également d'expliquer l'exécution d'une fonction et d'une procédure récursive (pile d'exécution). Si on reste dans un style purement fonctionnel, il est évidemment inutile d'utiliser ce schéma, une exécution symbolique suffit amplement. Rappelons quand même qu'une fois les fonctions correctement écrites, H. Abelson et G.Sussman [AbS89] s'intéressent aux mémoires (pointeurs) pour optimiser l'exécution de leurs fonctions.

4.1.2.2) Les procédures

En ce qui concerne les procédures nous donnons les besoins (plusieurs résultats et structurer les actions). Nous expliquons avec le schéma d'activation d'une procédure comment fonctionnent et à quoi servent les deux modes de passage des paramètres.

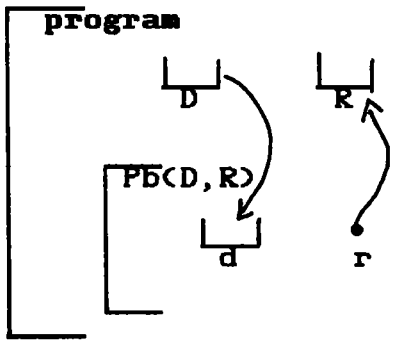


schéma d'activation d'une procédure Pb (d donnée et r résultat)

4.1.2.3) Importance des schémas d'activation d'un module

Il est vrai que cette manière de représenter une exécution d'un module est très opératoire et donc en contradiction avec le fait que nous préconisons l'écriture d'algorithme abstrait et si

possible dans un style fonctionnel, mais nous avons remarqué qu'un certain nombre non négligeable d'étudiants ne comprenaient pas correctement les deux modes de passage des paramètres (valeur et variable) et la notion de variable globale. Le fait de "dessiner" les emplacements mémoire montre bien:

- qu'après l'exécution d'un module avec un passage par valeur la variable qui a servi pour l'appel n'est pas modifiée.
- que l'on peut accéder au même emplacement mémoire par l'intermédiaire d'identificateurs de variables différents.

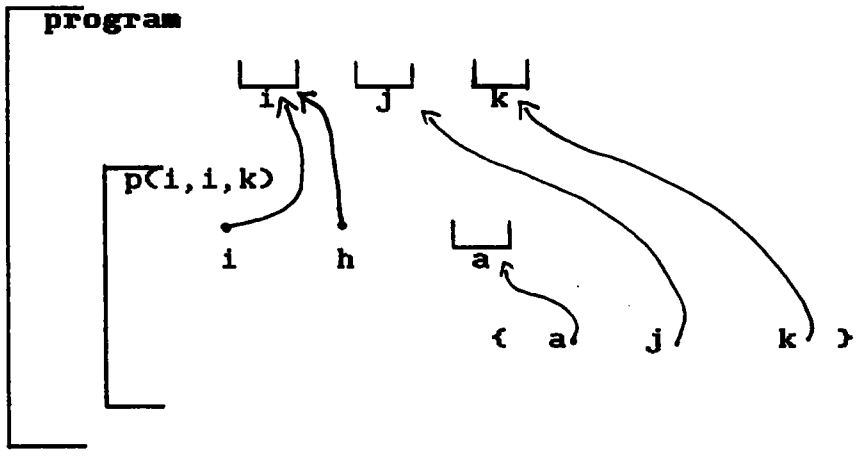
En plus d'une meilleure compréhension des modes de passages et de la notion de variable globale, nous introduisons de manière "abstraite" la notion de pile d'exécution. Nous avons comblé un peu le fossé entre la "forme" et le "faire en différé" [Duch90].

4.1.2) Exécution d'un programme

Soit le programme suivant:

```
program ex;
var i,j,k:integer;
  procedure p(var i,h:integer;a:integer);
  begin
     j:=a+k; i:=j+2; h:=i+3
   end;
begin
   k:=1;
   p(i,i,k)
end.
```

le schéma d'activation est le suivant:

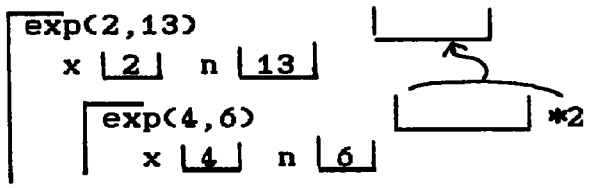


4.1.3) Exécution d'une fonction (récursive)

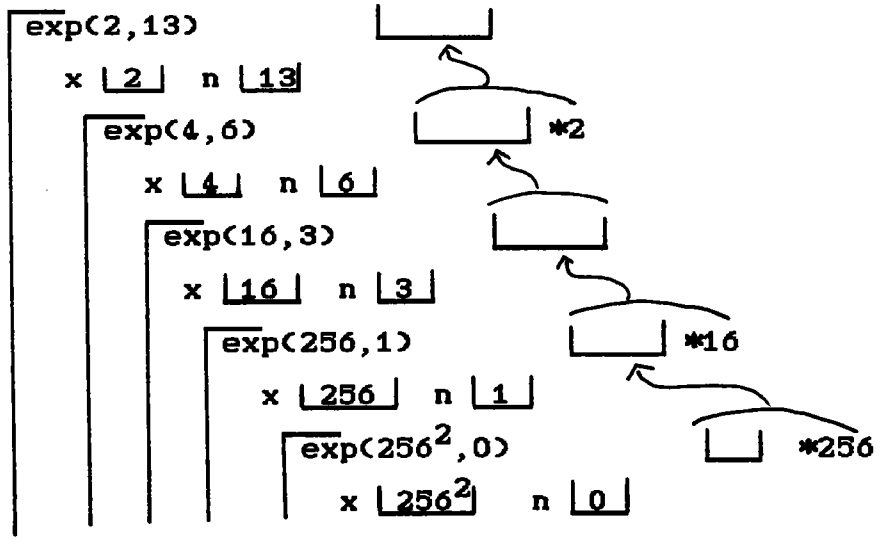
Nous prenons l'exemple de la fonction exp(x,n) qui calcule x^n en fonction de $(x^2)^{n/2}$.

```
function exp(x:real;n:integer):real;
begin
  if (n=0) then
    exp:=1
  else if pair(n) then
    exp:=exp(x*x,n div 2)
  else
    exp:=x*exp(x*x,n div 2)
end;
```

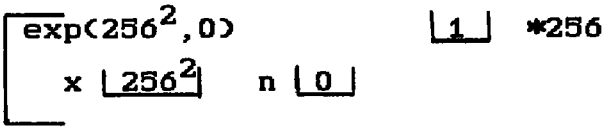
d'où l'exécution de exp(2,13) est la suivante:
1) une place mémoire est réservée pour le résultat de exp(2,13).
2) une place mémoire est réservée pour x et pour n où sont mises respectivement les valeurs 2 et 13.
3) comme 13≠0 et 13 est impair un appel à exp(2*2,13/2) c'est-à-dire exp(4,6) est exécuté, après cette exécution la valeur de exp(4,6) se trouve dans l'emplacement mémoire réservé à cet effet.
4) il reste à multiplier le résultat de exp(4,6) par la valeur de x, donc 2, et mettre le résultat de cette opération dans l'emplacement réservé en 1).



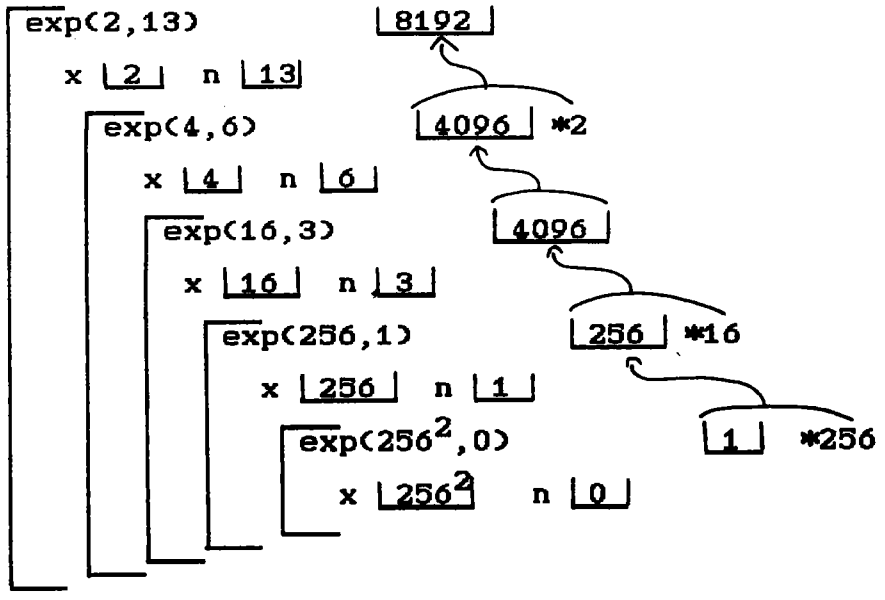
Etat des mémoires au début de l'appel de exp(256^2,0)



A ce moment comme n=0 la fonction exp donne son résultat comme suit:



la récurrence est amorcée, les calculs peuvent s'effectuer après chaque arrêt d'un appel comme suit:



4.2) Une leçon de décomposition

"pour savoir résoudre des problèmes, il faut en résoudre."
[Pol67], [Pol85]

Cette partie est pour nous une des plus importantes. Elle montre comment l'informaticien peut aborder un problème et comment il peut utiliser les techniques apprises précédemment pour écrire un programme qui résoudra de manière informatique son problème.

Il faut également mettre en garde l'étudiant: lorsqu'il veut décomposer un problème il se trouve souvent dans l'incapacité de le faire. Il rend l'algorithmique responsable de son échec, alors que c'est souvent une incompréhension du problème, qui ne peut se décomposer correctement que s'il est bien compris.

Au moment de définir les procédures nous avons déjà indiqué aux étudiants la nécessité de décomposer et les moyens de le faire

(fonction, procédure). Nous avons voulu, par l'intermédiaire d'une décomposition, montrer aux étudiants comment l'informaticien procède. Nous avons volontairement pris un exemple car il n'existe pas vraiment de règles pour décomposer un problème. Par contre nous leur donnons un fil directeur pour bien décomposer (une fois que l'idée est trouvée) qui est le suivant:

> A chaque fois nous essayerons d'écrire un algorithme général simple qui résoud notre problème. Celui-ci va utiliser des variables, des types et des sous-problèmes. Il y aura des contraintes sur ces variables et leurs types. Nous ne nous précipiterons pas pour définir un type si celui-ci n'est pas simple, nous attendrons d'en savoir plus sur l'utilisation de celui-ci, pour que sa réalisation soit la plus adaptée possible (REPRESENTER lorsque l'on a toutes les informations en main). Pour résoudre chaque sous-problèmes (REPARTIR), on procèdera de même en incluant les contraintes liées à son utilisation.

Pour la décomposition, nous donnons un exemple important par la taille du résultat, mais en écrivant à chaque fois des algorithmes courts et facilement compréhensibles par une simple lecture. Des exemples de ce genre manquent cruellement dans la plupart des ouvrages qui font référence à la programmation structurée.

4.2.1) Le problème: le jeu mastermind chinois

Ce jeu est une variante du jeu mastermind qui a été souvent utilisé comme exemple de programme pour les étudiants [Duc84]. C'est notre exemple "fétiche" [Can90b], car non seulement il nous permet de montrer comment l'informaticien décompose ses problèmes mais encore:

- l'importante de l'écriture abstraite d'un algorithme: nous ne construirons (REPRESENTER) les types (ici combinaison) qu'au moment où nous savons exactement comment il est utilisé.
- l'importance des schémas itératifs pour pouvoir écrire des algorithmes corrects sur le type combinaison.

Règle du jeu: On dispose d'une combinaison secrète de quatre chiffres, tous différents deux à deux. Le but de ce jeu est de trouver cette combinaison secrète en essayant successivement plusieurs combinaisons. A chaque tentative, le programme nous répondra en nous donnant le nombre de chiffres bien placés et le nombre de chiffres mal placés.

Exemple: combinaison secrète	essais successifs	nombre de bien placés	nombre de mal placés
1234	1307	1 (1)	1 (3)
	1243	2 (1 et 2)	2 (3 et 4)
	5678	0	0
	1234	4 (tous)	0

4.2.2) Décomposition du problème

Si nous désirons faire jouer le joueur plusieurs fois, nous pouvons obtenir l'algorithme suivant:

```
Voulez_vous_jouer(jouer);
while jouer do
begin
  jouer_une_fois;
  Voulez_vous_rejouer(jouer)
end
```

Cet algorithme impose les contraintes suivantes:

jouer: c'est une variable, son type doit être booléen (while jouer do). Comme le type Pascal boolean convient, il n'y a pas de type à définir. Elle devra contenir la valeur true si le joueur veut jouer et false sinon.

Voulez_vous_jouer: c'est une procédure, elle n'a pas de donnée, elle a un résultat qui est booléen. Elle demandera au joueur s'il veut jouer ou non. En fonction de sa réponse on donnera la bonne valeur au résultat.

jouer_une_fois: c'est une procédure, elle n'a ni donnée ni résultat. Elle simulera le comportement du jeu.

Voulez_vous_rejouer: c'est une procédure, elle n'a pas de donnée, elle a un résultat qui est booléen. Elle demandera au joueur s'il veut rejouer ou non. En fonction de sa réponse on donnera la bonne valeur au résultat.

4.2.3) Décomposition de jouer_une_fois

Cette procédure doit initialiser une combinaison secrète et faire faire plusieurs tentatives d'essais au joueur jusqu'à ce que celui-ci ait gagné. Nous pouvons ainsi obtenir l'algorithme suivant:

```
initialise(secrete);
gagne:=false; {pour pouvoir faire au moins un tour dans la boucle}
while not gagne do
begin
  initialise(essai);
  calcul_nombre_bien_place(nbp,essai,secrete);
  if nbp=4 then
     gagne:=true {le joueur a gagné}
  else
     writeln('nbp=',nbp,'nmp=',nombre_mal_place(essai,secrete))
end
```

Cet algorithme impose les contraintes suivantes:

nbp: c'est un entier qui contiendra le nombre de bien placés.

gagne: c'est un booléen, sa valeur sera true si le joueur a gagné et false sinon.

secrete: c'est une combinaison qui contiendra la combinaison secrète.

essai: c'est une combinaison qui contiendra la valeur des différents essais successifs.

combinaison: c'est un type pour pouvoir définir les variables secrete et essai. Nous utiliserons ce type sans le définir. Nous le définirons lorsque nous connaîtrons un peu mieux son utilisation.

initialise: c'est une procédure qui permet de lire une bonne combinaison. Elle a un résultat qui est de type combinaison et qui représente la combinaison lue.

calcul_nombre_bien_place: c'est une procédure qui calcule le nombre de bien placés dans la combinaison essai par rapport à la combinaison secrète. Les données sont ces deux combinaisons et le résultat est un entier.

nombre_mal_place: c'est une fonction qui calcule le nombre de mal placés dans la combinaison essai par rapport à la combinaison secrète. Les données sont ces deux combinaisons.

4.2.4) Décomposition de la procédure initialise

Une combinaison est une suite de quatre chiffres qui sont tous différents. Si on lit de n'importe quelle façon, nous n'aurons pas forcément une bonne combinaison. Donc nous utiliserons

l'algorithme suivant pour l'initialisation d'une combinaison c:

```
lire(c);
while not bon(c) do
  lire(c)
```

Cet algorithme impose les contraintes suivantes:

lire: c'est une procédure qui lit une combinaison. Elle a un résultat qui est de type combinaison et qui représente la combinaison lue.

bon: c'est une fonction dont le résultat est de type booléen. Elle retourne la valeur true si sa donnée de type combinaison est une combinaison correcte et false sinon.

4.2.5) Décomposition de la fonction bon

x est une bonne composition si x est une suite de chiffres tous différents. On obtient la définition suivante:

```
bon(x)=tous_chiffre(x) and tous_different(x)
```

Cet algorithme impose les contraintes suivantes:

tous_chiffre: c'est une fonction qui retourne true si sa donnée de type combinaison est composée de chiffres uniquement, et false sinon.

tous_different: c'est une fonction qui retourne true si sa donnée de type combinaison est composée de valeurs toutes différentes deux à deux, et false sinon.

4.2.6) Décomposition des procédures et des fonctions non terminées Celles qui dépendent du type combinaison

Pour ces procédures et fonctions, il faut connaître le type combinaison. Les procédures et fonctions non terminées qui sont:

tous_chiffre, tous_different, lire
calcul_nombre_bien_place et nombre_mal_place.

Pour chacune de ces procédures et fonctions, nous devons avoir accès aux quatre chiffres d'une combinaison:
Soit c1 c2 c3 c4 les quatre chiffres d'une combinaison, on a

```
tous_chiffre(c1 c2 c3 c4)=
      chiffre(c1) and chiffre(c2) and chiffre(c3) and chiffre(c4)
```

tous_different(c1 c2 c3 c4)=(c1≠c2) and (c1≠c3) and (c1≠c4) and (c2≠c3) and (c2≠c4) and (c3≠c4)

lire(c1 c2 c3 c4) on lit c1 puis c2 puis c3 puis c4 ou c1c2c3c4 d'un coup.

calcul_nombre_bien_place(nbp,c1 c2 c3 c4, s1 s2 s3 s4) calcule le nombre d'indice i tel que ci=si.

nombre_mal_place(c1 c2 c3 c4,s1 s2 s3 s4) c'est le nombre d'indice k tel que ck≠sk (pas bien placé) et ck apparait dans s1 s2 s3 s4.

Il nous reste maintenant à représenter le type combinaison et écrire les modules qui dépendent de ce type.

Remarque: Nous aurions du résoudre ces quatre sous-problèmes, quelle que soit la manière de concevoir l'algorithme structuré ou non. Ici, ces sous-problèmes apparaissent clairement, on sait ce que chacun d'eux doit résoudre. A part le type combinaison, qui sera commun à toutes ces structures, ils peuvent être traités séparément.

4.2.7) Vers le logiciel décompositeur

Le manque de commentaires dans les algorithmes que nous écrivons est volontaire. Nous préférons que les étudiants fassent une analyse saine arborescente descendante (récursive s'il le faut) avant l'écriture de leurs algorithmes. Par contre les commentaires sont importants pour la relecture d'un programme que l'on veut modifier; ils peuvent donc être insérés dans le texte du programme soit à la main soit avec des éditeurs syntaxiques qui, à chaque fois que l'on utilise une procédure ou une fonction, nous demandent les noms de ses paramètres ainsi que le nom de leur type pour tenir compte de ces nouvelles contraintes en plus de celles existantes, et d'ajouter un commentaire pour dire ce que ce module doit faire. Cet éditeur permettra de décomposer comme nous en avons l'habitude sur papier, et de remettre tout en place, lorsque tout est correctement défini. L'éditeur pourra même proposer à son utilisateur de décomposer un problème non encore traité. Nous donnerons plus de précision sur les spécifications de cet éditeur que nous avons appelé décompositeur (voir chapitre 10).

4.3) Utilisation des propriétés pour concevoir les algorithmes

Lors du chapitre 5 sur les schémas, nous montrerons également comment nos étudiants peuvent obtenir des algorithmes corrects à partir de propriétés et de schémas.

4.3.1) Passage du fonctionnel à l'actionnel

Comme en général nous demandons presque de manière systématique à nos étudiants d'écrire des fonctions, nous leur montrons comment on peut en déduire des procédures en gérant les variables qui vont contenir les résultats.

Exemple: tout le monde connaît la définition récursive (relation de récurrence) du quotient q(a,b) et du reste r(a,b) de la division euclidienne de a par b (a≥0 et b>0).

$$\begin{cases} q(a,b)=0 & \text{et } r(a,b)=a & \text{si } a<b \\ q(a,b)=q(a-b,b)+1 & \text{et } r(a,b)=r(a-b,b) & \text{sinon} \end{cases}$$

nous demandons à nos étudiants de montrer par récurrence que ces définitions récursives sont correctes puis nous leur demandons d'écrire des fonctions et éventuellement de les exécuter sur un exemple (2 à 3 appels récursifs). Par contre si nous leur demandons d'écrire une procédure (deux résultats) pour calculer le quotient q(a,b) et le reste r(a,b) nous leur demandons d'écrire l'entête de cette procédure et de définir une propriété (assertion) qui va spécifier le rôle de cette procédure.

```
procedure divise(a,b:integer;var q,r:integer);
{q=q(a,b) et r=r(a,b)}
```

cette assertion est l'objectif à atteindre pour chaque cas:

```
procedure divise(a,b:integer;var q,r:integer);
var q1,r1:integer;
begin
  if (a<b) then
    ⇓
    {q=q(a,b) et r=r(a,b)}  "amorce"
  else
   begin
     divise(a-b,b,q1,r1);
     {q1=q(a-b,b) et r1=r(a-b,b)} "par hypothèse de récurrence"
    ⇓
     {q=q(a,b) et r=r(a,b)}
   end
end;
```

Nous n'avons donc que deux problèmes à résoudre. Dans les deux cas la définition récursive de r(a,b) et q(a,b) en fonction de r(a-b,b) et q(a-b,b).

1er cas l'amorce

Pour réaliser (q=q(a,b)=0 et r=r(a,b)=a) il suffit de mettre 0 dans la variable q et a dans la variable r. Donc les deux affectations suivantes réaliseront l'amorce:

```
q:=0;
r:=a
```

nous appelons le cas d'arrêt ammorce car avec le théorème de récurrence: l'induction ∀n≥n0 $\mathcal{P}$(n)⇒$\mathcal{P}$(n+1) permet de construire la chaîne $\mathcal{P}$(n0)⇒$\mathcal{P}$(n0+1)⇒ ... ⇒$\mathcal{P}$(n), comme $\mathcal{P}$(n0) est vrai, ce vrai est comme une amorce qui va faire propager le vrai partout.

2ème cas (⇒)

Comme q(a,b)=1+q(a-b,b) et que la variable q1 contient q(a-b,b), il suffit de mettre dans la variable q la valeur de q1+1. Comme r(a,b)=r(a-b,b) et que la variable r1 contient r(a-b,b), les deux affectations suivantes réaliseront notre problème:

```
q:=q1+1;
r:=r1
```

4.3.2) Conception d'un algorithme

Les propriétés sont fondamentales, elle peuvent souvent nous guider dans la conception d'un algorithme. Un exemple: le tri par insertion:
Comme nous voulons trier un tableau nous pouvons nous servir de la propriété qui exprime le fait que les valeurs du tableau t entre les indices 1 et i sont triées.

Le principe:

Soit $\mathcal{P}$(i) la propriété suivante:

$\mathcal{P}$(i)=(les valeurs du tableau t de 1 à i sont triées)

Remarque: $\mathcal{P}$(1) est vraie car le tableau t de 1 à 1 ne contient qu'une valeur.

Si nous arrivons à trouver un algorithme qui réalise l'implication:

$\mathcal{P}$(i) ⇒ $\mathcal{P}$(i+1) pour tous les i≥1 alors nous pourrons réaliser $\mathcal{P}$(n) et donc le tableau sera trié.

Réalisation de $\mathcal{P}$(i) ⇒ $\mathcal{P}$(i+1)

En donnée nous avons le tableau t qui est trié de 1 à i.

Pour qu'il soit trié de 1 à i+1, un moyen est de prendre la valeur t[i+1] et de l'insérer à sa place dans les valeurs de 1 à i. Comme cela le tableau t contiendra bien toutes les valeurs du tableau initial et il sera trié de 1 à i+1.

Comme le principe de récurrence nous assure que comme $\mathcal{P}(1)$ est vraie et si l'on arrive à réaliser $\mathcal{P}(i) \Rightarrow \mathcal{P}(i+1)$ sous la forme d'une procédure inserer qui insérera t[i+1] dans t entre les indices 1 et i, alors la procédure récursive suivante:

```
procedure tri_insertion(var t:tableau;i:integer);
begin
  if (i<>1) then
   begin
    tri_insertion(t,i-1);
    inserer(t[i],t,i-1)
   end
end;
```

triera un tableau t de 1 à i.

Cette démarche est capitale car:

- elle nous permet d'être sûrs que si nous arrivons à réaliser la procédure inserer, la procédure de tri sera elle aussi correcte. Nous avons donc un nouveau problème à résoudre, et nous pouvons procéder de même pour le réaliser.
- L'étudiant ne pourra pas "déplier" son algorithme car l'utilisation de la récursivité présente l'avantage suivant: nous sommes obligés de donner un nom à tous les modules de la décomposition, ainsi l'étudiant pourra tester son algorithme et surtout le (re)comprendre plus facilement lors d'une relecture.
- De plus l'utilisation de l'analyse et de la programmation récursive lui permettra, plus tard, d'essayer de trouver la propriété (souvent simple) que doit vérifier son algorithme. Une fois celle-ci trouvée, il obtiendra un algorithme correct.

Remarque: Lorsque nous aborderons les listes linéaires nous pourrons donner une écriture fonctionnelle de cet algorithme:

```
function tri_insert(l:liste):liste;
begin
  if vide(l) then
    tri_insert:=consvide
  else
   tri_insert:=insert(valeur(l),tri_insert(suivant(l)))
end;
```

Anecdote

Une autre propriété peut nous amener vers la conception de la procédure de classement de l'algorithme de tri rapide (quicksort); nous avons, à ce propos, une anecdote que nous livrons ici:

L'exemple du quicksort est représentatif de la façon d'enseigner des informaticiens, qui utilisent la récursivité seulement quand ils sont incapables de s'en sortir sans pile, donc dans la procédure quicksort. Quant à la procédure de classement ou de segmentation qui prépare le tableau pour que les deux appels puissent correctement faire leur travail, elle est systématiquement écrite à l'aide de boucles. Au cours de nos premières années d'enseignement, nous procédions de même. A chaque fois que nous écrivions cette procédure, nous nous trompions en utilisant un ≤ à la place d'un < ou quelque chose du même ordre. En enseignant consciencieux nous demandions à nos étudiants d'aller tester leurs algorithmes sur les machines mises à leur disposition. Ils le faisaient et revenaient nous annoncer fièrement "Monsieur votre algorithme est faux". Au bout de trois années (l'inertie c'est quelque chose), nous décidions enfin d'expliquer de manière récursive cet algorithme. Tous les cas de l'analyse étaient simples à comprendre et à réaliser, l'algorithme devenait enfin clair et forcément correct, nous ne nous trompions plus.

