Livret - 4

Logiciel et algorithme

Conception descendante, algorithme, trace, facteur de qualité.

Outil utilisé : C#

RM di scala

Cours informatique programmation

Rm di Scala - http://www.discala.net

4: développement méthodique du Logiciel

Plan du chapitre:

Introduction

Le bon sens est la chose du monde la mieux partagée...la diversité de nos opinions ne vient pas de ce que les uns sont plus raisonnables que les autres, mais seulement de ce que nous conduisons nos pensées par diverses voies, et ne considérons pas les mêmes choses. Car ce n'est pas assez d'avoir l'esprit bon, mais le principal est de l'appliquer bien.
R Descartes Discours de la méthode, première partie, 1637.

Le développement méthodique d'un logiciel passe actuellement par une démarche de " descente concrète " de la connaissance que l'humain a sur la problématique du sujet, vers l'action élémentaire exécutée par un ordinateur. Le travail du programmeur étant alors ramené à une traduction permanente des actions humaines en actions machines (décrites avec des outils différents).

Nous pouvons en première approximation différentier cette " descente concrète " en un classement selon quatre niveaux d'abstraction :

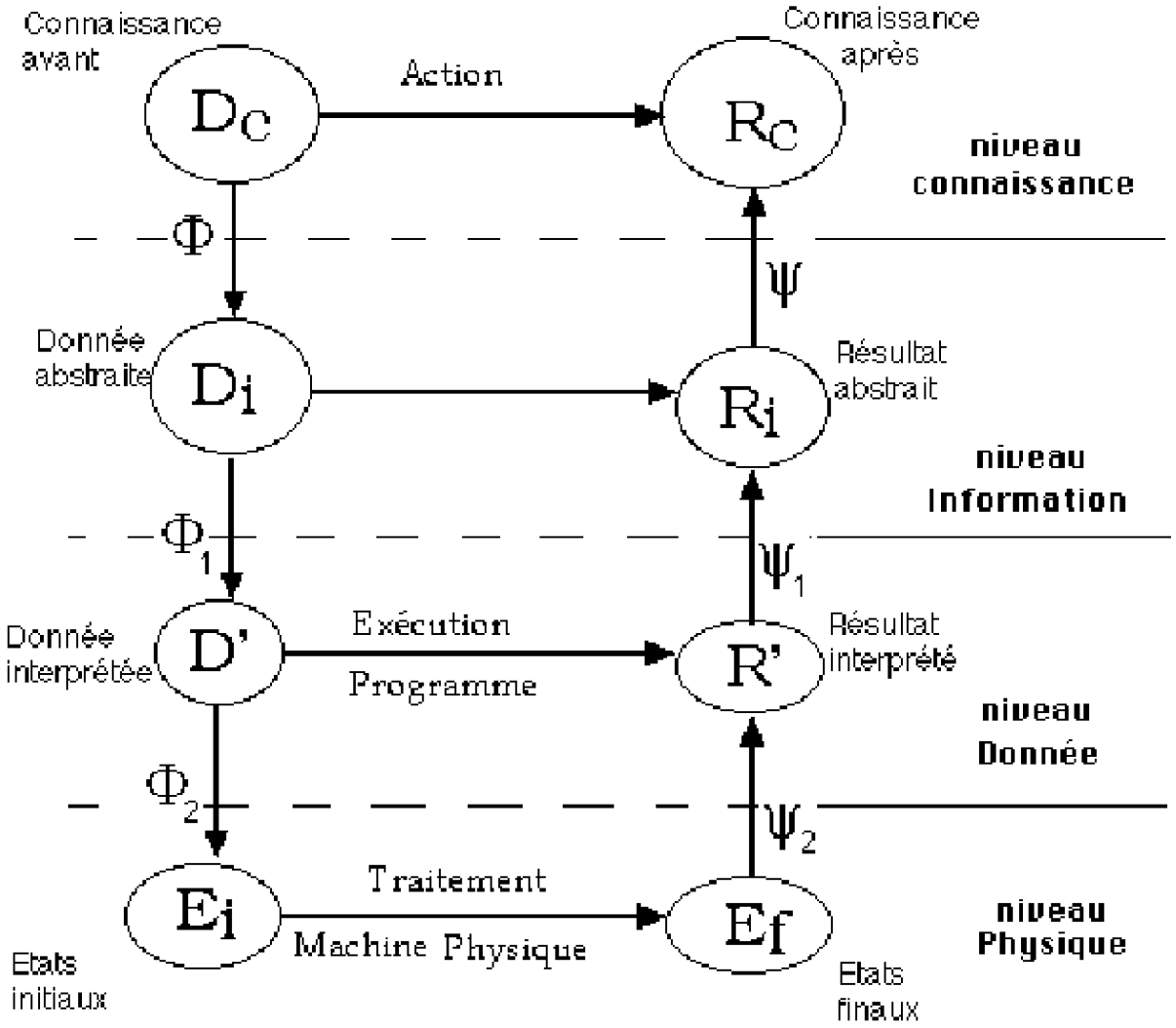


fig - schéma de descente concrète

Nous voyons que toute activité de programmation consiste à transformer un problème selon une descente graduelle de l'humain vers la machine. Ici nous avons résumé cette décomposition en 4 niveaux. La notion de programmation structurée est une réponse à ce type de décomposition graduelle d'un problème. L'algorithmique est la façon de décrire cette méthode de travail.

1. Production du logiciel

1.1 Génie logiciel

A une certaine époque, à ses débuts, l'activité d'écriture du logiciel ne reposait que sur l'efficacité personnelle du programmeur laissé pratiquement seul devant la programmation d'un problème.

De nos jours, le programmeur dispose d'outils et de méthodes lui permettant de concevoir et d'écrire des logiciels. Le terme **logiciel**, ne désigne pas seulement les programmes associés à telle application ou tel produit : il désigne en plus la documentation nécessaire à l'installation, à l'utilisation, au développement et à la maintenance de ce logiciel. Pour de gros systèmes, le temps de réalisation peut être aussi long que le temps du développement des programmes eux-mêmes.

Le **génie logiciel** concerne l'ensemble des méthodes et règles relatives à la production rationnelle des logiciels.

L'activité de développement du logiciel, vu les coûts qu'elle implique, est devenue une *activité économique* et doit donc être planifiée et soumise à des normes sinon à des attitudes équivalentes à celles que l'on a dans l'industrie pour n'importe quel produit.

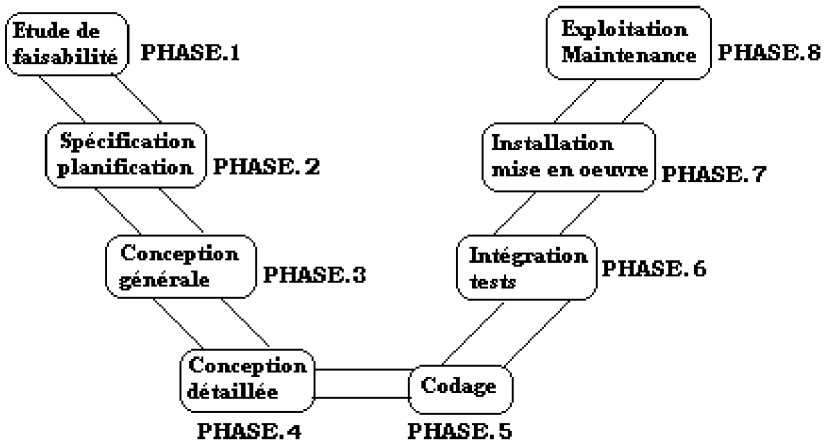
C'est pourquoi dans ce cours, le mot-clef est le mot **"composant logiciel"** qui tient à la fois de l'activité créatrice de l'humain et du composant industriel incluant une activité *disciplinée et ordonnée* basée pour certaines tâches sur des outils formalisés.

D'autre part le génie logiciel intervient lorsque le logiciel est trop grand pour que son développement puisse être confié à un seul individu ; ce qui n'est pas le cas pour des débutants, à qui il n'est pas confié l'élaboration de gros logiciels. Toutefois, il est possible de sensibiliser le lecteur débutant à l'habitude d'élaborer un logiciel d'une manière systématique et rationnelle à l'aide d'outils simples.

1.2 Cycle de vie du logiciel

Comme il faut un temps très important pour développer un grand système logiciel, et que d'autre part ce logiciel est prévu pour être utilisé pendant longtemps, on sépare fictivement des étapes distinctes dans ces périodes de développement et d'utilisation.

Le modèle dit de la cascade de Royce (1970) accepté par tout le monde informatique est un bon outil pour le débutant. S'il est utilisé pour de gros projets industriels, en supprimant les recettes et les validations en fin de chaque phase, nous disposons en initiation d'un cadre méthodologique. Il se présente alors sous forme de 8 diagrammes ou phases :



1.3 Maintenance d'un logiciel

Dans beaucoup de cas le coût du logiciel correspond à la majeure partie du coût total d'une application informatique. Dans ce coût du logiciel, la maintenance a elle-même une part prépondérante puisqu'elle est estimée de nos jours au minimum à **75% du coût total du logiciel**.

