Introduction à l'algorithmique

passeport pour la programmation

Christian Percebois

Département Informatique
IUT A Paul Sabatier

septembre 2010

TABLE DES MATIERES

Avant-propos

Les enseignements en Algorithmique et Programmation (AP), tels que définis dans le Programme Pédagogique National (PPN) du DUT Informatique, doivent permettre[1] :

- d'acquérir les connaissances nécessaires à la réalisation de logiciels,
- d'acquérir les concepts transversaux aux différents champs de l'informatique en terme de raisonnement, d'abstraction et de mise en œuvre de solutions,
- de développer des compétences permettant de comprendre, faire évoluer, d'assurer la maintenance et de déployer une application logicielle,
- d'apprendre à participer à un travail d'équipe en charge d'un projet et à être autonome dans la réalisation d'une mission.

Ce cours correspond à l'unité de formation INITIATION À L'ALGORITHMIQUE (TC-INFO-AP1) du pôle AP. Cette unité se caractérise par les points suivants :

Objectifs :

- Connaître un langage algorithmique élémentaire.

Compétences minimales :

- Savoir lire, comprendre, utiliser et tester un algorithme élémentaire.
- Savoir établir le lien entre un algorithme et un programme qui l'implémente.
- Savoir concevoir un algorithme similaire à un algorithme donné.

Contenu :

- Notion d'information et de modélisation.
- Structures algorithmiques fondamentales (séquence, choix, itération, etc.).
- Notion de type.
- Notion de sous-programme (fonction, procédure, méthode, etc.) et de paramètre.
- Implantation en langage de programmation.
- Premières notions de qualité (assertions, pré- et post-conditions, anomalies – élaboration d'un jeu d'essai).

Prolongements possibles :

- Notions de syntaxe et de sémantique.

Indications de mise en œuvre :

- Le choix du paradigme de programmation est laissé libre.

[1] extrait du PPN Informatique (septembre 2005)

Chapitre 1 : Le langage algorithmique

L'objectif de ce chapitre concerne l'écriture de schémas de calcul ou algorithmes. Après une présentation de la notion d'algorithme, nous détaillons les outils nécessaires à leur écriture.

1. NOTION D'ALGORITHME

Définition

On appelle algorithme la description, pour un exécutant donné, d'une méthode de résolution d'un problème sous la forme d'un enchaînement d'actions fournissant le résultat cherché.

Exemples

1) Dès l'antiquité, Euclide (mathématicien grec du $3^{ème}$ siècle avant J.-C.) a défini un algorithme pour calculer le plus grand commun diviseur (pgcd) de deux nombres entiers.

2) Depuis Jules Ferry (1832-1893), les élèves de l'école primaire apprennent à poser la multiplication de deux nombres.

3) Les notices constituent une autre exemple d'algorithme. Par exemple, la notice de remise à l'heure d'un réveil, les instructions d'assemblage d'un modèle réduit ou la recette pour un gâteau.

4) Les comportements de la vie courante peuvent aussi être modélisés par un algorithme. Par exemple, celui d'un salarié le matin avant de se rendre à son travail, celui d'un jardinier plantant un arbre ou d'un automobiliste changeant une roue à la suite d'une crevaison. Ce sont ces trois exemples simples que nous utiliserons par la suite pour illustrer ce cours.

En bref, chaque fois que nous sentons qu'une tâche peut se réaliser d'une manière purement routinière, il existe un algorithme formel, au sens informatique du terme, pour la même tâche. Un algorithme est donc la description, pour un exécutant donné, d'une méthode de résolution d'un problème, autrement dit d'une suite d'opérations conduisant à la solution à atteindre. Il exprime la décomposition de la tâche à réaliser en primitives de l'exécutant.

Remarques

1) La notion d'algorithme est très ancienne et a précédé celle d'ordinateur. Elle n'est donc pas spécifique à l'informatique.

2) Un algorithme s'exprime, dans sa forme finale, dans un langage de programmation pour être compris et exécuté par un ordinateur ; on parle alors de programme. On dit aussi qu'un programme correspond au codage d'un algorithme.

3) Un algorithme est indépendant du langage de programmation utilisé. L'algorithme d'Euclide programmé en Pascal, Ada ou Lisp reste toujours l'algorithme d'Euclide. Pour cela, nous introduirons dans la suite de ce cours un langage algorithmique permettant d'exprimer tout énoncé algorithmique.

2. L'ACTEUR

Définition

L'acteur est l'entité qui met en œuvre un algorithme.

Par rapport aux exemples cités, le salarié, le jardinier et l'automobiliste sont des acteurs ; si l'on parle de la mise en œuvre informatique de l'algorithme d'Euclide, l'acteur ou exécutant sera un ordinateur.

Il faut donc connaître précisément l'acteur avant d'écrire un algorithme. Un acteur se caractérise par les objets qu'il manipule et ses actions élémentaires, encore appelées actions primitives car non décomposables. Il sait exécuter chacune des actions élémentaires ; c'est son savoir-faire. Il sait aussi interroger l'état d'un objet au travers de conditions qui correspondent à des expressions logiques.

Exemples

1) Acteur : jardinier
Objets : brouette, arrosoir, ...
Actions : bêcher, arroser, planter, ...
Conditions : arrosoir plein, trou rebouché, …

2) Acteur : automobiliste
Objets : cric, manivelle, roue de secours, ...
Actions : lever le véhicule avec le cric, tourner la manivelle, …
Conditions : voiture en état de marche, cric disponible, …

Remarques

1) Ces notions existent aussi pour l'acteur ordinateur qui n'est rien d'autre qu'un acteur particulier. Il possède en tant qu'objets des « *variables* » ; il est muni d'actions élémentaires comme « *lire une donnée* », « *calculer la valeur d'une expression* », ... ; il peut aussi interroger son environnement par une question comme « *cette variable est-elle positive ?*», …

2) L'acteur est toujours parfaitement bien identifié au moment de l'écriture de l'algorithme. Il se caractérise essentiellement par son jeu d'actions élémentaires. Une action élémentaire d'un acteur s'exprime par un verbe à l'infinitif suivi de compléments. Le verbe donne un sens (une sémantique) à l'action ; le ou les compléments précisent l'action. On peut dès lors considérer un algorithme comme une suite ordonnée de prescriptions avec le programmeur pour donneur d'ordre et l'acteur en tant qu'exécutant.

3. LES STRUCTURES DE CONTROLE

Définition

Une structure de contrôle exprime un schéma d'ordonnancement des actions élémentaires de l'acteur.

Trois structures de contrôle fondamentales existent : la séquence, la sélection (ou choix) et la répétition ; en pratique, elles suffisent à écrire tout algorithme.

3.1. La séquence

Définition

La séquence exprime un enchaînement inconditionnel d'actions.

Syntaxe

$action_1$;	
$action_2$;	
…	où *action$_i$* est une action élémentaire
$action_i$;	de l'acteur
…	
$action_n$;	

Rappelons que les actions élémentaires d'un acteur s'expriment en pseudo-français sous la forme d'un verbe à l'infinitif muni éventuellement de ses compléments (groupe verbal). Le séparateur d'actions ';' (point-virgule) est un mot de la syntaxe du langage algorithmique.

Sémantique

L'*action$_i$* se déroule après l'*action$_{i-1}$* et avant l'*action$_{i+1}$*. Ceci exprime bien qu'il s'agit d'une suite inconditionnelle d'actions.

Exemples

1) Soit l'acteur automobiliste, victime d'une crevaison. On peut écrire :

```
prendre le cric ;
prendre la manivelle ;
prendre la roue de secours ;
```

2) Pour l'acteur jardinier :

```
creuser un trou ;
mettre l'arbre dans le trou ;
arroser l'arbre ;
```

3.2. La sélection (ou choix)

Définition

Elle exprime le choix entre deux actions (en général) en fonction de la valeur d'une condition.

Syntaxe

```
si condition alors
        action1 ;
sinon
        action2 ;
fin si ;
```

avec *action1* et *action2* définies comme des actions élémentaires réalisées par l'acteur et les mots *si*, *alors* et *fin si*, éléments de la syntaxe.

Les conditions peuvent être simples ou composées. Les conditions simples ne font intervenir qu'une seule interrogation, avec éventuellement un opérateur de négation ***non***. Par exemple, pour l'automobiliste, une condition simple sera « *la voiture est en état de marche* », alors que pour le jardinier, on pourra écrire « *l'arrosoir est plein* ». Une condition simple avec négation pour l'acteur jardinier prendra par exemple la forme « ***non*** *trou rebouché* ».

Les conditions composées combinent des conditions simples. Pour cela, on dispose de 2 opérateurs : l'opérateur ***et*** et l'opérateur ***ou***. Elles s'écrivent respectivement : *condition1* ***et*** *condition2*, et *condition1* ***ou*** *condition2*. Par exemple, on peut imaginer les formulations « *le cric fonctionne* » ***et*** « *la manivelle est disponible* » pour l'automobiliste ou encore « *le sécateur est affûté* » ***ou*** « *l'arrosoir est plein* » pour le jardinier.

La négation d'une condition est vérifiée si la condition ne l'est pas. Une condition composée ***et*** est vérifiée si les deux conditions simples le sont. Enfin, une condition composée ***ou*** est vérifiée si l'une des deux conditions (ou les deux) l'est (le sont).

Sémantique

La syntaxe d'une sélection montre qu'il y a deux cas à examiner : l'*action1* est exécutée si la condition (simple ou composée) est vérifiée, ou bien l'*action2* est exécutée si la condition (simple ou composée) n'est pas vérifiée.

Une seule action (*action1* ou *action2*) est exécutée. Jamais les deux, car il s'agit d'un choix !

Exemples

1) Soit l'acteur automobiliste devant changer la roue de son véhicule :

```
si le matériel est disponible alors
        changer la roue ;
sinon
        appeler un garagiste ;
fin si ;
```

La présence du matériel dans le coffre conditionne le comportement de l'acteur.

2) Pour l'acteur salarié au moment où il sort de chez lui :

```
si il pleut ou s'il y a du vent alors
        prendre l'imperméable ;
sinon
        prendre la veste ;
fin si ;
```

Il suffit qu'il pleuve ou bien qu'il y ait du vent (ou les deux à la fois) pour que la personne choisisse l'imperméable.

Cas particulier (sélection réduite)

Lorsque l'*action2* n'existe pas, la phrase devient :

```
si  condition alors
        action ;
fin si ;
```

Si la condition est vérifiée, l'acteur exécute l'action ; dans le cas contraire, il ne fait rien. C'est une sélection réduite.