L'analyse:

si binf≤bsup et t[binf]<a alors le résultat est le même que celui du classement entre binf+1 et bsup

si binf≤bsup et t[binf]≥a et t[bsup]>a alors le résultat est le même que celui du classement entre binf et bsup-1

si binf≤bsup et t[binf]≥a et t[bsup]≤a alors il suffit de permuter les deux valeurs t[binf] et t[bsup], et le résultat est le même que celui du classement entre binf+1 et bsup-1.

si binf>bsup ces deux bornes sont nos résultats.

D'où:

```
procedure classement(var t:tableau; a:V; binf,bsup:integer;
                                    var binf1,bsup1:integer);
begin
  if (binf>bsup) then
   begin
    binf1:=binf;bsup1:=bsup
   end
  else if (t[binf]<a) then
     classement(t,a,binf+1,bsup,binf1,bsup1)
```

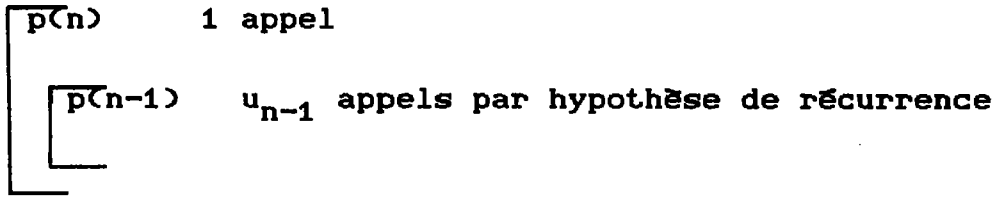
```
      else if (t[bsup]>a) then
         classement(t,a,binf,bsup-1,binf1,bsup1)
      else
       begin
          echange(t,binf,bsup);
          classement(t,a,binf+1,bsup-1,binf1,bsup1)
       end
    end;
```

Fin de l'anecdote

4.4) La notion de coût

De plus pour sensibiliser l'étudiant à la notion de coût, nous faisons calculer le nombre d'appels récursifs (démonstration par récurrence). Il faut au moins que l'étudiant connaisse le nombre d'appels à un module dans les cas suivant:

1) p(n)=if n=0 then f_0 else f(n,p(n-1))
lors d'un appel pour n=0 nous n'avons qu'un appel $u_0=1$
lors d'un appel pour n>0 nous avons:



d'où on obtient la suite suivante:

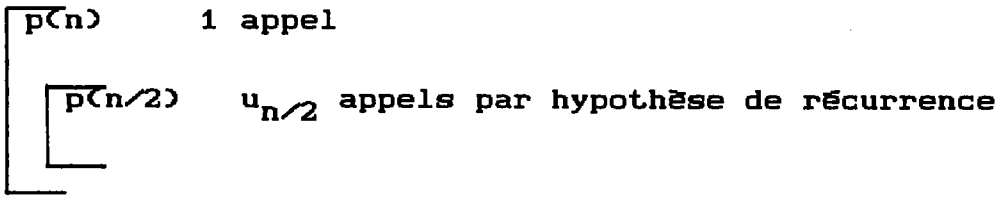
$u_0=1$
$u_n=1+u_{n-1}$ si n>0

nous pouvons alors faire démontrer par récurrence à l'étudiant que:

$\forall n \geq 0 \; u_n=n+1$

2) p(n)=if n=0 then f_0 else f(n,p(n/2))

pour n=0 nous avons 1 appel
pour n>0 nous avons:



Comme si $2^k \leq n < 2^{k+1} \Rightarrow 2^{k-1} \leq n/2 < 2^k$ nous ramenons l'étudiant sur une démonstration par récurrence plus "classique" en démontrant la propriété sur k suivante:

$\forall n$ tq $2^k \leq n < 2^{k+1}$ nba(p,p(n))=k+2

où nba(p,p(n)) est le nombre d'appel de p dans une exécution de p(n). Nous pouvons donc en déduire que:

nba(p,p(n))=2+E($\log_2$(n)) où E est la partie entière.

Chapitre 5

SCHEMAS

5.1) Schémas itératifs

Comme notre objectif est de montrer à chaque fois la correction des algorithmes de nos étudiants. Nous devons le faire dès le début lors de l'apprentissage de l'algorithmique de base. Si nous commençons à montrer la correction lorsque les algorithmes se compliquent, il risque d'y avoir un déphasage important entre l'enseignant et l'étudiant car celui-ci pourra ne pas être sensible à une modification brutale de notre langage. Nous utilisons donc le langage des assertions pour la composition séquentielle, les conditionnelles et les boucles. Pour ces dernières nous utilisons un théorème de récurrence sur les boucles for (pour) alors que pour les boucles while (tant que) nous avons préféré utiliser des propriétés.

5.1.1 Théorème de récurrence pour les boucles

Soit $\mathcal{P}(i)$ une propriété sur i un entier
Soit l'algorithme suivant:

```
for i:=binf to bsup do
    instr
```

Si $\mathcal{P}(binf-1)$ est vraie avant l'exécution de l'algorithme.
Si on peut montrer pour tout i plus grand que binf que si $\mathcal{P}(i-1)$ est vraie avant l'exécution de l'instruction instr $\mathcal{P}(i)$ est vraie après l'exécution de l'instruction instr alors:
$\mathcal{P}(bsup)$ est vraie après l'exécution de la boucle si binf≤bsup.
$\mathcal{P}(binf-1)$ est vraie sinon

Ce théorème est démontré par une exécution de la boucle et en utilisant le pas d'induction suivant:

```
        P(i-1)  {par hypothèse de récurrence}
        instr;
        P(i)
        i:=i+1;  {l'incrémentation de la boucle}
        P(i-1)
```

Une utilisation du théorème

Soit u_n la suite définie de manière récursive suivante:

u_{n0}=cst

$u_n = f(u_{n-1})$ $\forall n>n0$

Un algorithme

Pour calculer u_n pour un n donné on peut utiliser l'algorithme suivant:

```
u:=cst;
for i:=n0+1 to n do
  u:=f(u)
```

Correction

Soit la propriété P(i) suivante: $(u=u_i)$
au départ $u=cst=u_{n0}=u_{(n0+1)-1}$ donc P(n0) est vrai
$\{u=u_{i-1}\}$
u:=f(u)
$\{u=f(u_{i-1})=u_i\}$
donc à la fin de la boucle $P(n)=(u=u_n)$ est vraie.
l'algorithme est donc correct.

5.1.2) Les propriétés

Pour les raisons déjà évoquées, nous ne voulions pas faire un cours "classique" sur les algorithmes des tables, comme les algorithmes de recherche écrits à l'aide d'une boucle while gérée par un indice et une variable booléenne. Pour banaliser ces algorithmes au demeurant importants pour un informaticien confirmé, nous avons préféré définir le résultat de ceux-ci à l'aide d'une propriété. Nous sommes conscients que nous ne pouvons pas résoudre tous ces problèmes avec ce procédé. Par contre son avantage est qu'il nous permet de donner, dans un premier temps donner les propriétés que doit vérifier le résultat d'un algorithme ainsi l'étudiant obtiendra de manière systématique son algorithme sans effort de sa part. Dans un deuxième temps, nous

pourrons lui demander de trouver la bonne propriété, cela l'obligera à réfléchir un petit peu car il sait qu'une fois la propriété trouvée il obtiendra un algorithme correct. De plus, nous pourrons compléter ces schémas par des schémas de parcours (traitement et calcul). Comme nous le signalions, nous essayons par ce moyen de situer la réflexion de l'étudiant à un autre niveau d'abstraction et donc d'obtenir des algorithmes itératifs corrects (par rapport à la propriété) de manière systématique. L'analyse du problème n'est plus itérative (modification d'état), la boucle n'est plus qu'un outil de construction (ou d'obtention) d'algorithmes.

5.1.2.1) ∀ et ∃

Soit n une constante.
Soit tableau le type array[1..n] of T.
Soit t un tableau.
Soit $\mathcal{P}$ une propriété sur des valeurs de type T, calculable (existence d'un algorithme).

On veut calculer les deux propriétés suivantes:

1) tous_$\mathcal{P}$(t)= ∀ i∈(1,...,n) $\mathcal{P}$(t,i)
2) existe_$\mathcal{P}$(t)= ∃ i∈(1,...,n) $\mathcal{P}$(t,i)

Nous pouvons évidemment exprimer une de ces propriétés avec la négation de l'autre. Cela peut même servir à montrer la correction de l'une en fonction de l'autre (une seule démonstration). Par contre, pour spécifier leur problème, les étudiants utilisent soit la notation ∀ ou la notation ∃ en fonction de leur compréhension comme ils utilisent l'opérateur **and** ou **or** pour exprimer une expression booléenne.

Puis on donne aux étudiants les algorithmes ("classiques") qui calculent la valeur de la propriété. Ces algorithmes utilisent des boucles **while** gérées à l'aide de variables booléennes.

```
vtous_P:=true;
i:=1;
while (i<=n) and vtous_P do
begin
  if not P(t,i) then                }  vtous_P:=P(t,i)
     vtous_P:=false;                }
  i:=i+1
end
```

```
vexiste_P:=false;
i:=1;
while (i<=n) and not vexiste_P do
begin
  if P(t,i) then                    }   vtous_P:=P(t,i)
     vexiste_P:=true;               }
  i:=i+1
end
```

Puis nous montrons que ces algorithmes sont corrects, mais nous ne le faisons que pour ces deux là. L'étudiant, lorsqu'il aura trouvé sa propriété, pourra en déduire de manière systématique un algorithme. La réflexion se fait au niveau de la propriété.

Correction des algorithmes

Comme nous ne voulions pas utiliser un système de preuve complet pour montrer la correction des algorithmes (surtout l'utilisation des invariants) au niveau du DEUG première année et que l'obtention d'algorithmes corrects est une de nos priorités, nous avons décidé de prouver la correction de ces algorithmes à partir de la correction d'algorithmes équivalents utilisant des boucles for.

Nous avons les définitions suivantes:

$$\text{tous\_P}(t) = \operatorname{and}_{i=1}^{n} P(t,i)$$ {vraie si toutes vraies fausse si une fausse}

$$\text{existe\_P}(t) = \operatorname{or}_{i=1}^{n} P(t,i)$$ {fausse si toutes fausses vraie si une vraie}

Puis nous donnons les algorithmes avec des boucles for, qui calculent la valeur de l'expression (avec and ou or)

```
vtous_P:=true;
for i:=1 to n do
   vtous_P:=vtous_P and P(t,i);
tous_P:=vtous_P

vexiste_P:=false;
for i:=1 to n do
   vexiste_P:=vexiste_P or P(t,i);
existe_P:=vexiste_P
```

La correction de ces algorithmes utilisant des boucles for est simple en démontrant par récurrence les propriétés suivantes:

$$Q(i)=\bigwedge_{k=1}^{i} \mathcal{P}(t,k) \text{ pour tous\_}\mathcal{P}.$$

$$Q(i)=\bigvee_{k=1}^{i} \mathcal{P}(t,k) \text{ pour existe\_}\mathcal{P}.$$

Ces algorithmes peuvent par la suite être transformés en algorithmes équivalents (démontré lors de l'apprentissage des boucles) utilisant des boules while du type suivant:

```
vtous_P:=true;
i:=1;
while (i<=n) do
begin
  if not P(t,i) then
     vtous_P:=false;
  i:=i+1
end
```

et par la suite on montre que l'on peut arrêter cette boucle lorsque vtous_$\mathcal{P}$ est fausse (ou vexiste est vraie). Ainsi nous obtenons la correction des algorithmes.

Remarque: Nous pouvons également conseiller aux étudiants d'utiliser les boucles for lorsqu'il y a une grande chance que le résultat de tous_$\mathcal{P}$ soit vrai ou que le résultat de existe_$\mathcal{P}$ soit faux.

Exemples de propriétés

si toutes les valeurs d'un tableau t sont égales à v

$$\forall\, i\in(1,\ldots,n)\ t[i]=v$$

Soit v une valeur de type T. Si on veut chercher si v est une valeur contenue dans le tableau t, on peut se servir de la propriété suivante:

$$\exists\, i\in(1,\ldots,n)\ t[i]=v$$

Si on veut que toutes les valeurs d'un même tableau t soient différentes, on peut se servir de la propriété suivante:

$$\forall\, i\in(1,\ldots,n)\ \forall\, j\in(i+1,\ldots,n)\ t[i]\neq t[j]$$

cette propriété peut donc servir à écrire la fonction tous_different de la décomposition du jeu "le mastermind chinois".

Nous donnons également aux étudiants des algorithmes qui permettent de calculer l'indice i tel que la propriété $\mathcal{P}(t,i)$ soit vraie ainsi que ceux qui permettent de compter le nombre de i tels que la propriété $\mathcal{P}(t,i)$ soit vraie. Nous pourrons donc avec son aide faire écrire, à nos étudiants, un petit algorithme de tri très simple basé sur les propriétés suivantes:

$$trie(t)=(\forall i\in\{1, \ldots , n-1\}\ t[i]\leq t[i+1])$$
$$non\_trie(t)=(\exists i\in\{1, \ldots , n-1\}\ t[i]>t[i+1])$$

de cette dernière nous pouvons en déduire la fonction indice_pour_non_trie(t) qui nous donne un indice tel que t[i]>t[i+1]. Avec la procédure echange(var t,i) qui échange les valeurs des emplacements mémoire t[i] et t[i+1], nous obtenons l'algorithme de tri suivant:

```
while not trie(t) do
   echange(t,indice_non_trie(t))
```

5.1.3) Les algorithmes de parcours

Une fois les algorithmes sur les propriétés bien compris, nous pouvons compléter les connaissances des étudiants en étendant ces algorithmes aux traitements des informations contenues dans un tableau ainsi qu'aux calculs.

5.1.3.1) Les traitements

Soit n une constante.
Soit tableau le type array[1..n] of T.
Soit t un tableau.
Soit traitement un algorithme dont la donnée ou/et le résultat est une valeur de type T. Comme son utilisation se fera sur des informations qui appartiennent à un tableau nous noterons son utilisation de la manière suivante:

```
traitement(t,i)
```

Si l'on veut appliquer un traitement sur toutes les informations contenues dans un tableau nous pouvons décrire cette opération de la manière suivante:

parcours_traitement(t,traitement)
= ∀i∈(1,...,n) traitement(t,i)

L'algorithme:

```
for i:=1 to n do
 traitement(t,i)
```

Les différents traitements se font dans l'ordre croissant des indices.

on peut également faire le traitement seulement sur les informations qui possèdent une propriété $\mathcal{P}$.

```
for i:=1 to n do
 if P(t,i) then
   traitement(t,i)
```

ou simplement effectuer le traitement sur la première information qui possède la propriété $\mathcal{P}$. Pour cela il suffit de prendre l'algorithme de existe_$\mathcal{P}$ et d'effectuer le traitement lorsque l'on a trouvé la première valeur qui respecte la propriété $\mathcal{P}$ (vexiste:=true).

```
i:=1;
vexiste:=false;
while (i≤n) and not vexiste do
 if P(t,i) then
  begin
   traitement(t,i);
   vexiste:=true
  end
 else
   i:=i+1
```

Nous ne savons pas (actuellement) s'il faut d'autres schémas, car ceux que nous donnons ne sont pas suffisants pour résoudre tous les problèmes. Par contre il faut se méfier de la multiplicité des schémas pour ne pas dérouter les étudiants de DEUG surtout si ces schémas sont des abstractions d'algorithmes qui résolvent des problèmes spécifiques. P.C. Scholl et J.P. Peyrin ont déjà beaucoup réfléchi sur les schémas itératifs [ScP88], nous espérons qu'ils continuent de le faire (et d'autres également) pour que nous puissions encore mieux apprendre les boucles à nos étudiants de DEUG.

5.1.3.2) **Les calculs**

Les calculs que nous présentons, sont définis de manière récursive. Nous allons donc donner au préalable une définition récursive d'un tableau. Un tableau peut être considéré comme étant la représentation informatique de l'ensemble suivant:

$\{t[1],t[2],.....,t[n]\}$

soit la définition récursive suivante d'un ensemble de n valeurs suivante:

$$\{t[1],t[2],.....,t[n]\} = \{t[1]\} \text{ si } n=1$$
$$= \{t[1],t[2],...,t[n-1]\} \cup \{t[n]\} \text{ sinon}$$

on définit un calcul de la manière suivante:
Soit t un tableau
Soit f une fonction
f: TxB ⟶ B
Soit f_0 une fonction
f_0: T ⟶ B

On définit calcul récursivement de la manière suivante:

$$calcul(t,f,f_0,n) = f_0(t[1]) \text{ si } n=1$$
$$f(t[n],calcul(t,f,f_0,n-1)) \text{ sinon}$$

en fait

$$calcul(t,f,f_0,n) = f(t[n],f(t[n-1], \ldots f(t[2]),f_0(t[1])) ..))$$

L'algorithme

```
vcalcul:=f0(t[1]);
for i:=2 to n do
  vcalcul:=f(t[i],vcalcul)
```

Lorsque $f_0(t[1])=f(t[1],cst)$ on peut utiliser la définition récursive d'un ensemble suivant:

$$\{t[1],t[2],.....,t[n]\} = \emptyset \text{ si } n=0$$
$$= \{t[1],t[2],...,t[n-1]\} \cup \{t[n]\} \text{ sinon}$$

on définit un calcul de la manière suivante:

$$calcul_0(t,f,cst,n) = cst \text{ si } n=0$$
$$f(t[n],calcul_0(t,f,cst,n-1)) \text{ sinon}$$

et on peut utiliser l'algorithme suivant:

```
vcalcul0:=cst;
for i:=1 to n do
  vcalcul0:=f(vcalcul0,t[i])
```

Remarque: si l'on veut calculer:
$f(t[1],f(t[2], \ldots. f(t[n-1],f_0(t[n])) \ldots.))$
on peut utiliser les boucles downto

5.2) Schémas récursifs

En ce qui concerne les schémas d'algorithmes récursifs, nous en proposons quatre, deux pour chacune des deux classes, la classe des algorithmes dont la récursivité est terminale et celle dont la récursivité est non terminale. Pour chaque schéma, nous donnons et nous prouvons le comportement des procédures et le résultat des fonctions.

Les schémas sont les suivants:

5.2.1) Les procédures

Soit x une valeur d'un type S.
Soit b(x) une expression booléenne.
Soit g une fonction de S ⟶ S
Soit $\varphi(x)$ un algorithme, qui dépend éventuellement de x sans modifier x.
Soit $\psi(x)$ un algorithme, qui dépend éventuellement de x sans modifier x.
Soit $\varphi_0(x)$ un algorithme, qui dépend éventuellement de x sans modifier x.
Alors
les procédures récursives suivantes:

```
procedure p1(x:S);
begin
  if b(x) then
    begin
      φ(x);
      p1(g(x))
    end
  else
     φ0(x)
end;
```

```
procedure p2(x:S);
begin
  if b(x) then
   begin
    φ(x);
    p2(g(x));
    ψ(x)
   end
  else
    φ0(x)
end;
```

terminent si la condition d'arrêt suivante est vérifiée:

$\exists!k\geq 0$ tq b(x) and b(g(x)) and ... and $b(g^{k-1}(x))$ and not $b(g^k(x))$

et ont exécuté la suite d'instructions suivante:

$$\varphi(x);\ \varphi(g(x));\ \ldots.\ ;\ \varphi(g^{k-1}(x));\ \varphi_0 g^k(x))$$

pour p1 et

$$\varphi(x);\varphi(g(x));\ ..\ ;\ \varphi(g^{k-1}(x));\varphi_0(g^k(x));\psi(g^{k-1}(x));\ ..\ ;\psi(g(x);\psi(x)$$

pour p2

Si un tel k n'existe pas, elle ne termine pas et exécute la suite infinie d'instructions suivante:

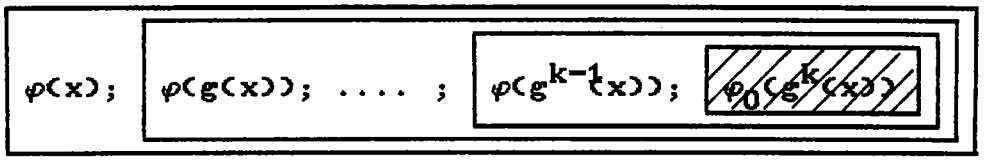
$$\varphi(x);\varphi(g(x));\ \ldots.\ ;\ \varphi(g^{k-1}(x));\varphi(g^k(x);\ \ldots.$$

La démonstration se fait en exécutant les procédures comme k est fixé (hypothèse), l'exécution des procédures est figée et nous savons donc exactement ce qu'ont exécuté ces procédures.

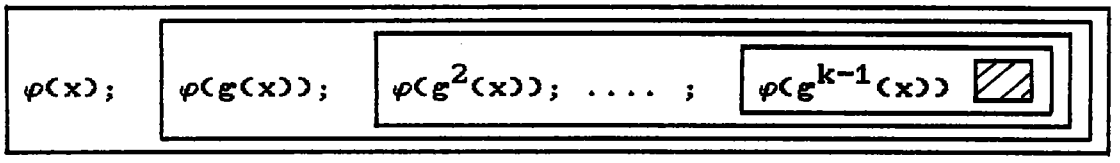
Structure récursive du comportement

Pour que nos étudiants puissent mieux trouver la procédure en fonction de ce qu'elle exécute, nous leurs donnons les structures récursives du comportement des schémas à l'exécution. Nous en donnons deux par procédures en distinguant le cas où quelque chose est exécutée à l'amorce et le cas où rien n'est exécuté à l'amorce. Nous donnons les structures suivantes:
(l'amorce est hachurée)

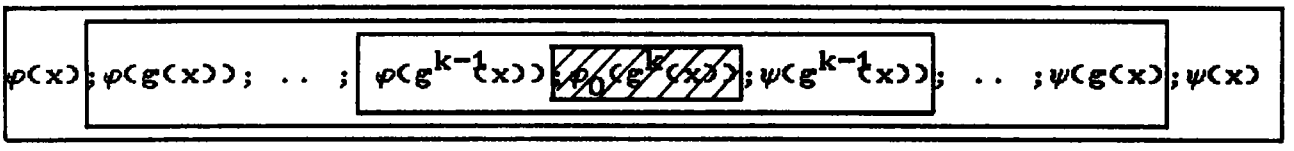
Pour p1 (cas de récursivité terminale)



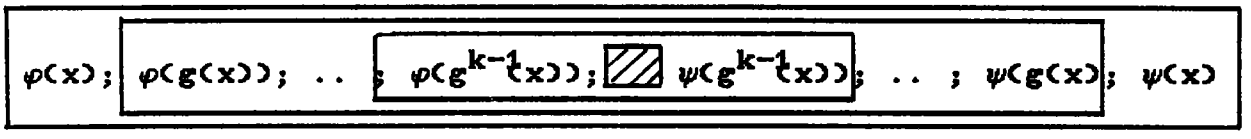
Pour p1 (cas de récursivité terminale) avec φ_0 ne rien faire



Pour p2 (deuxième cas de récursivité non terminale)



Pour p2 (cas de récursivité non terminale) avec φ_0 ne rien faire



5.2.2) Les fonctions

Pour les fonctions, nous donnons deux schémas (un par classe de récursivité).

Soit B le type du résultat de la fonction.
Soit x une valeur d'un certain type S.
Soit b(x) une expression booléenne.
Soit g une fonction de S ⟶ S
Soit f une fonction

f : SxB ⟶ B
(x,y) ⟶ f(x,y)

Soit f_0 une fonction

$$f_0 : S \longrightarrow B$$
$$x \longrightarrow f_0(x)$$

alors les deux fonctions suivantes:

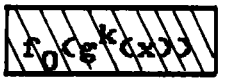
```
function p3(x:S):B;
begin
  if b(x) then
    p3:=p3(g(x))
  else
    p3:=f0(x)
end;
```

```
function p4(x:S):B;
begin
  if b(x) then
    p4:=f(x,p4(g(x)))
  else
    p4:=f0(x)
end;
```

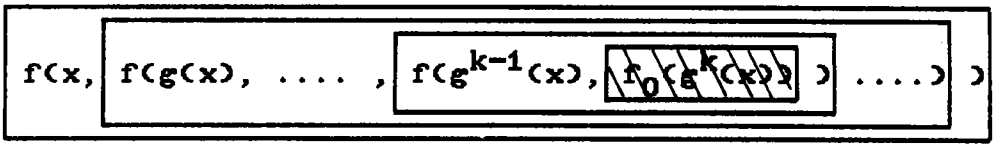
terminent si la même condition d'arrêt suivante est vérifiée:

et elles ont calculé

Pour p3 (cas de récursivité terminale)



Pour p4 (cas de récursivité non terminale)



Si un tel k n'existe pas, elle ne termine pas et donc ne donnera jamais de résultat.

5.2.3) Utilisation des schémas

Les schémas sont extrêmement importants, ils permettront à nos étudiants de lire autrement les algorithmes: ils pourront, par cette lecture, extraire d'un algorithme les informations essentielles que l'on retrouve dans toutes les écritures possibles de cet algorithme et d'en prevoir l'exécution. Les utilisations suivantes sont des exercices d'écoles pour la plupart, elles permettront d'initier les étudiants à cette lecture.