La maintenance est de trois sortes :

- adaptative (s'adapter à un nouvel environnement...)
- corrective (corrections d'erreurs...)
- perfective (améliorations demandées par le client...)

1.4 Production industrielle du logiciel

La production du logiciel étant devenue une activité industrielle et donc économique, elle n'échappe pas aux données économiques classiques. On répertorie un ensemble de caractéristiques associées à un projet de développement, chaque caractéristique se voyant attribuer un ratio de productivité.

Le ratio de productivité d'une caractéristique
C'est le rapport entre la productivité (exprimée en nombre d'Instructions Sources Livrées, par homme et par mois) d'un projet exploitant au mieux cette caractéristique, et la productivité d'un projet n'exploitant pas du tout cette caractéristique.

Le tableau suivant est tiré d'une étude de B.Boehm (Revue TSI 1982 : les facteurs de coût du logiciel):

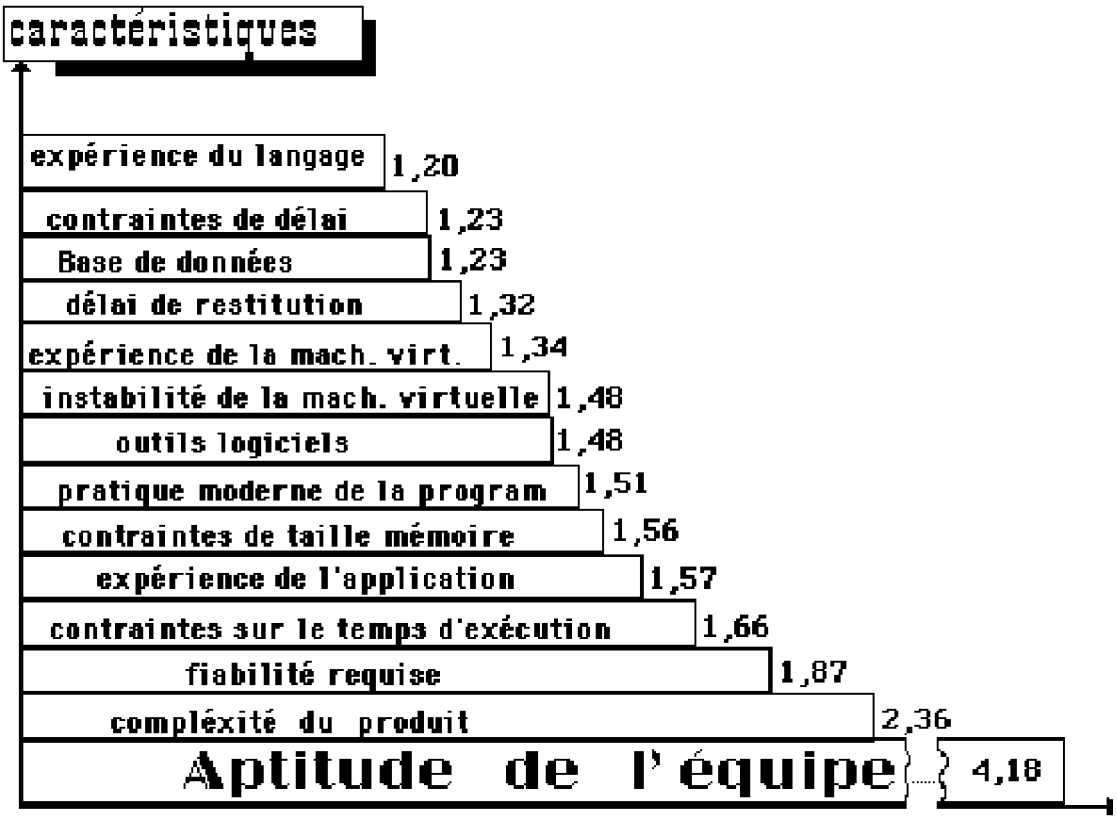


Tableau comparatif des divers ratios de productivité (B.Boehm)

Vous aurez remarqué en observant le graphique précédent que le facteur le plus important n'est pas l'expérience d'un langage (erreur commise par les néophytes). Ce qui explique entre autres arguments que l'enseignement de la programmation ne soit pas l'enseignement d'un langage.

Il apparaît que le facteur le plus coûteux reste un facteur sur lequel la technologie n'a aucune prise : l'aptitude qu'ont des individus à communiquer entre eux !

Pour l'élaboration d'un logiciel, nous allons utiliser deux démarches classiques : la méthode structurée ou algorithmique et plus tard une extension orientée objet de cette démarche.

2. Conception structurée descendante

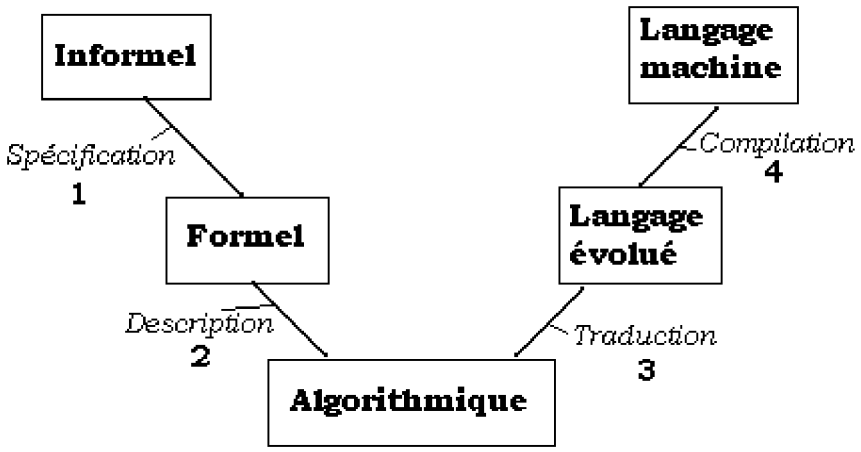
2.1 Critère simple d'automatisation

Un problème est **automatisable** (traitable par informatique) si :

- l'on peut parfaitement définir les données et les résultats,
- l'on peut décomposer le passage de ces données vers ces résultats en une suite d'opérations élémentaires dont chacune peut être exécutée par une machine.

Dans le cadre d'une initiation à la programmation, dans le cycle de vie déjà présenté plus haut, nous ne considérerons que les phases 2 à 6, en supposant que la faisabilité est acquise, et qu'enfin les phases de mise en œuvre et de maintenance sont mises à part.

Dans cette perspective, le schéma de la programmation d'un problème se réduit à 4 phases :



- La phase 1 de spécification utilisera les types abstraits de données (TAD),
- la phase 2 (correspondant aux phases 3 et 4 du cycle de vie) utilisera la méthode de programmation algorithmique,
- la phase 3 (correspondant à la phases 5 du cycle de vie) utilisera un traducteur manuel pascal,
- la phase 4 (correspondant à la phases 6 du cycle de vie) correspondra au passage sur la machine avec vérification et jeux de tests.

Nous utiliserons un " langage algorithmique " pour la description d'un algorithme résolvant un problème. Il s'agit d'un outil textuel permettant de passer de la conception humaine à la conception machine d'une manière souple pour le programmeur.
Nous pouvons résumer dans le tableau ci-dessous les étapes de travail et les outils conceptuels à utiliser lors d'une telle démarche.

ETAPES PRATIQUES	Matériel et moyens techniques à disposition
Analyse	Papier, Crayon, Intelligence, Habitude.
Mise en forme de l'algorithme	C'est l'aboutissement de l'analyse, esprit logique et rationnel.
Description	Utilisation pratique des outils d'une méthode de programmation, ici la programmation structurée.
Traduction	Transfert des écritures algorithmiques en langage de programmation.
Tests et mise au point	Mise au point du programme sur des valeurs tests ou à partir de programmes spécialisés.
Exécution	Phase finale : le programme s'exécute sans erreur.

2.2 Analyse méthodique descendante

Le second [précept], de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait et qu'il serait requis pour les mieux résoudre.
R Descartes Discours de la méthode, seconde partie, 1637.

Définir le problème à résoudre:
expliciter les données
  préciser: leur nature
  leur domaine de variation
  leurs propriétés
expliciter les résultats
  préciser: leur structure
  leur relations avec les données
fin définir;
Décomposer le problème en sous-problèmes**;**
Pour chaque sous-problèmes identifié faire
  si solution évidente **alors** écrire le morceau de programme
  sinon appliquer la méthode au sous-problème
  fsi
fpour.

démarche proposée par J.Arsac

Cette démarche méthodique a l'avantage de permettre d'isoler les erreurs lorsqu'on en commet, et elles devraient être plus rares qu'en programmation empirique (anciens organigrammes).