Pour l'automobiliste, on peut par exemple écrire :

```
si la voiture possède un enjoliveur alors
        ôter l'enjoliveur ;
fin si ;
```

ou bien lorsqu'il place le cric sous la voiture :

```
si le cric est disponible et la manivelle est disponible alors
        placer le cric sous la voiture ;
fin si ;
```

Dans ce dernier exemple, il faut que les deux conditions soient vérifiées pour que le cric soit placé sous la voiture par l'automobiliste.

Remarque

La construction :

```
si condition alors
        action1 ;
sinon
        action2 ;
fin si ;
```

est équivalente à :

```
si non condition alors
        action2 ;
sinon
        action1 ;
fin si ;
```

3.3. La répétition

Définition

La répétition exprime qu'une même action est exécutée tant qu'une condition reste vérifiée.

Syntaxe

```
tantque  condition faire
        action ;
fin tantque ;
```

où l'action est une action élémentaire de l'acteur, et *tantque*, *faire* et *fin tantque* sont des mots de la syntaxe du langage.

Sémantique

L'action est répétée tant que la condition reste vérifiée. On a donc un cycle : évaluation de la valeur de la condition ; si la condition est vérifiée, exécution de l'action ; puis retour au début du cycle. Ce cycle s'arrête lorsque la condition n'est plus vérifiée. La valeur de la condition doit être modifiée par le déroulement de l'action, sinon la répétition est infinie.

Exemples

1) Toujours pour l'acteur automobiliste, au moment où il enlève la roue :

```
tantque il reste un boulon à débloquer faire
        débloquer un boulon ;
fin tantque ;
```

On notera sur cet exemple que la boucle se termine puisque le nombre d'actions de la répétition dépend uniquement du nombre de boulons qui est connu a priori.

2) Pour l'acteur jardinier au moment de reboucher le trou :

```
tantque non le trou est rebouché faire
    ajouter de la terre ;
fin tantque ;
```

Le jardinier rajoute de la terre jusqu'à ce que le trou soit effectivement rebouché. Cette écriture est sémantiquement équivalente à la formulation :

```
tantque le trou n'est pas rebouché faire
    ajouter de la terre ;
fin tantque ;
```

Remarques

1) A priori, le nombre de fois où l'action d'une répétition est exécutée est inconnu du programmeur.

2) Ce nombre peut varier de zéro à l'infini : zéro si la condition n'est pas vérifiée dès la première évaluation, et l'infini si l'action ne modifie pas la valeur de la condition (celle-ci est toujours vérifiée).

Pour l'acteur salarié qui déjeune le matin, on peut écrire :

```
tantque il reste une tartine faire
    manger une tartine ;
fin tantque ;
```

Dans cette situation, il est clair qu'il est impossible de manger une tartine si le stock des tartines est épuisé.

Pour l'acteur automobiliste, une expression pourrait être :

```
tantque la voiture n'est pas suffisamment levée faire
    tourner la manivelle ;
fin tantque ;
```

Dans ce cas, le nombre de tours de manivelle n'est pas connu a priori ; on peut imaginer le cas d'un cric défectueux pour lequel les tours de manivelle sont sans effet : la boucle est alors infinie.

4. LES ENONCES ALGORITMIQUES

Définition

Un énoncé algorithmique est un texte en pseudo-français respectant la syntaxe définie par les structures de contrôle. Un algorithme est un énoncé dont l'exécution par l'acteur résout sémantiquement le problème.

Un énoncé algorithmique, et donc un algorithme, s'écrit dans un langage appelé langage algorithmique. Ce langage est composé de mots, symboles, commentaires... groupés en phrases qui obéissent à des règles qui déterminent de manière absolument stricte si une phrase est correcte ou non. L'analogie avec une langue naturelles comme le français est donc forte, la principale différence étant qu'une phrase en langue naturelle peut être formée de manière beaucoup moins rigoureuse et signifier néanmoins quelque chose.

Un énoncé dans un langage donné est dit syntaxiquement correct s'il respecte les règles de construction des phrases du langage et syntaxiquement incorrect sinon. Un énoncé syntaxiquement correct peut s'interpréter ; sa sémantique dépend des actions exécutées. A l'inverse, si sa syntaxe est incorrecte, il ne peut y avoir d'interprétation.

Pour l'acteur salarié, l'énoncé suivant ne respecte pas la syntaxe du langage algorithmique. « *Prendre la veste* » n'est pas une condition ; de même, « *il pleut ou il y a du vent* » n'exprime pas une action. Ce n'est donc pas un énoncé algorithmique.

.

```
si prendre la veste faire
      il pleut ou il y a du vent ;
alors ;
fin ;
```

Un énoncé syntaxiquement correct peut être asémantique, c'est-à-dire sémantiquement incorrect. Par exemple, pour l'acteur jardinier devant planter un arbre, il est illogique de « *mettre l'arbre dans le trou* » avant de « *creuser un trou* » :

```
arroser l'arbre ;
mettre l'arbre dans le trou ;
creuser un trou ;
```

Se méfier aussi de certaines formulations pouvant conduire à des incohérences sémantiques. Pour l'automobiliste par exemple, l'enchaînement des deux actions « *prendre le cric* » conduit à lui assigner un comportement irrationnel...

```
prendre le cric ;
prendre le cric ;
```

Remarques

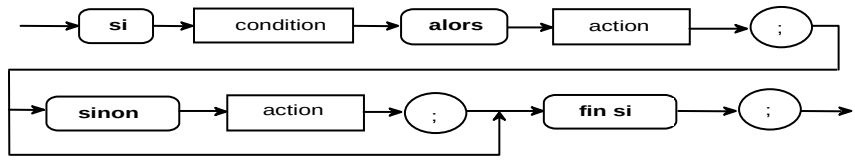
1) Un langage est un ensemble de symboles ou de groupes de symboles, appelés mots du langage. La syntaxe décrit l'ensemble des règles de construction des mots du langage (forme) alors que la sémantique précise la signification des mots (fond).

Un exemple bien connu de langage est celui de la plaque d'immatriculation du véhicule de l'automobiliste. Les mots du langage d'immatriculation sont des suites de symboles, lettres ou chiffres, construits selon les règles décrites par les textes de loi régissant les véhicules. La syntaxe de ce langage indique par exemple que la suite de symboles se termine par deux chiffres. Sa sémantique nous permet d'interpréter ces deux chiffres comme le numéro de département d'immatriculation du véhicule.

2) La syntaxe d'un langage se décrit par une grammaire[2]. Cette description peut se faire graphiquement ou textuellement. Les diagrammes ci-dessous, appelés diagrammes syntaxiques, visualisent graphiquement comment écrire une sélection en langage algorithmique. Pour être exhaustif, les symboles *condition* et *action*, dits symboles non-terminaux de la grammaire (représentés par des rectangles), par opposition aux symboles terminaux *si*, *alors*, *fin si*… (représentés par des rectangles aux bords arrondis), doivent eux aussi être spécifiés par des diagrammes de même type. Tel sera le cas des non-terminaux *séquence* et *répétition* ci-après pour définir un *énoncé algorithmique*.

Enoncé algorithmique

Sélection



[2] La grammaire du langage algorithmique figure en fin de document.

Chapitre 2 : Conception d'algorithmes

Nous détaillons maintenant la méthodologie d'écriture des algorithmes.

1. COMPOSITION D'ACTIONS

Dans les exemples développés jusqu'à présent, les actions étaient élémentaires (actions non exprimables en termes d'autres actions), ce qui limite fortement les possibilités d'écriture d'un énoncé algorithmique. Il faut donc disposer de moyens pour combiner ces actions. En pratique, pour pouvoir décrire toutes les combinaisons possibles de primitives d'un exécutant donné, il suffit de considérer qu'une action sera désormais :

- une action élémentaire,
- une action composée : séquence, sélection ou répétition.

Il est alors possible de définir des emboîtements d'actions : une sélection dans une répétition, une répétition dans une autre répétition, ... il suffit de respecter les règles syntaxiques énoncées précédemment en remplaçant l'action élémentaire par une action composée dans les schémas d'ordonnancement des paragraphes 3.1 à 3.3 du chapitre 1.

Cas général

Ainsi, par exemple, à partir du modèle général de la séquence des 4 actions *action1*, *action2*, *action3* et *action4* :

```
action1 ;
action2 ;
action3 ;
action4 ;
```

on peut écrire, en remplaçant l'action *action2* par une sélection et l'action *action3* par une répétition :

```
action1 ;
si condition2 alors
      action21 ;
sinon
      action22 ;
fin si ;
tantque condition3 faire
      action31 ;
      action32 ;
fin tantque ;
action4 ;
```

où l'action de la répétition est formée de la séquence des 2 actions *action31* et *action32*.

Exemples

1) Selon la météo, un salarié peut quitter son domicile sans prendre ni imperméable, ni veste. On comparera cet énoncé avec celui du chapitre 1, section 3.2.

```
si il pleut ou s'il y a du vent alors
        prendre l'imperméable ;
sinon
        si la température extérieure est inférieure à 25 degrés alors
                prendre la veste ;
        fin si ;
fin si ;
```

2) Pour retirer la roue crevée, l'automobiliste peut maintenant écrire la sélection réduite suivante dont l'action est une séquence composée d'une répétition (d'action « *débloquer un boulon* ») et de l'action élémentaire « *enlever la roue crevée* » :

```
si la manivelle est disponible alors
        tantque il reste un boulon à débloquer faire
                débloquer un boulon ;
        fin tantque ;
        enlever la roue crevée ;
fin si ;
```

Remarquer que les instructions composées sont délimitées par des mots-clés indiquant le début et la fin de la structure (*si* ... *fin si*, *tantque* ... *fin tantque*).

Il faut se méfier de certaines écritures. Ainsi, le texte ci-dessous très voisin du précédent provoque l'enlèvement de la roue crevée même lorsque la manivelle n'est pas disponible, car l'action « *enlever la roue crevée* » est toujours exécutée en séquence !

```
si la manivelle est disponible alors
        tantque il reste un boulon à débloquer faire
                débloquer un boulon ;
        fin tantque ;
fin si ;
enlever la roue crevée ;
```

3) Pour un étudiant en S1 informatique, on peut imaginer :

```
tantque toutes les notions ne sont pas assimilées faire
        étudier le cours ;
        faire les exercices proposés en TD et en TP ;
        si il subsiste des problèmes ou des erreurs alors
                poser des questions aux enseignants ;
        fin si ;
fin tantque ;
```

2. METHODOLOGIE D'ECRITURE DES ALGORITHMES

2.1. Affinages successifs

Jusqu'à présent, nous n'avons fourni que des outils pour exprimer des algorithmes. Mais nous n'avons toujours pas spécifié de méthodologie pour les écrire. Dans ce qui suit, nous proposons une méthode de développement d'algorithmes appelée méthode des affinages successifs.