Une première utilisation est l'identification du schéma correspondant à un algorithme récursif pour en déduire son

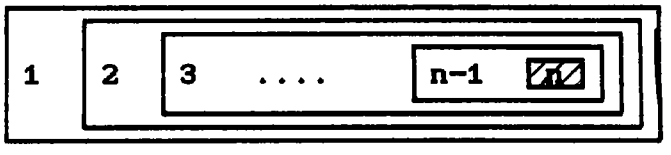
comportement (procédure) ou ce qu'il calcule (fonction).
Un exemple:
Soit la procédure suivante:

```
procedure p(x:S);
begin
  if b(x) then
   if c(x) then
    begin
     φ1(x);p(g(x))
    end
   else
    begin
     φ2(x);p(g(x));ψ2(x)
    end
end;
```

on constate que: la récursivité est non terminale sans else (φ_0)
le schéma correspondant est donc celui de p3 avec:

```
  x  ------>  x
b(x) ------>  b(x)
g(x) ------>  g(x)
φ(x) ------>  if c(x) then φ1(x) else φ2(x)
ψ(x) ------>  if not c(x) then ψ2(x)
```

Une deuxième utilisation est de faire le contraire c'est-à-dire qu'on leur donne un comportement d'un algorithme et ils doivent en déduire l'algorithme. C'est ce qui ce passe en réalité lorsque l'on veut écrire un algorithme.
Un exemple:
Si nous demandons aux étudiants de nous donner une procédure récursive terminale qui écrit les entiers de 1 à n, nous avons donc la structure récursive du comportement de la procédure suivante:



on en déduit facilement que:

```
   x ------> i
g(x) ------> g(i)=i+1
b(x) ------> i≠n
φ(x) ------> write(i)
φ0(x) ------> write(i)
```

comme b(x) vaut i≠n il faut que n soit également une donnée nous avons donc:

$x \longrightarrow (i,n)$
$g(x) \longrightarrow g(i,n)=(i+1,n)$
$b(x) \longrightarrow i \neq n$
$\varphi(x) \longrightarrow$ write(i)
$\varphi_0(x) \longrightarrow$ write(i)

d'où on obtient la procédure suivante:

```
procedure p(i,n:integer);
begin
  if (i<>n) then
   begin
    write(i);
    p(i+1,n)
   end
  else
    write(i)
end;
```

Une troisième application des schémas est d'obtenir des règles d'élimination de la récursivité. Nous donnons

* une règle pour les trois schémas de la classe des algorithmes dont la récursivité est terminale.
* deux règles pour les schémas de la classe des algorithmes dont la récursivité est non terminale.
  - une qui consiste à recalculer les valeurs de l'appel récursif précédent (existence de g^{-1}). Pour ces règles, nous proposons des transformations pour avoir des algorithmes plus rapides et moins gourmands (une seule boucle à la place des deux nécessaires).
  - une qui utilise une pile.

Exemple de transformation d'un algorithme itératif obtenu à partir de la version récursive à l'aide d'une règle d'élimination

Soit la version récursive de la fonction qui calcule x^n en fonction de $(x^2)^{n/2}$:

```
function exp(x:real;n:integer):real;
begin
  if (n=0) then
    exp:=1
  else if pair(n) then
    exp:=exp(x*x,n div 2))
  else
    exp:=x*exp(x*x,n div 2)
end;
```

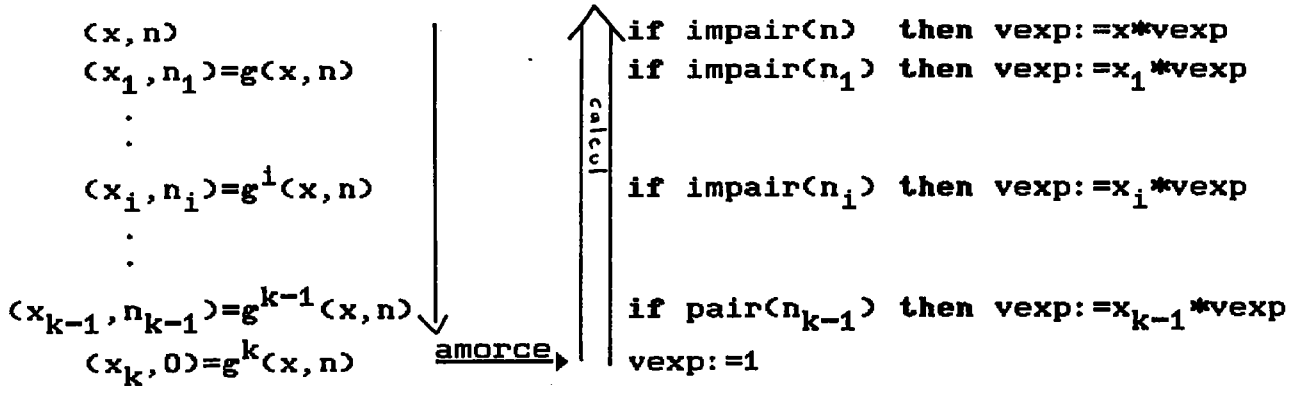
bien que g^{-1} n'existe pas, supposons qu'elle existe, nous avons donc:

$$
\begin{aligned}
x &\longrightarrow (x,n)\\
b(x) &\longrightarrow (n\neq 0)\\
g(x) &\longrightarrow g(x,n)=(x^2,n \text{ div } 2)\\
f_0(x) &\longrightarrow 1\\
f(x,y) &\longrightarrow f((x,n),y)= \begin{cases} y & \text{si } n \text{ est pair}\\ x.y & \text{si } n \text{ est impair}\end{cases}
\end{aligned}
$$

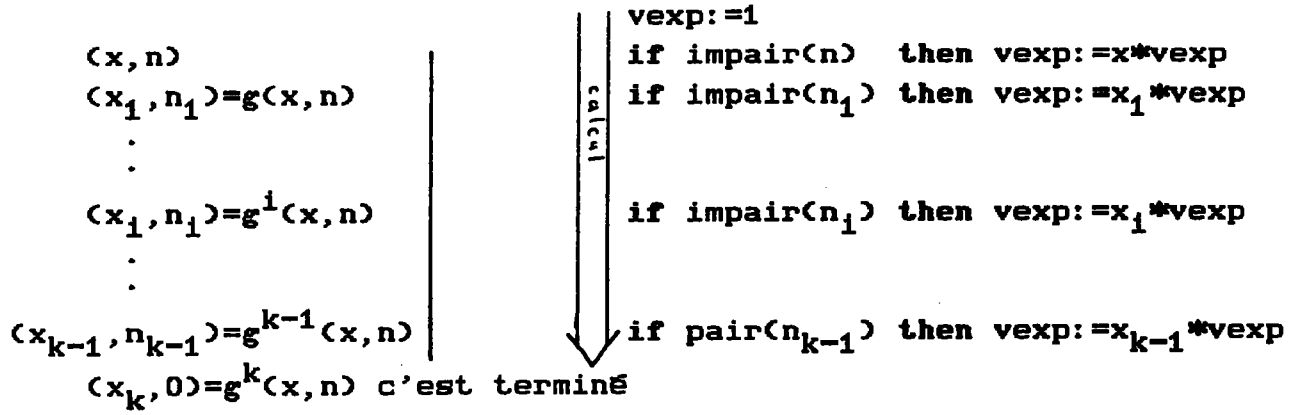
d'où en appliquant la règle adaptée [Can90c], nous obtenons la fonction itérative suivante:

```
function exp(x:real;n:integer):real;
var n0:integer;vexp:real;
begin
  n0:=n;
  while (n<>0) do
   begin
    x:=x*x;
    n:=n div 2
   end;
  vexp:=1;
  while (n0<>n) do
   begin
    x:=sqrt(x);
    n:=g_n^-1(n); {n'existe pas}
    if impair(n) then
     vexp:=x*vexp
   {else
     vexp:=vexp est inutile}
   end;
  exp:=vexp
end;
```

que calcule cette fonction:
schéma d'appel



Ici on peut calculer la valeur du résultat en descendant, comme le montre le schéma suivant:



d'où les étudiants en déduisent l'écriture de la fonction itérative avec une seule boucle suivante:

```
function exp(x:real;n:integer):real;
var vexp:real;
begin
 vexp:=1;
 while (n<>0) do
  begin
   if impair(n) then
     vexp:=x*vexp;
   n:=n div 2;
   x:=x*x
  end;
 exp:=vexp
end;
```

si nous donnions directement cet algorithme aux étudiants, ils auraient de grandes difficultés à le comprendre car il ne préserve

plus la propriété de la première version itérative qui est que la variable vexp contient toujours x^n si $g_n^{-1}(n)$ existait.

Il faut par contre freiner les étudiants car ils veulent utiliser cette technique systématiquement. Nous leur proposons donc la variante de x^n en fonction de $x^{n/2}$ suivante:

$x^0=1$

$x^n=(x^{n/2})^2$ si n est pair

$x^n=x.(x^{n/2})^2$ si n est impair

Le schéma du calcul est le suivant:

$(x_i,n_i)=g^i(x,n)$ **if** pair(n_i) **then** vexp:=vexp*vexp
else vexp:=x*vexp*vexp

Certains nous assurent que leur boucle calcule bien x^n car avec ce type d'algorithme ils sont persuadés qu'il n'y a aucun problème avec les nombres pairs, ils essayent donc avec 1, 3, 5, 7 voire 9. Comme la décomposition binaire de ces entiers est symétrique, ils trouvent la bonne solution. Il suffit dans ce cas de leur faire exécuter leur boucle avec $n=2^k$.

Signalons que lorsqu'un étudiant connaît le résultat de son algorithme, celui-ci triche volontairement ou pas pour que son algorithme faux calcule la bonne solution. Cela n'arrive pas qu'en informatique, on ne compte plus le nombre d'étudiants qui se trompent en démontrant un théorème et arrivent quand même au bon résultat.

Chapitre 6

L'APPROCHE FONCTIONNELLE POUR LES LISTES LINEAIRES

La base de ce chapitre est l'article "PASCAL, un langage applicatif" [Can91] que nous avons présenté lors des journées langages applicatifs dans l'enseignement de l'informatique [LAE91].

Les types abstraits ont déjà été utilisés pour enseigner les listes [PMS88], [GSF87], [ScP88] (type abstrait séquence) et le besoin de cacher la représentation dans des fonctions de construction s'est déjà fait ressentir, tout au moins au niveau des arbres [BoM83], pour obtenir des programmes plus lisibles et donc plus compréhensibles; mais à part le livre de P.-C. Scholl et J.-P. Peyrin [ScP88], cela s'adresse à des étudiants de second cycle.

En ce qui concerne les listes linéaires, nous avons une approche tout à fait similaire à celle de H. Abelson et G.J. Sussman [AbS89], nous adoptons tout d'abord une approche fonctionnelle qui permettra à nos étudiants d'avoir une vision abstraite et inductive des listes linéaires et les obligera donc à utiliser une analyse abstraite, récursive et applicative des problèmes sans se soucier de la représentation machine. PASCAL, le langage que nous employons ne possède pas de type inductif, il suffit donc de définir ce type et ses fonctions de base proprement, comme nous le proposons par la suite. Nos étudiants pourront, sans trop d'effort de leur part, déduire de leurs analyses des fonctions PASCAL. Nous avons voulu situer la réflexion de nos étudiants à un niveau d'abstraction différent, nous voulions qu'ils aient un cadre où ils puissent raisonner proprement car l'expérience montre que les étudiants sont en général débordés et déroutés par l'utilisation des pointeurs et des effets de bords. Puis, toujours à cause du coût d'un algorithme à l'exécution, on apprendra aux étudiants à transformer leur fonctions correctes:

1) en fonctions qui utilisent pour leur résultat les mêmes emplacements mémoires que les listes transmises en données comme le conseillent H. Abelson et G.J. Sussman [AbS89].

2) en procédures qui utilisent également pour leurs résultats la variable qui sert de donnée (passage par variable) et cela pour faire des insertions ou des suppressions ou des

modifications de manière physique dans des variables de type liste. Nous obtiendrons par ce moyen les procédures récursives "classiques" sur les listes avec l'appel récursif pour la variable l^.suivant.

6.1) Définition récursive de L(V)

On définit de manière récursive l'ensemble L(V) des listes linéaires de la manière suivante:

$$L(V) = \{()\} \cup \{(a,l) \text{ tq } a \in V \text{ et } l \in L(V)\}$$

liste vide liste non vide

Construction de la liste qui contient les n valeurs suivantes:
$a_1, a_2, \ldots, a_n$

$() \in L(V)$

$(a_n,()) \in L(V)$ car $a_n \in V$

$(a_{n-1},(a_n,())) \in L(V)$ car $a_{n-1} \in V$

.
.

$(a_2,(a_3,(\ldots\ldots,(a_{n-1},(a_n,()))\ldots\ldots))) \in L(V)$

$(a_1,(a_2,(a_3,(\ldots\ldots,(a_{n-1},(a_n,()))\ldots\ldots)))) \in L(V)$ car $a_1 \in V$

$\in V$; $\in L(V)$

Remarque: Les parenthèses sont importantes car elles pousseront les étudiants à avoir une vision non globale d'une liste. Ils raisonneront sur une liste non vide en terme de valeur et suivant et non en terme d'un objet qui contient les n valeurs $a_1, a_2, \ldots. a_n$.

6.2) Fonctions de base sur L(V)

6.2.1) Fonctions de consultation

vide : L(V) ⟶ {vrai,faux}
() ⟶ vrai
(a,l) ⟶ faux

valeur : L(V)-{()} ⟶ V
(a,l) ⟶ a

suivant : L(V)-{()} ⟶ L(V)
(a,l) ⟶ l

Remarque: Avec ces trois fonctions de base, nous pouvons consulter toutes les valeurs d'une liste:

VALEUR : L(V) ⟶ $\mathcal{P}$(V)
() ⟶ ∅
(a,l) ⟶ {a} ∪ VALEUR(l)

VALEUR((a,l))={valeur((a,l))} ∪ VALEUR(suivant((a,l)))

6.2.2) Fonctions de construction

Il faut pouvoir construire des listes linéaires. Comme il y a deux cas, la liste vide et les listes non vides, nous aurons deux fonctions de construction.

consvide : ⟶ L(V)
⟶ ()

cons : VxL(V) ⟶ L(V)
a,l ⟶ (a,l)

6.3) Utilisation abstraite

Puis nous assurons aux étudiants que nous réussirons sans problème (avec les pointeurs) à définir le type liste qui représente L(V) et à en déduire les fonctions de base PASCAL associées, et nous leur demanderons d'utiliser ces fonctions de base pour régler tous leurs problèmes sur les listes linéaires. Comme les étudiants n'ont qu'une vision abstraite de ces listes, ils ne se trompent guère lorsqu'ils définissent leurs fonctions. Le travail fourni sera payant lorsque nous passerons aux arbres car l'approche pourra être identique. L'étudiant ne sera pas effrayé, il ne sera pas confronté à deux problèmes en même temps, la compréhension de la récursivité, la compréhension du type arbre et de ses manipulations.

6.3.1) Analyse récursive et applicative

En ce qui concerne la résolution d'un problème sur les listes, nous conseillons de procéder de la manière suivante:

1) Si la fonction n'est pas simple, une analyse abstraite s'impose
2) On en déduit une fonction

6.3.1.1) Exemple de fonction sur les listes: l'insertion d'une valeur v dans une liste ordonnée

Analyse abstraite

si l=() alors insert(v,())=(v,())
car la liste qui ne contient que la valeur v est ordonnée
si l=(a,l') et v≤a alors insert(v,(a,l'))=(v,(a,l'))=(v,l)
car si l est ordonnée et v≤a, v sera plus petit que toutes les valeurs de l' donc de l et par conséquent (v,l) est ordonnée
si l=(a,l') et v>a insert(v,(a,l'))=(a,insert(v,l'))
car par récurrence (induction) insert(v,l') est ordonnée et a est plus petit que v ainsi que toutes les valeurs de l' donc la liste obtenue est ordonnée

D'où l'on obtient la fonction suivante:

```
function insert(v:V;l:liste):liste;
begin
  if vide(l) then
    insert:=cons(v,consvide)
  else if (v<=valeur(l)) then
    insert:=cons(v,l)
  else
    insert:=cons(valeur(l),insert(v,suivant(l)))
end;
```

6.3.1.2) Exécution symbolique

```
insert(3, (1,(2,(4,(5,(6,()))))) )
||
cons(1, insert(3, (2,(4,(5,(6,())))) ) )
|||      ||
|||      cons(2, insert(3, (4,(5,(6,()))) ) )
|||      |||      ||
|||      |||      cons(3, (4,(5,(6,()))) )
|||      |||      ||
(1,      (2,      (3,(4,(5,(6,()))))))
```

6.3.2) Théorème d'induction sur L(V)

Nous pouvons de plus prouver nos fonctions à l'aide du théorème

suivant comme cela est conseillé dans [Gra86]:

Théorème:

> Soit $\mathcal{P}(l)$ ume propriété sur $l \in L(V)$
> Si 1) $\mathcal{P}(())$
> 2) $\forall a \in V, \forall l \in L(V)\ \mathcal{P}(l) \Rightarrow \mathcal{P}((a,l))$
> alors
> $\forall l \in L(V)\ \mathcal{P}(l)$

On peut démontrer que la fonction insert est correcte en démontrant la propriété suivante [Can90c]:

$\forall l \in L(V), \forall v \in V\ ord(l) \Rightarrow ord(insert(v,l))$

6.3.3) Les schémas de fonction sur les listes

nous avons dégagé deux schémas importants sur les listes qui permettent d'avoir deux "sens de parcours" des listes linéaires qui sont:

1) dans le sens d'apparition des valeurs dans la liste.
2) dans le sens inverse d'apparition des valeurs dans la liste.

Soit f une fonction de VxB dans B et f_0 une constante de type B:

$f: VxB \longrightarrow B \qquad f_0 \longrightarrow B$

$(a,b) \longrightarrow f(a,b) \qquad \longrightarrow f_0$

Soit $l=(a_1,(a_2, \ldots (a_n,()) \ldots))$

si nous voulons calculer les expressions suivantes:

1) $f(a_1,f(a_2, \ldots f(a_n,f_0) \ldots))$ (f_0 si $l=()$ donc $n=0$)
2) $f(a_n,f(a_{n-1}, \ldots f(a_1,f_0) \ldots))$ (f_0 si $l=()$ donc $n=0$)

nous pouvons utiliser les fonctions suivantes:

1) si nous posons

$l=(a_1,(a_2, \ldots (a_n,()) \ldots))$

$p(l)=p((a_1,(a_2, \ldots (a_n,()) \ldots)))$

$=f(a_1,f(a_2, \ldots f(a_n,f_0) \ldots))$

nous avons

a) si $l=()$ $p(())=f_0$
b) si $l \neq ()$ donc $n>0$

$f(a_2, \ldots f(a_n,f_0) \ldots)=p((a_2, \ldots (a_n,()) \ldots))$

$=p(suivant(l))$

et donc

$p((a_1,(a_2, \ldots (a_n,()) \ldots)))=f(a_1,p(suivant(l)))$

$=f(valeur(l),p(suivant(l)))$

d'où nous obtenons la fonction suivante:

```
    function p(l:liste):B;
    begin
      if vide(l) then
         p:=f0
      else
         p:=f(valeur(l),p(suivant(l))
    end;
```

2) si nous posons

$l=(a_1,(a_2, \ldots (a_n,()) \ldots))$

$p(l)=p((a_1,(a_2, \ldots (a_n,()) \ldots)))$

$=f(a_n,f(a_{n-1}, \ldots f(a_1,f_0) \ldots))$

nous avons

$p(())=f_0$

$p(l)=f(acces\_en\_queue(l),p(supp\_en\_queue(l)))$ si $l\neq()$

comme les listes linéaires ne sont pas adaptées aux accès et suppression en queue, nous préférons donner la fonction suivante:

```
    function p(l:liste):B;
    begin
      p:=p'(l,f0)
    end;
```

avec

```
    function p'(l:liste,vp:B):B;
    begin
      if vide(l) then
         p':=vp
      else
         p':=p'(suivant(l),f(valeur(l),vp))
    end;
```

Preuve: soit $l=(a_1,(a_2, \ldots , (a_n,()) \ldots))$

et comme

$p(l)=p'((a_1,(a_2, \ldots , (a_n,()) \ldots)) , f_0)$

$=p'((a_2, \ldots (a_n,()) \ldots) , f(a_1,f_0))$

$= \ldots$

$=p'((a_i, \ldots (a_n,()) \ldots) , f(a_{i-1}, \ldots f(a_1,f_0) \ldots))$

$=p'((a_{i+1}, .. (a_n,()) ..) , f(a_i,f(a_{i-1}, .. f(a_1,f_0) ..)))$

$= \ldots$

$=p'(() , f(a_n,f(a_{n-1}, \ldots f(a_1,f_0) \ldots)))$

$=f(a_n,f(a_{n-1}, \ldots f(a_1,f_0) \ldots))$

cqfd

6.4) La réalisation Pascal à l'aide de variable dynamique

Comme nous avons choisi le langage de programmation PASCAL nous pouvons, pour que nos étudiants programment leurs fonctions, mettre toutes les déclarations qui suivent dans une librairie. Par contre nous pensons utile de ne rien leur cacher [DMa90] pour qu'ils puissent mieux comprendre ce qui se passe au niveau de la machine pour mieux s'en abstraire. De plus le fait d'expliquer ce qu'est un pointeur à nos étudiants nous permettra de leur apprendre:

- ce qu'est une adresse et le besoin de cette notion pour stocker une liste
- les différentes allocations mémoires (statique, en pile, en tas).
- les techniques d'optimisation des algorithmes sur les listes.

Ce qui est le plus important est surtout l'approche fonctionnelle, elle sera un refuge pour nos étudiants qui ne comprennent pas correctement la notion de pointeur, elle permettra à nos étudiants d'analyser leurs problèmes, d'écrire des fonctions et de les exécuter sur une machine. Ils pourront donc avoir du temps pour assimiler la représentation des listes. L'enseignant doit préserver son approche fonctionnelle dans l'enseignement des listes de liste, arbres ...

6.4.1) le type liste

Il faut pour que le résultat d'une fonction puisse être de type liste qu'une variable de type liste soit du style:

et avec le contenu de cette variable nous devons pouvoir accéder aux deux informations de la liste valeur et suivant. Car, si nous considérons que cette variable contient les deux informations, il faudra également que l'information suivant en contienne deux autres. Nous arrivons donc au type tableau.

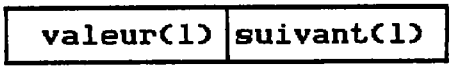
Comment un informaticien peut-il, avec ce contenu, accéder aux deux informations?

1) les deux informations étant de types différents, le type qui doit contenir ces deux informations doit être:

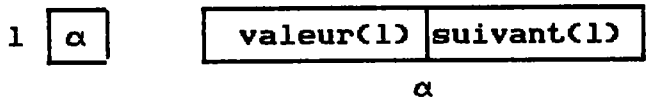
```
record
  valeur:V;
 suivant:liste
end
```

chaque type défini devant avoir un nom, nous l'appellerons place.

une variable de type place sera représentée de la manière suivante:



2) Pour accéder à la variable de type place qui contient les deux informations valeur(l) et suivant(l), la variable l doit contenir une information qui nous permette d'y accéder d'où la nécessité de connaître l'adresse dans la zone de mémoire de cette place.



Nous avons l'habitude de noter cela de la manière suivante:

Donc le type liste est le suivant:

```
type    { V doit être défini s'il n'est pas prédéfini }
        liste = "adresse d'une place";
        place = record
                  valeur:V;
                 suivant:liste
                end
```

dont l'écriture en PASCAL est la suivante:

```
type liste = ^place;
     place = record
               valeur:V;
              suivant:liste
             end;
```

6.4.2) Réalisation des fonctions de base

6.4.2.1) Réalisation des fonctions de consultation

```
function vide(l:liste):boolean;
begin
  vide:=(l=nil)
end;

function valeur(l:liste):V;
begin
  valeur:=l^.valeur
end;

function suivant(l:liste):liste;
begin
  suivant:=l^.suivant
end;
```

Remarque: Le domaine de définition des fonctions valeur et suivant est l'ensemble des listes non vide. Si les fonctions PASCAL associées sont appelées avec la liste vide, il y aura une erreur à l'exécution. Cela est pour nous une chose saine. Plus tard avec de nouveaux compilateurs nous espérons que ces erreurs pourront être détectées à la compilation. Nous pensons l'inclure dans le logiciel décompositeur que nous avons l'intention de développer (voir chapitre 10).

6.4.2.2) Réalisation des fonctions de construction

```
function consvide:lste;
begin
  consvide:=nil
end;

function cons(a:V;l:liste):liste;
var vcons:liste;
begin
  new(vcons);
  vcons^.valeur:=a;
  vcons^.suivant:=l;
  cons:=vcons
end;
```

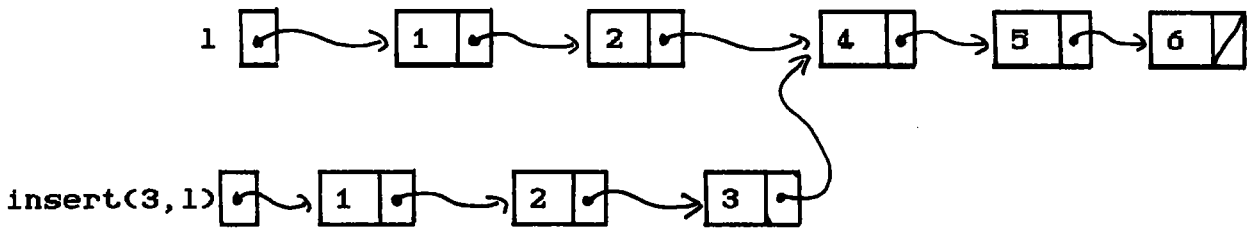
Remarque: C'est dans la fonction cons que l'on cache toutes les actions qui permettent la demande d'une place sur le tas ainsi qu'une affectation d'une valeur dans chaque variable de la place. C'est cette fonction qui nous permet de passer

- de la programmation impérative (actionnelle) à la programmation applicative (fonctionnelle)
- de l'analyse applicative (raisonnement applicatif) à une mise en oeuvre impérative.

6.4.3) Un vrai passage par valeur

Avec les pointeurs les fonctions nous assurent d'une propriété extrêmement importante: c'est un passage par valeur d'une liste et non un passage par valeur d'une adresse.

Nous prenons l'exécution précédente de l'insertion:



Après l'exécution de la fonction insert la liste l n'est pas modifiée (au sens liste et pas uniquement au sens pointeur).

Donc après l'exécution de cette fonction nous pouvons nous servir de la liste l si nous restons dans un style fonctionnel. Par contre si nous mélangeons dans nos algorithmes le style fonctionnel et la technique d'effet de bord, il faudra bien évidemment faire un peu plus attention car l et insert(a,l) peuvent partager une partie des places, comme l et cons(a,l) partagent tous les places de la liste l.