Il apparaît donc plusieurs niveaux de décomposition du problème (niveaux d'abstraction descendants). Ces niveaux permettent d'avoir une description de plus en plus détaillée du problème et donc de se rapprocher par raffinements successifs d'une description prête à la traduction en instructions de l'ordinateur.

Afin de pouvoir décrire la décomposition d'un problème à chaque niveau, nous avons utilisé un langage algorithmique (et non pas un langage de programmation) qui emprunte beaucoup au langage naturel (le français pour nous).

2.3 Analyse ascendante

Le troisième [précept], de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu, comme par degrés, jusqu'à la connaissance des plus composés; et supposant même de l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous essaierons de partir de l’existant (les fichiers sources déjà écrits sur le même sujet) et de reconstruire par étapes la solution. Le problème dans cette méthode est d’assurer une bonne cohérence lorsque l’on rassemble les morceaux.

Les méthodes objets que nous aborderons plus loin, sont un bon exemple de cette démarche. Nous n'en dirons pas plus dans ce paragraphe en renvoyant le lecteur intéressé au chapitre de la programmation orientée objet de cours.

2.4 Programmation descendante avec retour sur un niveau

Comme partout ailleurs, une attitude appuyée sur les deux démarches est le gage d'une certaine souplesse dans le travail. Nous adopterons une démarche d'analyse essentiellement descendante, avec la possibilité de remonter en arrière dès que le développement paraît trop complexe.

Nous adopterons dans tout le reste du chapitre une telle méthode descendante (avec quelques retours ascendants). Nous la dénommerons " programmation algorithmique ".

Nous utilisons les concepts de **B.Meyer** pour décomposer un problème en niveaux logiques puis en raffinant successivement les différentes étapes.

2.5 Machines abstraites et niveaux logiques

Principe :
On décompose chacune des étapes du travail en niveaux d'abstractions logiques. On suppose en outre qu'à chaque niveau logique fixé, il existe une machine abstraite virtuelle capable de comprendre et d'exécuter la description du problème sous la forme algorithmique en cours. Ainsi, en descendant de l'abstraction vers le concret, on passe graduellement d'un énoncé de problème au niveau humain à un énoncé du même problème à un niveau où la machine devient capable de l'exécuter.

Niveau logique	Machine abstraite	Enoncé du problème en
0	M_0 = l'humain	A_0 = langage naturel
1	M_1 = mach. Abstraite	A_1 = lang.algorithmique
. . .	. . .	. . .
n	M_n = machine+OS	A_n = langage évolué
n+1	M_{n+1}= machine physique	A_{n+1}= langage binaire

A partir de cette décomposition on construit un " arbre " de programmation représentant graphiquement les hiérarchies des machines abstraites.

Voici un exemple d'utilisation de cette démarche dans le cas de la résolution générale de l'équation du second degré dans **R**.

Le problème se décompose en deux sous-problèmes :

- le premier concerne la résolution d'une équation du premier degré strict
- le second est relatif à la "résolution d'une équation du second degré strict".

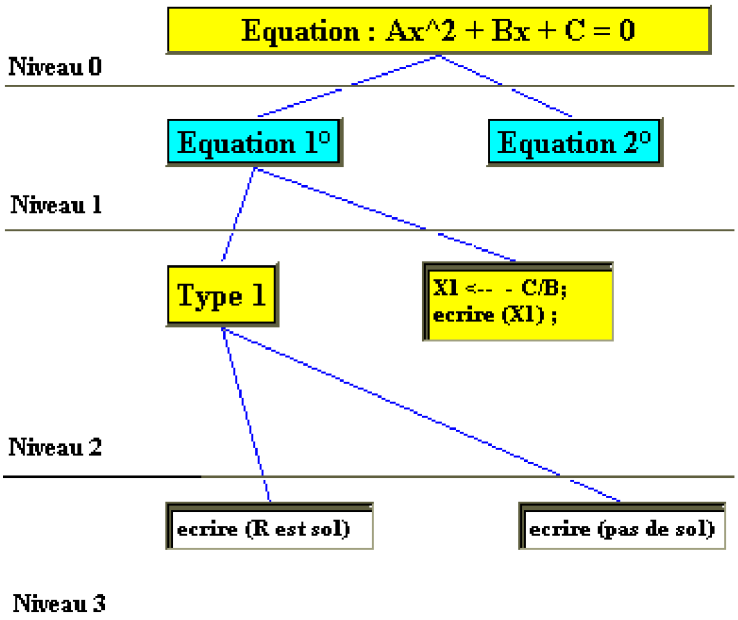


figure de la branche d'arbre 1er degré

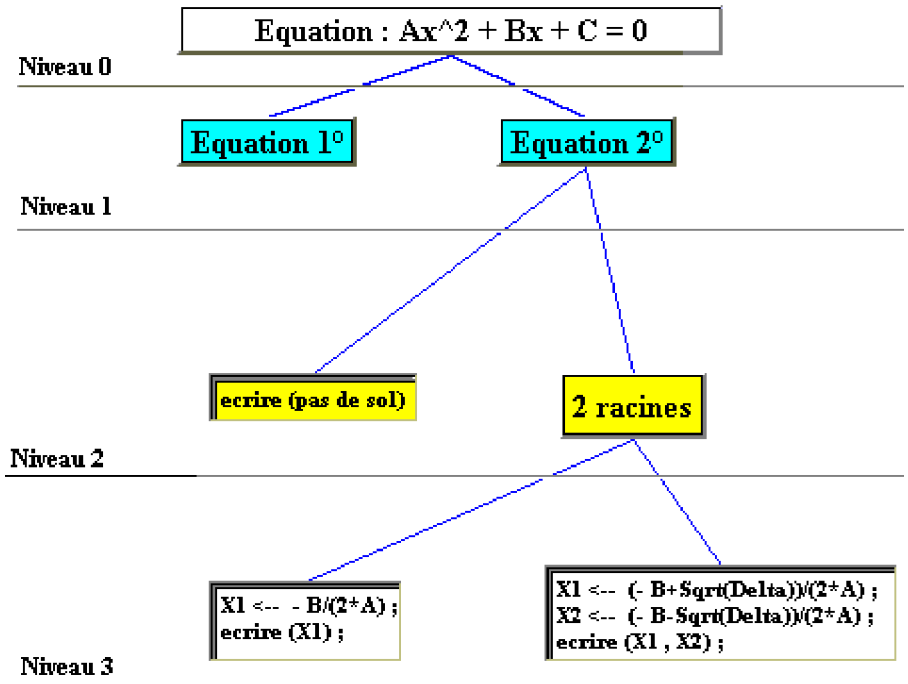


figure de la branche d'arbre 2ème degré

Nous avons utilisé comme langage de description des étapes intermédiaires un langage algorithmique basé sur des mots du français. Nous le détaillerons plus tard.

3. Notion d'ALGORITHME

Définition ***(D.E. Knuth)***

Un algorithme est un ensemble de règles qui décrivent une séquence d'opérations en vue de résoudre un problème donné bien spécifié. Un algorithme doit répondre aux 5 caractéristiques suivantes :

- La finitude
- La précision
- Le domaine des entrées
- Le domaine des sorties
- L'exécutabilité

Notons qu'un algorithme exprime donc un procédé séquentiel (or dans la vie courante tout n'est pas nécessairement séquentiel comme par exemple écouter un enseignement et penser aux prochaines vacances), et ne travaille que sur des problèmes déjà transformés de la phase 1 à la phase 2 (la spécification). I

Il n'est pas demandé aux débutants de travailler sur cette étape du processus. C'est pourquoi la plupart des exercices de débutant sont déjà spécifiés dans l'énoncé, ou bien leur spécification est triviale.

Indiquons les éléments de définition des cinq autres caractéristiques demandées à un algorithme :

- **Finitude** : Le nombre d'étapes d'un algorithme doit être fini. Le temps d'exécution pourra être évalué.
- **Précision** : Chaque étape doit être parfaitement définie. Toutes les actions élémentaires doivent être connues.
- **Domaine des entrées** : Le champ des données d'entrée doit être spécifié.
- **Domaine des sorties** : Un algorithme ayant un résultat, il faut donner les champs correspondants aux résultats de sortie, ou du moins les relations entre les données d'entrée et les données de sortie.
- **Exécutabilité** : Un algorithme doit déboucher sur un programme exécutable en un temps fini et raisonnable.

Environnement

On appelle environnement d'un algorithme l'ensemble des entités utilisés par le processeur pendant le déroulement de l'algorithme.

Nous allons définir un langage de description des algorithmes qui nous permettra de décrire les arbres de programmation et le fonctionnement des machines abstraites de la programmation structurée.