Définitions

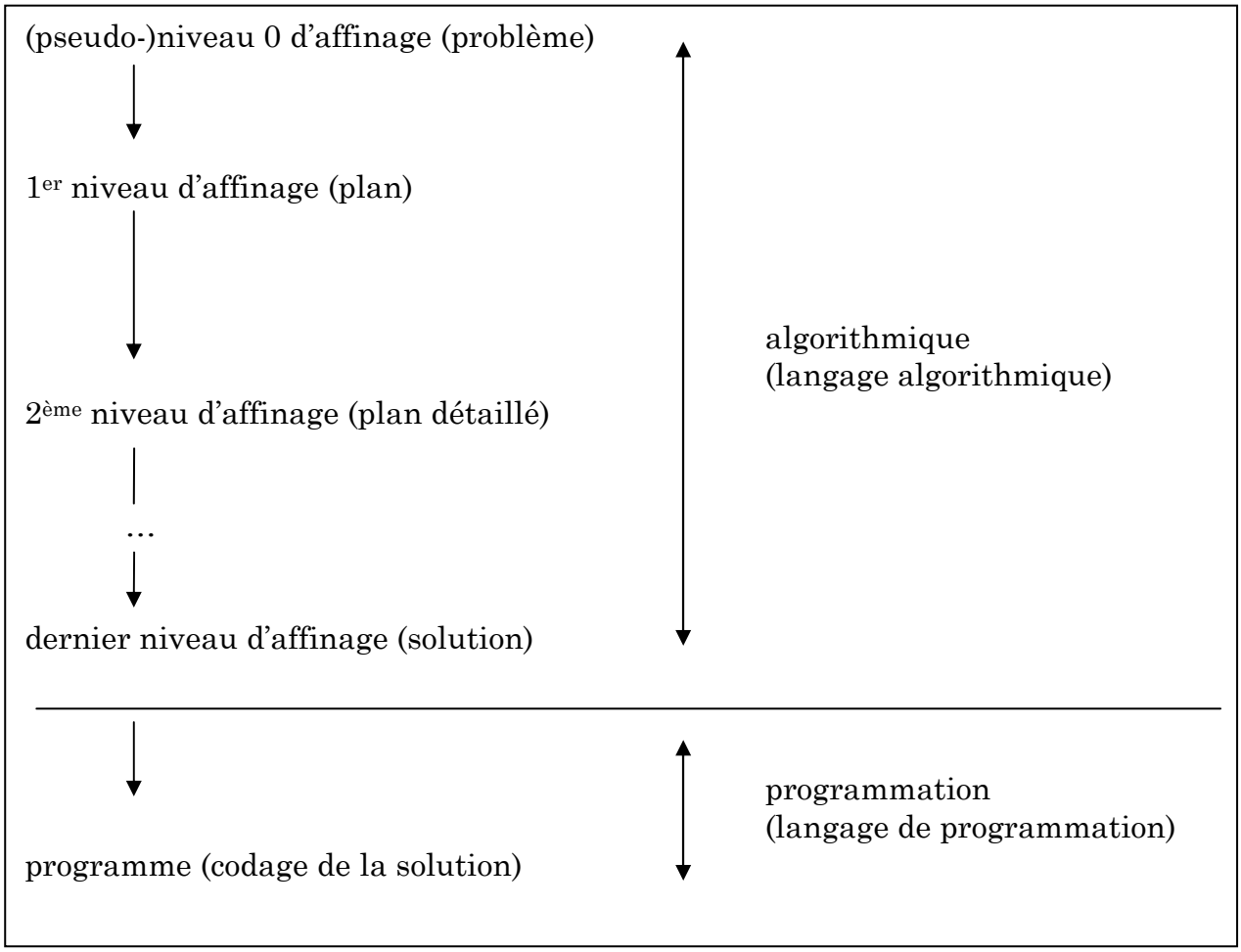
On appelle affinage le processus de décomposition d'une action complexe en termes d'actions moins complexes, plus simples, de telle manière que le développement soit lui-même une séquence, une sélection ou une répétition. La décomposition doit se substituer, en termes sémantiques, à l'action complexe.

On appelle affinages successifs une suite d'affinages résolvant le problème. Chaque niveau d'affinage exprime un niveau de détails. Ainsi, affinant les détails de plus en plus, on finit par atteindre le niveau des actions élémentaires de l'acteur qui constitue le programme proprement dit.

Démarche

Pour illustrer la démarche, faisons l'analogie avec le génie civil. Pour construire un pont, il ne suffit pas qu'un maçon décide de se mettre au travail avec son sable, son ciment, son fil à plomb et sa truelle. La construction nécessite des plans : plans partiels, puis plus détaillés pour les parties délicates. Seulement après on entreprend la réalisation.

Il en est de même pour la programmation. Programmer de façon descendante, c'est commencer par faire le plan de son programme. Puis, pour les parties dont la réalisation n'est pas évidente, on recommence ce processus et ce, jusqu'à atteindre les actions que l'on sait réaliser.



L'algorithmique (partie supérieure du schéma) implique un important travail de réflexion ; la programmation (partie inférieure) correspond simplement à une phase de traduction, appelée codage. Il s'agit donc d'une approche descendante, niveau par niveau. Chaque niveau d'affinage constitue un niveau de détails résolvant le problème. L'arrêt d'un affinage intervient dès que l'action considérée peut se traduire directement en actions élémentaires de l'acteur.

Lors de la programmation, tous les problèmes algorithmiques doivent être résolus ; le programme final est obtenu par une simple substitution de texte : les actions d'un niveau d'affinage sont remplacées par les actions développées. Ce procédé se poursuit si ces actions ont été à leur tour développées en actions plus élémentaires.

Avec cette méthode des niveaux de détails successifs, on définit d'une part l'abstraction (isoler une action complexe sans la décrire : quoi ?) et, d'autre part, son affinage (décrire l'action complexe précédemment isolée en termes d'actions plus simples : comment ?).

Convention[3] **(propre à l'IUT)**

Cette séparation *quoi-comment* se traduira explicitement dans le codage du programme par la convention suivante :

```
-- quoi
comment
```

où la notation -- introduit un commentaire pour ce qui suit jusqu'à la fin de la ligne. Elle a pour principal objectif de mémoriser les prises de décision du concepteur au sein même de l'énoncé algorithmique.

Par exemple, si le niveau N d'affinage d'un problème est :

```
si la manivelle est disponible alors
        ôter la roue crevée ;
fin si ;
```

et si l'on suppose que l'action « *ôter la roue crevée* » s'affine en :

```
tantque il reste un boulon à débloquer faire
        débloquer un boulon ;
fin tantque ;
enlever la roue crevée ;
```

on écrira au niveau N+1 d'affinage, par une simple substitution de texte :

```
si la manivelle est disponible alors
        -- ôter la roue crevée
        tantque il reste un boulon à débloquer faire
                débloquer un boulon ;
        fin tantque ;
        enlever la roue crevée ;
fin si ;
```

Pour maintenir cette traçabilité, nous introduisons une convention de codage appelée règle des paragraphes. Cette règle permet donc d'accompagner le processus de décomposition : chaque action complexe devient un commentaire pour le niveau inférieur où elle s'explicite.

Par définition, nous appelons paragraphe tout commentaire spécifiant une action de l'algorithme (*quoi* ou titre du paragraphe) immédiatement suivi de la mise en œuvre de l'action (*comment* ou contenu du paragraphe). Nous convenons ainsi qu'un paragraphe commence par son titre et se termine soit au début du paragraphe suivant de même niveau d'indentation, soit au prochain niveau d'indentation inférieur au niveau courant (caractérisé par un mot-clé ou le début du paragraphe suivant), soit à la fin du texte.

[3] Les conventions de codage sont détaillées dans le fascicule « Conventions d'écriture des programmes en langage algorithmique ».

A titre d'exemple, le niveau N+1 d'affinage de l'action « *ôter la roue crevée* » correspond au final au paragraphe :

```
-- ôter la roue crevée
tantque il reste un boulon à débloquer faire
      débloquer un boulon ;
fin tantque ;
enlever la roue crevée ;
```

Remarques

1) Chaque étape d'affinage est marquée par une phase de validation, en supposant les actions complexes réalisées par l'acteur. Si l'algorithme est incorrect à ce niveau de détails, revenir en arrière et remettre en cause le niveau d'affinage précédent. Parfois, on peut être amené à remettre en cause le premier niveau. Lorsque l'erreur est détectée, reprendre le processus de décomposition.

2) Dans la démarche des affinages successifs, le concepteur de l'algorithme est aussi acteur ; son jeu d'actions complexes n'est pas a priori spécifié. Il s'obtient par diverses tentatives d'écriture de l'algorithme au niveau considéré. C'est sans doute là une des difficultés majeures de l'approche.

Avantages et inconvénients

Les avantages de cette méthodologie sont :

- raisonnement descendant avec en particulier la possibilité de garder la trace de l'affinage précédent en commentaires ; la documentation de l'algorithme et du programme est alors pertinente et n'entraîne aucun surcroît de travail.
- séparation des difficultés : une seule difficulté à la fois, ce qui évite le foisonnement de détails, souvent source d'erreurs.
- validation de chaque niveau d'affinage obtenu.
- séparation de deux activités différentes que sont l'algorithmique et la programmation ; ainsi, à un algorithme donné correspond plusieurs codes possibles selon le langage de programmation considéré.

Il existe aussi des inconvénients qui sont :

- propagation d'erreurs lorsque les premiers niveaux d'affinage sont incorrects : les premiers choix sont fondamentaux et une mauvaise écriture au premier niveau peut remettre en cause l'architecture du programme.
- absence de mécanisme de modularité : le texte final obtenu est à plat et sa taille peut constituer un frein à sa compréhension ; il faut alors prévoir des méthodes de conception plus élaborées.

2.2. Exemple

Soit un réveil à affichage digital possédant 3 boutons notés *A*, *B* et *C* ainsi que deux registres (tiroirs contenant des valeurs entières) notés *hr* et *mr* mémorisant respectivement les heures et les minutes du réveil.

La notice de cet appareil décrit la manière de le remettre à l'heure :

- Le bouton *A* met le réveil en mode 'remise à l'heure'.
- Une première pression sur le bouton *B* autorise la modification des minutes.
- Chaque pression sur le bouton *C* permet alors d'augmenter d'une unité la valeur du registre des minutes *mr* modulo 60, sans modifier le registre *hr*.
- Le bouton *B* stoppe ce processus et autorise alors la modification de la valeur du registre des heures *hr* : à chaque pression du bouton *C*, il pourra ainsi être augmenté d'une unité, modulo 24, sans modification du registre *mr*.
- Toute pression sur *A* permet de sortir du mode 'remise à l'heure' et de reprendre le mode 'normal' du fonctionnement du réveil.