6.5) Transformation des fonctions

Si notre problème ne doit pas reconstruire, c'est-à-dire ne pas utiliser la fonction cons pour faire des modifications physiques sur les listes, nous pouvons transformer les fonctions:

6.5.1) En fonction

H. Abelson et G.J. Sussman [AbS89] proposent pour transformer leurs fonctions d'utiliser la technique de l'effet de bord pour optimiser l'exécution de celles-ci (set-car et set-cdr). Personnellement nous n'apprécions pas l'utilisation des fonctions que nous donnons dans ce paragraphe. Nous ne les enseignons pas, nous voulons simplement montrer qu'il possible de les écrire dans le langage de programmation que nous avons choisi. Nous préférons

lorsqu'il s'agit de "bricoler" les emplacements mémoires utiliser le style actionnel.

Dans l'exemple de la fonction insert il faut éviter l'évaluation de cons(valeur(l),insert(v,suivant(l))), nous obtenons donc la fonction suivante:

```
function insert(v:V;l:liste):liste;
begin
  if vide(l) then
     insert:=cons(v,consvide)
  else if (v≤valeur(l) then
     insert:=cons(v,l)
  else
   begin
     l^.suivant:=insert(v,suivant(l));
     insert:=l
   end
end;
```

Nous pouvons même rester "indépendant" de la représentation (l^.suivant) au niveau de l'utilisation, en écrivant les deux fonctions de modification des champs valeur (set-car) et suivant (set-cdr) d'une variable de type liste non vide suivantes:

```
function modif_valeur(l:liste;v:V)liste;
begin
  l^.valeur:=v;
  modif_valeur:=l
end;

function modif_suivant(l:liste;lsuiv:liste):liste;
begin
  l^.suivant:=lsuiv;
  modif_suivantt:=l
end;
```

et nous pouvons donc écrire la fonction insert de la manière suivante:

```
function insert(v:V;l:liste):liste;
begin
  if vide(l) then
    insert:=cons(v,consvide)
  else if (v<=valeur(l)) then
    insert:=cons(v,l)
  else
    insert:=modif_suivant(l,insert(v,suivant(l)))
end;
```

6.5.2) En procédure: Approche procédurale

Comme l est modifiée cette variable est donnée et résultat de notre problème (l:=insert(v,l)) nous pouvons écrire une procédure qui résoudra notre problème. L'obtention de procédure n'est pas simple donc nous conseillons à nos étudiants:

1) d'écrire systématiquement la fonction, elle est souvent simple, on a peu de chances de se tromper.
2) d'en déduire une procédure avec un passage par variable pour la donnée qui est en même temps le résultat. Comment? c'est très simple: la fonction donne une ossature au niveau des tests à la procédure car elle doit faire les mêmes tests. Nous considérons donc la fonction comme une analyse saine de notre problème. Puis dans chaque cas la fonction donne la valeur du résultat, il faut simplement l'affecter à la variable qui doit contenir ce résultat.

L'ossature de la procédure est la suivante, le résultat de la fonction se trouve entre accolades:

```
procedure insert(v:V; var l:liste);
begin
  if vide(l) then
            {cons(v,consvide)}
  else if (v<=valeur(l)) then
            {cons(v,l)}
  else
            {cons(valeur(l),insert(v,suivant(l)))}
end;
```

Pour les deux premiers cas, la réponse est simple il suffit de mettre le résultat de la fonction dans l. Pour le dernier cas, nous ne voulons pas exécuter le cons. Il suffit de prendre la place dont l'adresse est dans l, elle contient déjà valeur(l) dans le champ valeur, puis de mettre dans le champ suivant la valeur de insert(v,suivant(l)). La procédure que l'on est en train d'écrire va résoudre le problème; nous obtenons donc:

```
procedure insert(v:V; var l:liste);
begin
  if vide(l) then
    l:=cons(v,consvide)
  else if (v<=valeur(l)) then
    l:=cons(v,l)
  else
    insert(v,l^.suivant)
end;
```

Cet exemple est encore relativement simple, montrons que l'obtention des procédures n'est pas toujours si aisée. Il faut que l'étudiant (et l'enseignant) fasse très attention car il doit gérer et connaître les variables qui contiendront les (ou le) résultats de sa procédure. C'est un art difficile que nous montrons à nos étudiants pour qu'ils procèdent toujours avec rigueur, car si nous ne leur donnons que des exemples simples comme la procédure insert ils sont souvent persuadés que le passage d'une fonction à une procédure n'est qu'une traduction des fonctions où l'on remplace valeur(l) par l^.valeur et suivant(l) par l^.suivant.

6.5.2.1) Un tri par insertion

A l'aide de la fonction insert précédente, nous pouvons sans problème écrire une fonction tri_insertion suivante:

```
function tri_insertion(l:liste):liste;
begin
  if vide(l) then
   tri_insertion:=consvide
  else
   tri_insertion:=insert(valeur(l),tri_insertion(suivant(l)))
end;
```

Dans ce cas la fonction est simple. De plus, le principe d'induction sur les listes nous assure que la liste tri_insertion(l) sera ordonnée et contiendra bien toutes les valeurs de la liste l:

* si l=() tri(l)=() donc () est ordonnée et contient bien toutes les valeurs de la liste l.

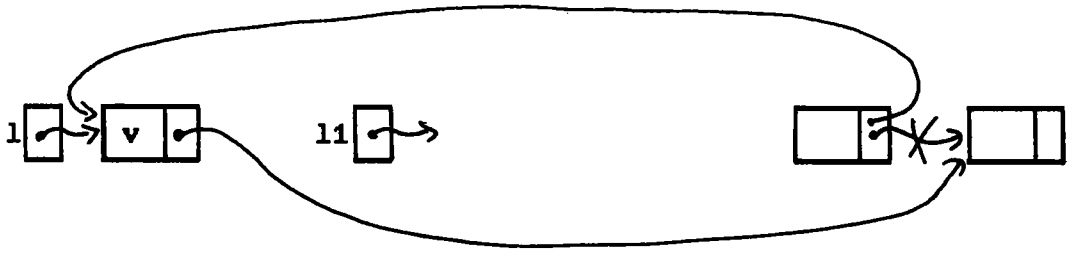
* si l=(a,l') alors tri_insertion(l)=insert(a,tri_insertion(l')) donc par hypothèse de récurrence (ou d'induction) tri_insertion(l') est ordonnée et contient bien toutes les valeurs de la liste l' et insert(a,tri_insertion(l')) sera une liste ordonnée, de plus elle contiendra toutes les valeurs de tri_insertion(l') donc toutes les valeurs de l' (par hypothèse d'induction) plus la valeur a donc toutes les valeurs de la liste l.

Réalisation de la procédure tri (le but étant de ne pas utiliser la fonction cons car toutes les valeurs ont déjà une place sur le tas).

```
procedure tri_insertion(var l:liste);
begin
  if vide(l) then
    l:=consvide
  else
   begin
     tri_insertion(l^.suivant);
     {le résultat est dans l^.suivant}
     insert(valeur(l),l^.suivant);
     {le résultat est dans l^.suivant}
     l:=l^.suivant
     {le résultat est dans l}
   end
 end;
```

Bien que cette procédure utilise moins de cons que la fonction, elle utilise n cons où n est la longueur de la liste, car la procédure insert en utilise un à chaque appel. Pour régler notre problème il faut que la procédure insert insère la place qui contient valeur(l), donc dont l'adresse se trouve dans l, dans la liste l^.suivant.

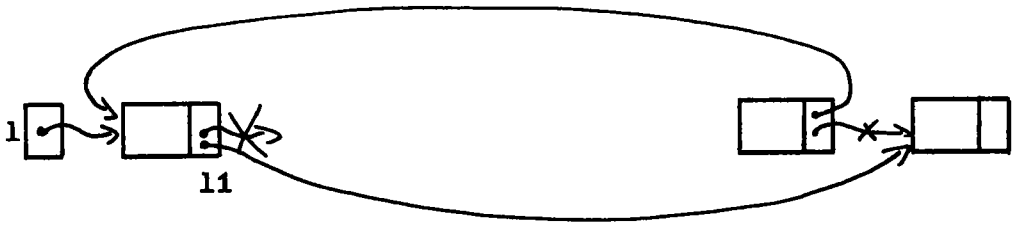
```
procedure insert'(l:liste; var l1:liste);
{fait comme insert(valeur(l),l1) mais sans cons}
begin
  if vide(l1) then
   begin
    l1:=l;l^.suivant:=consvide          {cons(v,consvide)}
   end
  else if (valeur(l)<=valeur(l1)) then
   begin
    l^.suivant:=l1;l1:=l                {cons(v,l1)}
   end
  else
    insert'(l,l1^.suivant)
end;
```



si l'on écrit la procédure de tri suivante:

```
procedure tri_insertion(var l:liste);
begin
  if vide(l) then
   l:=consvide
  else
   begin
     tri_insertion(l^.suivant);
     insert'(l,l^.suivant);
     l:=l^.suivant
   end
 end;
```

on peut remarquer qu'elle est fausse car la fonction insert' modifie l1 et l^.suivant et lors de l'appel ce sont les mêmes variables.



nous avons tout simplement perdu la valeur du résultat, il faut donc utiliser une autre variable pour la sauvegarder. Nous obtenons donc:

```
procedure tri_insertion(var l:liste);
var l1:liste;
begin
  if vide(l) then
   l:=consvide
  else
   begin
     l1:=l^.suivant;
     tri_insertion(l1);
     insert'(l,l1);
     l:=l1
   end
 end;
```

Chapitre 7

EVALUATIONS

Nous allons proposer un certain nombre de sujets de DEUG que nous avons eu l'occasion de donner lors de contrôles. Nous donnerons et interpréterons les résultats obtenus par les étudiants. Comme nous n'avons pas assuré l'enseignement initial des huit premières semaines de DEUG première année depuis un certain temps, sauf lors de semaines de remise à niveau en DEUG deuxième année, nous ne donnerons pas de sujet pour cet enseignement.

7.1) Sujet de DEUG1 (PIE+TM)

Voici le sujet de DEUG première année donné en 1988-89, année où nous avons introduit le calcul des propriétés. A cette époque, à cause d'un manque d'enseignants en informatique, nous avons regroupé les étudiants de la filière TM et PIE. Les étudiants de PIE étant plus nombreux (128), les résultats sont plus fiables. Remarque: les étudiants étant nombreux (160) nous n'avions pas voulu faire un cours "classique" sur les boucles while, pour faciliter la tâche des chargés de TD. Il fallait que tous les étudiants fassent les mêmes exercices et surtout qu'ils les traitent de la même façon. A part notre remarque sur l'exercice 1, nous sommes assez optimistes pour l'avenir.

7.1.1) Le sujet

Université de Metz — Mercredi 24 mai 1989
Faculté des Sciences
UFR MIM + SFA
DEUG A1 TM + PIE
Année 1988-89
In2.2 : D. Cansell

CONTROLE D'INFORMATIQUE

Documents manuscrits du cours et TD autorisés

Exercice 1) Soient les déclarations suivantes:

```
const n=20;m=3;
type cas=(cas1,cas2,cas3,cas4,cas5);
       T=array[cas] of 0..n;
      t1=record
           c:char;
           d:array[1..m] of T
         end;
       t2=record
            ff:array[cas] of t1;
            i,j:integer;
             c:cas;
            t:t1
          end;
var cask:cas; x:T; h,k:t1; y:t2;
```

Dites si les expressions ou les instructions suivantes sont correctes. Justifiez vos réponses.

1) x[cask]
2) h.d.c[4]
3) y.t.c
4) x[succ(cask)]
5) y.t:=y.ff[cas3]
6) y.ff[y.c].d[6][y.c]:=15 mod y.i
7) y:=k
8) h.c:=x[cas6]

Exercice 2) Exécutez le programme suivant:

```
program ex2;
const n=7;
var i,j,x:integer;
  procedure p(var i,k:integer);
  var x:integer;
     function f(n,m:integer):integer;
     begin
       f:=2*n-m
     end;
   begin
      x:=f(n,k);k:=k-2;
      i:=x+k-j
   end;
begin  j:=5;x:=8;
  p(i,x);
  writeln(i,j,x);
  p(j,j);
  writeln(i,j,x)
end;
```

Exercice 3) Un petit algorithme de tri.

Soient les déclarations suivantes:

```
const n=30;
type tableau=array[1..n] of integer;
var T:tableau;
```

On dit que le tableau T est trié (ordonné) si et seulement si:

$\forall\ i \in (1,\ldots,n-1)\ \ T[i] \leq T[i+1]$

a) Ecrivez une fonction Pascal trie qui donnera le résultat vrai si un tableau t est trié et faux sinon. (la fonction a une donnée qui est t, son nom est trie, son résultat booléen).

b) Si le tableau t n'est pas trié il sera intéressant de connaître un des indices i pour lequel on a t[i]>t[i+1] afin de pouvoir permuter t[i] et t[i+1] pour rendre le tableau t "un peu plus trié".

Soit t un tableau non trié, écrivez une fonction Pascal indice qui nous donnera la valeur d'un indice i tel que t[i]>t[i+1].(la donnée de la fonction est t, son nom indice, son résultat un entier).

c) Soit la procédure echange, dont l'utilisation est la suivante:
echange(i,j) qui permute les valeurs des deux variables i et j.

Utilisez cette procédure et vos deux fonctions pour écrire une procédure tri qui modifiera le tableau t pour qu'il contienne les mêmes valeurs mais triées.

Principe de l'algorithme. Tant que le tableau n'est pas trié on cherche un indice i tel que t[i]>t[i+1] et on échange les variables t[i] et t[i+1].

d) Rappel d'In1.0: Ecrivez la procedure echange.

7.1.2) Les résultats

Exercice 1)

17 étudiants n'ont pas fait l'exercice.
24 étudiants ont essayé de le faire sans succès.
17 étudiants ont fait la moitié en faisant des fautes.
22 étudiants ont correctement fait la moitié.
13 étudiants ont fait plus que la moitié en faisant des fautes.
29 étudiants ont fait correctement l'exercice.

6 étudiants ont tout fait correctement et de plus ils ont fait une remarque sur la correction à l'exécution sur les indices.

Remarque: en TD, la majorité des étudiants avaient fait un exercice similaire qui leur permettait de faire correctement la moitié de celui proposé, sauf les étudiants d'un des groupes car le chargé de ce TD ne l'avait pas traité, cette vingtaine d'étudiants n'a pas fait cet exercice.

Exercice 2)

24 étudiants n'ont pas fait cet exercice.
31 étudiants ont essayé de le faire sans résultat.
35 étudiants ont compris le passage par valeur.
22 étudiants ont bien compris le passage par valeur et ont fait des fautes avec le passage par variables. Ils ont quand même compris le passage par variable.
4 étudiants ont bien compris les deux passages par variables mais se sont arrêtés lorsqu'un même emplacement mémoire était référencé par deux variables.
11 étudiants ont fait correctement cet exercice.

Exercice 3)

24 étudiants n'ont pas abordé cet exercice.
30 étudiants ont essayé de le faire sans succès
15 étudiants ont écrit correctement une fonction
9 étudiants ont écrit les deux fonctions
8 étudiants sont arrivés jusqu'à la procédure de tri en faisant de petites fautes.
17 étudiants sont arrivés jusqu'à la procédure de tri
25 étudiants ont fait correctement tout l'exercice.

Conclusions

* Les étudiants (68%, 80% si on prend en compte la remarque) ont compris et manipulent correctement le type enregistrement.
* Les étudiants (56%) ont compris le passage par valeur et (30%) ont compris le passage par variable.
* Les étudiants (58%) arrivent à déduire des fonctions de propriétés.
* 40% des étudiants ont bien traité l'exercice sur les propriétés, ce qui est très rassurant pour nous.

Les étudiants font beaucoup de fautes pour exécuter des procédures. Ces fautes sont souvent dues à un manque de rigueur de leur part. Il faut dire qu'en deuxième année les étudiants n'ont

plus trop de mal à exécuter des fonctions et des procédures, ce qui est rassurant pour nous.

7.1.3) Deux autres sujets

Lors de la session de rattrapage, nous avons donné le sujet qui suit. Les étudiants se sont bien comportés dans l'ensemble, deux étudiants ont réussi à tout faire (20/20). Voici le sujet.

Université de Metz
Faculté des Sciences
UFR MIM + SciFA
DEUG A1 TM + PIE
In2.x : D. Cansell

2ème session

Documents manuscrits du cours et des TD autorisés

Exercice 1)

Soient n une constante entière, x,y des variables entières.
Soit p une procédure dont l'entête est la suivante:

```
procedure p(i,j:integer;var k:integer);
```

Soit f une fonction dont l'entête est la suivante:

```
function f(n,m:integer):integer;
```

Dire si les instructions Pascal suivantes sont correctes ou non. Justifier vos réponses.

1) p(4,3,x)
2) f(x+1,7)
3) p(f(4,6),8,x)
4) x:=f(x+7,x)
5) p(x,x,6)
6) x:=f(x,p(1,2,x))
7) p(f(x,x),x,x)
8) p(x,x+1,f(x,x))

Exercice 2) type joueur (de tennis)

Pour un joueur de tennis il faut connaître les renseignements suivants:

- son nom
- son prénom
- sa nationalité
- son classement ATP
- son âge

s'il est droitier ou gaucher
son poids
sa taille
sa résidence (ville + pays)

2.1) Définir correctement les types suivants:
- un type qui contient les valeurs droitier et gaucher
- un type qui représente une résidence
- un type qui représente un joueur.

2.2) Soit t un tableau de n joueurs (t:array[1..n] of joueur). Ecrire une fonction qui compte le nombre de gauchers parmi les n joueurs.

2.3) Ecrire une procédure qui affiche tous les renseignements du joueur classé premier à l'ATP.

Problème

Soient n une constante, V un type dont les valeurs sont comparables.
Soit tableau le type array[1..n] of V.
Soient t1,t2 deux tableaux.

Définition $t1 \leq_1 t2$
On dit que $t1 \leq_1 t2$ si et seulement si $\forall i \in (1, ,n)$ t1[i]≤t2[i]

Définition $t1 \leq_2 t2$ (ordre lexicographique (celui du dictionnaire)
On dit que $t1 \leq_2 t2$ si et seulement si

(∀ i t1[i]=t2[i])
ou
(∃ i tq ∀ j∈(1,..,i-1) t1[j]=t2[j] et t1[i]<t2[i])

1) Ecrire une fonction inf1 qui aura comme données t1 et t2 deux tableaux et dont le résultat sera $t1 \leq_1 t2$ (résultat booléen).

2) Ecrire la fonction inf2 qui aura comme données t1 et t2 deux tableaux et dont le résultat sera $t1 \leq_2 t2$

Nous donnons également, le sujet que nous avions donné en 1989-90, aux étudiants de TM1.

Université de Metz
UFR MIM
In2.2: D. Cansell
Année 1989-90

PREMIERE SESSION

Documents manuscrits du cours et des TD autorisés

Exercice 1)

Soit p une procédure. Soient x,y,z des variables entières.

Nous avons le droit d'appeler la procedure p de la manière suivante:

```
p(x,3,y,z)
p(x,x,y,x+1)
p(z,z+6,z,9*(z-3))
```

Nous n'avons pas le droit d'appeler la procédure p de la manière suivante:

```
p(1,2,3,4)
p(x+1,x,x,x+2)
```

1.1) Comment peut être l'entête de la procédure p ?

Exercice 2)

Soit le programme suivant:

```
program exam;
var i,j,k:integer;
  procedure p(i:integer;var x:integer);
  var h:integer;
    procedure r(var k:integer);
    begin
      k:=x+h
    end;
  begin
    h:=3;
    r(i);
    x:=i+4*k
   end;
begin
  i:=5;
  k:=2;
  p(i,j);
  writeln(i,j)
end.
```

2.1) Donner le schéma d'activation de ce programme et dire ce qu'il écrit.

Problème

Soient vinf et vsup et n trois constantes entières.
Soit tableau le type suivant:
```
tableau=array[1..n] of vinf..vsup
```
Soit compte_occurence le type suivant:
```
compte_occurence=array[vinf..vsup] of integer
```

3.1) Soit t un tableau. Soit v une valeur entière. Ecrire un algorithme sous forme d'une fonction qui compte le nombre d'indices i du tableau t qui possède la propriété suivante: t[i]=v. Le nom de cette fonction sera compte_occ, sa donnée v et t, son résultat un entier.

3.2) En déduire une procédure compte_toute_occ, dont le paramètre donnée est t un tableau et dont le paramètre résultat est occ de type compte_occurence. A la fin de la procédure occ possèdera la propriété suivante:
∀ v∈(vinf,...,vsup) occ[v] contient le nombre d'indice i du tableau t qui possède la propriété t[i]=v

3.3) En déduire un algorithme de tri sachant que vinf est la plus petite valeur possible pour un tableau t et que le nombre de valeur égale à vinf dans le tableau t peut être calculé à l'aide des questions précédentes.

3.4) En déduire un programme qui lira un tableau le triera à l'aide de votre procédure et affichera les valeurs du tableau trié.

7.2) Deux sujets de DEUG2 (PIE+TM)

Nous avons l'habitude de faire deux sujets dont l'un porte sur la récursivité en général et l'autre sur les listes linéaires.

7.2.1) Sujet 1)

Il s'agit du premier sujet donné en 1988-89. Il y avait 72 étudiants en PIE2.

Université de Metz
Faculté des Sciences
DEUG A 2 P.I.E. + T.M.
Année 1988-1989
In3.2: D. Cansell

CONTROLE N° 1

Documents manuscrits autorisés

Exercice 1) Montrer par récurrence que

$\forall\ n \geq 0 \quad 1^3+2^3+\ldots+n^3 \leq n^4$

Exercice 2) Soit la fonction suivante:

```
function rac(n,i,x,s:integer):integer;
begin
  if x>n then
    rac:=i
  else
    rac:=rac(n,i+1,x+s,s+2)
end;
```

2.a) Exécuter rac(16,0,1,3).

2.b) Soit g(n,i,x,s)=(n,i+1,x+s,s+2).
Montrer que $g^k(n,0,1,3)=(n,k,(k+1)^2,2k+3)$.

2.c) En déduire que
$rac1(n)^2 \leq n < (rac1(n)+1)^2$ avec rac1(n)=rac(n,0,1,3)

2.d) Dérécursiver la fonction rac.

Exercice 3)

3.a) Donner une écriture récursive de x^n en fonction de $(x^2)^{n/2}$
c.à.d. exp(x,n) en fonction de exp(x*x,n div 2).

3.b) Ecrire la fonction Pascal associée exp.

3.c) Exécuter exp(2,9).

3.d) Dérécursiver votre fonction.

Exercice 4) Soit V un type.

Soit le type tab=array[1..n] of V.
Soit t1,t2 deux variables de type tab.

On dit que egal(t1,t2)⇔(∀ i∈(1,..n) t1[i]=t2[i]).

4.a) Ecrire une fonction récursive egal dont l'entête est la suivante: **function** egal(t1,t2:tab;i:integer):boolean; qui est vraie si (∀ j∈(1,..,i) t1[j]=t2[j]) et fausse sinon.

On dit que EGAL(t1,t2)⇔(∀ i∈(1,..,n) ∃ j∈(1,..,n) tq t1[i]=t2[j]
et
∀ j∈(1,..,n) ∃ i∈(1,..,n) tq t2[j]=t1[i])

4.b) Ecrire une fonction récursive estdans dont l'entête est la suivante: **function** estdans(x:V;t:tab;i:integer):boolean qui est vraie si (∃ j∈(1,..,i) tq x=t[j]) et fausse sinon.

4.c) Ecrire une fonction récursive tousdedans dont l'entête est la suivante: **function** tousdedans(t1,t2:tab;k:integer):boolean; qui est vrai si (∀ i∈(1,..,k) ∃ j∈(1,..,n) tq t1[i]=t2[j]).

4.d) En déduire une fonction EGAL.

4.e) Dérécursiver vos fonctions.

7.2.2) Résultats

Exercice 1)

8 étudiants n'ont rien fait.
10 étudiants n'ont montré que $\mathcal{P}(n0)$.
5 étudiants se sont trompés dans la démonstration, mais le théorème est bien utilisé.
49 étudiants ont fait correctement l'exercice.

Exercice 2)

4 étudiants n'ont rien fait.
65 étudiants ont fait l'exécution (a).
39 étudiants ont fait la démonstration par récurrence (b).
8 étudiants ont réussi à montrer la question c.
57 étudiants ont éliminé la récursivité.

Récapitulatif: (5 a) (20 a,d) (3 d) (6 a,b) (26 a,b,d) (1 a,c,d) (7 a,b,c,d). Avec la notation suivante (i l) où i représente le

nombre d'étudiants qui ont traité les questions de la liste 1.

Exercice 3)

15 étudiants n'ont rien fait (10 sont hors sujet).
13 étudiants ont fait les 3 premières questions en faisant des fautes.
23 étudiants ont fait correctement les trois premières questions.
20 étudiants ont fait correctement l'exercice.
1 étudiant a réussi à optimiser son algorithme itératif en n'utilisant qu'une seule boucle; à l'époque nous n'avions donné que des indications pour le faire.

Exercice 4)

44 étudiants n'ont rien fait.
7 étudiants ont fait l'égalité 1 (dont 3 correctement).
13 étudiants ont fait l'égalité 1 et la fonction estdans.
4 étudiants ont fait les deux égalités.
3 étudiants ont presque tout fait.

Conclusions

Le sujet était assez long et dans la majorité des cas, les étudiants ont bien enchaîné les questions. Le dernier exercice étant plus difficile, seul les très bons étudiants l'ont traité car il leur restait encore suffisamment de temps. Il faut savoir que certains étudiants de PIE peuvent être acceptés en licence d'informatique, il faut donc s'assurer de leurs capacités à poursuivre.

En ce qui concerne les résultats nous avons constaté les points suivants:

* Le théorème de récurrence est très bien assimilé (75%).
* Les étudiants(90%) savent exécuter des fonctions et des procédures récursives.
* Les étudiants (70%) parviennent à écrire une fonction récursive, lorsqu'ils ont trouvé la définition récursive. Dans l'exercice 3 une dizaine d'étudiants ont redonné la définition de x^n en fonction de $(x^{n/2})^2$ du cours.
* Les étudiants (79%) savent utiliser les règles d'élimination, ils trouvent toujours la bonne règle même si quelquefois ils font des erreurs.