Voici classiquement ce que tous les auteurs utilisent comme système de description d'un algorithme lorsqu'ils le font avec un langage. *Les deux sous-paragraphes qui suivent, fournissent les définitions des éléments fondamentaux d'un tel langage algorithmique, le paragraphe d'après construit un langage algorithmique fondé sur ces éléments fondamentaux.*

Nous verrons que l'algorithmique est par nature plus proche de l'étudiant que la machine. En effet dans la suite du cours, l'étudiant s'apercevra par exemple, que les nombres rationnels ne sont pas représentables simplement en machine, encore moins les nombres réels. Les langages d'implémentations impératifs comme Pascal, Java, C# etc… étant relativement pauvres à cet égard.

L'étudiant ne doit pas croire que l'informatique s'est résignée à ne travailler que sur les entiers et les décimaux, mais plutôt se rendre compte qu'il existe une palette importante de certains produits informatiques qui traitent plus ou moins efficacement les insuffisances des langages classiques par exemple vis à vis des rationnels (les systèmes de calcul formel comme MAPLE (étudié en Taupe),MATHEMATICA,... sont une réponse à ce genre d'insuffisance). Nous ne nous préoccupons absolument pas, dans un premier temps en algorithmique, ni de la vérification, ni du contrôle, ni des restrictions d'implantation des données. Notre préoccupation première est d'écrire des algorithmes justes qui fonctionnent sur des données justes.

3.1 Objets de base d'un langage algorithmique

Contenant

Nous appelons **contenant** toute cellule mémoire d'une machine abstraite d'un niveau fixé.

Contenu

Nous appelons **contenu** l'information représentée par l'état du contenant.

Atomes

Pour un contenant fixé on note A l'ensemble de tous ses états possibles, on dit aussi ensemble des **atomes** du niveau n (niveau du contenant).

Remarques :

a) un atome de niveau n est donc un état possible d'un contenant,
b) pour un niveau logique fixé, il y a un nombre d'atomes fini,
c) lorsque l'on est au niveau machine :

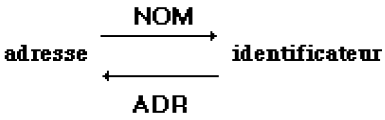
- le contenant est à p positions binaires (p est le nombre de bits du mot, p>1).
- A ={0,1}x....x {0,1} , p fois

Adresse fictive

Toute machine abstraite de niveau fixé dispose d'autant de cellules mémoires que nécessaire. Elles sont repérées par une **adresse fictive** (à laquelle nous n'avons pas accès).

Nom

Par définition, à toute adresse nous faisons correspondre bijectivement par l'opération **nom**, un identificateur unique définissant pour l'utilisateur la cellule mémoire repérée par cette adresse :



Nous définissons aussi un certain nombre de fonctions :

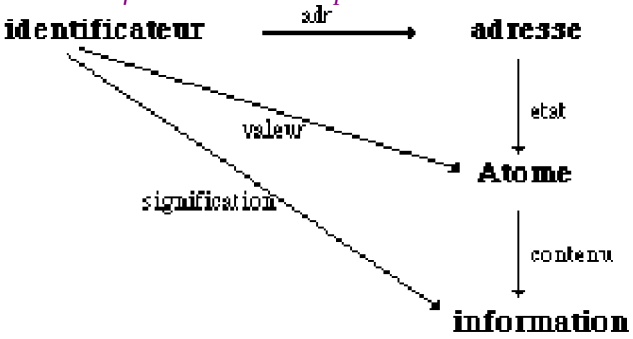
Etat : Adresse →Atome *(donne l'état associé à une adresse)*

valeur: identificateur →Atome *(donne l'état associé à un identificateur, on dit la valeur)*

contenu: Atome →information *(donne le contenu informationnel de l'atome)*

signification: identificateur →information *(sémantique de l'identificateur)*

Ces 4 fonctions sont liées par le schéma suivant :



3.2 Opérations sur les objets de base d'un langage algorithmique

Les parenthèses d'énoncé en LDFA seront algol-like : nous disposerons d'un marqueur du genre **debut** et d'un second du genre **fin** .

Exécutant ou processeur algorithmique

Nous appelons **exécutant** ou **processeur**, la partie de la machine abstraite capable de lire, réaliser, exécuter des opérations sur les atomes de cette machine, ceci à travers un langage approprié.

***Remarque**: l'opérateur formel **exécutant** dépend du temps.*

Instruction simple

C'est une instruction exécutable en un temps fini par un processeur et elle n'est pas décomposable en sous-tâches exécutables ou en autres instructions simples. Ceci est valable à *un niveau fixé*.

Instruction composée

C'est une instruction simple, ou bien elle est décomposable en une suite d'instructions entre parenthèses.

Composition séquentielle

Si i,j,...,t représentent des instructions simples ou composées, nous écrirons la composition séquentielle avec des " ; ". La suite d'instructions " i ; j; ..... ; t " est appelée une **suite d'instructions séquentielles**.

Schéma fonctionnel

C'est :

- soit un identificateur,
- soit un atome,
- soit une application **f** à n variables (où n>0):
  f : (identificateur)^n $\rightarrow$ identificateur

Espace d'exécution

L'espace d'exécution d'une instruction, c'est le n-uplet des n identificateurs ayant au moins une occurrence dans l'instruction (ceci à un niveau fixé).

Soit une instruction i_k, l'ensemble E_k des variables, ayant au moins une occurrence dans l'instruction i_k est noté : $E_k = \{x_1, x_2, ..., x_p\}$ (espace d'exécution de l'instruction i_k)

Environnement

C'est l'ensemble des objets et des structures nécessaires à l'exécution d'un travail donné pour un processeur fixé (niveau information).

Action

C'est l'opération ou le traitement déclenché par un événement qui modifie l'environnement (ou bien toute modification de l'environnement);

Action primitive

Pour un processeur donné (d'une machine abstraite d'un niveau fixé)une action est dite primitive, si l'énoncé de cette action est à lui seul suffisant pour que le processeur puisse l'exécuter sans autre éléments supplémentaires. Une action primitive est décrite par une *instruction simple* du processeur.

Action complexe

Pour un processeur donné (d'une machine abstraite d'un niveau fixé)une action complexe est une action **non-primitive**, qui est **décomposable** en actions primitives (à la fin de la phase de conception elle pourra être exprimée soit par un **module de traitement**, soit par une **instruction composée**).

Remarques :

- Ce qui est action primitive pour une machine abstraite de niveau n, peut devenir une action complexe pour une machine abstraite de niveau n+1, qui est l'expression de la précédente à un plus bas niveau (d'abstraction).
- Les instructions du langage doivent être les mêmes pour tous les niveaux de machine abstraite, sinon la programmation devient trop lourde à gérer.
- Tout langage de description de machine abstraite n'est pas implantable sur ordinateur (*au plus partiellement sinon ce serait tout simplement un langage de programmation*). Il ne peut servir qu'à décrire en partie la spécification et la conception. De plus il doit utiliser les idées de la programmation structurée descendante modulaire.

4. Un langage de description d'algorithme : LDFA

Avertissement

L'apprentissage d'un langage de programmation ne sert qu'aux phases 3 et 4 (traduction et exécution) et ne doit pas être confondu avec l'utilisation d'un langage algorithmique qui prépare le travail et n'est utilisé que comme plan de travail pour la phase de traduction. En utilisant la construction d'une maison comme analogie, il suffit de bien comprendre qu'avant de construire la maison, le chef de chantier a besoin du plan d'architecte de cette maison pour passer à la phase d'assemblage des matériaux ; il en est de même en programmation.

Enonçons un langage simple, extensible, qui est utilisé dans tout le reste du document et qui va servir à décrire les algorithmes. Nous le dénotons dans la suite du document comme **LDFA** pour **L**angage de **D**escription **F**ormel d'**A**lgorithme (terminologie non standard utilisée par l'auteur pour dénommer rapidement un langage algorithmique précis).

4.1 Atomes du LDFA

- Les ensembles de nombres comme **N**, **Z**, **Q**, **R** (les vrais ensembles classiques des mathématiques et leurs structures connues).
- La grammaire mathématique et celle du français.
- {**V**,**F**} comme éléments logiques (({**V**,**F**},¬ , ∧ , ∨) étant une algèbre de Boole)
- Les prédicats.
- Les caractères du français et les chaînes de caractères C des machines.

4.2 Information en LDFA

On rappelle qu'une information en LDFA est obtenue par le contenu d'un atome et se construit à l'aide de :

- la grammaire du français et le sens commun des mots,
- les théorèmes et les résultats obtenus des théories mathématiques (le sens étant le sens habituel donné à tous les symboles),
- toutes les manipulations générales (algorithmes en particulier) sur les structures de données.

4.3 Vocabulaire terminal du LDFA

V_T = **{** <—— , Ω , **lire( ) , ecrire( ) , si , tantque , alors , ftant , faire , fsi , sinon , sortirSi, pour , repeter , fpour , jusque , ; , entrée , sortie , Algorithme , local , global , principal , modules , specifications , types-abstraits , debut , fin , (,) , [,] , * , + , - , / ,** ¬ , ∧ , ∨ **}**

4.4 Instructions simples du LDFA :

Instruction vide

syntaxe : Ω

sémantique : ne rien faire pendant un temps de base du processeur.