L'heure de référence, donnée par une horloge externe, s'exprimera à l'aide de 2 valeurs précisant les heures et minutes qu'on affectera respectivement à deux registres *he* et *me* par l'action *lire (he, me)*.

Problème

A partir de cet énoncé, il s'agit d'écrire un algorithme de remise à l'heure du réveil pour un acteur dont les seules actions élémentaires sont :

Action	Description de l'action
lire (he, me)	Lire l'heure et les minutes de référence
appuyerSurA	Appuyer sur le bouton A
appuyerSurB	Appuyer sur le bouton B
appuyerSurC	Appuyer sur le bouton C

On suppose que cet acteur est capable d'enchaîner ces actions élémentaires en séquences, choix et répétitions. Il lui sera possible de comparer le contenu du registre *hr* (respectivement *mr*) avec la valeur externe de référence *he* (respectivement *me*) grâce aux opérateurs =, /=, >, >=, <, <=. On négligera le temps de manipulation dû aux différentes actions pour effectuer la remise à l'heure du réveil.

Solution

Ne pas se hâter, et donc ne pas rédiger directement l'algorithme en utilisant les actions élémentaires de l'acteur. Cette approche tend à négliger la phase d'analyse du problème à résoudre et ne peut s'envisager que pour des exemples immédiats à coder. Ainsi, on bannira toute tentative directe de la forme :

```
appuyerSurA ;
lire (he, me) ;
si hr = he alors
        appuyerSurB ;
        …
```

Il faut au contraire écrire des affinages successifs, en définissant des actions complexes exprimant la solution au problème posé.

On commence donc par se donner un jeu d'actions complexes. Par exemple :

- *modifier les minutes du réveil*,
- *mettre le réveil en mode 'normal'*,
- *modifier les heures du réveil*,
- *mettre le réveil en mode 'remise à l'heure'*,
- et *lire l'heure et les minutes de l'horloge externe*.

Ensuite, on cherche un enchaînement logique de ces actions complexes résolvant le problème de la remise à l'heure du réveil. On conçoit ainsi un algorithme général, appelé premier niveau d'affinage, qui pourrait ici se définir comme suit :

```
mettre le réveil en mode 'remise à l'heure' ;
lire l'heure et les minutes de l'horloge externe ;
modifier les minutes du réveil ;
modifier les heures du réveil ;
mettre le réveil en mode 'normal' ;
```

Il faut maintenant affiner l'action complexe « *modifier les minutes du réveil* ». Au deuxième niveau d'affinage, en considérant l'action « *ajouter une minute aux minutes* », on peut écrire :

```
-- modifier les minutes du réveil
tantque la mise à jour des minutes n'est pas finie faire
        ajouter une minute aux minutes ;
fin tantque ;
```

De façon similaire, on affine l'action complexe « *modifier les heures du réveil* ». On obtient pour le deuxième niveau de cette action :

```
-- modifier les heures du réveil
tantque la mise à jour des heures n'est pas finie faire
        ajouter une heure aux heures ;
fin tantque ;
```

D'où le deuxième niveau complet d'affinage du problème. Il suffit de remplacer les actions complexes du premier niveau par les actions affinées du deuxième niveau :

```
mettre le réveil en mode 'remise à l'heure' ;
lire l'heure et les minutes de l'horloge externe ;
 -- modifier les minutes du réveil
tantque la mise à jour des minutes n'est pas finie faire
      ajouter une minute aux minutes ;
fin tantque ;
-- modifier les heures du réveil
tantque la mise à jour des heures n'est pas finie faire
      ajouter une heure aux heures ;
fin tantque ;
-- remettre le réveil en mode 'normal'
mettre le réveil en mode 'normal' ;
```

Remarquer que le deuxième niveau d'affinage complet du problème est une simple substitution de texte.

Le programme final est décrit ci-dessous ; il est exécutable par une personne sachant réaliser les actions prescrites par la notice du réveil. A ce niveau, on doit gérer les détails les plus fins. En particulier, avant de modifier les minutes (respectivement les heures), il faudra autoriser la modification des minutes (respectivement des heures).

Finalement, on obtient :

```
-- mettre le réveil en mode 'remise à l'heure'
appuyerSurA ;
-- lire l'heure et les minutes de l'horloge externe
lire (he, me) ;
-- modifier les minutes du réveil
appuyerSurB ;
tantque mr /= me faire
      appuyerSurC ;
fin tantque ;
-- modifier les heures du réveil
appuyerSurB ;
tantque hr /= he faire
      appuyerSurC ;
fin tantque ;
-- mettre le réveil en mode 'normal'
appuyerSurA ;
```

Remarques

1) Pour respecter la relation *quoi-comment* souhaitée en termes de traces d'affinage, on notera que la simple recopie de l'affinage d'une action ne permet pas toujours d'obtenir une mise en page correcte ; il conviendra alors de modifier le texte final en conséquence. Ceci justifie l'ajout du commentaire *remettre le réveil en mode 'normal'* au deuxième niveau complet d'affinage, afin que l'action correspondante ne soit pas englobée avec l'action complexe « *modifier les heures du réveil* » de l'affinage.

2) L'écriture d'une répétition constitue un point délicat qu'il convient d'appréhender avec la plus grande rigueur. Une répétition met en jeu différentes composantes : son action d'initialisation, sa condition d'arrêt et son action de progression permettant d'amorcer l'itération suivante. Elle se caractérise aussi par un traitement spécifique au problème posé. On peut résumer sa structuration sémantique comme suit :

```
action d'initialisation ;
tantque condition faire
        traitement ;
        action de progression ;
fin tantque ;
```

Après identification de la condition d'arrêt, on doit notamment porter son effort sur la complémentarité des actions d'initialisation et de progression. Pour la boucle de modification des minutes du réveil, on peut établir les correspondances suivantes :

Composante	Code
Action d'initialisation	appuyerSurB
Condition	mr /= me
Traitement	appuyerSurC
Action de progression	appuyerSurC

Sur cet exemple, le traitement et l'action de progression sont confondus. C'est l'action *appuyerSurC* qui réalise à la fois le traitement (affichage des minutes du réveil) et la progression (augmentation d'une unité du registre *mr*). Comme cette action modifie le registre *mr*, le cycle répétitif se termine ; considérant l'état initial d'entrée dans la répétition *mr /= me*, l'état de calcul *mr = me* sera nécessairement atteint au cours d'une itération donnée de par l'opération modulo sur le registre *mr*. On notera aussi qu'*appuyerSurB* initialise la boucle puisque c'est cette action qui autorise la modification des minutes *mr* du réveil.

3) Dans le cas où le réveil est déjà à l'heure, les seules opérations réalisées par l'acteur seront dans l'ordre « *appuyerSurA* », « *appuyerSurB* », « *appuyerSurB* » et « *appuyerSurA* ». Les actions des deux répétitions « *appuyerSurC* » ne seront donc jamais exécutées. Une optimisation de l'algorithme consiste alors à examiner si le réveil est déjà à l'heure ; on peut imaginer le premier niveau d'affinage suivant :

```
lire l'heure et les minutes de l'horloge externe ;
si le réveil n'est pas à l'heure alors
        mettre le réveil en mode 'remise à l'heure' ;
        modifier les minutes du réveil ;
        modifier les heures du réveil ;
        mettre le réveil en mode 'normal' ;
fin si ;
```

Ce niveau d'affinage se traduit en actions plus élémentaires par le programme ci-dessous où l'unique difficulté réside dans l'expression de la condition « *le réveil n'est*

pas à l'heure ». On l'obtient aisément en considérant la négation de l'expression « *le réveil est à l'heure* » :

hr = he et mr = me

soit :

non (hr = he et mr = me)
≡ non (hr = he) ou non (mr = me) -- *loi de De Morgan*
≡ hr /= he ou mr /= me.

On notera également que la prise en compte de cette nouvelle fonctionnalité de l'algorithme a exigé d'inverser par rapport à la version initiale les deux actions :

- « *mettre le réveil en mode 'remise à l'heure'* »,
- et « *lire l'heure et les minutes de l'horloge externe* ».

Ainsi, aucune action *appuyer* ne sera réalisée si le réveil est déjà à l'heure :

```
-- lire l'heure et les minutes de l'horloge externe
lire (he, me) ;
-- mettre à l'heure le réveil par rapport à l'heure de référence
si hr /= he ou mr /= me alors
    -- mettre le réveil en mode 'remise à l'heure'
    appuyerSurA ;
    -- modifier les minutes du réveil
    appuyerSurB ;
    tantque mr /= me faire
        appuyerSurC ;
    fin tantque ;
    -- modifier les heures du réveil
    appuyerSurB ;
    tantque hr /= he faire
        appuyerSurC ;
    fin tantque ;
    -- mettre le réveil en mode 'normal'
    appuyerSurA ;
fin si ;
```

En conclusion, se méfier donc de certaines optimisations hâtives qui oublieraient de reconsidérer le problème et qui souvent conduisent à des programmes faux, ne respectant pas le mode d'emploi du réveil. Un tel écueil est le résultat d'une méthodologie ascendante consistant à modifier directement le programme final, sans reconsidérer l'algorithme initial.

4) En pratique, la démarche par affinages successifs implique souvent de nombreux essais d'écriture et donc des remises en cause des niveaux précédents lorsque des erreurs sont détectées. La corbeille à papier est donc un ustensile à considérer avec sérieux… De plus, l'exercice implique de l'entraînement et la connaissance de quelques schémas d'algorithmes parfaitement identifiés, appelés

algorithmes fondamentaux[4], comme la recherche d'une donnée, l'insertion d'un élément dans une suite, le tri d'un ensemble de valeurs… Il n'y a donc pas de recette et chaque situation adresse un nouveau problème. La programmation est par essence une activité humaine qui combine à la fois rigueur et imagination.

3. SPECIFICATION DES ACTIONS D'UN ALGORITHME

3.1. Précondition et postcondition d'une action

La méthode d'analyse descendante présentée précédemment ne permet de raisonner séparément sur chacune des actions complexes à affiner de l'algorithme que si cette action ne définit avec précision son état initial et son état final. La notion d'assertion[5] permet alors très simplement d'exprimer cette représentation. Ainsi la spécification d'une action complexe s'exprimera par deux assertions :

- l'assertion d'entrée décrivant l'état du calcul avant l'exécution de l'action, appelée précondition, et qui spécifie les conditions nécessaires pour que l'action puisse s'exécuter correctement,
- l'assertion de sortie décrivant l'état du calcul après exécution de l'action, appelée postcondition, associée à la fonctionnalité que doit remplir l'action.