7.2.3 Sujet 2)

Université de Metz — Mardi 23 janvier 1990
UFR MIM + SciFA
DEUG A2 TM + PIE
In3.2: D. Cansell

CONTROLE NUMERO 2

Documents manuscrits du cours et des TD autorisés.

Exercice 1) *Insertion en queue de liste.*

Soit la fonction insertenqueue définie en cours suivante:

```
function insertenqueue(v:V;l:liste):liste;
begin
  if vide(l) then insertenqueue:=cons(v,consvide)
  else  insertenqueue:=cons(valeur(l),insertenqueue(v,suivant(l)))
end;
```

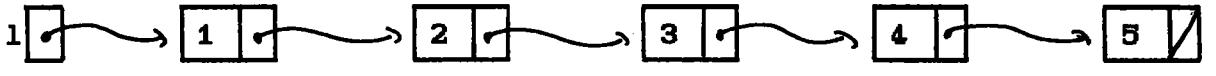
1.1) Montrer par récurrence sur la longueur n de la liste l que le nombre d'appels à la fonction cons dans un appel à insertenqueue(v,l) est égale à n+1.

1.2) Déduire de la fonction insertenqueue une procédure qui insère de manière physique une valeur v dans une liste l, l'entête de la procédure sera la suivante:

```
  procedure insertenqueue(v:V;var l:liste);
```

Exemple d'exécution
Avant l'appel

insertenqueue(5,l)
Après l'appel
(1,2,3,4 sont dans les mêmes emplacements mémoire avant et après l'appel)



1.3) Nous ne pouvons pas éviter de parcourir toute la liste pour inserer une valeur en queue d'une liste. Que proposez-vous pour éviter ce parcours?

Exercice 2) *Panachage de deux listes.*

Définition: Soient l1 et l2 deux listes linéaires sur V.

```
  l1=(a1,(a2,( ....,(an,()) ....)))
  l2=(b1,(b2,( ........ (bm,()) ........ )))
```

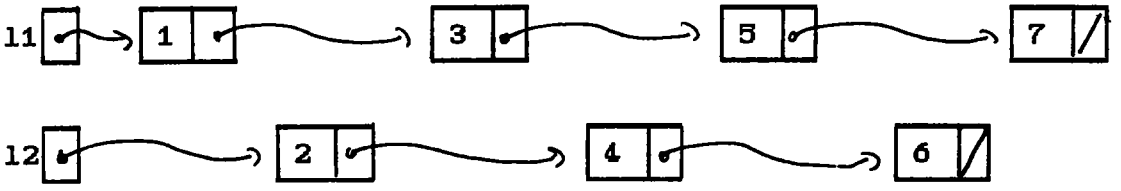
On définit panach(l1,l2) de la manière suivante:

panach(l1,l2)= si n≤m

(a1,(b1,(a2,(b2,(....,(an,(bn,(bn+1,(...,(bm,())...))))....)))))
panach(l1,l2)= si n≥m
(a1,(b1,(a2,(b2,(....,(am,(bm,(am+1,(...,(an,())...))))....)))))

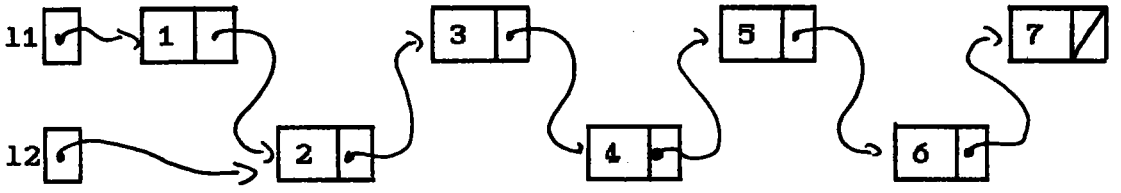
2.1) Ecrire une fonction Pascal panach dont l'entête est la suivante:
function panach(l1,l2:liste):liste; qui construit panach(l1,l2)

2.2) Transformer votre fonction en procédure pour qu'elle utilise les mêmes emplacements mémoires de l1 et l2.
Exemple d'exécution:
Avant l'appel de la procédure



panach(l1,l2)

Après l'appel



Exercice 3) *Suppression de valeur redondante dans une liste ordonnée.*

Le but de cet exercice est de supprimer les valeurs qui apparaissent plus d'une fois et de manière consécutive dans une liste.
exemple:
supp((1,(1,(1,(2,(2,(3,(4,(4,()))))))))=(1,(2,(3,(4,()))))

3.1) Ecrire une fonction supptoutdebut dont l'entête est la suivante: **procedure** supptoutdebut(v:V;l:liste):liste; qui construit la liste de toutes les valeurs de l sauf toutes les valeurs v qui sont au début de la liste l.
Exemple:
supptoutdebut(1, (1,(1,(1,(2,(2,(3,()))))))))=(2,(2,(3,())))
supptoutdebut(2, (3,(3,(5,())))))=(3,(3,(5,())))

3.2) En déduire la fonction supp.

3.3) Traduire vos fonctions en procédures pour ne pas utiliser la fonction cons (suppression physique de valeurs dans une liste).

7.2.4) Les résultats

Il y avait 86 étudiants cette année là.

Exercice 1)

pour la démonstration par récurrence:
- 18 étudiants n'ont pas fait la démonstration.
- 2 étudiants n'ont démontré que P(n0).
- 66 étudiants ont fait correctement l'exercice.

pour la transformation de la fonction en procédure:
- 31 étudiants ne l'ont pas fait ou correctement fait.
- 5 étudiants n'ont pas utilisé cons (new non plus).
- 50 étudiants l'ont correctement fait.

Exercice 2)

Pour la fonction:
- 31 étudiants ne l'ont pas fait, ou de manière incorrecte.
- 11 étudiants ont écrit la fonction avec de petites erreurs.
- 44 étudiants l'ont correctement écrite.

Pour la procédure:
- 69 étudiants ne l'ont pas fait ou correctement fait.
- 7 étudiants ont fait de petites erreurs.
- 10 étudiants ont correctement écrit la procédure.

Exercice 3)

Pour les fonctions:
- 27 étudiants n'ont pas écrit les fonctions, ou de manière incorrecte.
- 20 étudiants ont écrit la première.
- 39 étudiants ont écrit les deux.

Pour les procédures:
- 76 étudiants n'ont pas écrit ces procédures, ou de manière incorrecte.
- 5 étudiants ont fait de petites erreurs.
- 5 étudiants ont correctement écrit ces procédures.

Conclusions

* Les étudiants (77%) maîtrisent bien le théorème de récurrence.
* Les étudiants (63%) transforment bien les fonctions en

procédures lorsque la fonction est simple et connue (insert_en_queue à été écrite en cours ou en TD) .

* en ce qui concerne les fonctions sur l'ensemble du sujet, plus de 60 étudiants (70%) ont écrit correctement au moins une des fonctions.

Nous ne pensons pas que les résultats sur les procédures aillent à l'encontre de l'approche procédurale. Il faut savoir que les étudiants ont été initiés à cette approche lors de leur dernier cours. Ils ne la maîtrisent donc pas correctement et comme le sujet est assez long, ils ont préféré traiter les fonctions qu'ils comprennent mieux et sur lesquelles nous avons insisté davantage

Un autre sujet:

Université de Metz
Faculté des Sciences
UFR SFA + MIM
DEUG A 2 PIE + TM
Année 1988-1989
In3.2 : D. Cansell

CONTROLE N° 2

Documents manuscrits du cours et des TDs autorisés

Exercice 1) Pour cet exercice l'ensemble des listes est L(Z)
Soient n et p deux entiers tq 0<p<n
Le but de cet exercice est de construire les listes suivantes:

atoisansp(n,p)=(n,(n-1,...(p+1,(p-1,(...,(1,())...)))...))

iotasansp(n,p)=(1,(2,...(p-1,(p+1,(...(n,())...)))...))

1.1) Ecrire une fonction atoisansp qui construit la liste atoisansp(n,p) à l'aide de la fonction atoi(k) qui construit la liste (k,(k-1,(...(1,())...))) pour un k donné et de la fonction supp(a,l) qui construit la liste qui contient toutes les valeurs de la liste l (dans le même ordre) sauf la valeur a.
l'entête de la fonction sera:

```
function atoisansp(n,p:integer):liste
```

1.2) Ecrire une fonction atoinp qui construit la liste atoinp(n,p) suivante:

(n,(n-1,(....(p+1,())....)))

l'entête de la fonction sera: `function atoinp(n,p:integer):liste`

1.3) Modifier la fonction atoinp en vous servant de la fonction atoi pour qu'elle vous construise atoisansp(n,p)

1.4) Compter le nombre de fois où vos fonctions précédentes utilisent la fonction cons en fonction de n et p (démontrer votre résultrat). Laquelle des fonctions écrites en 1.1 et 1.3 est la meilleure ? Peut-il y avoir une fonction encore meilleure ?

1.5) Ecrire une fonction iotasansp qui construit la liste iotasansp(n,p) en utilisant n-1 fois la fonction cons. (Ecrire une fonction iotanp qui construit la liste suivante: (p,(....(n,())...)) avec une technique de balancement et en déduire la fonction iotasansp)

Exercice 2) Un tri par fusion d'une liste linéaire sur V où V est un ensemble ordonné.

Principe du tri Soit l une liste de longueur n que l'on veut trier. On découpe la liste l en deux listes l1 et l2 telles que l1 contienne les n/2 premières valeurs de la liste l et l2 les autres. On trie les deux listes et on les fusionne (avec la fonction fusion(l1,l2) écrite en TD).

Rappel
```
function fusion(l1,l2:liste):liste;
begin
  if vide(l1) then fusion:=l2
  else if vide(l2) then fusion:=l1
  else if valeur(l1)≤valeur(l2) then
       fusion:=cons(valeur(l1),fusion(suivant(l1),l2))
  else fusion:=cons(valeur(l2),fusion(l1,suivant(l2)))
end;
```

2.1) Soit l une liste de longueur n, soit p<n. Ecrire une fonction premier qui construit la liste des p premières valeurs de l. L'entête sera la suivante: **function** premier(l:liste;p:integer):liste

premier((a$_1$,(...(a$_p$,(...(a$_n$,())...))...))...),p)=(a$_1$,(...(a$_p$,())...))

2.2) Soit l une liste de longueur n, soit p<n. Ecrire une fonction dernier qui construit la liste des valeurs de la liste l sauf les p premières. L'entête sera la suivante:

function dernier(l:liste;p:integer):liste;

dernier((a$_1$,(...(a$_{p+1}$,(...(a$_n$,())...))...)),p)=(a$_{p+1}$,(...(a$_n$,())...))

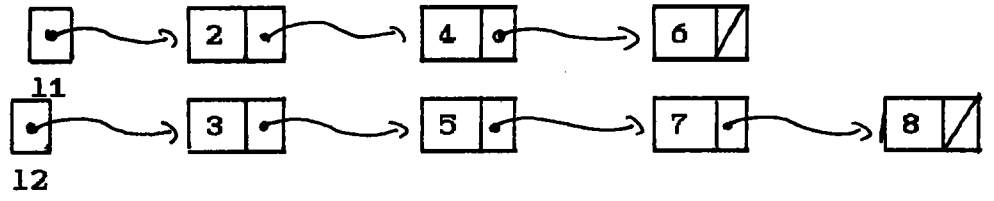
2.3) Soit l une liste de longueur n. En déduire la fonction tri dont l'entête est : **function** tri(l:liste;n:integer):liste , et qui construit la liste ordonnée qui contient toutes les n valeurs de la liste l . Comment utiliser votre fonction pour trier une liste?

2.4) Traduire la fonction fusion en procédure, pour qu'elle n'utilise pas la fonction cons; l'entête sera la suivante:

procedure fusion(var l1:liste;l2:liste) qui construit fusion(l1,l2) en utilisant les mêmes emplacements mémoires que les listes l1 et l2 et qui met le résultat dans la variable l1.

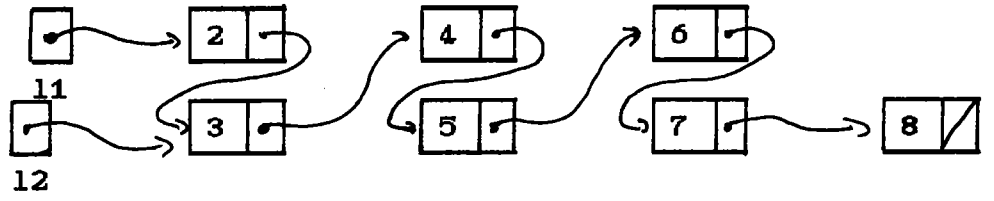
Exemple:

Avant



fusion(l1,l2)

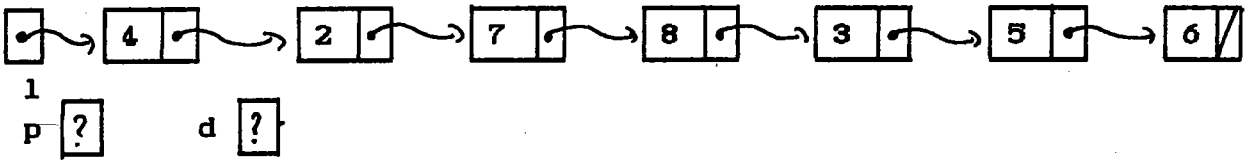
Après



2.5) Traduire vos fonctions premier et dernier en une procédure premder qui utilise également les mêmes emplacements mémoires que la liste l

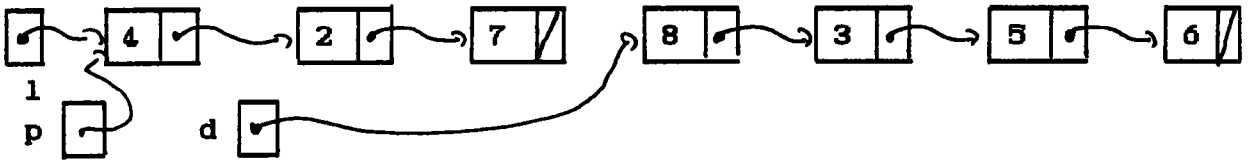
Exemple:

Avant



premder(l,3,p,d)

Après



2.6) En déduire une procédure de tri par fusion d'une liste l qui n'utilise jamais la fonction cons.

Remarque: Nous avons utilisé les tris sur les listes également pour évaluer les connaissances des étudiants sur les algorithmes de tris. Nous avons également donné lors l'année précédente, un sujet sur l'algorithme quicksort sur les listes. Les étudiants ont obtenu de bons résultats avec ce type de sujet, ce qui nous a encouragé à continuer.

Chapitre 8

CONCLUSIONS

8.1) Choix du langage de programmation PASCAL

Il vaut mieux bien faire avec le langage X que mal faire avec le langage Y. Par contre si nous savons bien faire avec ces deux langages de programmation lequel doit-on choisir? La réponse sera sans doute apportée par les réflexions commune du groupe Abalone et de SPECIF.

Dans l'état actuel, si on nous demande de nous servir de SCHEME ou ML comme langage de programmation nous le ferons sans avoir à changer quoi que ce soit de fondamental à notre enseignement.

Par contre, nous ne pensons pas que le choix d'un langage comme PROLOG ou d'un langage objet soit une bonne chose pour des étudiants de DEUG mais le fait d'initier correctement nos étudiants à la logique favorisera l'apprentissage de PROLOG comme l'utilisation de type abstrait dès le DEUG favorisera l'initiation à un langage objet.

A propos de PASCAL, rappelons qu'au moment où nous avons commencé à enseigner, la question du choix du langage ne se posait pas. Nous nous servions de celui qui existait c'est-à-dire PASCAL, nous n'avons donc pas choisi. Notre seul but était de bien faire avec le langage de programmation que nous utilisions et dans assumer toutes les conséquences.

Nous écrivons tous nos algorithmes en PASCAL car:

- c'est un langage de programmation algorithmique.
- nous n'encadrons pas beaucoup nos étudiants dans les salles machines (interventions ponctuelles). C'est un choix délibéré de notre part nous avons donc voulu lier un peu plus fortement l'algorithmique et la programmation mais sans tomber dans l'excès d'un cours orienté apprentissage d'un langage.
- à travers lui nous faisons passer les concepts de base de l'algorithmique. Cela peut être considéré comme un acte démagogique car par le biais de l'apprentissage du langage, nous pensons apprendre à nos étudiants l'algorithmique. Nous

savons qu'il existe des pressions de nos collègues non informaticiens (même parfois informaticiens) et de la part des étudiants pour que l'enseignement de l'informatique se limite à apprendre la programmation et ses astuces. Même si notre choix n'a pas été influencé par ces pressions, il peut être une réponse aux envies de ces personnes.

- car nous avons voulu situer la réflexion de l'étudiant sur la conception de l'algorithme à un niveau différent (propriétés, comportement, analyse abstraite pour l'approche fonctionnelle des listes linéaires, etc ...).

De plus,

* c'est un choix stratégique, car pour enseigner la récursivité et les listes, nous aurions pu choisir un langage applicatif comme Lisp, mais avec le temps qui nous était imparti nous ne voulions pas initier nos étudiants à un deuxième langage et, comme nous l'avons rappelé, les physiciens ou les mécaniciens voulaient que leurs étudiants connaissent des langages comme Basic ou Fortran, avec une analyse récursive des problèmes et les règles d'élimination de la récursivité. Nous pouvons justifier auprès de nos collègues non informaticiens que leurs étudiants connaissent la programmation itérative et qu'ils pourront obtenir des programmes Basic ou Fortran à l'aide de traduction de programme PASCAL. Nous avons déjà eu beaucoup de mal à imposer à nos collègues l'algorithmique (récursive ou pas) à travers le langage PASCAL, alors si nous choisissions un langage comme SCHEME ou LISP, nous nous marginaliserions par rapport aux autres disciplines, qui ne demanderaient pas mieux pour prendre en charge leur enseignement de l'informatique "presse bouton". De plus, avec Pascal, l'étudiant peut prendre conscience de sa capacité à construire ses types ainsi que les fonctions de base qui lui sont associées pour s'abstraire lui-même de la machine. Il pourra ainsi avoir, comme nous le lui conseillons, une analyse applicative et récursive de ses problèmes; le langage PASCAL lui permettra sans trop d'effort de traduire sous forme de fonction ces analyses. L'étudiant pourra, par la suite, s'initier ou être initié sans problème à un langage applicatif pour l'avoir ainsi pratiqué.

* nos étudiants qui ne poursuivront pas en informatique, connaîtront un langage impératif, qui est le style de langage le plus utilisé dans le monde du travail (Fortran, Cobol, Basic ..).

8.2) Les machines

Un cours d'algorithmique peut très bien se faire sans machine,

comme un cours de mathématique. Mais en informatique, si la machine est bien utilisée, elle peut avoir des vertus pédagogiques indéniables car un algorithme pourra, avec la façon dont nous l'écrivons, donner naissance sans aucun problème à un programme que nous pourrons exécuter sur des données importantes. Dans certains cas, nous pourrons corriger ce programme, lorsque les résultats de celui-ci sont erronés. Nous insisterons sur le fait qu'en aucun cas la bonne exécution d'un programme ne peut assurer le concepteur de celui-ci de sa correction. C'est sur une feuille de papier que le concepteur construit correctement son algorithme en étant sur qu'il est juste avant de l'écrire. Seules de petites erreurs pourront être corrigées facilement.

Le fait qu'un algorithme donne lieu à l'écriture d'un programme est une chose saine pour le concepteur car il doit faire en sorte que tout ce qu'il définit et tout ce qu'il utilise dans un algorithme soit réalisable au niveau de son langage de programmation. Si parfois un algorithme peut ne pas être complet, il faut absolument que tout le soit au niveau du programme, car la machine ne nous fera pas de cadeau, la sanction est immédiate. Il faut aller jusqu'au bout. C'est une école de rigueur.

Par contre, il faut mettre en garde les étudiants. La machine est un outil formidable, mais un outil quand même, dont il faut apprendre à se servir le plus correctement possible. Une trop grande précipitation à l'utilisation de la machine peut avoir plusieurs effets néfastes qui sont:

* La sanction immédiate qui peut provoquer chez l'étudiant un rejet pur et simple de la machine et par conséquent de la discipline informatique.
* Une frénésie d'utilisation, car l'étudiant est persuadé que c'est au contact de la machine qu'il apprendra l'informatique, et cela au détriment des autres disciplines et même de l'informatique. Beaucoup trop de nos étudiants, spécialistes de la machine, ont échoué au DEUG. Lorsqu'ils l'obtenaient l'année suivante, ils choisissaient la Licence d'Informatique et se retrouvaient limités par des carences énormes en mathématique, matière qu'ils avaient négligée pour se consacrer à des séances de programmation.

Pour la première catégorie d'étudiants, il faut dédramatiser et banaliser l'utilisation de la machine. Il faut donc leur décrire les composantes de base d'une machine (unité et mémoire centrale, disquette, disque, clavier, écran, imprimante ...) et les commandes de base d'un système (fichier, éditeur, compilation, exécution) pour qu'ils puissent utiliser la machine mise à leur disposition et retrouver dans la documentation de celle-ci les commandes de base afin d'écrire, de compiler et d'exécuter leurs programmes corrects bien compris avant de se trouver devant la

machine.

Pour la seconde catégorie, il faut, si nous nous en rendons compte, les inciter à ne pas négliger les autres matières au risque d'échouer.

Correction d'erreur

La correction d'erreur est une chose extrêmement difficile: si ce sont de petites erreurs de syntaxe ou des coquilles et si le compilateur peut s'en rendre compte, dans ce cas la correction est simple; si ce sont des erreurs à l'exécution, cela peut provenir de coquilles ou plus grave d'erreurs dans la conception de l'algorithme. De plus la correction de certaines erreurs peut en entrainer d'autres!!!. L'utopie est de ne jamais faire d'erreur, la réalité est que personne n'y échappe. Pour pouvoir les corriger, il faut essayer de regarder la valeur des résultats (parfois les résultats intermédiaires) et d'analyser ceux-ci pour voir où est vraiment la faute. Il existe des techniques de construction des programmes pour corriger plus facilement ces derniers.

* Il s'agit de la décomposition des programmes en utilisant la démarche arborescente descendante à l'aide des fonctions et des procédures. Avec cette méthode, un algorithme est toujours très court donc bien lisible et par conséquent plus compréhensible. Cet algorithme donnera naissance à une procédure ou à une fonction qui sera donc facile à tester (corriger) et qui pourra, une fois supposée correcte, être utilisée en toute confiance.
* il s'agit de la récursivité pour les raisons déjà évoquées.

Le meilleur conseil que nous pouvons donner aux étudiants est:

"Efforcez-vous donc toujours d'écrire des algorithmes courts, vous verrez vous serez récompensés!"

8.3) Bilan et perspectives

Avec ce simple objectif qui était de vouloir que nos étudiants décomposent leurs problèmes, un étudiant d'un DEUG scientifique de l'Université de Metz aura abordé tous les points suivants:

* les bases de l'algorithmique
* des notions de logique (expression booléenne, implication ...)
* faire des démonstrations
  (correction d'algorithme, démonstration par récurrence, ..)
* évaluer la performance de ses algorithmes (coût)
* l'architecture des ordinateurs

(présentation de la machine, algorithmes de codage et de calcul)
* des notions de système
(éditeur, compilateur, fichier donnée, fichier exécutable, ..)
* programmer ses algorithmes (le langage PASCAL)
* décomposer un problème
* définir des propriétés
* définir les types en fonction de l'algorithme pour qu'ils lui soient le plus adaptés possible
* un peu d'informatique de gestion (exemple de banque, gestion de la scolarité de l'UFR MIM [Can90b])
* Etude de schéma d'algorithme (itératif et récursif)
* définir des types pour mieux s'abstraire de la machine. Notion de type abstrait (pile, liste, ...)
* programmer de manière applicative (approche fonctionnelle des listes linéaires)
* d'abord réfléchir puis en déduire un algorithme (définir une propriété, définir de manière récursive ou non le résultat d'une fonction, trouver le schéma d'un algorithme en trouvant la structure récursive d'un comportement d'une procédure, déduire d'algorithmes récursifs des algorithmes itératifs à l'aide de règles, n'avoir une approche procédurale qu'une fois que la fonction est écrite ....)

etc ...

En ce qui concerne nos objectifs:

- nous partons de zéro en DEUG première année, nous sommes en accord avec ce qui est fait au lycée, car seulement 2% des bacheliers qui viennent s'inscrire dans les Universités françaises ont suivi l'option informatique dans leur lycée. D'ailleurs, les étudiants qui ont déjà des connaissances en informatique ont l'impression de tout savoir, ce qui rend notre tâche d'enseignant difficile.

- les étudiants connaissent la récursivité et lui font confiance, ils pourront ainsi aborder avec un bagage théorique plus important des cours d'informatique des deuxièmes cycles où notre discipline prend une part importante. Ils pourront choisir en toute connaissance de cause la poursuite de leurs études.

- En relisant la liste de ce qu'ont fait les étudiants en DEUG, et en regardant la réalisation on peut facilement se rendre compte que ce qu'ils ont appris est cohérent et relativement complet.

Nous pensons donc avoir atteint nos objectifs.

Tout cela n'a été possible, que grâce à la rigueur constante que

nous avons exigée de nos étudiants, en mettant à leur disposition un cadre où ils peuvent raisonner proprement pour construire des algorithmes corrects et en leur "imposant" un style fonctionnel et abstrait qui les forcera à plus de rigueur encore et à plus de créativité.

Nous revendiquons comme le groupe de réflexion Abalone [Aba91] les points forts suivant:

- Les rapports avec les étudiants de DEUG: ils sont plus que positifs. Nos étudiants en général apprécient notre style d'enseignement et tant mieux s'ils poursuivent leurs études en informatique.
- La productivité de l'approche: nous essayons également de leur donner les réflexes de la modularité et de modélisation de leur problème.