Affectation

syntaxe : a <—— α
où : a∈ **identif, et** α est un schéma fonctionnel.

sémantique :
1) si α=identificateur alors val(a)=val(α)
2) si α est un atome alors val(a)=α
3) si α est une application / α : $(id_1,.....,id_p) \rightarrow \alpha(id_1,.....,id_p)$
alors val(a)= α'(val(id_1),.....,val(id_p))
où α' est l'interprétation de α sur l'ensemble des valeurs des val(id_k)

Lecture

syntaxe : **lire**(a) (où a ∈ **identif**)

sémantique : le contexte de la phrase précise où l'on lit pour "remplir" a, sinon on indique **lire**(a) **dans** .....

Elle permet d'attribuer une valeur à un objet en allant lire sur un périphérique d'entrée et elle range cette valeur dans l'objet.

Ecriture

syntaxe : **ecrire**(a) (où a ∈ **identif**)

sémantique : le contexte de la phrase précise où l'on écrit pour "voir" a, sinon on indique **ecrire**(a) **dans** .....

Ordonne au processeur d'écrire sur un périphérique (Ecran, Imprimante, Port, Fichier etc...)

Condition

syntaxe : **si** P **alors** E1 **sinon** E2 **fsi**
où P est un prédicat ou proposition fonctionnelle,
E1 et E2 sont deux instructions composées.

sémantique : classique de l'instruction conditionnelle, si le processeur n'est pas lié au temps on peut écrire :

si P **alors** E1 **sinon** Ω **fsi** = **si** P **alors** E1 **fsi**

Nous notons = la relation d'équivalence entre instructions. Il s'agit d'une équivalence sémantique, ce qui signifie que les deux instructions donnent les mêmes résultats sur le même environnement.

Boucle tantque

syntaxe : **tantque** P **faire** E **ftant**
où P est un prédicat et E une instruction composée)

sémantique :
tantque P **faire** E **ftant** = **si**P **alors** (E ; **tantque** P **faire** E **ftant**) **fsi**

Remarques :

Au sujet de la relation "=" qui est la notation pour l'équivalence sémantique en LDFA, on considère un "programme" LDFA non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E0, puis on évalue la modification de cet environnement que chaque action provoque sur lui:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow \ldots\ldots\ldots \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

où action n+1 : $\{E_n\} \rightarrow \{E_{n+1}\}$.

On obtient ainsi une suite d'informations sur l'environnement :$(E_0, E_1, \ldots, E_{k+1})$

Nous dirons alors que deux instructions (simples ou composées) sont sémantiquement équivalentes (notation =) si leurs actions associées sur le même environnement de départ provoquent la même modification.

A chaque instruction est associée une action sur l'environnement, c'est le résultat qui est le même (même état de l'environnement avant et après) :
Soient : Instr1 → action1 (*action associée à Instr1*),
Instr2 → action2 (*action associée à Instr2*),

Soient E et E' deux états de l'environnement,
si nous avons : **{E} action1 {E'}** et **{E} action2 {E'} ,**alors Instr1 et Instr2 sont sémantiquement équivalentes, nous le noterons :

Instr1 = Instr2

Boucle répéter

syntaxe : **repeter** E **jusqua** P
(où P est un prédicat et E une instruction composée)

sémantique :
repeter E **jusqua** P = E ; **tantque** **not** P **faire** E **ftant**

Exemple d'équivalence entre itérations:

tantque P **faire** E **ftant** = **si P alors (repeter E jusqua not** P) **fsi**
repeter E **jusqua** P = E ; **tantque not** P **faire E ftant** *(par définition)*

Boucle pour

syntaxe : **pour x** <-- a **jusqua b faire** E **fpour**
(où E est une instruction composée, x une variable, a et b des expressions dans un ensemble fini F totalement ordonné, la relation d'ordre étant notée ≤ , le successeur d'un élément x dans l'ensemble est noté Succ(x) et son prédécesseur pred(x))

sémantiques :
Cette boucle fonctionne à la fois en suivant automatiquement l'ordre croissant dans l'ensemble fini F ou en suivant automatiquement l'ordre décroissant, cela dépendra de la position respective de départ de la borne a et de la borne b. La variable x est appelée un indice de boucle.

sémantique dans le cas ordre croissant à partir du tantque :

```
x <-- a ;
tantque x ≤ succ(b)faire
  E ;
  x <-- succ(x) ;
ftant
```

sémantique dans le cas ordre décroissant à partir du tantque :

```
x <-- a ;
tantque x ≥ pred(b) faire
  E ;
  x <-- pred(x) ;
ftant
```

Exemple simple :

- **E = N** *(entiers naturels)* et la relation d'ordre : ≤ = *inférieur ou égal dans* **N**
- **pour** i ≤ x **jusqà** y **faire** R **FinPour**
  (ici i prendra toutes les valeurs successives dans **N** comprises entre x et y soient, x+y-1 valeurs et s'incrémentera de 1 à chaque fois)

Sortie de boucle

syntaxe : **SortirSi** P (où P est un prédicat ou une instruction vide)ne peut être utilisée qu'à l'intérieur d'une itération (tantque, répéter, pour).
sémantique : termine par anticipation et immédiatement l'exécution de la boucle dans laquelle l'instruction **SortirSi** se trouve.

Exemple récapitulatif complet

Reprenons l'exemple précédent de l'équation du second degré en décrivant dans l'arbre de programmation l'action de la machine abstraite de chaque niveau à l'aide d'instructions du langage algorithmique LDFA :