Une action complexe, que l'on pourra dès lors manipuler sans connaître la façon dont elle est effectivement réalisée, est alors notée et spécifiée :

{précondition} action *{postcondition}*

Exemples

1) Pour la remise à l'heure du réveil, on peut définir, si A et B représentent deux valeurs quelconques :

{hr=A et mr=B} modifier les minutes *mr* du réveil *{hr=A et mr=me}*

De même, on écrira :

{hr=A et mr=B} modifier les heures *hr* du réveil *{hr=he et mr=B}*

2) Considérant l'algorithme général de remise à l'heure du réveil, on pourra spécifier l'état initial avant exécution par une précondition et l'état final après exécution par une postcondition :

[4] Le fascicule « Recueil des algorithmes fondamentaux » fournit un éventail d'algorithmes que tout informaticien digne de ce nom se doit de connaître par cœur.

[5] proposition que l'on énonce et que l'on soutient comme vraie ; affirmation

{hr=A et mr=B}
remettre à l'heure le réveil
{hr=he et mr=me}

3.2. Etats de calcul d'un algorithme

Préconditions et postconditions permettent notamment de spécifier les états de calcul d'un algorithme, parfois aussi appelées situations. Un état de calcul exprime les valeurs des objets manipulés par l'acteur en un point donné de l'exécution. Ainsi, la connaissance des « situations » par lesquelles est supposé passer un algorithme lors de son exécution joue un rôle fondamental aussi bien initialement pour le concevoir, qu'ensuite pour le comprendre ou le modifier.

Considérons l'acteur salarié et supposons que des travaux l'obligent à modifier son itinéraire habituel. L'algorithme suivant décrit les actions qu'il a à entreprendre pour suivre la déviation mise en place :

emprunter la première avenue à gauche ;
avancer jusqu'au deuxième feu rouge ;
au stop, tourner à gauche ;
à la première intersection, continuer tout droit ;

Pour être compréhensible par le programmeur, cet algorithme est à compléter en exhibant les différentes situations où se trouve l'automobiliste au cours de son nouveau trajet : où est-il supposé être au début ? par où passe-t-il ? où se trouve-t-il à la fin de la déviation ?

Un état de calcul se décrit en général par une assertion exhibant l'état de calcul en ce point. L'algorithme est ainsi décoré par différentes assertions, notées par la suite entre accolades et considérées comme des commentaires, en des points particuliers.

Exemples

1) La déviation décorée par des assertions devient :

{gare Matabiau}
emprunter la première avenue à gauche ;
avancer jusqu'au deuxième feu rouge ;
{place St Michel}
au stop, tourner à gauche ;
à la première intersection, continuer tout droit ;
{avenue de Rangueil}

2) L'algorithme qui suit montre comment les actions, considérant l'assertion initiale *{hr=A et mr=B}* qui traduit que les valeurs pour les registres *hr* et *mr* du réveil sont quelconques, modifient ces registres jusqu'à atteindre l'état *{hr=he et mr=me}*, assertion finale exprimant que le réveil est à l'heure :

```
{hr=A et mr=B}
lire (he, me) ;
si hr /= he ou mr /= me alors
        appuyerSurA ;
        appuyerSurB ;
        tantque mr /= me faire
                appuyerSurC ;
        fin tantque ;
        {hr=A et mr=me}
        appuyerSurB ;
        tantque hr /= he faire
                appuyerSurC ;
        fin tantque ;
        {hr=he et mr=me}
        appuyerSurA ;
fin si ;
{hr=he et mr=me}
```

Remarque

Spécifier une action permet de détecter des incohérences d'enchaînement au sein d'un algorithme. Par exemple, en spécifiant l'action élémentaire « *prendre le cric* » de l'acteur automobiliste comme suit :

```
{le cric est disponible}
prendre le cric
{le cric n'est pas disponible}
```

on identifie aisément que la deuxième action de la séquence :

```
prendre le cric ;
prendre le cric ;
```

viole la précondition qu'exige « *prendre le cri* » :

```
{le cric est disponible}
prendre le cric ;
{le cric n'est pas disponible}
prendre le cric ;
{ ?}                          -- état non atteignable
```

Chapitre 3 : Les sous-programmes

Lors de l'écriture des algorithmes, nous avons mis en évidence des actions complexes, puis affiné ces actions complexes par niveaux de détails successifs pour atteindre le programme final. Des mécanismes encourageant cette décomposition existent dans les langages de programmation : il s'agit des sous-programmes. Dans une première approche, un sous-programme permet de considérer un groupe d'instructions comme une entité indépendante susceptible d'être appelée à partir du programme principal.

Définition

Un sous-programme est un module indépendant de code de l'acteur ordinateur (associé, en ce qui nous concerne, à une abstraction fonctionnelle de l'algorithme).

Intérêt

Dans les langages de programmation, cette notion de sous-programme vise un double objectif assez similaire à la décomposition algorithmique :

- permettre le découpage d'un programme complexe en unités logiques plus petites, susceptibles d'être conçues, codées, validées et maintenues séparément,
- permettre l'utilisation répétitive d'un même ensemble d'actions (appelées instructions) en différents points du programme.

Dans la suite de ce cours, nous considérons deux types de sous-programmes : les procédures et les fonctions. Procédures et fonctions permettent de représenter n'importe quelle abstraction fonctionnelle ; les fonctions de plus rendent toujours un résultat unique. Nous considérons tout d'abord le cas particulier des procédures.

1. EN-TETE D'UN SOUS-PROGRAMME

Définition

Un en-tête de sous-programme est une interface entre le sous-programme et le monde extérieur (le programme).

Définir un en-tête de sous-programme consiste à donner un nom (un identificateur) à une action complexe.

Exemple

On s'intéresse à l'algorithme d'un éditeur de texte (simplifié) :

```
-- éditer un texte
créer un texte initialement vide ;
lire une commande de l'utilisateur ;
tantque la commande n'est pas la commande « quitter » faire
        exécuter la commande ;
        afficher le texte ;
        lire la commande suivante de l'utilisateur ;
fin tantque ;
```

Dans cet exemple, on peut associer un sous-programme à chacune des actions complexes, par exemple « *créer un texte initialement vide* », « *lire la commande de l'utilisateur* », « *exécuter la commande* » et « *afficher le texte* ». On est ainsi amené à spécifier des sous-programmes reflétant la structure de l'algorithme.

L'en-tête (ou déclaration) d'un sous-programme est aussi appelé spécification du sous-programme ; il consiste à donner un nom significatif à une action complexe de l'algorithme (quoi ?). En aucun cas, ne sont mentionnés dans une spécification les détails de mise en œuvre du sous-programme (comment ?) appelé corps du sous-programme ; ce point fera l'objet du chapitre 5.

Ainsi, en reprenant notre exemple, nous écrirons pour l'en-tête du sous-programme de *création d'un texte vide*, ici de type procédure :

```
procédure créerTexte ;
```

et pour *la lecture d'une commande, son exécution* et *l'affichage du texte*, toujours considérés comme des procédures :

```
procédure lireCommande ;
```

```
procédure exécuterCommande ;
```

```
procédure afficherTexte ;
```

Remarque

Dans ce cours, nous utilisons la même syntaxe pour écrire des algorithmes et des programmes[6]. Pour les algorithmes, seules les constructions séquence, sélection et répétition sont utilisées en exprimant les actions et conditions en pseudo-français. Pour les programmes, de nouveaux mots-clés seront introduits au fur et à mesure des besoins, comme par exemple ici ***procédure***. De plus, afin de ne pas obscurcir les concepts de programmation introduits, la structure d'un programme en termes d'unités de compilation ne sera pas précisée ici.

[6] La grammaire figurant en fin de document précise la syntaxe des « deux » langages.

2. APPEL D'UN SOUS-PROGRAMME

Définition

Un appel de sous-programme est une action (instruction) du langage algorithmique qui invoque l'exécution du sous-programme.

L'appel d'un sous-programme consiste donc à exécuter ce sous-programme. On mentionne alors son nom (son identificateur) dans une instruction d'appel. Comme l'appel s'apparente à une action, on parle alors d'action nommée.

Exemple

Si l'on a spécifié le sous-programme de lecture d'une commande, on peut exploiter ce sous-programme dans l'algorithme. De même, pour l'exécution de la commande.

L'en-tête d'un sous-programme précise le protocole d'appel du sous-programme. On invoque le sous-programme en indiquant simplement son nom dans le programme. On écrira en langage algorithmique :

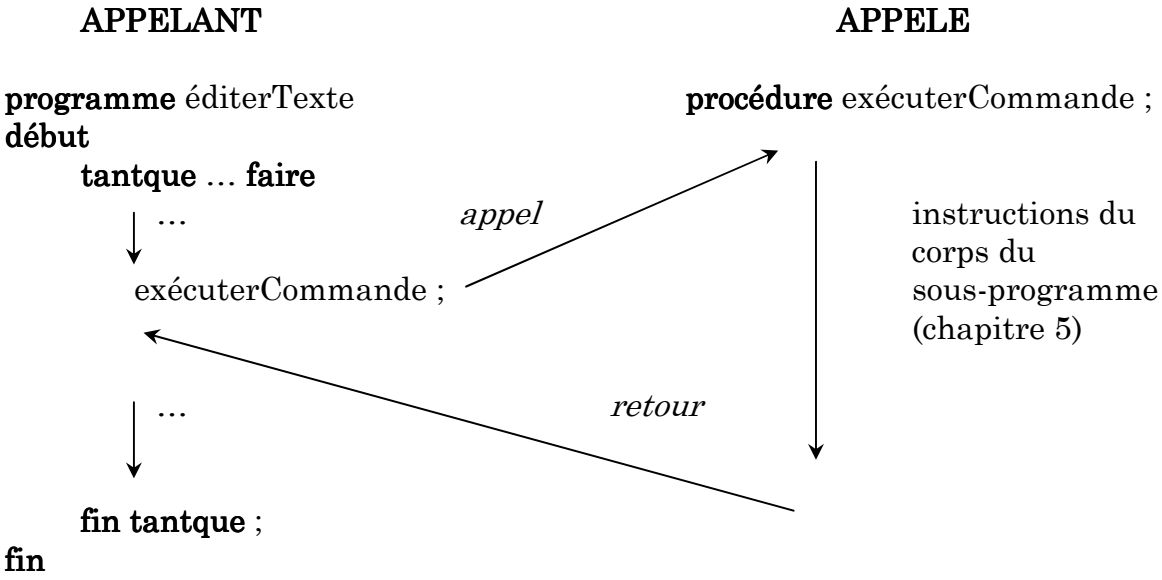
```
programme éditerTexte

début
    …
    tantque ... faire
        -- exécuter la commande
        exécuterCommande ;
        -- afficher le texte
        afficherTexte ;
        -- lire la commande suivante de l'utilisateur
        lireCommande ;
    fin tantque ;
fin
```

Dans cette écriture, le mot-clé ***programme*** introduit un programme en langage algorithmique en spécifiant son nom. Les mots-clés ***début*** et ***fin*** délimitent les actions (instructions) du programme.