Le travail à faire est encore immense, c'est à l'ensemble de la communauté informatique de s'investir dans cette tâche. De notre côté nous continuerons notre effort (voir chapitre 10)

TROISIEME PARTIE: "Ailleurs et demain"

Chapitre 9

COMPARAISON AVEC D'AUTRES FAÇONS D'ENSEIGNER EN DEUG

Nous avons pris comme éléments de comparaison un certain nombre d'ouvrages d'algorithmique cités dans le compte-rendu des journées de Nantes [SPE89] ainsi que d'autres ouvrages empruntés à la bibliothèque de l'Université de Metz.

9.1) [Ars80]

Arsac: Premières leçons de programmation (1980) chez Fernand Nathan.

C'est une initiation à l'algorithmique. Le tort de cet ouvrage est que les algorithmes sont expliqués en LSE qui ressemble à du Basic. Cela nuit à la lisibilité donc à la compréhension de ceux-ci. La notion de variable, les types entier, réel ..., tableau, ainsi que les expressions, l'affectation et les conditionnelles sont supposés être connus. Les explications commencent aux itérations pour lesquelles l'auteur utilise la récursivité. La programmation structurée se fait par l'intermédiaire de définitions de procédure et de fonction. Par contre les conseils de l'auteur sont toujours judicieux.

9.2) [Ars83]

Arsac: les bases de la programmation (1983) chez DUNOD informatique

L'auteur présente l'algorithmique à l'aide des trois piliers qui sont récurrence, récursivité et itération; personnellement, nous n'en utilisons qu'un: la récursivité. Il se sert d'un système de preuves pour construire des algorithmes justes et les améliore pour les rendre plus clairs ou plus performants. La lisibilité des algorithmes est difficile, c'est dommage pour le contenu qui semble intéressant d'autant plus qu'en 1983, il existait des langages de programmation plus clairs que celui choisi par l'auteur.

9.3) [Lig85]

P. Lignelet: Algorithmique Méthodes et Modèles (1985 3ème édition) C.N.A.M.

L'auteur sépare les algorithmes en deux parties: ceux qui sont prêts à l'emploi et ceux que l'on peut créer. Beaucoup des mots forts qu'il cite dans son avant-propos ne sont pas employés ni vraiment définis. Il insiste par exemple sur la lisibilité et la modularité, or la lisibilité n'est pas toujours évidente et la modularité n'apparait que sur une page pour la définition des procédures et des fonctions. Il y a beaucoup d'exemples classiques et d'idées fortes communs aux nôtres au départ, mais ils ne sont pas traités de la même façon. Il insiste sur la construction démonstrative des programmes en proposant de penser d'abord, mais il donne des définitions mathématiques des résulats puis l'algorithme. L'auteur axe son cours sur les boucles avec sa notation propre (pour i:= 1 to ∞ tant que non c(i)) et il prouve ses algorithmes à l'aide des invariants.

9.4) [CoK87]

J. Courtin et J. Kowarsky: Initiation à l'algorithmique et aux structures de données, chez DUNOD informatique.

Tome1) Programmation structurée:

C'est un ouvrage bien présenté et agréable à lire, par contre, l'initiation est faite rapidement. Il est axé sur les boucles où le raisonnement par récurrence est utilisé pour montrer qu'un programme termine et est correct. **Où est la structure grenobloise?** Il est trop axé sur les tris itératifs qui passent après les algorithmes sur les fichiers.

Tome2) récursivité et structure de données avancées: Les auteurs ne font pas d'initiation à la récursivité. Pour les listes, il y a l'approche effet de bords et l'approche procédurale. Ils présentent toujours une version itérative et récursive (très souvent dans cet ordre). En ce qui concerne les IUT, il existe un programme national, mais nous savons qu'un contenu peut-être enseigné de différentes façons.

9.5) [ScP88]

P.C. Scholl, J.P. Peyrin: Schémas algorithmiques fondamentaux, séquences et itérations chez Masson

Nous avons déjà fait référence à ce livre et à l'importance des schémas dans un enseignement de l'informatique. Nous donnerons

donc les différences (non fondamentales) entre nos points de vues respectifs du type abstrait séquence (ou liste).

Après une présentation de l'algorithmique de base (décompositions simples et paramétrisation) où il est demandé que l'on résolve les problèmes à l'aide de pré et post conditions, les auteurs nous présentent la machine caractère, qui permet à l'étudiant d'avoir une vision abstraite et simple d'une machine. Elle leur permet également:

- d'expliquer ce qu'est une analyse itérative (recherche d'invariant à l'aide de relation de récurrence)
- de définir le type séquence marquée et les algorithmes (schémas) sur ces séquences.

Tous les problèmes sont simples et donc la résolution algorithmique de ces problèmes est facile à comprendre. Il n'y a pas d'exemple d'une décomposition d'un problème important mais ce n'est pas l'objectif de ce livre (schémas).

Le type séquence permet d'unifier les types "classiques" table, liste (voire fichier séquentiel). Contrairement à nous leur type séquence possède les fonctions d'accès en tête et en queue et donc les constructeurs associés (e.S et S.e).

Nous avons préféré ne pas unifier les tables et les listes mais nous avons donné à chaque fois une (voire des) définition récursive (et les algorithmes associés) [Can90c]
des tables:

a)
$\{t[1],t[2], \ldots, t[n-1],t[n]\} = \{t[1]\}$ si n=1
$\{t[1],t[2], \ldots, t[n-1]\} \cup \{t[n]\}$

b)
$\{t[i], \ldots, t[n-1],t[n]\} = \{t[n]\}$ si i=n
$\{t[i]\} \cup \{t[i+1], \ldots, t[n]\}$

c)
$\{t[i], \ldots,t[j]\} = \{t[i]\}$ si i=j
$\{t[i], .., t[\frac{i+j}{2}]\} \cup \{t[\frac{i+j}{2}+1], .. ,t[j]\}$

des listes linéaires:
$(a_1,(a_2, \ldots (a_n,()) \ldots))$

et comme nous l'avons signalé lors du chapitre 6, nous n'avons pas voulu que nos étudiants écrivent les fonctions sur les listes avec des accès et des suppressions en queue (coûteuses en cons). Comme les auteurs ont choisi le style itératifs pour tous leurs algorithmes la définition d'une séquence sous la forme S.e leur permet d'utiliser plus facilement les invariants:

Ce livre à lui seul ne peut pas suffire pour un enseignement de DEUG, il doit par contre en faire partie comme cela est fait à Grenoble [BeS90].

9.6) [Gre86]

Grégoire (C.N.A.M.): Informatique-Programmation (Tome 1 et 2 2ème edition chez Masson).

Ce cours, qui est destiné aux auditeurs du C.N.A.M., est l'oeuvre d'une équipe pédagogique et son but est d'apporter de la rigueur à des auditeurs engagés dans la vie professionnelle à la demande même de la profession. Les auteurs ont donc choisi d'orienter leur cours sur les boucles et les invariants (tome 1) et la récursivité sous forme de fonction dans le tome 2. Nous avons pris cet exemple car ce genre de cours est fait en DEUG.

Tome 1) Programmation structurée:

Les auteurs commencent par un cours de logique pour préparer les auditeurs aux preuves, ce qui nous parait sain. Pour chacune des instructions de base, ils donnent une règle pour prouver celles-ci; ils utilisent la même démarche pour les procédures et les fonctions pour lesquelles ils expliquent l'allocation dynamique en pile. Ensuite, les boucles apparaissent avec les invariants, puis les auteurs proposent une méthode pour la construction systématique de programmes fondée sur la relation de récurrence qui est résumée ici:

1) Inventer l'hypothèse de récurrence, les suites qui l'expriment et l'invariant qui définit les propriétés de cette récurrence.
2) Etablir le test d'arrêt (test supplémentaire).
3) construire le système de définitions récurrentes associé aux suites (respect de l'invariant).
4) déterminer les conditions initiales.
5) Prouver l'arrêt (terminaison).

Ils donnent deux exemples qui sont les suivants:

le calcul de A^B et la recherche (séquentielle) dans une chaine (tableau)

a) A^B (idée se servir de la décomposition binaire de B)

1) ils proposent de trouver trois suites (x_i) (y_i) (z_i) (?) telles que l'invariant soit

$$A^B = z_i * x_i^{y_i}$$

2) Critère d'arrêt:

$i^* = PP$ ((i≥0), y_i=0)

résultat=$z_i^*=z^*$

3) $x_{i+1}=x_i^2$

z_{i+1}= si impair(y_i) alors $z_i * x_i$ sinon z_i finsi

$y_{i+1}=y_i$ div 2

+ preuve du respect de l'invariant (voir [Gre86] page 188)

4) conditions initiales x_0=A y_0=B z_0=1

5) convergence $y_{i+1}<y_i$ donc y_i atteindra bien 0

d'où on obtient l'algorithme classique.

Pour cet exemple nous avons donné, une variante séquentielle de x^n que nous avons prouvée à l'aide du théorème de récurrence. Nous pensons que l'algorithme précédent est représentatif de notre choix et nous préférons l'introduire au moment de la récursivité, en donnant une écriture récursive de x^n en fonction de $(x^2)^{n/2}$. Nous attendons les règles d'élimination de la récursivité pour que les étudiants obtiennent exactement le même algorithme itératif comme nous l'avons montré en 5.3.

b) recherche x dans CH

Les auteurs posent CH = CH<c_1 c_2 .... c_n>

invariant: P : (x∉CH<c_1 .. c_{i-1}>) et (non vide(CH<c_i .. c_n>))

arrêt: i^* = PP ((i≥1),(c_i=x) ou (i=n))

résultat x∈CH c_i^* = x

récurrence: on parcourt la chaine par c_{i+1} = suivant(c_i,CH)

initialisation: CH non vide

c1 = premier(CH)

et ils obtiennent un algorithme de recherche classique, ils sont par contre obligés de calculer la longueur de la chaine; si c'est une liste, il faut donc la parcourir en entier et la reparcourir pour savoir si x est dans cette chaine.

Nos propositions sont les suivantes:

* Comme leur chaine ressemble à un tableau, nous pouvons utiliser la propriété suivante:

∃i∈(1,...,n) x = c_i (=CH[i])

* Pour les DEUG 2ème année nous pouvons utiliser la variante dichotomique récursive de cette recherche (voir annexe 2 chapitre 5).

* Si leur chaine est une liste nous proposons une analyse récursive du problème:
Nous posons CH = (c_1, (c_2, ... (c_n,()) ...))
Analyse:
si CH = () alors x ∉ CH
si CH = (c, CH') alors si c = x alors x ∈ CH
si c ≠ x alors x ∈ CH ⇔ x ∈ CH'

et l'on obtient la fonction récursive suivante:

```
function recherche(x:entier;CH:"liste d'entier"):boolean
begin
 if vide(CH) then
   recherche:=false
 else if (x=valeur(CH)) then
   recherche:=true
 else
   recherche:=recherche(x,suivant(CH))
end;
```

Comme la récursivité est terminale, il n'y a aucun problème pour obtenir une version itérative sans calculer la longueur.

Remarque: Les auteurs introduisent la notion de liste ou chaine, ils donnent des fonctions de base sur ces chaines et ne développent pas les listes davantage, à part un ou deux algorithmes. On n'y trouve pas les réalisations de ce type ainsi que des fonctions de base. Par contre, les auteurs expliquent que l'on peut représenter les listes avec des tableaux (représentation consécutive), avec une représentation dispersée sans citer les pointeurs ou à l'aide de fichiers (développée). Ils affirment également que lorsque l'on utilise la représentation dispersée, l'insertion et la suppression d'un élément sont plus faciles, par contre, la lecture d'une suite est plus couteuse que la représentation contigue. C'est mal connaitre les listes car entre les opérations suivantes qui sont évaluées n fois (n+1 pour le test, où n est le nombre de valeurs):

i≤n	vide(l) {l=nil}
t[i]	valeur(l) {l^.valeur}
i:=i+1	suivant(l) {l^.suivant}
représentation contigüe	représentation dispersée

nous ne voyons pas la différence annoncée, à moins que les auteurs ne calculent la longueur de la liste avant de la parcourir!

La programmation structurée est bien présentée mais il n'y a pas de décomposition d'un problème.

Les auteurs font également une remarque assez pertinente sur l'affectation unique, qui est la suivante:
"il serait souhaitable de ne jamais changer la relation entre un nom et l'objet qu'il désigne (variable et valeur qu'elle contient). Ainsi les programmmes seraient plus clairs, moins compliqués et certainement plus fiables".

ils ajoutent "on ne peut pas toujours le faire" sans doute à cause des boucles.

A nous d'ajouter: "on peut toujours le faire en utilisant la récursivité, car lors d'un appel récursif un emplacement mémoire est alloué à chaque paramètre, donnée et résultat. Ces emplacements contiendront à chaque fois une unique valeur, sauf si nous les réutilisons"

Tome 2) La spécification récursive et l'analyse des algorithmes.
Les auteurs disent assez vite que l'on doit utiliser la récursivité:
"si un problème ou les données qu'il traite sont définis récursivement alors une solution récursive peut paraitre appropriée et facile à programmer sans erreur"

Pour nous le "peut paraitre" est de trop.

Ils poursuivent: "Dans bien des cas une solution itérative peut-être plus efficace que la solution récursive. Encore faut-il trouver cette solution itérative.
En résumé, il est souvent fiable de spécifier une solution récursive, mais par souci d'efficacité, la solution récursive doit être écartée au profit d'une solution itérative dans les cas suivants:

- la solution itérative est évidente
- une étude de la solution récursive montre que la profondeur de la récursivité est d'un ordre supérieur à $n\log_2(n)$, et on sait qu'il existe une solution itérative (rappelons qu'elle n'existe pas nécessairement)."

C'est ce genre de remarques qui ont fait énormément de tort à la récursivité car elle n'est plus correctement utilisée, elle perd toutes ses vertus pédagogiques pour devenir une technique destinée à résoudre certains problèmes insolubles autrement. C'est pour ces raisons que les auteurs insistent sur les boucles. Par souci de

rigueur, qui les caractérise, ils prouvent leurs algorithmes. Ce qui est correctement fait et correspond à leur orientation pédagogique.

Par contre, ils proposent des règles d'élimination de la récursivité pour les fonctions. Il nous semble bizarre qu'ils trouvent d'abord une table d'association à partir de l'analyse de leur problème, qu'ils en déduisent une version itérative puis qu'ils donnent seulement la version récursive.

Conclusion

Les auditeurs du C.N.A.M. sont souvent des adultes motivés qui viennent se perfectionner ou se remettre à niveau en informatique. Par contre nos étudiants de DEUG suivent d'autres cours qu'ils estiment plus fondamentaux comme les mathématiques ou la physique. Si nous leur apprenons l'informatique de cette manière nous pensons qu'ils ne seront pas nombreux à poursuivre cette matière. Nous croyons même qu'ils abandonneront volontiers notre discipline au profit d'autres, car nos étudiants sont pour la plupart novices. Il faut absolument leur apprendre les bases mais surtout ne pas les "matraquer" avec notre "jargon", il faut les intéresser, tout en leur inculquant de la rigueur. Nous utilisons donc le théorème de récurrence pour prouver la correction des boucles for et nous obtenons les algorithmes classiques avec les boucles while à l'aide d'une propriété que respecte le résultat de notre algorithme. Il est vrai que ces propriétés peuvent être considérées comme des invariants, mais elles sont plus souples d'utilisation et nous ne devons pas faire appel à un système de preuves complet pour montrer que ces algorithmes sont corrects.

9.7) [Duc84]

A. Ducrin: Programmation, Méthode déductive

Nous avions adapté cette méthode pour nos étudiants des DEUG première année en 1984. En 1985 nous ne l'utilisions plus car nous pensions déjà à l'époque que ce n'était pas de cette manière que nous pourrions intéresser nos étudiants, car la méthode est trop lourde à gérer: lourde pour les étudiants car dès que le problème se complique la table devient immense et donc incompréhensible.

A la lecture de l'ouvrage "Programmation" d'Amédée Ducrin destiné aux étudiants ainsi qu'à toute personne soucieuse d'apprendre à programmer, le principe de la méthode est clair:

"partez du résultat de votre problème, remontez jusqu'aux données de celui-ci, redescendez votre chemin et vous obtiendrez un algorithme."

La réalisation de cette méthode est faite de la manière suivante: En remontant vers les données, remplissez à chaque pas la table avec la définition qui vous a fait remonter (décomposition d'expression). A la fin numérotez les définitions et l'algorithme apparait.

nous pouvons faire les constatations suivantes:

La méthode est simple.
Sa réalisation l'est beaucoup moins.

De plus, d'une manière générale,

- c'est un ouvrage d'initiation d'exemples, jamais A. Ducrin n'a formalisé ses méthodes (abstraction de l'exemple), si ce n'est avec des organigrammes ou de petits résumés à la fin d'un chapitre. L'étudiant doit faire sa propre abstraction sur les exemples. Nous avons préféré montrer à nos étudiants que les instructions qu'ils avaient à leur disposition ne leur suffisaient pas pour résoudre un exemple particulier. Puis nous donnions la définition (syntaxe + sémantique) pour réaliser grace à elle d'autres exemples.

- rien ou presque n'est démontré, comme si le fait de mettre toutes les informations dans la table suffisait pour montrer que l'analyse est correcte.

- il y a beaucoup d'instructions ou de définitions d'écriture. Nous pensons que ces instructions ne font pas réellement partie de l'algorithme proprement dit. Nous avons toujours considéré qu'un programme est composé de trois parties:

```
Initialisation des données D
Pb(D,R) { l'algorithme }
Exploitation des résultats R
```

pour que l'étudiant dissocie bien qu'il y a deux choses à faire, sans compter nos remarques sur programme= représentation d'un algorithme (voir Ana Gram). Il y a évidemment, et nous ne l'ignorons pas, certains cas où l'on doit afficher ou lire des informations durant l'exécution d'un algorithme.

- il y a beaucoup de définitions sur les boucles comme:

t pour i de p à q
dernier t pour i de p à q
t_p suivi de t pour i de p+1 à q
t pour i de p jusqu'à arrêt exclu
t pour i de p jusqu'à arrêt

pour calculer les suites récurrentes A. Ducrin utilise l'ancienne valeur pour calculer la nouvelle, ce qui est une technique connue et très utilisée dans la programmation itérative. Par contre, dans ces cas, il utilise sa notation propre, en étant obligé de définir une deuxième table, comme le montre l'exemple suivant du calcul de la suite:

u_0="valeur de u_0"
$u_n=f(u_{n-1})$

* calcul de u_n		
- u (type) contient u_n	3	résultat = ecrire u
	2	n = donnée
	1	u_0= "valeur de u_0"
u pour p de 1 à n		
	1	u=f(@u)

- les conditionnelles sont introduites après les boucles car, pour cette méthode, elles sont plus délicates à utiliser.

- les conditions sont très proches du résultat alors qu'elles se trouvent généralement au début dans les programmes.

- la programmation structurée et la récursivité occupent une petite place. Par contre ces quelques pages sur la récursivité ont été pour nous réconfortantes, car quand A. Ducrin donne ses tables pour la variante dichotomique de x^n ainsi que celle de hanoi, elles deviennent enfin lisibles, car tout le reste se trouve caché dans la récursivité:

f(D) est son résultat, D sa donnée (donnée et résultat ne peuvent pas être plus liés), c'est la définition récursive de f(D) qui va décomposer, à l'exécution, l'expression récursive f(D) et ce jusqu'à arriver à une donnée où l'on connaît le résultat. C'est

la fonction qui retrouvera son chemin (arrêt des appels récursifs) en remontant et qui effectuera les calculs nécessaires jusqu'à calculer f(D).

Ce genre de remarque aurait pu diriger A. Ducrin vers une autre voie, une voie applicative (qui est pourtant la sienne), avec laquelle il aurait pu, au lieu de décomposer son résultat avec des expressions qui utilisent des variables, décomposer son expression à l'aide de fonctions. Ainsi, il n'aurait pas été obligé de mettre toutes ses définitions dans une unique table.

Nous avons l'impression qu'A. Ducrin a voulu écrire un algorithme résoudre_un_problème. Le principe de son algorithme est le suivant: il faut partir du résultat; ce principe est simple. Pour le mettre en oeuvre A. Ducrin choisit ses structures de données: une table qui contiendra toutes les définitions, à la fin il suffira d'affecter un numéro à chaque définition (ordre d'exécution). Pour les identificateurs qui sont résultats d'une boucle (suite récurrente) il faut définir une sous-table. Lorsqu'une définition dépend de cas, il faut que tous les cas apparaissent dans la table. Pour les analyses A. Ducrin utilise des organigrammes.

9.8) [Gra86]

Anna Gram: Raisonner pour programmer

C'est un ouvrage qui propose plusieurs moyens ou stratégies pour arriver, à partir d'un problème donné (cahier des charges), à construire un programme qui résoud un problème (solution informatique). Nous n'allons pas exposer tous ces moyens, par contre nous essayerons de donner les nôtres avec leur notation. Il nous semble que le nombre de ces stratégies soit trop important pour qu'un étudiant de DEUG puisse l'utiliser, la boite à outils "résoudre les problèmes" en contient trop. Ses possibilités de résoudre un problème étant trop importantes l'étudiant ne choisira pas forcément la bonne stratégie. Les auteurs en sont conscients car leurs travaux s'adressent à des personnes qui disposent déjà d'une expérience certaine en analyse et programmation; ces travaux s'adresseront donc à nos étudiants une fois que ceux-ci seront correctement formés.

Nous allons par contre essayer d'utiliser A. Gram pour expliquer nos convictions. Revenons à notre idée de moule unique: lorsque nous disons qu'il doit être le plus rigide possible, cela signifie qu'il doit donner peu d'outils aux étudiants pour construire leurs premiers algorithmes ou programmes. Par contre il doit être plus ou moins représentatif des techniques diversement employées, pour

que nos étudiants, une fois sortis de ce moule, soient capables d'évoluer et de s'adapter pour comprendre d'autres techniques.

Toutes les stratégies partent d'une idée capitale pour pouvoir résoudre n'importe quel problème. Notre objectif est, pour un étudiant de DEUG, d'arriver assez vite de son idée à un algorithme correct, car en général nos étudiants ont de bonnes idées et lorsqu'ils n'ont pas trop d'antécédents informatiques, celles-ci sont généralement simples. Par contre c'est dans la réalisation de leur idée, donc dans l'écriture ou l'obtention de l'algorithme qu'ils ont le plus de problèmes. C'est pour celà que nous n'utilisons pas ITERATION (sauf pour des cas extrêmement simples). Pour obtenir ces algorithmes nous proposons que l'étudiant exprime (SPECIFIER) son idée à l'aide d'une propriété. Nous utilisons DECOMPOSITION dans le cas où le problème à résoudre est statique. Chaque fois que l'on décompose un problème en plusieurs sous-problèmes, pour chacun de ces sous-problèmes on repart (REPARTIR) avec des contraintes (PARAMETRER, DESIGNER, TYPER) (voir annexe 1 chapitre 12). Nous n'utilisons REPRESENTER que le plus tard possible comme conseillé par les auteurs ".. Le choix réalisé à ce stade est déterminant du fait de l'existence d'un grand nombre de solutions, difficiles à imaginer et surtout à comparer, les conséquences de ces maladresses pouvant très bien n'apparaitre qu'après un long développement."

Ce qui confirme notre équation

Programme = Représenter(Algorithme ⊢— Structures de données)

Pour les étudiants de DEUG deuxième année nous préconisons INDUCTION avec DECOMPOSITION, car nous voulons que nos étudiants spécifient de manière récursive tous leurs problèmes récursifs; si lors de la décomposition, d'autres sous-problèmes apparaissent, il faut les traiter de la même façon (voir exemple du tri par insertion). Cette stratégie PROPRE-CORRECT peut-être résumée par:

PROPRE-CORRECT= Idée, puis démontrer (s'il le faut)
puis (DECOMPOSITION avec INDUCTION)*
puis REPRESENTER

ou

PROPRE-CORRECT= simple

Si les étudiants veulent des algorithmes itératifs ils utiliseront le paradigme ELIMINER. En fait la stratégie ITERATION est la suivante:

ITERATION= INDUCTION puis ELIMINER
puis OPTIMISER ou TRANSFORMER avec REPRESENTER.

Les listes et les arbres sont décrits de manière fonctionnelle. Par contre ces types sont peu utilisés, le plus courant étant le type tableau. On sent assez facilement que les auteurs avaient le type tableau en tête avant l'analyse. L'exemple qui nous a le plus frappé est celui des sous-suites ascendantes. Les auteurs supposent que la suite initiale est dans une liste et ne font que des accès en queue de liste (S.e) [ScP88]. Rappelons que leur objectif était d'obtenir un algorithme itératif et que sous cette forme (tableau), les listes s'adaptent bien aux invariants qui permettront d'obtenir des boucles. Notre analyse applicative et récursive de ce problème est la suivante:

Soit l une liste (la suite)
Si l=() alors la sous-suite ascendante de l ne peut être que la liste vide.
Si l=(a,l') alors la sous-suite ascendante de l peut être une sous-suite ascendante de l' ou alors elle peut commencer par a et par conséquent le suivant de a dans la sous-suite sera une sous-suite ascendante de l' qui commence par une valeur plus grande que a.
Pour les longueurs il suffira dans le cas d'une suite non vide de prendre le maximum entre les deux plus longues proposées.
D'où on obtient:

```
function lg_max(l:liste):liste;
begin
if vide(l) then
  lg_max:=0
else
  lg_max:=max(lg_max(suivant(l),
              1+lg_max_sup(suivant(l),valeur(l))
end;
```

Cette décomposition a fait apparaitre un autre problème qui est presque du même ordre:

Analyse de lg_max_sup(l,v) soit une sous-suite ascendante de l qui commence par une valeur plus grande que v.

si l=() alors une telle sous-suite ne peut-être que vide
si l=(a,l') alors une telle sous-suite peut être une sous-suite de l' qui commence par une valeur plus grande que v par contre si a≥v alors une telle sous-suite peut-être une sous-suite qui commence par a et dont le suivant est une sous-suite ascendante de l' qui

commence par une valeur plus grande que a.

Donc pour les longueurs il faudra prendre le maximum et on obtient la fonction suivante:

```
function lg_max_sup(l:liste,v:V):liste;
begin
if vide(l) then
  lg_max_sup:=0
else
  lg_max_sup:=max(lg_max_sup(suivant(l),v),
              1+lg_max_sup(suivant(l),valeur(l))
end;
```

Cette fonction ne fait appel qu'aux fonctions suivant, vide et valeur, elle peut donc être écrite sans problème avec un tableau à la place d'une liste.