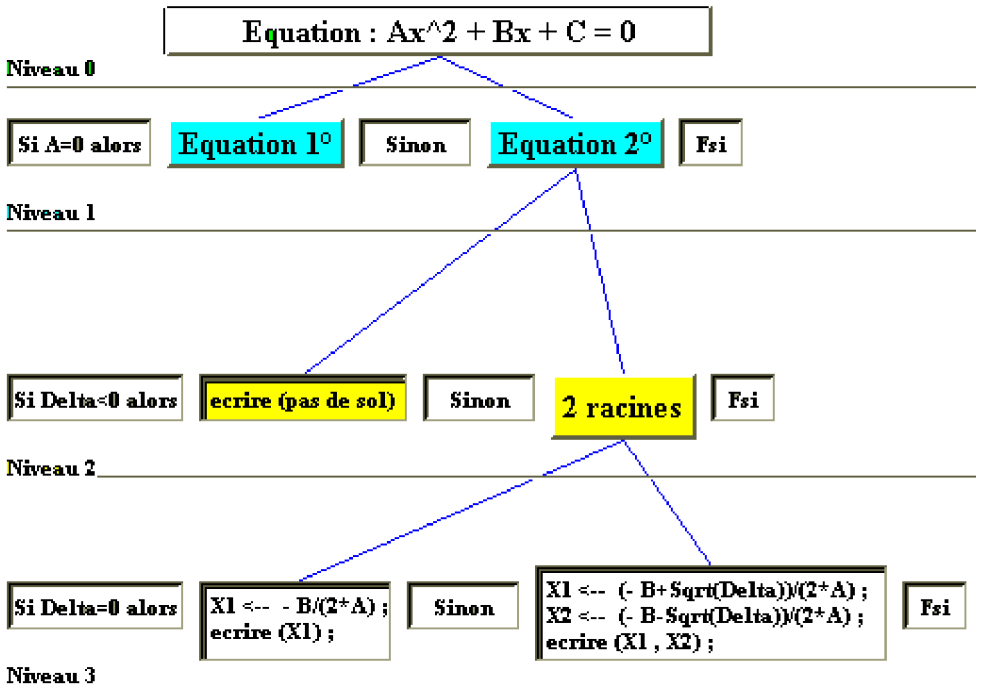


figure de la branche d'arbre 2ème degré

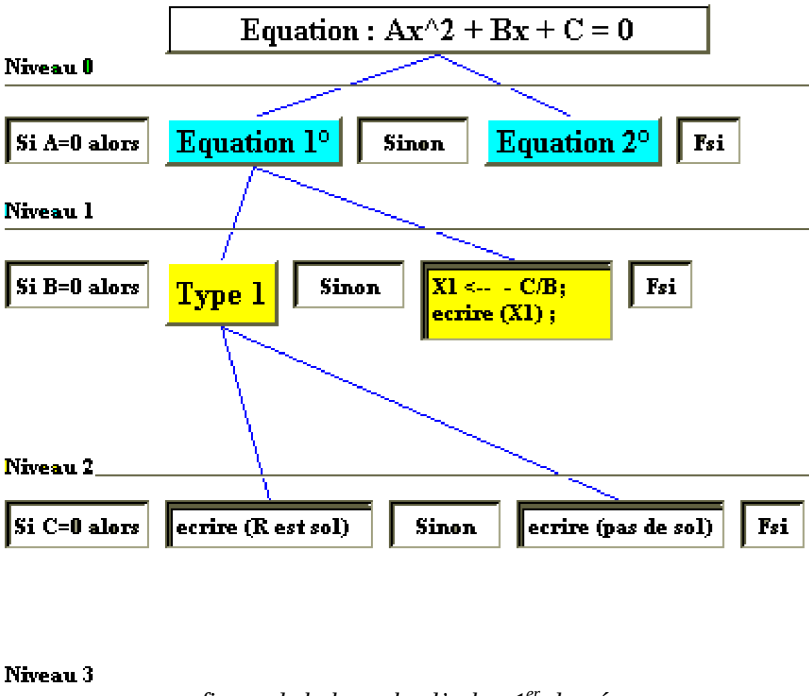


figure de la branche d'arbre 1er degré

Ecriture de l'algorithme

En relisant cet arbre selon un parcours en préordre *(il s'agit de parcourir l'arbre en partant de la racine et descendant toujours par le fils le plus à gauche, puis ensuite de passer au fils droit suivant etc…)*, l'on obtient après avoir complété l'algorithme une écriture linéaire comme suit :

```
Algorithme Equation
   Entrée: A,B,C ∈ R3
   Sortie: X1 ,X2 ∈ R2
   Local: Δ ∈ R
début
    lire(A,B,C);
    Si A=0 alors  { A=0 }
      Si B = 0 alors
        Si C = 0 alors
          écrire(R est solution)
        Sinon { C ≠ 0 }
          écrire(pas de solution)
        Fsi
      Sinon  { B ≠ 0 }
        X1 ← -C/B;
        écrire (X1)
      Fsi
    Sinon  { A ≠ 0 }
```

```
Sinon  { A ≠ 0 }
    Δ ← B² - 4*A*C ;
    Si Δ < 0 alors
          écrire(pas de solution)
    Sinon   { Δ ≥ 0 }
      Si Δ = 0 alors
         X1 ← -B / (2*A);
         écrire(X1)
      Sinon  { Δ > 0 }
         X1 ← (-B+sqrt(Δ)) / (2*A);
         X2 ← (-B-sqrt(Δ)) / (2*A);
         écrire( X1 , X2 )
      Fsi
    Fsi
  Fsi
FinEquation
```

Nous regroupons toutes les informations de conception dans un document que nous appelons le dossier de développement.

5. Le Dossier de développement

C'est un document dans lequel se trouvent consignés tous les éléments relatifs à la construction et à l'écriture de l'algorithme et du programme résolvant le problème cherché. Nous le divisons en 5 parties.

5.1 Enoncé et spécification

Enoncé du problème résolu par ce logiciel.

- **Spécifications opérationnelles** des abstractions de plus haut niveau du logiciel, en exprimant celles-ci à l'aide de types abstraits et de spécifications de plus bas niveau.
- **Spécifications des types abstraits de données** utilisés.
- **Spécifications d'interface** pour les abstractions de plus bas niveau.

On utilisera ces trois techniques de spécification de manière descendante, quitte à remonter corriger des spécifications de niveau plus haut lorsque des erreurs seront apparues dans une spécification de plus bas niveau. Ces spécifications sont destinées au niveau " concepteur de logiciel ", plutôt qu'à l'utilisateur. Cette partie rassemble les définitions abstraites des composants. Un utilisateur de base n'ayant à priori pas à consulter ce paragraphe, les termes employés seront les plus rigoureux possibles relativement à un formalisme éventuel.

Analyse des besoins :
son utilité principale est de fournir à l'utilisateur la description des services que lui rendra ce logiciel. Les termes utilisés doivent être compris par l'utilisateur.

5.2 Méthodologie

Dans ce paragraphe se situent tous les documents et les explications qui ont pu mener à la décision de résoudre le problème posé par la méthode que l'on a choisie. Le programmeur dispose ici de toute latitude pour s'exprimer à l'aide de texte en langue naturelle, de représentation graphique, d'outils ou de supports permettant au lecteur de se faire une idée précise du pourquoi des choix effectués.

5.3 Environnement

L'étudiant pourra présenter sous forme d'un tableau les principales informations concernant les données de son algorithme.

Exemple :

Nom	**genre**	**localisation**	**utilisation**
PHT	reel	Entrée	prix hors taxe
TVA	reel	local	TVA en %
PTTC	reel	sortie	Prix TTC

5.4 Algorithme en LDFA

Ici se situe la description de l'algorithme proposé pour résoudre le problème proposé. Il est obtenu entre autre à partir de l'arbre de programmation construit pendant l'analyse et la conception. Ci-dessous le modèle général d'un algorithme :

```
Algorithme XYZT;
    global :
    local :
    entrée :
    sortie :
    modules utilisés :
Spécifications : (TAD)
  Types Abstraits de Données utilisés
début
  ( corps d'algorithme en LDFA)
fin XYZT.
```

Nous verrons ailleurs ce que représentent les notions de TAD et de module.

5.5 Programme en langage de programmation (Pascal par exemple)

Dans ce paragraphe nous ferons figurer la " traduction " en langage de programmation de l'algorithme du paragraphe précédent.

6. Trace formelle d'un algorithme

Et le dernier [précept], de faire partout des dénombrements si entiers, et des revues si générales, que je fusse assuré de ne rien omettre.
R Descartes Discours de la méthode, seconde partie, 1637.

Nous proposons au débutant de vérifier l'exactitude de certaines parties de son algorithme en utilisant un petit outil permettant l'exécution formelle (c'est à dire sur des valeurs algébriques ou symboliques plutôt que numériques) de son algorithme. La trace numérique et les vérifications associées seront effectuées lors de l'exécution par la machine.

6.1 Espace d'exécution d'une instruction composée

On appelle espace d'exécution d'une séquence ou d'un bloc d'instructions $i_1 ... i_n$ l'ensemble $E = \bigcup_{k=1}^{n} E_k$ où E_k est l'espace d'exécution de l'instruction i_k.

Rappelons que l'on peut considérer un " programme " LDFA sous un autre point de vue : non pas comme une suite d'instructions, mais comme un environnement donné avec un état initial E_0 , puis on évalue la modification de cet environnement que chaque instruction provoque sur lui.

On considère les instructions i_k comme des transformateurs d'environnement E_n:
$\{E_0\} \rightarrow \{E_1\} \rightarrow \{E_2\} \rightarrow ..... \rightarrow \{E_k\} \rightarrow \{E_{k+1}\}$

L'instruction i_{n+1} fait alors passer l'environnement de l'état E_n à l'état E_{n+1}.
Nous écrirons ainsi : $i_{n+1} : \{E_n\} \rightarrow \{E_{n+1}\}$

Ces actions déterminent alors une suite d'états de l'environnement $(E_0, E_1, ...E_{k+1})$ que l'on peut observer.

C'est ce point de vue qui permet d'exécuter un suivi d'exécution symbolique d'un algorithme. Nous le nommerons " trace formelle ".

On adoptera pour une trace formelle une disposition en tableau de l'espace d'exécution comme suit :

Etats	V_1	V_2	.....	V_n
E_1	--	--		**y**
E_2	**x**	--		**y+1**

La colonne Etats représente donc les états successifs de l'environnement (ou espace d'exécution) figuré ici par les variables $V_1,V_2,...,V_n$. Les contenus des cellules du tableau sont les valeurs symboliques des variables au cours du déroulement de l'exécution. On peut considérer l'image mentale suivante de la trace formelle comme étant une succession de "photographies instantanées" de l'environnement prises après chaque instruction.

6.2 Exemple complet avec trace formelle

Nous traitons un exemple complet avec son dossier de développement et une trace formelle.

Enoncé

Calculer $S=\sum_{i=1}^{n} i$ sans utiliser de formule (*car l'on sait que S=(n+1)n/2*)

Spécification : flux d'information

En Entrée	**En Sortie**
Un nombre $n \in N^*$	Ecrire la somme voulue S.

Méthodologie

S est la somme des termes d'une suite récurrente :

$$s_i \begin{cases} s_0 = 0 \\ s_i = s_{i-1} + i \end{cases}$$

Environnement

Nom	**genre**	**localisation**	**utilisation**
N	Entier	Entrée	Nombre d'éléments à saisir
S	Entier	Sortie	Variable de cumul pour la somme
I	Entier	local	Gestion des boucles : compteur

Algorithme

Algorithme Somentier
| N ∈ **N***
| S , I ∈ **N**^{2}
|
Début *{Somentier}*
| *(E_0)*
| **Lire** (N) ;
| *(E_1)* ⟸
| S ← 0;
| *(E_2)* ⟸
| I ← 1;
| *(E_3)* ⟸
| **TantQue** I ≤ *? ? ?* **faire**
| | *(E_4)* ⟸
| | S ← S+I;
| | *(E_5)* ⟸
| | I ← I+1;
| | *(E_6)* ⟸
| **FinTant**;
| *(E_7)* ⟸
| Ecrire(S);
Fin Somentier

Ceci est un algorithme incomplet dans lequel on a déjà intercalé les états (E_n) entre les instructions. On ne sait pas exactement quel sera le test d'arrêt de la boucle (remplacé par ? ? ?), on sait seulement que c'est la valeur de la variable de compteur **I** qui le fournira.