A l'appel d'un sous-programme, l'appelant transmet le contrôle au corps du sous-programme appelé, les instructions du corps du sous-programme sont exécutées, puis, le contrôle est rendu à l'appelant.

Sur cet exemple, la procédure *exécuterCommande* sera appelée autant de fois que l'action de la répétition sera exécutée.

On obtient alors le schéma suivant :



Remarque

Pour être capable d'utiliser un sous-programme, on précisera dans l'en-tête son rôle par un court commentaire ; ce commentaire, introduit par les caractères --, est très important car il complète la spécification du sous-programme. Par exemple, on écrira :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande ;
```

3. PARAMETRES D'UN SOUS-PROGRAMME

3.1. Notion de paramètre

Définition

Un paramètre est une information spécifiant le contexte de calcul du sous-programme. Cette information est soit transmise, soit produite par le sous-programme.

Il s'agit donc de pouvoir appliquer un même sous-programme à des contextes de calcul différents.

Exemple

Pour l'algorithme de l'éditeur, on constate que l'exécution de la commande doit considérer la dite commande. Cette commande varie en fonction de l'action à réaliser sur le texte.

Nous devons donc introduire cette commande et ce texte en tant que paramètres du sous-programme. Dans ce cas, l'en-tête du sous-programme *exécuterCommande* inclut deux paramètres spécifiés entre parenthèses : *uneCommande* et *unTexte.* Il s'écrit donc :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande (uneCommande, unTexte) ;
```

De même, pour *créerTexte, lireCommande* et *afficherTexte*, on peut imaginer les spécifications suivantes :

```
-- crée un texte initialement vide
procédure créerTexte (unTexte) ;
```

```
-- lit une commande de l'utilisateur
procédure lireCommande (uneCommande) ;
```

```
-- affiche un texte
procédure afficherTexte (unTexte) ;
```

Le programme d'édition d'un texte devra lire les différentes commandes effectives et les exécuter au fur et à mesure. On écrira par exemple :

```
programme éditerTexte

début
    …
    tantque ... faire
        -- exécuter la commande
        exécuterCommande (laCommande, leTexte);
        -- afficher le texte
        afficherTexte (leTexte) ;
        -- lire la commande suivante de l'utilisateur
        lireCommande (laCommande) ;
    fin tantque ;
fin
```

Dans la spécification du sous-programme, le paramètre introduit est appelé paramètre formel. Dans l'instruction d'appel, le paramètre spécifié remplace le paramètre formel lors de l'activation : c'est un paramètre effectif (ou paramètre réel). Les paramètres formels sont mis en correspondance avec les paramètres effectifs, un à un, de gauche à droite. Dans notre exemple, *uneCommande* et *unTexte* sont des paramètres formels (articles indéfinis), alors que *laCommande* et *leTexte* sont des paramètres effectifs (articles définis).

3.2. Paramètres et variables

Dans le cas général, un paramètre (effectif ou formel) n'est rien d'autre qu'une variable, objet le plus élémentaire manipulé par l'acteur ordinateur. Une variable mémorise à tout instant une ou plusieurs valeurs ; on peut donc la caractériser comme une entité formée de 2 éléments : sa valeur, information destinée à être traitée, et son identificateur (son nom) qui la symbolise dans l'algorithme. On distingue les variables simples et les variables structurées. Une variable simple ne mémorise qu'une valeur alors qu'une variable structurée regroupe plusieurs valeurs sous un même nom

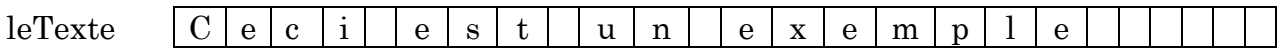
collectif. Pour l'éditeur, *laCommande* est une variable simple (un caractère par exemple) alors que *leTexte* est une variable structurée (qui mémorise en général plusieurs caractères).

On peut représenter schématiquement une variable par un rectangle à côté duquel on écrit son identificateur, le contenu du rectangle exprimant la valeur de la variable. Une variable est donc définie par un doublet (identificateur, valeur).

Exemples

1) La variable simple *laCommande* ayant pour valeur le caractère 'i' (pour insertion) peut se représenter par :

2) La variable structurée *leTexte* mémorise plusieurs caractères, comme dans l'exemple *"Ceci est un exemple"* :



Remarques

1) Les paramètres formels ne servent qu'à l'écriture du traitement réalisé par le sous-programme, encore appelé corps du sous-programme. Dès lors, on les nomme par des identificateurs génériques, comme *uneCommande* et *unTexte.* On dit aussi que ce sont des variables muettes. A l'inverse, les paramètres effectifs expriment précisément le contexte de calcul que l'action liée au sous-programme gère et modifie. On parle ainsi de la *commande* à exécuter pour mettre en forme le *texte* en cours d'édition, soient respectivement *laCommande* et *leTexte*. Noter ainsi l'usage de l'article indéfini (*un*, *une*, …) pour les paramètres formels et défini (*le*, *la*, …) pour des paramètres effectifs.

2) Un paramètre effectif peut avoir le même identificateur que son homologue formel. Il s'agit cependant de deux variables distinctes : l'une appartenant au contexte de l'appelant et l'autre au contexte de l'appelé.

3.3. Modes de transmission des paramètres

Il existe trois utilisations possibles d'un paramètre par un sous-programme : le mode entrée, le mode sortie et le mode mise à jour. Ces modes d'utilisation correspondent à des modes logiques qui déterminent la façon dont le paramètre est utilisé par le sous-programme.

3.3.1. Le mode *entrée*

Définition

C'est une valeur transmise par l'appelant qui sert au calcul et qui n'est pas modifiée par le sous-programme. La valeur transmise est celle du paramètre effectif au moment de l'appel. Le mode entrée (spécifié par le mot-clé ***entrée***) indique que le paramètre est uniquement consulté par le sous-programme.

Exemples

1) Le paramètre *uneCommande* est un paramètre en mode entrée pour le sous-programme *exécuterCommande*. Dans ce cas, l'en-tête spécifie :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande (entrée uneCommande, unTexte) ;
```

2) Il en est de même pour *afficherTexte* :

```
-- affiche un texte
procédure afficherTexte (entrée unTexte) ;
```

3.3.2. Le mode *sortie*

Définition

Le mode sortie (spécifié par le mot-clé ***sortie***) est un paramètre calculé par le sous-programme et transmis à l'appelant en tant que résultat. La valeur transmise est celle du paramètre formel à la fin de l'exécution du sous-programme.

Exemples

1) La procédure *lireCommande* doit acquérir la commande de l'utilisateur. C'est donc un paramètre dont la valeur est déterminée par l'appelé :

```
-- lit une commande de l'utilisateur
procédure lireCommande (sortie uneCommande) ;
```

2) L'éditeur travaille initialement sur un texte vide. C'est au sous-programme *créerTexte* de créer un tel texte :

```
-- crée un texte initialement vide
procédure créerTexte (sortie unTexte) ;
```

3.3.3. Le mode *mise à jour*

Définition

Le mode mise à jour (spécifié par le mot-clé ***màj***) correspond à un paramètre transmis par l'appelant, et modifié par l'appelé avant d'être communiqué à l'appelant.

Exemple

Toute commande agit sur le texte. Celui-ci est fourni au sous-programme *exécuterCommande* qui le modifie (en général), puis le retourne à l'appelant. L'en-tête du sous-programme sera dès lors :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande (entrée uneCommande, màj unTexte) ;
```

3.4. Sémantique des modes de transmission

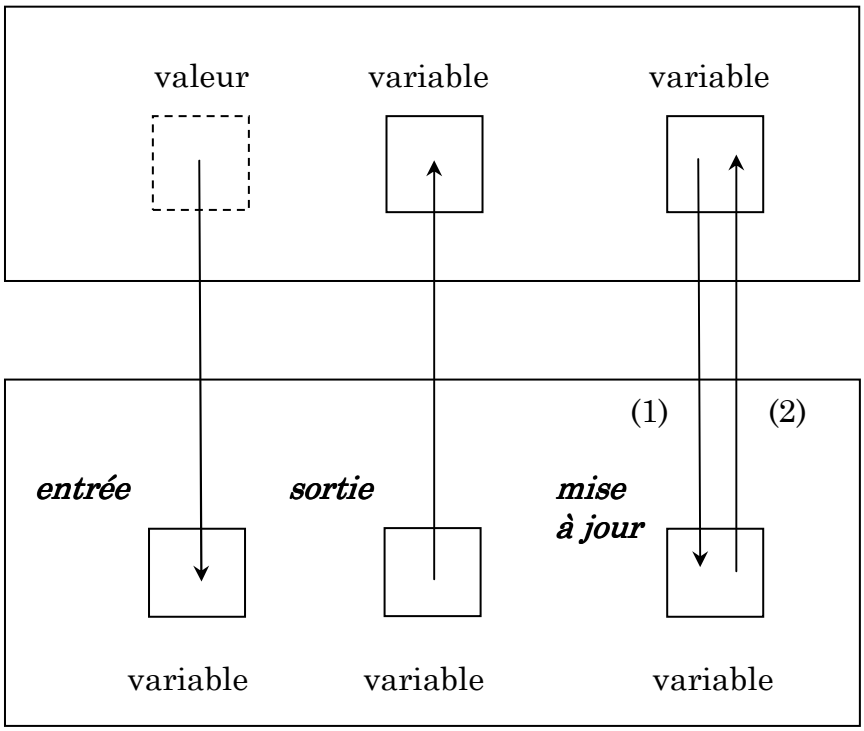
En pratique, un paramètre transmis en mode entrée est considéré comme une constante pour le sous-programme ; la valeur de ce paramètre est déterminée par le paramètre effectif transmis au moment de l'appel du sous-programme. En langage algorithmique, une recopie de la valeur du paramètre effectif dans le paramètre formel est engagée au moment de l'appel.

De la même façon, un paramètre transmis en mode sortie est une valeur calculée par le sous-programme, puis affectée au paramètre effectif transmis ; cette valeur sera donc retournée dans le paramètre effectif après l'exécution du sous-programme. En langage algorithmique, cela se traduit par une recopie de la valeur du paramètre formel dans le paramètre effectif lors du retour.

Pour un paramètre en mode mise à jour, la valeur initiale est fournie à l'entrée par le paramètre effectif et sa valeur finale est celle modifiée par le sous-programme. Cette nouvelle valeur sera donc retournée dans le paramètre effectif après l'exécution du sous-programme. En langage algorithmique, il y a copie à l'appel de la valeur du paramètre effectif dans le paramètre formel (1), et en fin d'exécution du sous-programme, copie de la valeur du paramètre formel dans le paramètre effectif (2).