[Souvenir]
[Pour ce qui est de l'exemple des quilles, il me touche particulièrement à cause d'une anecdote. J. GIMIE D'ARNAUD un mathématicien en poste à l'Université de Metz, m'avait demandé un algorithme qui lui donnerait toutes les répartitions de b boules sur t tas (un tas pouvant être vide), cet algorithme devait être itératif pour le programmer en Basic, son unique langage.Il en avait besoin pour faire des probabilités. Je lui ai dit qu'apparemment ce problème n'était pas difficile (pour moi) et qu'il aurait la solution pour le lendemain. Le soir même j'ai fait mon analyse récursive, j'ai écrit un algorithme récursif, puis j'ai éliminé la récursivité. Comme convenu, je lui ai apporté ma solution le lendemain. Il m'a dit, après avoir exécuté mon programme, que le sien trouvait les mêmes solutions; par contre, il n'était pas sur de sa correction, il avait des excuses, il n'était pas informaticien. Je regrette qu'il ne soit plus de ce monde pour pouvoir lire mon cours, afin de lui montrer noir sur blanc ce qu'est pour moi l'informatique.

Depuis, je donne ce problème à mes étudiants de licences d'informatique et de mathématique. A chaque fois que je leur demande de compter le nombre de répartitions pour préparer l'algorithme qui les donnera toutes, il y a quatre ou cinq étudiants qui essayent de trouver une formule avec de t! b! et (b+t)! ou des variantes avec t+1, b+1 et b+t+1. A chaque fois je leur explique qu'une telle formule n'est pas exploitable pour caractériser toutes les répartitions. Ceci pour répondre à la remarque faite par les auteurs "Une première idée est de chercher

le calcul de manière analytique. Elle est abandonnée devant la difficulté de trouver une formule." Dans leur cas, il s'agissait de trouver le nombre et non toutes les répartitions.

```
procedure repartir(a:array[1..t] of integer;b,i:integer);
var k:integer;
begin
 if (i=t) then
  begin
   a[i]:=b; afficher(a)
  end
 else
  begin
    for k:=0 to b do
     begin
      a[i]:=k;repartir(a,b-k,i+1)
     end
  end
end;
```

Pour éliminer la récursivité, comme il y a b+1 appels récursifs, il faut une pile sur laquelle on empile i et k pour savoir s'il y a une autre suite ou non.

[fin du souvenir]

Chapitre 10

POURSUITES

10.1) Les arbres et les graphes

Nous nous intéressons également aux arbres pour lesquels nous avons déjà donné un aperçu de ce que pouvait être notre démarche dans l'exemple du dictionnaire. Comme nous l'avons également signalé lors du chapitre 6 consacré aux listes linéaires, nous devons absolument poursuivre le même effort en adoptant une approche fonctionnelle pour les arbres. Nous avons déjà une grande expérience (6 années) dans l'enseignement des arbres et graphes pour les étudiants de Licence d'informatique et de mathématique. Nous avons volontairement laissé ce cours d'algorithmique pour prendre en charge, depuis l'année universitaire 1990-91, une partie des TD d'algorithmique sur les arbres et graphes pour les étudiants du DEUG deuxième année (Mathématique-Electronique -Informatique). Les résultats déjà obtenus sont très prometteurs pour l'avenir de notre discipline.

Le problème des graphes est l'arrêt des algorithmes et leur correction. Notre démarche est encore simple: nous voulons, avant d'écrire un algorithme sur les graphes, être sûr qu'il est correct. Dans ce cas nous utilisons une approche équationnelle et la théorie du point fixe pour montrer que nos algorithmes sont corrects.

Un exemple: le calcul de $\hat{\Gamma}(x)$ l'ensemble des descendants de x dans le graphe.

Définitions
Soit $G=[X,\Gamma]$ un graphe X est un ensemble de sommets et Γ la fonction successeur:

$$\Gamma : X \longrightarrow \mathcal{P}(X)$$
$$x \longrightarrow \Gamma(x)$$

nous pouvons appliquer Γ à un sous-ensemble de X de la manière suivante:
soit $A \subseteq X$ alors $\Gamma(A) = \bigcup_{x \in A} \Gamma(x)$

cela nous permet donc de définir Γ^i de la manière suivante:

$$\Gamma^0 = id$$
$$\Gamma^{i+1} = \Gamma \circ \Gamma^i$$

on définit donc $\hat{\Gamma} : X \longrightarrow \mathcal{P}(X)$ de la manière suivante:

$$\hat{\Gamma}(x) = \bigcup_{i\geq 1} \Gamma^i(x)$$

d'où $\hat{\Gamma}(x)$ est la plus petite solution (au sens de l'inclusion) de l'équation ensembliste suivante:

$$Y = \Gamma(\{x\} \cup Y)$$

Preuve (classique en point fixe):

1) $\hat{\Gamma}(x)$ est une solution de l'équation

$$\begin{aligned}
\Gamma(\{x\} \cup \hat{\Gamma}(x)) &= \Gamma(\{x\} \cup \bigcup_{i\geq 1} \Gamma^i(x)) \\
&= \Gamma(x) \cup \Gamma(\bigcup_{i\geq 1} \Gamma^i(x)) && \text{car } \Gamma(A\cup B)=\Gamma(A)\cup\Gamma(B) \\
&= \Gamma(x) \cup \bigcup_{i\geq 1} \Gamma(\Gamma^i(x)) && \text{car } \Gamma(A\cup B)=\Gamma(A)\cup\Gamma(B) \\
&= \Gamma(x) \cup \bigcup_{i\geq 1} \Gamma^{i+1}(x) && \text{car } \Gamma(\Gamma^i(x))=\Gamma^{i+1}(x) \\
&= \Gamma(x) \cup \bigcup_{i\geq 2} \Gamma^i(x) \\
&= \bigcup_{i\geq 1} \Gamma^i(x) \\
&= \hat{\Gamma}(x)
\end{aligned}$$

d'où $\hat{\Gamma}(x)$ est une solution de l'équation: $Y = \Gamma(\{x\} \cup Y)$

2) $\hat{\Gamma}(x)$ est la plus petite solution
Soit Y une autre solution on montre par récurrence que:

$\forall n\geq 1 \ \Gamma^n(x) \subseteq Y$

a) $\{x\} \subseteq \{x\} \cup Y$ donc $\Gamma(\{x\}) = \Gamma^1(x) \subseteq \Gamma(\{x\} \cup Y) = Y$

b) $\forall n\geq 1 \ \Gamma^n(x) \subseteq Y \Rightarrow \Gamma^{n+1}(x) \subseteq Y$

$\Gamma^{n+1}(x) = \Gamma(\Gamma^n(x)) \subseteq \Gamma(Y) \subseteq \Gamma(\{x\} \cup Y) = Y$

d'où comme $\forall n\geq 1 \ \Gamma^n(x) \subseteq Y$ on a $\bigcup_{i\geq 1} \Gamma^i(x) = \hat{\Gamma}(x) \subseteq Y$

et nous obtenons l'algorithme qui calcule $\hat{\Gamma}(x)$ en construisant une

suite croissante (au sens de l'inclusion) d'ensemble (démonstration par récurrence). Lorsque la suite est stationnaire (ce qui est le cas lorsque l'ensemble des sommets du graphe est fini) elle atteint le plus petit point fixe (c'est un point fixe et plus petit que le plus petit car $\emptyset \subseteq \hat{\Gamma}(x)$)

```
Y0:=∅;
Y1:=Γ(x);
while (Y0≠Y1) do
begin
 Y0:=Y1;
 Y1:=Γ({x}∪Y1)
end;
```

$\hat{\Gamma} := Y_0$

Cela n'empêche pas de donner les autres parcours de graphes (profondeur d'abord, largeur, arc nouveau).

10.2) Compilation: Optimisation de code

De plus, pour que notre discours soit plus fort encore, il faut développer en compilation la partie optimisation de code, pour pouvoir écrire de façon plus lisible les programmes, afin que la récursivité ne soit plus critiquée et qu'elle puisse ainsi être aussi performante que les boucles à l'exécution. Nous rappelons que les compilateurs peuvent, depuis longtemps, détecter les sous-expressions communes; combien de temps faudra-t-il attendre pour que les concepteurs d'algorithmes ne s'occupent plus du travail de la machine, en décomposant les expressions pour ne pas recalculer plusieurs fois la même sous-expression?

Il faut que la stratégie ITERATION soit assurée par les compilateurs.

10.3) Les logiciels d'aides

10.3.1) Pour les propriétés

Ce logiciel permettrait aux étudiants:
- de définir une propriété ou un parcours et il nous fournirait l'algorithme associé qui pourrait être testé.
- (éventuellement) de le transformer.

Réalisation: Elle est simple si on utilise un logiciel comme yacc

ou lex. Il suffit de définir une grammaire dont le langage généré est exactement l'ensemble des propriétés et d'ajouter des attributs et des actions sémantiques qui génèrent l'algorithme ou qui construisent l'arbre syntaxique de l'algorithme pour pouvoir l'interpréter.

10.3.2) Un dérécursiveur

Transformateur automatique d'un texte récursif en texte itératif: un dérécursiveur.

Réalisation: Pour ce logiciel, également, nous pouvons nous servir des mêmes techniques de compilation. Donnons un exemple:

si l'on se trouve dans le cas de réduction suivant:

```
if exb then
  inst1
else
  inst2
```

on ajoute les attributs pour chaque instruction:
$b(x)$, $g(x)$, $\varphi(x)$, $\psi(x)$, $\varphi_0(x)$ pour les procédures

donc soit les attributs pour instr1:
$b1(x)$, $g1(x)$, $\varphi1(x)$, $\psi1(x)$, $\varphi_0 1(x)$

donc soit les attributs pour instr2:
$b2(x)$, $g2(x)$, $\varphi2(x)$, $\psi2(x)$, $\varphi_0 2(x)$

d'où $b(x)$=exb **and** $b1(x)$ **or not** exb **and** $b2(x)$ avec transformation si $b1(x)$ ou $b2(x)$ sont des constantes (vrai ou faux)

```
φ(x)= if exb then
        φ1(x)
      else
        φ2(x)
```

avec transformation si $\varphi1(x)=\varphi2(x)$ ou si $\varphi1(x)$ ou $\varphi2(x)$ sont vides.

$g(x)$, $\psi(x)$ et $\varphi_0(x)$ sont similaires à $\varphi(x)$

remarques:
- si $\psi(x)$ est vide alors la récursivité est terminale.
- si $g1(x)\neq g2(x)$ et $\varphi1(x)\neq\varphi2(x)$ on peut calculer φ et g en même temps (dans le même if)

Pour les transformations des fonctions en des algorithmes qui calculent la valeur dans la première boucle on peut le faire ou ne pas le faire dans des cas classiques et connus. Par contre, pour les autres cas, le logiciel pourra proposer l'optimisation et le faire tester soit par comparaisons des résultats d'exécution des deux versions de l'algorithme (récursif et itératif) sur un grand nombre de valeurs ou soit en demandant ces valeurs à l'utilisateur; ceci ne pourra être en aucun cas une preuve de correction si tous les tests ont été concluants, comme une exécution correcte d'un programme n'assure pas sa correction.

En ce qui concerne les algorithmes avec les tableaux il faudra s'assurer si oui ou non les indices sont dans les bonnes bornes à chaque fois. Dans ce cas nous pouvons utiliser des techniques d'encadrement de valeur de variable, s'il y a un risque de débordement il faudra enlever le test de b(x) et le réinjecter dans le corps de la première boucle.

10.3.3) Logiciel de support de cours

Un logiciel de présentation du cours. On peut facilement imaginer un logiciel qui présente la notion ou le concept qui va être développé dans le prochain cours et un autre logiciel qui conforte la notion ou le concept après le cours ou les TD. On peut également imaginer en séance initiatrice un logiciel de présentation des capacités de la machine à résoudre certains problèmes connus. Ce genre de logiciel et bien d'autres sont développés dans le Laboratoire ARCADE [Pey91].

10.3.4) Utilisation de logiciel écrit par des étudiants

> Nous mettons en garde le lecteur, il ne s'agit pas de faire de la publicité en présentant une liste de logiciels développés par nos étudiants à notre demande. Il s'agit pour nous de montrer les résultats que nous avons obtenus.

Il s'agit d'utiliser des interpréteurs de mini-langage écrits par des étudiants de Licence d'informatique lors de leur projet du cours d'algorithmique. Cela permettra aux étudiants de DEUG d'être confrontés à des travaux d'étudiants de second cycle afin de leur montrer de quoi ils seront capables s'ils poursuivent en informatique. Celui qui concerne le plus le cours de DEUG est un interpréteur d'un mini-Lisp (restriction aux entiers avec +,-,*,/ ... et listes d'entiers avec les fonctions de base prédéfinies)

écrit par les étudiants de Licence d'Informatique de Metz (1988-89).

Ce logiciel permettait de définir des fonctions et de les évaluer comme le montre l'exemple de la fonction concat suivant:

```
concat(l1,l2) = if vide(l1) then
                  l2
                else
                  cons(valeur(l1),concat(suivant(l1),l2))
'concat((1,2),(3,4,5))
(1,2,3,4,5)
```

les étudiants avaient plusieurs options de vérification de type:
- à l'exécution; arrêt si une fonction de base est mal utilisée exemple: concat((1),2) lors de cons(1,2) 2 n'est pas une liste
- à la définition de la fonction; sur l'exemple précédent avec vide(l1) l1 est une liste (valeur(l1) et suivant(l1) confirme) comme le résultat de la fonction est une liste (résultat de cons) l2 ne peut être qu'une liste.

Ce langage a été utilisé par les étudiants de la filière Mathématique et Informatique qui ont apprécié cette séance ce qui nous encourage à tenter à nouveau l'expérience cette année. De plus, pour ces étudiants, nous pouvons, lors d'un cours complémentaire, utiliser les logiciels suivants que nous avons donnés à faire:

- Lors de l'année 1987-88 nous avions donné avec D. Méry un mini interpréteur PROLOG. Ce logiciel permettait de définir des axiomes (vrai) puis des clauses de Horn (même récursives) et de poser une question; leur interpréteur cherchait un chemin en utilisant les clauses jusqu'à n'avoir que des axiomes. Nous avions donné l'algorithme d'unification en TD.

- Lors de l'année 1989-90 nous avions donné un langage qui n'utilisait que des procédures avec les deux passages de paramètres. Les types utilisés étaient les entiers, les listes et les arbres (binaires) d'un type défini. On pouvait donc définir des listes de listes ou d'arbres etc ... . On pouvait également avoir un appel avec la variable qui contient suivant(l), gauche(a) ou droit(a) en appelant p(... var suivant(l)).

- Nous pouvons également, comme il y a une initiation aux expressions régulières et aux grammaires hors contexte dans ce cours complémentaire, utiliser le générateur automatique d'analyseur lexical et syntaxique sur lequel on peut rajouter des attributs et des actions sémantiques (de manière automatique). Ce

logiciel est écrit par les étudiants de maitrise d'informatique durant le cours de compilation.

10.3.5) Un décompositeur de problème

Nous avons déjà décrit sommairement ce logiciel (chapitres 4 et 6); il aiderait à décomposer un problème en suivant le fil directeur donné précédemment (chapitre 4):
Au départ il demanderait le nom du problème à résoudre ainsi qu'un commentaire décrivant le problème.

Pour chaque problème il procèderait de la manière suivante:

Si le problème n'existe pas le logiciel donnera les contraintes (identificateur des types des données et résultats) dues à la décomposition ainsi que le commentaire associé au problème. Le logiciel demandera le nom des paramètres ainsi que l'algorithme qui résoud le problème. Si cet algorithme utilise des variables qui ne sont pas des paramètres il pourra demander s'il s'agit d'une variable d'un problème d'un autre niveau; si la réponse est non, il pourra soit déterminer le type de manière explicite et le précisera, soit le demander (identificateur). Si cet algorithme utilise des sous-problèmes, il en déduira des contraintes sur les types des paramètres et demandera si ce problème existe déjà (existence d'un problème de même nom avec les mêmes types pour les paramètres; stratégie du moindre effort). Il y aura constamment des tests de cohérence des contraintes (vérification de type et de bonne utilisation).

L'utilisateur pourra tenter de représenter ces types et cela de manière conversationnelle (proposition de choix en fonction d'un descriptif d'une valeur de ce type) et d'écrire les sous-problèmes dépendant de la représentation du type. En fonction du choix de la représentation, le logiciel pourra proposer à l'utilisateur des schémas d'algorithmes, surtout sur les tables, comme les propriétés (logiciel 1) ou les parcours. En ce qui concerne les types inductifs, l'utilisateur pourra définir son type de manière récursive et définir les fonctions de bases identificateurs pour les projections (fonctions de consultation) et de construction. La représentation du type et des fonctions de base sera produit automatiquement. Pour les adeptes des procédures, ils pourront utiliser soit le type pointeur qu'ils définiront de manière explicite (comme le permet le langage de programmation) soit les fonctions qui utilisent les mêmes emplacements ou utiliser une notation du style:

procedure pb(**var** l:liste) pour la déclaration
pb(**var** suivant(l)) pour l'utilisation

le décompositeur pourra également interpréter l'algorithme d'un sous-problème complet sur des exemples de données. Il peut à la

limite le faire de manière systématique (correction remontante des programmes).

L'utilisateur pourra choisir le problème qu'il veut décomposer en voyageant dans l'arbre de décomposition. A défaut le décompositeur pourra lui en proposer un.

Lorsque la décomposition sera terminée, le décompositeur donnera en résultat un programme que l'on pourra compiler (0 erreur de compilation) et exécuter (aucune assurance d'avoir 0 erreur; si oui aucune chance que l'informatique soit une discipline).

Ce logiciel pourra dans une faible mesure répondre aux souhaits des auteurs d'Anna Gram [Gra86], qui demandent que des langages de méta-programmation se développent. Rien ne nous empêchera de rajouter des spécifications constructives en terme de pré et post conditions comme celà est fait dans eiffel [Mey90], mais il faut que celles-ci construisent l'algorithme solution car "les développeurs fondés sur la sémantique, ce serait mieux" [Gra86].

BIBLIOGRAPHIE

[Aba91]: **Groupe Abalone**, Enseignement de l'informatique utilisant les langages fonctionnels dans les premiers cycles universitaires, Journées de Rennes, novembre 1991.

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

[AHU87]: **A. Aho, J. Hopcroft, et J. Ullman**, Data structure and algorithms, Addison Wesley.
Version française: stuctures de données et algorithmes, InterEditions, 1987.

[AhU73]: **A. Aho, et J. Ullman**, The Theory of parsing, Translation and compiling, (2 Tomes), Prentice-Hall, Englewood Cliffs N.J, 1973

[AhU79]: **A. Aho, et J. Ullman**, Principles of compiler design, Addison Wesley, 1979.
Version française: Compilateurs Principes, technique et outils, InterEditions, 1989.

[Ars80]: **J. Arsac**, Premières leçons de programmation, Fernand Nathan, 1980.

[Ars83]: J. **Arsac**, les bases de la programmation, DUNOD informatique, 1983.

[Ars88]: **J. Arsac**, La didactique de l'informatique: un problème ouvert?, Premier colloque francophone sur la didactique de l'informatique, Paris, 1988.

[Ars90]: **J. Arsac**, Vous avez dit algorithmique?, Deuxième colloque sur la didactique sur l'informatique, Namur, 1990.

[BeB89]: **P. Berlioux, et P. Bizard**, Algorithmique (2 tomes), DUNOD informatique, 1989.

[BeS90]: **J.-P. Bertrandias, P.-C. Scholl**, L'informatique, discipline fondamentale en DEUG A: un exemple à Grenoble, Annexe 6 des journées SPECIF de Nantes [SPE90], mars 1990.

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

[Can90a]: **D. Cansell**, L'informatique en tant que discipline dans les DEUG scientifiques cela existe. Un exemple à l'Université de Metz, annexe 5 des journées SPECIF de Nantes [SPE90], mars 1990.

[Can90b]: **D. Cansell**, Initiation à l'algorithmique et à la programmation PASCAL par l'exemple, Polycopié de cours de DEUG 1, 1990.

[Can90c]: **D. Cansell**, Initiation à la récursivité et à la programmation récursive, Polycopié de cours de DEUG 2, 1990.

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

[CFD88]: Actes du premier colloque francophone sur la didactique de l'informatique, Paris, 1988.

[CFD90]: Actes du deuxième colloque sur la didactique de l'informatique, Namur, 1990.

[ClB84]: **G. Clavel, et J. Biondi**, Introduction à la programmation (3 tomes), Masson, 1984.

[CoK87]: **J. Courtin et J. Kowarsky**, Initiation à l'algorithmique et aux structures de données, chez DUNOD informatique, 1987
Tome1) Programmation structurée:
Tome2) récursivité et structure de données avancées:

[Cou88]: **P. Cousot**, Introduction à l'algorithmique numérique et à la programmation en Pascal, McGraw-Hill, 1988.

[Dij76]: **E. W. Dijkstra**, A discipline of programming, Prentice-Hall, Englewood Cliffs, 1976.

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

[Duc84]: **A. Ducrin**, Les bases de la programmation, Dunod informatique, 1984

[Duch90]: **C. Duchâteau**, Synthèse d'un colloque en trois temps ou la valse est un art difficile, Deuxième colloque francophone sur la didactique de l'informatique, 1990.

[Gar87]: **Y. Gardan**, une nouvelle approche pour la conception de système de CAO: application aux générateurs dans SACADO, Rapport de recherche LRIM-Metz, 1987.

[Gal188]: Y. Gardan, J.-P. Jung, J.-N. Kolopp, C. Minich, W. Totino, Une nouvelle approche de la convivialité application au système SACADO, MICAD 88, Paris, 1988.

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

[GoL86]: L. Goldschlager, et A. Lister, Informatique et algorithmique, InterEdition, 1986.

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

[Gre86]: Grégoire (C.N.A.M.), Informatique-Programmation, 2ème edition, Masson, 1986
Tome 1) Programmation structurée:
Tome 2) La spécification récursive et l'analyse des algorithmes.

[Hof85]: D. Hofstadter, Godel Escher Bach, les brins d'une guirlande éternelle, InterEditions, 1985.

[LAE91]: Actes de la journée sur les langages applicatifs dans l'enseignement de l'informatique, revue BIGRE juin 1991.

[Lan90]: G. Languy, L'enseignement de l'informatique dans les études d'ingénieurs, Deuxième colloque sur la didactique de l'informatique, Namur, 1990.

[Lig85]: P. Lignelet, Algorithmique Méthodes et Modèles (3ème édition), C.N.A.M., 1985.

[Luc83]: M. Lucas, Algorithmique et représentation des données, tome 2: évaluations, arbres, graphes, analyse de textes, Masson, 1983.

[LPS83]: M. Lucas, J.P. Peyrin, P.C. Scoll, Algorithmique et représentation des données, tome 1: files, automates d'états finis, Masson, 1983.

[MeB78]: B. Meyer, et C. Beaudoin, Méthodes de programmation, Eyrolles, 1978.

[Mey90]: B. Meyer, Conception et programmation par objets, InterEditions, 1990.

[Pai88]: C. Pair, L'apprentissage de la porgrammation, Premier

colloque francophone sur la didactique de l'informatique, Paris, 1988.

[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.

[Pol67]: G. Polya, La découverte des mathématiques, collection SIGMA 9, Dunod Paris, 1967.

[Pol85]: G. Polya, Comment poser et résoudre un problème, 2ème édition, J. Gabay, 1985.

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

[ScP89]: P.C. Scholl, et J.P. Peyrin, Schémas algorithmiques fondamentaux, Masson, 1989.

[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.

ANNEXE 1

METZ: UN EXEMPLE

Journées SPECIF 1990
Nantes - 27, 28 et 29 Mars 1990

Annexe 5

L'INFORMATIQUE EN TANT QUE DISCIPLINE DANS LES DEUG SCIENTIFIQUES CELA EXISTE.

UN EXEMPLE A L'UNIVERSITE DE METZ

Dominique CANSELL (Metz)

1 PRESENTATION

1.1 DEUG rénové

Les DEUG A et B de l'université de Metz, sont gérés par les UFR scientifiques qui sont l'UFR MIM (Mathématique, Informatique, Mécanique et Automatisme) et l'UFR SciFA (Sciences Fondamentales Appliquées). Ce sont des DEUG rénovés dont les avantages sont les suivants:

* huit semaines d'orientation (d'adaptation à la vie universitaire) durant lesquelles tous les étudiants suivent à peu près les mêmes cours. Ils peuvent donc au terme de ces semaines conforter ou changer l'orientation de leurs études.
* 150 étudiants par séances de cours et 32 par groupes de travaux dirigés (TD), ce qui n'est pas excessif par rapport au nombre d'élèves dans une classe de lycée.
* des contrôles fréquents, pour que l'étudiant puisse organiser et répartir ses efforts tout au long de l'année.

1.2 Le DEUG A

Il y a trois sections dont les thèmes principaux apparaissent dans leur sigle. Ces trois sections sont:

M.I.E. (Mathématique, Informatique, Electronique)
T.M. (Technologie et Matériaux)
P.I.E. (Physique, Informatique, Electronique)

1.3 Le DEUG B

Une seule section:

C.B.B. (Chimie, Biologie, Biochimie)

1.4 L'informatique en DEUG

Tous les étudiants des DEUG scientifiques ont un module d'informatique. Vu le nombre d'heures (370 pour les MIE et plus de 160 pour les autres) et le contenu de ces modules, chaque étudiant peut prétendre avoir eu une bonne formation de base en informatique.

2 MODALITES PRATIQUES

2.1 Nombre d'heures

	cours	travaux dirigés	salle machine
MIE	83	130	170
TM	59	79	80
PIE	57	79	70
CBB	56	56	50

2.2 Matériel

Les étudiants ont à leur disposition une salle machine de 16 PC du plan informatique pour tous.

Remarque: Les étudiants de TM utilisent pour certaines heures les PC de la filière technologique.