Utilisation de la trace formelle

Nous allons montrer à l'aide de la trace formelle que cet algorithme fournit bien la somme des n premiers entiers dans la variable S, relativement aux préconditions { S = 0 et i = 1}.

Nous allons donc faire de la démonstration de programme :

Précondition	**Action**	**Postcondition**
{S= 0 et i= 1}	**Algorithme**	{ S = $\sum_{i=1}^{n} i$ }

Nous pouvons avoir une hésitation quant à la borne du test "**TantQue** I ≤ ??? ", faut-il s'arrêter à N, N-1 ou N+1 ?

Posons comme hypothèse que le test s'arrête à la valeur N, soit : "**TantQue** I ≤ N …

Exécutons manuellement et pas à pas l'algorithme précédent en supposant que le test d'arrêt n'est pas franchi, c'est à dire que l'on a I > N.

Voici le début des résultats de sa trace formelle dans le tableau ci-dessous :

Etats	I	N	S
E_0	-	-	--
E_1	-	**n**	--
E_2	-	**n**	**0**
E_3	**1**	**n**	**0**
$E_4=E_3$	**1**	**n**	**0**
E_5	**1**	**n**	**1**
E_6	**2**	**n**	**1**
$E_4=E_6$	**2**	**n**	**1**
E_5	**2**	**n**	**3**
E_6	**3**	**n**	**3**
$E_4 = E_6$ etc..	**3**	**n**	**3**

isolons les deux premiers " tours " de boucle :

$E_4 = E_6$	**2**	**n**	**1**
$E_4 = E_6$…	**3**	**n**	**3**

Nous voyons que juste avant la sortie de boucle (état E_6) au premier tour I=2 et S=1, au deuxième tour I=3 et S=3 .

Nous posons l'hypothèse de récurrence qu'au kième tour i=k+1 et S=$\sum_{i=1}^{k} i$ (somme des k premiers entiers). Nous allons utiliser l'exécution formelle pas à pas d'un tour de boucle afin de voir si après un tour de plus cette hypothèse se vérifie au rang k+1 :

Etats	I	N	S
.....	**...**	**...**	**...**
$E_4 = E_6$	**k+1**	**n**	**S=**$\sum_{i=1}^{k} i$
E_5	**k+1**	**n**	**S=**$\sum_{i=1}^{k} i$ **+ k+1**
E_6	**k+2**	**n**	**S=**$\sum_{i=1}^{k} i$ **+ k+1**

Or S=$\sum_{i=1}^{k} i$ +k+1 = $\sum_{i=1}^{k+1} i$ (la somme des k+1 premiers entiers).

Nous venons donc de montrer qu'à l'état E6 cet algorithme donne :

Etats	I	N	S
E_6	k+1	n	$S=\sum_{i=1}^{k} i$

En particulier, lorsque k = n nous avons dans S la somme des n premiers entiers $\sum_{i=1}^{n} i$:

Etats	I	N	S
E_6	n+1	n	$S=\sum_{i=1}^{n} i$

Nous pouvons déjà écrire que : $\forall$ n, n > 0, $S = \sum_{i=1}^{n} i$

En plus ce dernier tableau nous permet immédiatement de trouver la valeur exacte de la variable de contrôle de la boucle (ici la variable I qui vaut n+1) et donc d'écrire un test d'arrêt de boucle juste.

On peut alors choisir comme test I<>n+1 ou bien I< n+1 etc... ou tout autre prédicat équivalent.

Il était possible de programmer directement cet algorithme avec les deux autres boucles (**pour**... et **répéter**...). Ceci est proposé en exercice au lecteur.

7. Traducteur élémentaire LDFA - C# / Pascal

Nous venons de voir qu'un algorithme devait se traduire en langage de programmation (dit évolué). Nous fournirons ici un tableau qui sera utile à l'étudiant pour la traduction des instructions algorithmiques en langage de programmation.

7.1 Traducteur

Afin de bien montrer que l'écriture algorithmique est plus abstraite qu'un langage de programmation nous donnons un tableau de traduction LDFA dans deux langages : en Pascal de base et en C# restreint aux instructions seulement: (dans le tableau, P est un prédicat et E une instruction composée)

LDFA	C#	Pascal
Ω (instruction vide)	pas de traduction	pas de traduction
debut i1 ; i2; i3; ..... ; ik **fin**	**{** i1 ; i2; i3; ..... ; ik **}**	**begin** i1 ; i2; i3; ..... ; ik **end**
x ← a	x = a ;	x := a
;	pas de traduction	(ordre d'exécution) ;
Si P **alors** E1 **sinon** E2 **Fsi**	**if** (P) E1 ; **else** E2 ; *(attention, pas de fermeture !)*	**if** P **then** E1 **else** E2 *(attention, pas de fermeture !)*
Tantque P **faire** E **Ftant**	**while** (P) E ; *(attention, pas de fermeture)*	**while** P **do** E *(attention, pas de fermeture)*
répeter E **jusquà** P	**do** E ; **while** (! P) ;	**repeat** E **until** P
lire (x1,x2,x3.....,xn)	System.Colnsole.Read( ) System.Colnsole.ReadLine( )	read(fichier,x1,x2,x3.....,xn) readln(x1,x2,x3.....,xn) Get(fichier)
ecrire (x1,x2,x3.....,xn)	System.Colnsole.Write( ) System.Colnsole.WriteLine( )	write(fichier,x1,x2,x3.....,xn) writeln(x1,x2,x3.....,xn) Put(fichier)
pour x ← a **jusquà** b **faire** E **Fpour**	**for** (**int** x= a; x <= b; x++) E ;	**for** x:=a **to** b **do** E *(croissant)* **for** x:=a **downto** b **do** E *(décroissant)* *(attention, pas de fermeture)*
SortirSi P	**if** (P) break **;**	**if** P **then** Break

Ce tableau de traduction permet déjà d'écrire très rapidement des programmes Pascal et C# simples à partir d'algorithmes étudiés et écrits.

7.2 Exemple

En appliquant le traducteur à l'algorithme de l'équation du second degré nous obtenons le programme Pascal suivant :

```
program equation;
var
   A,B,C:real;
   X1,X2:real;
   Delta:real;
begin
  readln(A,B,C);
  if A = 0 then {A=0}
    if B = 0 then
      if C = 0 then
         writeln('R est solution')
      else
         writeln('pas de solution')
```

```
    else
    begin
      X1 := - C/B;
      writeln('x=',X1)
    end
else
begin
    Delta := B*B-4*A*C;
    if Delta < 0 then
          writeln('pas de solution')
    else
     if Delta=0 then
     begin
       X1 := -B/(2*A);
       writeln('x=',X1)
     end
     else
     begin
       X1 := (-B + Sqrt(Delta)) / (2*A);
       X2 := (-B - Sqrt(Delta)) / (2*A);
       writeln('x1=',X1,'x2=',X2)
     end
end
end.
```

En appliquant le traducteur C# à ce même algorithme de l'équation du second degré, nous obtenons le squelette de programme C# suivant :

```
double a, b, c, delta, x, x1, x2 ;
.....
if (a ==0)
  if (b ==0)
    if (c ==0)
      System.Colnsole.WriteLine( ) ("tout reel est solution") ;
    else
     System.Colnsole.WriteLine( ) ("il n'y a pas de solution") ;
  else  {
     x = -c/b ;
     System.Colnsole.WriteLine( ) ("la solution est " + x) ;
   }
 else  {
    delta = b*b -4*a*c ;
    if (delta <0)
      System.Colnsole.WriteLine( ) ("il n'y a pas de solution dans les reels") ;
    else
      if (delta == 0)  {
          x1 = -b / (2*a) ;
          System.Colnsole.WriteLine( ) ("il y a une solution double : "+x1) ;
      }
       else   {
       x1 = (-b + Math.Sqrt(delta)) / (2*a) ;
       x2 = (-b - Math.Sqrt(delta)) / (2*a) ;
       System.Colnsole.WriteLine( ) ("il y deux solutions égales a "+x1+" et " + x2) ;
       }
 } etc…
```

7.3 Sécurité et ergonomie

L'utilisation du traducteur manuel LDFA ––> C# fournit une version préliminaire de programme pascal fonctionnant sur des données correctes sans aucune présentation.

Il appartient au programmeur de compléter dans un deuxième temps la partie sécurité associée aux contraintes du domaine de définition des variables et aux contraintes matérielles d'implantation. Enfin, dans un troisième temps, l'ergonomie (forme de l'échange d'information entre le programme et le futur utilisateur) sera envisagée et programmée.

Voyons sur l'exemple de la somme des n premiers entiers déjà cité plus haut, comment ces trois étapes s'articulent .