On peut dès lors représenter ainsi la vue en coupe d'un sous-programme :

programme appelant : paramètres effectifs



sous-programme appelé : paramètres formels

Rappelons que les paramètres effectifs sont mis en correspondance avec les paramètres formels, un à un, et de gauche à droite.

Remarques

1) Un paramètre effectif associé à un paramètre formel en mode sortie ou en mode mise à jour est utilisé pour mémoriser l'effet de l'appel au sous-programme. C'est donc nécessairement une variable de l'appelant. Par contre, pour un paramètre effectif correspondant à un paramètre formel d'entrée, seule sa valeur est utilisée : cette valeur peut donc être définie par le contenu de la variable (sa valeur), une constante ou encore une expression.

Ainsi, supposant qu'en fin de session l'éditeur choisisse de sauvegarder systématiquement le texte, on pourrait, dans le corps du programme *éditerTexte*, rajouter l'appel :

```
-- mémoriser le texte dans un fichier de sauvegarde
-- 's' correspond à la commande de sauvegarde
exécuterCommande ('s', leTexte) ;
```

2) Les paramètres formels d'un sous-programme sont toujours des variables : elles permettent de nommer les informations d'entrée et/ou de sortie de l'appelant.

4. PROCEDURES ET FONCTIONS

4.1 Procédures

Définition

Une procédure (spécifiée par le mot-clé ***procédure***) correspond à l'abstraction d'une action ; elle enrichit le jeu d'actions élémentaires d'un acteur.

Les paramètres effectifs d'une procédure peuvent être transmis indifféremment dans les trois modes entrée, sortie et mise à jour.

Exemple

Comme souligné précédemment, le sous-programme d'exécution d'une commande est une procédure :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande (entrée uneCommande, màj unTexte) ;
```

Puisqu'une procédure s'assimile à une action, son utilisation reste inchangée.

4.2. Fonctions

Définition

Une fonction (spécifiée par le mot-clé ***fonction***) correspond à l'abstraction d'un calcul ; elle enrichit le jeu des valeurs élémentaires de l'acteur.

Les paramètres formels d'une fonction tiennent lieu de constantes locales dont les valeurs sont fournies par les paramètres effectifs ; en conséquence, le mode de transmission des paramètres pour les fonctions est toujours le mode entrée. Cette restriction se justifie par le souhait qu'une même fonction appelée avec les mêmes paramètres doit nécessairement fournir toujours le même résultat, ce qui ne peut être obtenu qu'avec le mode entrée.

Exemple

Considérons la commande de l'éditeur permettant de comptabiliser le nombre de mots du texte. On spécifie ce calcul par la fonction *nombreDeMots* qui fournit, en toutes circonstances, le nombre de mots présents dans le texte. Cette fonction a pour en-tête :

```
-- calcule le nombre de mots d'un texte
fonction nombreDeMots (entrée unTexte) ;
```

On peut ainsi imaginer chez l'appelant *éditerTexte*, de façon concise :

```
-- calculer et comparer le nombre de mots du texte
si nombreDeMots (leTexte) >= 100 alors
        …
fin si ;
```

Avec une procédure, l'écriture s'avère plus rigide car elle nécessite d'introduire un paramètre en mode ***sortie***, dans notre cas *nbMots*, comme indiqué dans l'en-tête :

```
-- calcule le nombre de mots d'un texte
procédure calculerNombreDeMots (entrée unTexte, sortie nbMots) ;
```

Remarquer aussi le changement de nom du sous-programme car l'appel correspond maintenant à un traitement et non à un calcul.

Chez l'appelant *éditerTexte*, la formulation équivalente à la précédente devient :

```
-- calculer le nombre de mots du texte
calculerNombreDeMots (leTexte, nbMots) ;
-- comparer le nombre de mots du texte
si nbMots >= 100 alors
        …
fin si ;
```

Remarque

Un appel de fonction exprime un calcul et ne peut jamais être considéré comme une action en soi, comme l'est une procédure. Noter également que l'exécution d'une fonction produit un résultat « prenant la place de l'appel ». Il serait donc faux d'écrire dans le corps du programme *éditerTexte* :

```
-- calculer le nombre de mots du texte
nombreDeMots (leTexte) ;
```

5. LE CONTRAT APPELANT-APPELE

L'en-tête d'un sous-programme peut être considéré d'un double point de vue :

- celui de l'appelant pour lequel il représente la fonctionnalité attendue,
- celui de l'appelé pour lequel il constitue le traitement à remplir.

Ce double point de vue lie l'appelant et l'appelé par un contrat. Ceci permet notamment de séparer logiquement l'utilisation d'un sous-programme et sa réalisation effective. De plus, dès lors qu'un appel de sous-programme correspond à un traitement (action pour une procédure et calcul pour une fonction), on peut formaliser le contrat appelant-appelé à l'aide d'une précondition et d'une postcondition.

Le contrat appelant-appelé s'exprime par une précondition (balise ***nécessite***) et une postcondition (balise ***entraîne***) traduisant l'effet, sur le contexte courant de calcul, de l'action subordonnée à un appel du sous-programme. La précondition, ou assertion d'entrée, définit le domaine des valeurs d'entrée des paramètres et précise donc les

obligations de l'appelant. La postcondition, ou assertion de sortie, décrit les relations établies par le sous-programme entre les paramètres d'entrée et de sortie et donc précise les engagements de l'appelé.

Définition

L'en-tête d'un sous-programme, aussi appelé spécification du sous-programme, définit l'interface, ou protocole d'appel, entre le sous-programme et le monde extérieur. Un en-tête de sous-programme en langage algorithmique fournira les informations suivantes : la nature du sous-programme (procédure ou fonction), son nom, son rôle en un court commentaire incluant éventuellement sa précondition (clause ***nécessite***) et sa postcondition (clause ***entraîne***), ses paramètres en entrée, ses paramètres en sortie, ses paramètres en mise à jour.

Exemples

1) Soit *taille* une fonction applicable à un texte qui détermine son nombre de caractères. La postcondition *taille (unTexte) = 0* du sous-programme *créerTexte* exprime le fait qu'aucun caractère n'a encore été mémorisé par *unTexte* :

```
-- crée un texte initialement vide
-- entraîne taille (unTexte) = 0
procédure créerTexte (sortie unTexte) ;
```

2) La commande d'insertion d'un caractère *c* au rang *i* dans le *texte* admet pour en-tête :

```
-- insère le caractère c dans le texte unTexte au rang i
-- nécessite 1 ≤ i ≤ taille (unTexte) + 1
-- entraîne taille (unTexte') = taille (unTexte) + 1 et unTexte'i = c
procédure insérerCaractère (màj unTexte, entrée i, entrée c) ;
```

Dans cet énoncé, *unTexte* référence le texte avant modification et *unTexte'* après. De façon similaire, $unTexte_i$ exprime la valeur du *ième* caractère du texte avant sa mise à jour et $unTexte'_i$ après.

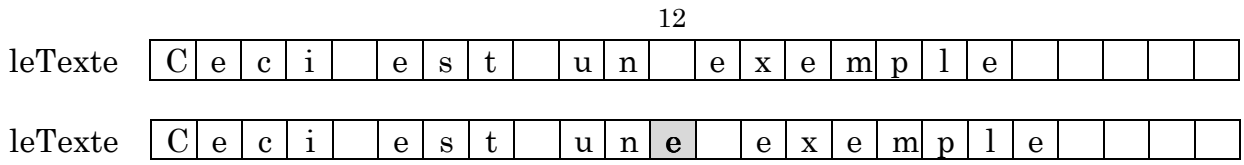
En pratique, la postcondition doit aussi indiquer ce que sont devenus les caractères du texte autres que celui inséré :

```
-- entraîne taille (unTexte') = taille (unTexte) + 1
--     et ∀k ∈ [1 .. i−1], unTexte'k = unTexte k
--     et unTexte'i = c
--     et ∀k ∈ [i+1 .. taille (unTexte')], unTexte'k = unTexte k-1
```

On peut paraphraser la postcondition come suit :

- les caractères du texte avant le rang d'insertion *i* ne sont pas modifiés,
- le caractère *c* est inséré au rang *i* du texte,
- les caractères situés à partir du rang d'insertion *i* sont décalés vers la droite.

Pour l'appel spécifique *insérerCaractère (leTexte, 12, 'e')*, le texte évolue comme ci-dessous où la première ligne représente le texte avant insertion et la seconde après. On peut ainsi vérifier sur ce schéma que le paramètre effectif *leTexte* est modifié conformément à la postcondition de la procédure *insérerCaractère*.



3) Pour une fonction déterminant le nombre de caractères d'un mot du texte, on pourra spécifier :