2.3 Logiciel

Besoins: un éditeur de texte,
un compilateur Pascal

Choix: un TURBO-Pascal

2.4 Documentation

A part leurs notes de cours et leurs documents personnels, les étudiants ont à leur disposition:
* Le manuel du TURBO-Pascal
* Le manuel d'utilisation du système

2.5 Nombre d'étudiants (1989-90)

MIE1	187	MIE2	67
TM1	74	TM2	53
PIE1	186	PIE2	94
CBB1	268	CBB2	133

3 FONCTIONNEMENT

3.1 Les cours

Ils sont assurés en totalité par des informaticiens de l'UFR MIM (24-01).

3.2 Les Travaux Dirigés

Ils sont assurés en totalité par des informaticiens de l'UFR MIM (24-01) et des vacataires informaticiens. La règle adoptée est que le responsable du cours prend en charge un des groupes de TD et prépare toutes les séances (seul ou avec les autres intervenants) pour que tous les étudiants d'une même section abordent les mêmes exercices.

3.3 Les heures machine

Ces heures machine servent surtout à conforter les notions apprises en cours et utilisées en TD, par des exercices de programmation (algorithmes écrits en TD) et elles servent également à développer un projet bien "dégrossi" en TD.

Pour que l'étudiant puisse utiliser l'éditeur de texte et le compilateur Pascal, les premières séances (4 heures) sont encadrées. Le reste du temps n'est pas encadré, mais l'étudiant n'est pas livré à lui-même. Les travaux de programmation sont demandés ou développés en TD et il peut dialoguer avec les enseignants, les personnels IATOS qui s'occupent de tout le matériel du service informatique, et surtout avec des étudiants plus compétents que lui dans ce domaine. La qualité de certains travaux rendus et le taux d'occupation des salles montrent que cela est largement suffisant.

4 PROGRESSION

Le fil directeur de tous les cours d'informatique de DEUG est l'algorithmique. Nous considérons que c'est la matière de base. Par son biais nous abordons des notions:
- de système
- de langage de programmation impérative
- d'architectures, en décrivant les mécanismes d'allocation de mémoires, d'appel de sous-programme et en écrivant les algorithmes de codages et des opérations sur les entiers, ...
- de type abstrait et de programmation fonctionnelle en utilisant une approche fonctionnelle pour les listes linéaires

4.1 DEUG A et B première année

Même si le nombre d'heures diffère, la progression et le message à faire passer sont les mêmes. L'année est découpée en deux parties.

4.1.1 Les 8 premières semaines

Cours	16 h
TD	24 h (16 pour les CBB)
Machine	8

L'objectif de ce cours est de présenter la discipline informatique et d'aborder les points suivants:
* algorithmique de base (affectation, conditionnelles, boucles, tableau)
* notion de correction d'algorithme (assertions)
* notion de coût d'algorithme (comparaison d'algorithmes)
* historique, présentation, description et utilisation de la machine (système MS/DOS, éditeur de texte, compilateur, exécution de programme).

4.1.2 Les 20 semaines suivantes

	MIE	TM+PIE+CBB
cours	24 h	20 h
TD	36 h	20 h
Machine	60 h	20 h

L'objectif de ce cours est de présenter la programmation structurée et de l'utiliser dans la décomposition de problème. Les points suivants sont abordés:

* programmation structurée (fonction et procédure)
* décomposition d'un problème (programmation arborescente descendante)
* calcul de propriétés (obtention d'un algorithme classique sur les tableaux à l'aide d'une propriété, boucle while avec variable booléenne)
* type et algorithme de base sur les enregistrements
* type et algorithme de base sur les fichiers (fichier standard et TURBO)
* graphisme

Remarque: Les étudiants de MIE développent en TD une partie d'un gros projet. Toutes ces parties sont regroupées à la fin, et donne lieu à une exécution d'un gros programme.

4.2 Le DEUG A Deuxième année

En deuxième année les objectifs et donc les contenus diffèrent entre le DEUG A et B.

4.2.1 Les 14 premières semaines

	MIE	TM+PIE
cours	21	21
TD	35	35
Machine	42	28 (sur l'année)

L'objectif de ce cours est d'expliquer les vertus de la programmation récursive. La récursivité est présentée comme un outil puissant aussi bien pour la conception que pour la compréhension des algorithmes, et cela quel que soit le type de problème abordé (calcul, comportement, tri, liste). L'élimination de la récursivité est enseignée pour que les étudiants puissent faire le lien avec ce qu'ils ont appris en première année, mais surtout pour qu'ils puissent, plus tard dans leur vie professionnelle, programmer leurs algorithmes récursifs (proprement et correctement écrits) en utilisant des langages non récursifs. L'approche fonctionnelle des listes linéaires leur présente la notion de type abstrait et un autre style de programmation, le style fonctionnel. Les points abordés sont les suivants:

* récursivité (théorème de récurrence, arrêt et exécution d'un algorithme récursif, élimination de la récursivité)
* tri (les principes et les algorithmes récursifs de la plupart des tris, notion de coût)
* les listes linéaires (approche fonctionnelle, notion de tas, approche procédurale par transformation des fonctions pour une optimisation des algorithmes en place et en temps)

4.2.2 Les 14 semaines suivantes

Seules les sections MIE et TM ont un cours d'informatique durant cette période.

4.2.2.1 Les MIE

cours	21 h
TD	35 h
Machine	42 h

L'objectif de ce cours est de compléter la connaissance de l'étudiant en définissant et en utilisant les structures de donnée plus complexes telles que les automates, les arbres et les graphes. C'est également d'essayer de trouver la structure de donnée la plus appropriée à un problème donné. Les points suivants sont abordés:

* automate
* grammaire (spécification syntaxique)
* arbres (représentation par table, par liste, dynamique; arbres binaires de recherche; parcours; arbres à critère d'équilibre).

* graphes (exemples; définitions; représentation par matrice, par table, par liste; algorithme de parcours, de recouvrement, du chemin le plus court ..)

4.2.2.2 Les TM

Cours	2 h
Machine	10 h

L'objectif de ce cours est de comprendre les concepts de base en utilisant le logiciel de C.A.O, SACADO (Système Adaptatif de Conception Assistée et de Développement par Ordinateur) développé par l'équipe du professeur Y. Gardan du LRIM (Laboratoire de Recherche en Informatique de Metz).
Les mécaniciens ont créé une MST CFMAO où l'informatique prend une part non négligeable (Algorithmique, Architecture et système, Base de donnée et C.A.O)

4.3 Le DEUG B deuxième année

cours	20 h
TD	20 h
Machine	20 h

L'objectif de ce cours est de donner des compléments d'algorithmique sur les tableaux à la demande des responsables des deuxièmes cycles de chimie et d'initier les étudiants à l'utilisation des tableurs.

5 EVALUATION

Chaque cours d'informatique est évalué de la manière suivante:
- 2 contrôles écrits (de 2 heures)
- 1 note de projet

La règle de calcul de la moyenne est propre à chaque enseignant (commune pour chaque section). Il en existe deux:
1) la note de projet compte comme une note d'écrit
2) la note de projet compte comme une note d'écrit si le projet n'est pas rendu ou comme des points ajoutés (ou enlevés) à la moyenne de l'écrit si le projet est rendu.

Remarque: Les petits projets sont corrigés par simple lecture du dossier de programmation rendu. Les plus conséquents nécessitent une vérification sur machine.

6 OBJECTIFS

Les objectifs particuliers à chaque enseignement ont déjà été donnés. L'objectif général est que chaque étudiant de DEUG puisse dire qu'il a abordé des notions plus ou moins importantes d'informatique théorique (système, architecture, algorithmique et structure de donnée) et qu'il a mis en oeuvre ces notions, au niveau de la machine, par le biais de la programmation Pascal. Il pourra donc exploiter ses connaissances soit dans sa vie professionnelle, soit pour aborder des cours d'informatique plus conséquents d'un deuxième cycle. L'informatique lui apporte également une certaine rigueur, une forme de démonstration constructive (il ne suffit pas de savoir qu'une solution existe encore faut-il pouvoir l'exhiber, la construire), et de plus dans leur travaux de programmation ils doivent aller jusqu'au bout de leurs idées.

7 CONCLUSION

L'informatique, telle qu'elle est enseignée dans les DEUG scientifiques de l'Université de Metz, participe à la formation générale de tous les étudiants. Elle n'est pas reléguée au rang de prestataire de service du syle "informatique=programmer". Elle peut donc sans complexe, à l'Université de Metz, prétendre au titre de DISCIPLINE à part entière.

Annexe 2

REALISATION LOCALE
DE NOTRE ENSEIGNEMENT

A2.0) Intégration de l'informatique au sein du DEUG

Il est évident qu'un enseignement d'informatique en DEUG doit s'intégrer dans l'enseignement global du DEUG, il faut donc lui allouer un nombre suffisant d'heures de cours et de TD pour faire autre chose que de la simple figuration (apprentissage de langage ou pire initiation à certains logiciels), des salles et des machines pour que tous les étudiants de DEUG puissent programmer et des postes d'enseignant-chercheur pour assurer convenablement cet enseignement. Tout ceci est le rôle de l'UFR Mathématique, Informatique, Mécanique et Automatique. Il faut dire que cette UFR joue pleinement son rôle en accordant une priorité assez forte à l'enseignement de notre discipline en DEUG, de même que l'autre UFR scientifique, l'UFR Sciences Fondamentales appliquées (Les DEUG A et B sont gérés par ces deux UFR scientifiques). Notre gros problème est l'augmentation constante des effectifs en DEUG, malgré tous les efforts de l'UFR qui ne peut pas avoir l'informatique comme unique priorité, nous manquons toujours d'enseignants et de machines pour les étudiants de DEUG.

La politique de recrutement d'ATER et la nomination de moniteurs a en partie résolu le problème du manque d'enseignants mais pour assurer la totalité de l'enseignement de DEUG, certains informaticiens font encore trop d'heures complémentaires.

En ce qui concerne le manque de machines, le problème se règle actuellement car les membres du conseil de l'UFR Mathématique, Informatique, Mécanique et Automatique, conscients de ce manque, ont donné cette année des moyens supplémentaires pour l'achat d'une quinzaine de machines et les deux UFR ont alloué une salle pour y placer ces nouvelles machines. De plus, pour l'année prochaine, les UFR scientifiques ont l'intension de demander une subvention à la Région pour l'achat de machines à l'usage des étudiants des DEUG A et B. Par contre, le problème le plus crucial est le manque de locaux mais cette situation devrait être moins délicate lorsque les mécaniciens seront installés dans de nouveaux batiments (actuellement en construction).

Equipe pédagogique

Pour donner une image "sérieuse" de l'informatique il faut surtout des enseignants motivés qui s'investissent dans cet enseignement. Il ne faut donc pas cantonner des enseignants dans ce type d'enseignement ni les laisser livrer à eux-mêmes. En ce qui nous concerne, chaque enseignant-chercheur en poste a la responsabilité d'au moins un enseignement de DEUG et d'un enseignement de second cycle. Chaque nouvel enseignant profite s'il le désire de l'acquis d'enseignants qui ont ou avaient la même charge. Chaque chargé du cours est également libre de la manière dont il veut faire passer son message, par contre, lorsqu'il s'agit de modifier le contenu, il doit en référer à l'ensemble des enseignants.

Cours-TD-salle machine

Nous pensons que les cours sont indispensables car nous avons à enseigner à nos étudiants, des concepts et des notions importants qui seront renforcés et durant les Travaux Dirigés et confortés devant la machine.

La totalité des cours d'informatique de DEUG A et B sont assurés par des enseignants-chercheurs informaticiens (24-01) de l'UFR Mathématique, Informatique, Mécanique et Automatique.

La totalité des TD sont assuré par des enseignants-chercheurs ou de chercheurs informaticiens.

Seul deux séances d'initiation dans la salle machine sont encadrées par les chargés de TD. Par contre, le reste du temps les étudiants ne sont pas livrés à eux-mêmes car:

les travaux de programmation sont demandés ou développés en TD
les projets sont bien "dégrossis" en TD
il existe un dialogue entre les étudiants et

- leurs enseignant (cours et TD)
- les personnels IATOS qui s'occupent du matériel du service informatique
- les moniteurs IPT (étudiants de licence d'informatique) qui sont mis à l'unique disposition des étudiants de DEUG
- les étudiants plus confirmés qu'eux qui sont légion dans les salles machines (DEUG, Licence et Maitrise)

Uniformité

Nous demandons une uniformité par section, pour cela chaque chargé du cours pour une section prend également en charge un groupe de TD pour s'assurer de la bonne harmonisation entre le cours et les TD et surtout pour préparer (seul ou avec les autres intervenants) les séances de TD pour que tous les étudiants d'une

même section abordent les mêmes exercices.

A2.1) DEUG1

A2.1.1) Contenu

Voir [Can90b] consacrée à l'initiation à l'algorithmique et à la programmation Pascal par l'exemple à l'usage des étudiants des DEUG scientifiques.

A2.1.2) Progression

Les volumes horaires que nous donnons sont ceux alloués à tous les étudiants de DEUG A des UFR scientifiques de l'Université de Metz.

A2.1.2.1) Le premier semestre

Le premier semestre des DEUG A et B dure 8 semaines. Lors de ce semestre nous préconisons l'enseignement de l'algorithmique de base ainsi qu'une initiation à l'utilisation de la machine. Nous conseillons donc la progression suivante:

Cours 1 (2h)
Introduction et présentation (chapitre 0 et 1 de [Can90b])
TD1 (2x2h)
1) (coursTD) Les expressions (chapitre 2 de [Can90b])
2) exercice sur les expressions

Cours 2 (2h)
L'affectation et les notions de base (chapitre 3 de [Can90b])
TD2 (2x2h)
1) Exercices sur la composition séquentielle et simulation d'exécution d'algorithme.
2) (coursTD) Présentation d'une machine (chapitre 4 de [Can90b])

Cours 3 (2h)
Dialogue, les entrées-sorties (chapitre 5 de [Can90b])
TD3 (2x2h)
1) (coursTD) Fin de la présentation d'une machine (chapitre 4 de [Can90b]) et exercices sur les entrées-sorties.
2) TD sur machine: utilisation des commandes du système et de

l'éditeur, compilation et exécution d'un programme écrit dans un TD précédent.

Cours 4 (2h)

Rudiment de logique (chapitre 2 de l'annexe 1) et les conditionnelles (chapitre 6 de [Can90b]).

TD 4 (2x2h)

1) Exercice de logique (table de vérité) et exercice sur les conditionnelles (recherche de condition). Ecriture d'algorithmes utilisant des conditionnelles.

2) Ecriture d'algorithmes utilisant les conditionnelles.

Cours 5 (2h)

Itération: Les boucles pour (chapitre 7 de [Can90b])

TD 5 (2x2h)

1) Exécution manuelle d'algorithme avec des boucles, exercices sur les suites récurrentes.

2) Correction d'algorithmes avec des boucles (récurrence) et écriture d'algorithmes.

Cours 6 (2h)

Itération: La boucle tant que (chapitre 8 de [Can90b])

TD 6 (2x2h)

1) Exercice sur les boucles tant que.

2) TD en salle machine, écriture et exécution de programmes.

Cours 7 (2h)

Variable indicée: Le type tableau (chapitre 9 de [Can90b])

TD 7 (2x2h)

1) Exercices sur les vecteurs et les matrices.

2) Les combinaisons

Cours 8 (2h)

Résolution d'un problème, qui utilise le plus de notions apprises, jusqu'à l'écriture d'un programme. Faire apparaitre le besoin d'utiliser la décomposition. Présentation des autres domaines de l'informatique

TD 8 (2x2h)

1) Les tableaux surdimentionnés.

2) Codage des polynomes (chapitre 17 de [Can90b]).

A2.1.2.2) Le deuxième semestre

A part les étudiants de DEUG A à dominante Mathématique et Informatique qui ont 12 semaines de cours et 18 semaines de TD car ils développent un grand projet, tous nos étudiants des DEUG A et B ont dix semaines de cours et de TD. Lors de ces dix séances nous

préconisons le déroulement suivant:

Cours 1 (2h)
Programmation structurée: Les fonctions: leurs définitions (chapitre 10 de [Can90b]).
TD 1 (2h)
Ecriture des fonctions du programme second degré, ainsi que d'autres (chapitre 10 de [Can90b]).

Cours 2 (2h)
Programmation structurée: Les fonctions: leurs utilisations et leurs évaluations (chapitre 10 de [Can90b]). Limitation des fonctions et besoin de décomposer.
TD 2 (2h)
Exécution de deux fonctions (expression simple et non simple). Définition d'autres fonctions.

Cours 3 (2h)
Programmation structurée: Les procédures: leurs définitions, leurs exécutions, passage des paramètres, portée des identificateurs (chapitre 11 de [Can90b]).
TD 3 (2h)
Exécutions de procédures, Définitions de procédures.

Cours 4 (2h)
Une décomposition: le jeu du Mastermind (chapitre 12 de [Can90b])
TD 4 (2h)
Analyse et éciture des modules qui dépendent du type décomposition, sensibilisation aux besoins des propriétés. Le programme entier sera donné aux étudiants).

Cours 5 (2h)
Le calcul des propriétés (chapitre 13 de [Can90b])
TD 5 (2h)
Algorithme de tri (bulle) à l'aide des propriétés (chapitre 13 de [Can90b]).

Cours 6 (2h)
Le calcul des propriétés (suite), schémas de parcours.
TD 6 (2h)
Les différentes égalités de tableaux. Ecriture de l'algorithme qui calcule si un carré est magique ou non.

Cours 7 (2h)
Le type enregistrement (chapitre 14 de [Can90b]).
TD 7 (2h)

Le type étudiant et quelques algorithmes de la gestion de l'UFR.

Cours 8 (2h)
Les fichiers (chapitre 15 de [Can90b]).
TD 8 (2h)
Le panachage et la fusion de deux fichiers.

Cours 9 (2h)
Graphisme (chapitre 16 de [Can90b]) et présentation des commande du graphisme spécifique au langage que l'on utilise. Analyse du programme du jeu de tennis (programme distribué aux étudiants).
TD 9 (2h)
Programme qui simule à l'écran une boule qui roule sur une droite puis tombe et rebomdi sur le sol pour remonter de moitier ect ...

Cours 10 (2h)
Codages et algorithmes de codages des entiers (chapitre 17 de [Can90b]).
TD 10 (2h)
D'autres algoritmes de codage.

A2.2) DEUG2

A2.2.1) Contenu

Voir [Can90c] consacrée à l'initiation à la récursivité et à la programmation récursive à l'usage des étudiants des DEUG scientifiques.

A2.2.2) Progression

Tous les étudiants des DEUG A ont un cours d'informatique sur 14 semaines, lors du premier semestre.

Cours 1 (1h30)
Théorème de récurrence et son utilisation en algorithmique (chapitre 1 de [Can90c]). Les Fonctions exponentielles, exécution, correction et cout) (chapitre 2 de [Can90c]).
TD 1 (2h30)
Exercice de démonstration par récurrence ($2^n > n$, $2^n > n^2$, la somme des n premiers nombres impairs, $10^n - 1 = 0 [9]$ ...).

Cours 2 (1h30)

Fonctions récursives suite: les combinaisons, quotient et reste de la division euclidienne (chapitre 2 de [Can90c]).

TD 2 (2h30)

Ecritures récursives, fonctions Pascal asociées et exécution des fonctions suivantes: a+b, a*b, pgcd(a,b).

Cours 3 (1h30)

Procédures récursives: besoin, exécution (quotient et reste, hanoi ...) (chapitre 3 de [Can90c]).

TD 3 (2h30)

Fibonacci, divise(a,b) en fonction de divise(a,2b), hanoi avec les deux jeux de n disques (chapitre 10, à finir chez eux).

Cours 4 (1h30)

Classification, schéma, arrêt et comportement des algorithmes récursifs (chapitre 4 de [Can90c]).

TD 4 (1h30)

Exercices sur les affichages de nombre (non donné en TD)

- qu'affiche cette procédure?
- quelle procédure affiche cette suite de nombre?

Etude des schémas:

if b(x) then if c(x) then $\varphi_1(x)$;p(g1(x))
else $\varphi_2(x)$;p(g1(x))
{else $\varphi_0(x)$}

et

if b(x) then if c(x) then $\varphi_1(x)$;p(g1(x));$\psi_1(x)$
else $\varphi_2(x)$;p(g1(x));$\psi_2(x)$
{else $\varphi_0(x)$}

Cours 5 (1h30)

Récursivité sur les ensembles (chapitre 5 de [Can90c])

TD 5 (2h30)

Ecriture des fonctions qui calcule le minimum de n valeurs, qui calcule le pgcd de n valeurs, le nombre d'occurrence d'une valeurs parmi n valeur, l'indice du minimum, l'indice de la place d'une valeur (pour le tri par insertion).

Cours 6 (1h30)

Elimination de la récursivité sans pile (chapitre 7 de [Can90c]). Règles 1 à 6.

TD 6 (1h30)

Utilisation des régles sur des exemples dont certain avec optimisation (exp). Exemples avec des données de type tableau recherche séquentielle et dichotomique d'une valeur dans un tableau. Recherche de règles pour les schémas donnés en étude lors du TD 4 (cas g1=g2 et g1≠g2).

Cours 7 (1h30)

Elimination de la récursivité à l'aide d'une pile. Besoin et description fonctionnelle de la pile. Réalisation des fonctions de base sur les piles. Règles 7 à 9.

TD 7 (2h30)

Règles pour le schéma récursif non terminal du TD précédent avec g1≠g2. La fonction x^n en fonction de $x^{n/2}$. Un exemple avec un appel avec n/2. etc...

Cours 8 (1h30)

Elimination de la récursivité: Règle 10. Initiation aux étiquettes. Application à la procédure hanoi.

TD 8 (2h30)

La fonction d'Ackermann. Un exemple de procédures avec plusieurs cas et plusieurs suite. La fonction Fibonacci.

Cours 9 (1h30)

Les Tris séquentiels. Introduction, notion de cout d'un tri. Principe, algorithme et cout du tri par insertion les différentes variantes (chapitre 9 de [Can90c]).

TD 9 (2h30)

Une exécution. Analyse et écriture de la procédure décale. L'élimination de la récursivité pour la procédure de tri par insertion. Le principe, l'algorithme et le cout du tri par sélection.

Cours 10 (1h30)

Les tris dichotomiques principe et algorithme: fusion et quicksort.

TD 10 (2h30)

Le tri Von Neumann. Cout de l'algorithme de tri par fusion. Présentation du radixsort.

Cours 11 (1h30)

Les listes linéaires: l'approche fonctionnelle. Définition récursive du type liste et des fonctions de base. Utilisation abstraite de ces listes sur des exemples.

TD 11 (1h30)

Analyse abstraite et écriture des fonctions Pascal associées aux fonctions suivantes: les accès, les insertions, les suppressions, la concaténation, la fusion. Les procédures d'affichage d'une liste et les fonctions d'initialisation par lecture.

Cours 12 (1h30)

Réalisation des fonctions de base. Notion de tas et d'allocation mémoire. Variable de type pointeur et leur utilisation dans les paramètres des procédures.

TD 12 (2h30)

Utilisation abstraite des listes. Analyse et écriture de la fonction de tri selon le principe du quicksort. Les listes d'entiers la fonction atoi, iota, atoisansp et iotasansp.

Cours 13 (1h30)

L'approche procédurale. Transformation des fonction en procédure pour faire des opérations physiques sur les listes (insertion, suppression, utilisation de place déjà existante).

TD 13 (2h30)

La procédure qui insère une valeur en queue, la procédure qui concatène deux listes. La procédure tri par insertion. La procédure renverse (chapitre 13 de [Can90c])

Cours 14 (1h30)

Principe d'induction sur les listes (Enoncé et un exemple d'utilisation). Description des autres listes: les files (avec une queue pour inserer), en anneau, etc .. . Définition des listes de listes, application à des ensembles d'ensembles.

TD 14 (2h30)

C'est un coursTD où nous avons initié les étudiants à la récursivité sur les chaines de caractères (lecture) (chapitre 6 de [Can90c]). Structure récursive du fichier d'entrée. Reconnaissance des langages L1, L2, L3, L4, L5. Technique de buffer (création d'une fenêtre de lecture) et lectures des entiers.

L'UFR Mathématique, Informatique, Mécanique et Automatique propose également des cours complémentaires aux étudiants de deuxième année de DEUG

- à dominante Mathématique et Informatique dont le programme actuel est le suivant:
  * arbre, graphe, automate et grammaire
- à dominante Technologie et Mécanique dont le programme actuel est le suivant:
  * comprendre les concepts de base de la C.A.O. en utilisant le logiciel de C.A.O. SACADO (Système Adaptatif de Conception Assisté et de Développement par Ordinateur) développé par l'équipe du professeur Y. Gardan du Laboratoire de recherche en informatique de Metz.

A2.3) Une évaluation sur machine

Nous avons choisi, pour les raisons déjà évoquées, de ne pas faire d'évaluation sur machine.

Par contre tous les étudiants des DEUG scientifiques doivent réaliser un projet sur machine. La plupart du temps ces projets

sont bien "dégrossi" en TD pour que les étudiant ne se trouvent pas pris au dépourvu seul devant une machine. Ce projet est pris en compte dans le calcul de la moyenne.

A2.4) Notation

Pour chaque semestre les étudiants ont un ou deux contrôles écrits et un projet à rendre (sauf pour les huit premières semaines). Le calcul de la moyenne de l'étudiant est calculé des manières suivantes:

- s'il n'y a pas de projet c'est la moyenne des notes d'écrits.
- s'il y a un projet nous pouvons adopter plusieurs stratégies:
  1) la note de projet compte comme une note d'écrit.
  2) si le projet n'est pas rendu la note de projet compte comme une note d'écrit donc 0. Par contre si le projet est rendu on peut ajouter des points de 0 à 2 sur la moyenne de l'écrit.

Nous avons personnelement choisi la deuxième stratégie pour les raisons suivantes:

- cela donne plus d'importance à l'écrit, comme cela les étudiants s'investissent plus en TD ou chez eux pour préparer leurs contrôles. Il ne passe pas tout leur temps dans les salles machines.
- cela minimise les tricheries car comme le projet est en général fait en binôme et en dehors des heures encadrées même si ce projet est bien "dégrossi" et suivi en TD.