Etape de traduction-*somme des n premiers entiers*

Texte final de l'algorithme de départ :	Texte de sa traduction en C# de base :
Algorithme Somentier N ∈ **N*** S , I ∈ **N**^{2} **Début** *{Somentier}* **Lire** (N) ; S ← 0; I ← 1; **TantQue** I < N+1 **faire** S ← S+I; I ← I+1; **FinTant**; Ecrire(S); **Fin Somentier**	**class** Somentier **{** **static int** N ; **static int** S,I ; **static void** Main()**{** n = Int32.Parse(System.Console.ReadLine( )); S = 0 ; I = 1 ; **while** (I < N+1) **{** S = S + I; I = I+1; **}** System.Console.WriteLine(S); **}** **}**

Etape de sécurisation-*somme des n premiers entiers*

Sécurité due aux domaines de définition des données

La traduction ne permet pas d'écrire les domaines de définition des variables : en l'occurrence ici la variable N∈ **N*** est traduite par " **var** N : **integer** ", or le type prédéfini **integer** est un sous-ensemble des entiers relatifs **Z**, il est donc nécessaire d'éliminer les entiers négatifs ou nuls comme choix possible.

Dès que l'utilisateur aura entré son nombre, le programme devra tester l'appartenance au bon intervalle afin de protéger la partie de code, par exemple avec une instruction de condition :

```
if ( N > 0 ) {   // code protégé  }
```

Protection programmée dans les pointillés en dessous dans le cadre de droite :

```
class Somentier {
  static  int N  ;
  static   int S,I ;
 static void Main(){
    n = Int32.Parse( System.Console.ReadLine( ));

  S = 0 ;
  I = 1 ;
   while (I < N+1) {
      S = S + I;
      I = I+1;
   }
    System.Console.WriteLine(S);

 }
}
```

```
class Somentier {
  static  int N  ;
  static   int S,I ;
static void Main(){
    n = Int32.Parse( System.Console.ReadLine( ));
  if ( N > 0 ) {
    S = 0 ;
    I = 1 ;
    while (I < N+1) {
      S = S + I;
      I = I+1;
     }
     System.Console.WriteLine(S);
  }
 }
}
```

Sécurité due aux contraintes d'implantation

Si nous exécutons ce programme pour la valeur N=65537, la valeur fournie en sortie est "-2147450880" sur un entier signé 32 bits « **int** », le résultat n'est pas correct. Nous sommes confrontés au problème de la représentation des entiers machines déjà cité. Ici, le type **int** est restreint à l'intervalle *[*-2147483648, +2147483647*]* ; il y a manifestement dépassement de capacité (overflow) et le système a allègrement continué les calculs malgré ce dépassement. En effet, la somme vaut 65537*65538/2 soit **2147581953**, cette valeur n'appartient pas à l'intervalle des **int**.

Le programmeur doit donc remédier à ce problème par un effort personnel de sécurisation de son programme en n'autorisant les calculs que pour des valeurs valides offrant un maximum de sécurité.

Ici la variable S contient la somme $\sum_{i=1}^{k} i$, nous savons que $\sum_{i=1}^{k} i$ = k(k+1)/2, donc il suffira de résoudre dans **N** l'inéquation k(k+1)/2 ≤ 2147483647où n est l'inconnue. L'unique solution positive a pour partie entière 65537, qui est la valeur maximale avant dépassement de capacité. Donc il suffit de protéger le code par un test supplémentaire sur la variable N :

```
if (N > 0 & N < 65538) {
   S = 0 ;
   I = 1 ;
   while (I < N+1) {
      S = S + I;
      I = I+1;
    }
   System.Console.WriteLine(S);
}
```

En vérifiant sur l'exécution, nous trouvons que S = **2147581953** pour N = **65537**. Ce qui nous donne la version suivante du programme :

```
class Somentier {
  static  int N  ;
  static   int S,I ;
 static void Main(){
    n = Int32.Parse( System.Console.ReadLine( ));
   if ( N > 0 & N < 65538 ) {
      S = 0 ;
      I = 1 ;
      while (I < N+1) {
        S = S + I;
        I = I+1;
      }
      System.Console.WriteLine(S);
   }
  }
}
```

Etape d'ergonomie-*somme des n premiers entiers*

Dans cet exemple, l'information à échanger avec l'utilisateur est très simple et ne nécessite pas une interface spéciale. Il s'agira de lui préciser les contraintes d'entrée et de lui présenter d'une manière claire le résultat.

```
class Somentier {
  static  int N  ;
  static   int S,I ;
 static void Main(){
    System.Console.WriteLine(" Entrez un entier entre 0 et 65537 "  ) ;
    n = Int32.Parse( System.Console.ReadLine( ));
   if ( N > 0 & N < 65538 ) {
      S = 0 ;
      I = 1 ;
      while (I < N+1) {
        S = S + I;
        I = I+1;
      }
      System.Console.WriteLine(" La somme des "+N+  " premiers entiers = "+ S ) ;
      System.Console.WriteLine(S);
   }
   else
       System.Console.WriteLine(" Calcul impossible ") ;
  }
}
```

Vous remarquerez que les adjonctions supplémentaires de code *(en italique)* dans le programme final se montent à environ 50% du total du code écrit, car un logiciel n'est pas uniquement un algorithme traduit.

En continuant d'appliquer le principe de la programmation structurée, il est bon de bien séparer lors du développement la partie algorithmique des parties sécurité et ergonomie. Le programmeur débutant y gagnera en clarté dans sa méthode de travail.

8. Facteurs de qualité du logiciel

B.Meyer et G.Booch

Constat

Un utilisateur, lorsqu'il achète un produit comme un appareil électroménager ou une voiture, attend de son acquisition qu'elle possède un certain nombre de **qualités** (fiabilité, durabilité, efficacité, ...). Il en est de même avec un logiciel.

Voici une liste minimale de critères de qualité du logiciel (proposée B.Meyer, G.Booch):

Correction	**Robustesse**	**Extensibilité**
Réutilisabilité	**Compatibilité**	**Efficacité**
Portabilité	**Vérificabilité**	**Intégrité**
Facilité utilisation	**Modularité**	**Lisibilité**
Abstraction		

Reprenons les définitions communément admises par ces deux auteurs sur ces facteurs de qualité.

Correction

La **correction** est la qualité qu'un logiciel a de respecter les spécifications qui ont été posées.

Robustesse

La **robustesse** est la qualité qu'un logiciel a de fonctionner en se protégeant des conditions de dysfonctionnement.

Extensibilité

L'extensibilité est la qualité qu'un logiciel a d'accepter des modifications dans les spécifications et des adjonctions nouvelles.

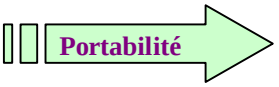
Réutilisabilité

La **réutilisabilité** est la qualité qu'un logiciel a de pouvoir être intégré totalement ou partiellement sans réécriture dans un nouveau code.

Compatibilité

La **compatibilité** est la qualité qu'un logiciel a de pouvoir être utilisé avec d'autres logiciels sans autre effort de conversion des données par exemple.

L'efficacité est la qualité qu'un logiciel a de bien utiliser les ressources.

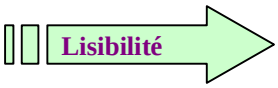


La **portabilité** est la qualité qu'un logiciel a d'être facilement transféré sur de nombreux matériels, et insérable dans des environnements logiciels différents.

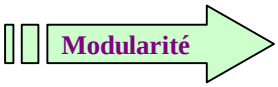
La **vérificabilité** est la qualité qu'un logiciel a de se plier à la détection des fautes, au traçage pendant les phases de validation et de test.

L'intégrité est la qualité qu'un logiciel a de protéger son code et ses données contre des accès non prévus.

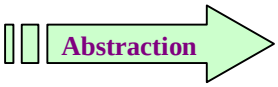
La **facilité** d'utilisation est la qualité qu'un logiciel a de pouvoir être appris, utilisé, interfacé, de voir ses résultats rapidement compris, de pouvoir récupérer des erreurs courantes.



La **lisibilité** est la qualité qu'un logiciel a d'être lu par un être humain.



La **modularité** est la qualité qu'un logiciel a d'être décomposable en éléments indépendants les uns des autres et répondants à un certain nombre de critères et de principes.



L'abstraction est la qualité qu'un logiciel a de s'attacher à décrire les opérations sur les données et à ne manipuler ces données qu'à travers ces opérations.

La production de logiciels de qualité n'est pas une spécificité des professionnels de la programmation ; c'est un état d'esprit induit par les méthodes du génie logiciel. Le débutant peut, et nous le verrons par la suite, construire des logiciels ayant des " qualités " sans avoir à fournir d'efforts supplémentaires.

Bien au contraire la réalité a montré que les étudiants " bricoleurs " passaient finalement plus de temps à " bidouiller " un programme que lorsqu'ils décidaient d'user de méthode de travail. Une amélioration de la qualité générale du logiciel en est toujours le résultat.