```
-- détermine le nombre de lettres du mot délimité
-- par les rang i (début de mot) et j (fin de mot) du texte unTexte
-- nécessite 1 ≤ i ≤ j ≤ taille (unTexte) et ∀k∈[i .. j],  lettre (unTexte k) = vrai
-- entraîne résultat = j - i + 1
fonction nbLettres (entrée unTexte, entrée i, entrée j) ;
```

où la pseudo-variable ***résultat*** dans la postcondition incarne le résultat de la fonction *nbLettres*.

Remarque

La précondition porte sur les paramètres d'entrée. La postcondition exprime le traitement réalisé par le sous-programme en termes de relations mettant en jeu les paramètres transmis en mode sortie ou en mode mise à jour ; pour cela, elle peut faire référence aux valeurs transmises en entrée.

Chapitre 4 : Variables et types

Dans ce chapitre, nous présentons l'objet de base manipulé par l'acteur ordinateur : la variable. Nous lui associons ensuite la notion de type qui caractérise l'ensemble de ses valeurs et l'ensemble de ses opérations.

Dans l'appel suivant :

exécuterCommande (laCommande, le Texte)

laCommande et *leTexte* sont des paramètres effectifs. Ce sont aussi des variables : *laCommande* est une variable élémentaire et *leTexte* est une variable structurée.

1. LES VARIABLES ELEMENTAIRES

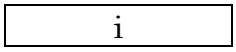
Définition

Une variable élémentaire est l'objet le plus simple manipulé par l'acteur ordinateur. Elle représente implicitement une valeur ; à chaque instant, une variable élémentaire ne peut représenter qu'une seule valeur. On désigne une variable par un nom appelé identificateur de la variable ; à ce nom est associé un emplacement de mémoire, dispositif physique permettant la sauvegarde et la restitution de valeurs (voir Cours d'Architecture des ordinateurs).

On peut représenter schématiquement une variable par un rectangle à côté duquel on écrit son identificateur, le contenu du rectangle exprimant la valeur de la variable. Une variable est donc définie par un doublet (identificateur, valeur).

Exemple

La variable *laCommande* ayant pour valeur le caractère 'i' peut se représenter par :



Remarques

1) Les variables d'un programme en langage algorithmique sont déclarées au sein d'un glossaire (mot-clé ***glossaire***). La réservation d'un emplacement de mémoire pour une variable est effective dès que la variable est déclarée.

2) Toute variable utilisée dans un programme algorithmique doit être déclarée, une fois et une seule ; une déclaration n'affecte pas le contenu de l'emplacement de mémoire associé à la variable.

3) Les variables désignant des paramètres formels de sous-programme sont implicitement déclarés par l'en-tête du sous-programme. Leur contenu est géré par le mode de transmission choisi (entrée, sortie ou mise à jour).

4) En mathématiques, les noms des variables sont souvent formés d'une seule lettre. Dans les algorithmes, cette restriction n'existe pas et les noms des variables sont aussi explicites que possible afin de favoriser la compréhension du texte.

2. LES VARIABLES STRUCTUREES

Reprenons l'appel :

exécuterCommande (laCommande, leTexte)

Il apparaît que la variable *leTexte* n'est pas une variable élémentaire dans la mesure où elle ne mémorise pas simplement un caractère, mais plusieurs !

C'est ainsi qu'il est parfois nécessaire de regrouper logiquement des variables élémentaires. Ceci se vérifie particulièrement en programmation descendante où les données et les traitements sont affinés simultanément.

Pour cela, l'acteur ordinateur propose deux constructeurs de données : le tableau et l'enregistrement (appelé aussi article ou structure).

2.1. Le *tableau*

Définition

Un tableau est une séquence de longueur fixe d'éléments de même nature dans laquelle chaque élément est identifié selon sa position (encore appelé indice).

Exemple

Le tableau *leTexte* de 1000 caractères mémorisant la chaîne de caractères *"Ceci est un exemple"* peut se représenter comme suit :

	1	2	3	4	5	…																		1000
leTexte	C	e	c	i		e	s	t		u	n		e	x	e	m	p	l	e						

Au chapitre 5 (Corps d'un sous-programme), nous montrerons comment accéder en consultation ou en modification à un élément de tableau.

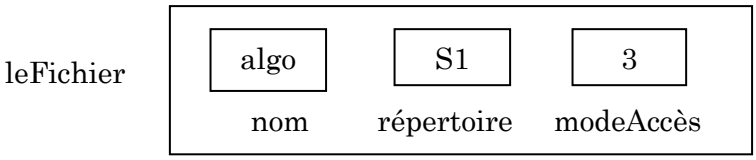
2.2. L'*enregistrement*

Définition

Un enregistrement (encore appelé article ou structure) est un regroupement fini de composants accessibles directement par l'intermédiaire d'une sélection. L'intérêt d'un tel constructeur est de pouvoir manipuler globalement tout un ensemble structuré et cohérent d'informations.

Exemple

Soit la variable *leFichier* désignant un fichier en mémoire secondaire dont le contenu est susceptible d'être chargé dans le tableau *leTexte* de l'éditeur. En cours d'exécution, cette variable pourra prendre des valeurs pour chacun de ses composants. Par exemple, la chaîne de caractères ″algo″ pour le nom du fichier, la chaîne ″S1″ pour le nom du répertoire et l'entier 3 pour le mode d'accès en mise à jour, ce que l'on traduira schématiquement :



Au chapitre 5 (Corps d'un sous-programme), nous montrerons comment accéder en consultation ou en modification à un composant (ou champ) d'un enregistrement.

3. LES TYPES

A une variable est associée une valeur, mais aussi un type. Une variable est donc un triplet (identificateur, valeur, type).

Définition

Le type d'une variable caractérise son ensemble de valeurs et son ensemble d'opérations.

Exemple

Pour la variable caractère *laCommande* (on dit que la variable *laCommande* est de type *Caractère*), le type *Caractère* précise l'ensemble des valeurs possibles de la variable (les caractères) et l'ensemble des opérations qu'on peut lui appliquer (l'égalité avec un autre caractère, le prédécesseur dans l'alphabet, le successeur, ...).

3.1. Les types élémentaires

Le tableau suivant résume les principaux types élémentaires que l'on rencontre dans la plupart des langages, dits aussi types prédéfinis ; il s'agit essentiellement des entiers, des réels, des caractères et des booléens. En langage algorithmique, on introduira donc les types *Entier*, *Réel*, *Caractère* et *Booléen* correspondants spécifiés comme suit pour leurs ensembles de valeurs et d'opérations :

Type	Valeurs	Opérations[7]
Entier	$min_{entiers}$, ..., 0, ... $max_{entiers}$	+, -, *, /, <, >, =, div, mod, …
Réel	$min_{réels}$, ..., 0.0, ... $max_{réels}$	+, -, *, /, <, >, =…
Caractère	'A', 'B', ..., 'Z', 'a', …, 'z', …	pred, succ, pos, val, <, =, …
Booléen	VRAI, FAUX	non, et, ou, …

[7] div : division entière, mod : reste de la division entière, pred : prédécesseur, succ : successeur, pos : rang du caractère, val : valeur du caractère de rang donné.

A partir de cette définition des types élémentaires de données, le programmeur peut définir des variables particulières ; par exemple *laCommande* (de type *Caractère*) et le composant *modeAccès* de l'enregistrement *leFichier* (de type *Entier*).

On dit alors que ces variables sont des représentants ou des instances du type. Ainsi, implicitement, la variable *laCommande* de type *Caractère*, pourra prendre une valeur comprise entre 'A' et 'z'. De même, on pourra lui appliquer une des opérations qui caractérisent le type, en écrivant par exemple *pred ('B')* qui fournira le caractère *'A'*.

En langage algorithmique, un type sera attaché à toute variable déclarée dans un glossaire. La syntaxe utilisée sera *v* <*T*> où *v* désigne une variable, <> introduit la marque de type et se lit *est de type*, et *T* désigne le type de la variable.

Exemple

Le glossaire ci-dessous (mot clé ***glossaire***) déclare la variable *laCommande*. La déclaration mentionne son nom et son type ; elle s'accompagne, par convention, d'un court commentaire qui précise son rôle dans l'algorithme.

```
glossaire
    laCommande <Caractère> ;        -- la commande à exécuter
```

Remarque

Une déclaration de variable n'affecte pas le contenu de l'emplacement de mémoire qui lui est associé. Ainsi, l'effet de cette déclaration peut se concrétiser comme suit :

où le symbole « ? » indique que son contenu est encore indéterminé : l'emplacement pour la variable a été réservé, mais aucune valeur ne lui a été encore affectée.

Au chapitre 5 (Corps d'un sous-programme), nous expliciterons le moyen de donner une valeur à une variable.

3.2. Les types composés

Les deux principaux constructeurs de type sont le constructeur tableau et le constructeur enregistrement (appelé aussi article ou structure). Il existe aussi le constructeur pointeur (voir Cours de Structures de Données) et le constructeur fichier (voir Cours de Système de Gestion de Fichiers). Noter l'analogie entre le constructeur de types (données) et la composition d'actions (traitement).

3.2.1. Le type *tableau*

Pour déclarer un représentant du type tableau, c'est-à-dire une variable structurée tableau, il conviendra de définir auparavant le type correspondant, en

utilisant conjointement les mots-clés ***type*** et ***tableau*** où le mot-clé ***type*** introduit toujours une définition de type.

Exemple

Pour déclarer la variable *leTexte*, on doit définir préalablement son type, c'est-à-dire son modèle. Soit *TabCaractères* ce type décrivant une zone tampon où sont mémorisés les caractères du texte : un tableau de 1000 caractères.

```
-- définition du type TabCaractères
type TabCaractères : tableau [1 à 1000] de <Caractère> ;

glossaire
    -- déclaration de la variable leTexte
    leTexte <TabCaractères> ;
```

Dans la définition du type *TabCaractères*, la notation *[1 **à** 1000]* définit le domaine d'indice du tableau *leTexte*, c'est-à-dire la borne inférieure de l'indice (valeur 1) et sa borne supérieure (valeur 1000). Ce type définit donc un modèle de 1000 variables élémentaires désignées par un indice variant de 1 à 1000.

Noter aussi que chaque élément du tableau *leTexte* est de type *Caractère*, et donc qu'on peut lui appliquer toutes les opérations des caractères... C'est aussi cette spécification qu'apporte le typage.

Remarques

1) Comme pour une variable élémentaire, la déclaration d'un tableau n'affecte pas son contenu. Une fois la déclaration de la variable *leTexte* enregistrée, son contenu peut se schématiser comme suit :

	1	2	3	4	5	…																			1000
leTexte	?	?	?	?	?	..																			?

2) L'ensemble des valeurs possibles d'un tableau se définit comme l'union des valeurs possibles de chacun de ses éléments. Parmi les opérations applicables à un tableau, détaillées au chapitre 5 (Corps d'un sous-programme), figure l'accès en consultation et en modification à un de ses élément.

3.2.2. Le type *enregistrement*

Comme pour une variable structurée tableau, déclarer un représentant du type enregistrement, c'est-à-dire une variable structurée enregistrement, nécessite de définir auparavant le type correspondant, en utilisant conjointement les mots-clés ***type*** et ***enregistrement***.

Exemple

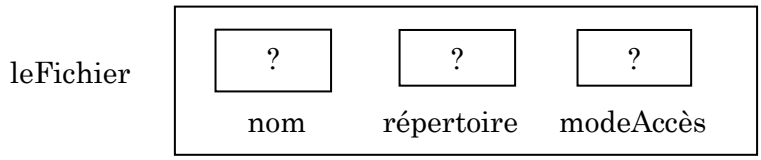
Pour déclarer la variable *leFichier*, il convient d'écrire :

```
-- définition du type EnrFichier
type EnrFichier : enregistrement
        nom <Chaîne>,
        répertoire <Chaîne>,
        modeAccès <Entier> ;

-- définition du type Chaîne
constante LG_MAX <Entier> = 25 ;
type Chaîne : tableau [1 à LG_MAX] de <Caractère> ;

glossaire
        -- déclaration de la variable leFichier
        leFichier <EnrFichier> ;
```

Remarquer que l'on spécifie aussi les types des champs de l'enregistrement, toujours avec la notation <>, et que cet exemple induit la définition du type *Chaîne* : un tableau de *LG_MAX* caractères, où *LG_MAX* représente une constante quelconque introduite par le mot-clé ***constante***.

Remarques

1) La déclaration d'un enregistrement n'affecte pas son contenu. En mémoire, la variable *leFichier* a l'issue de la déclaration se présente comme suit :



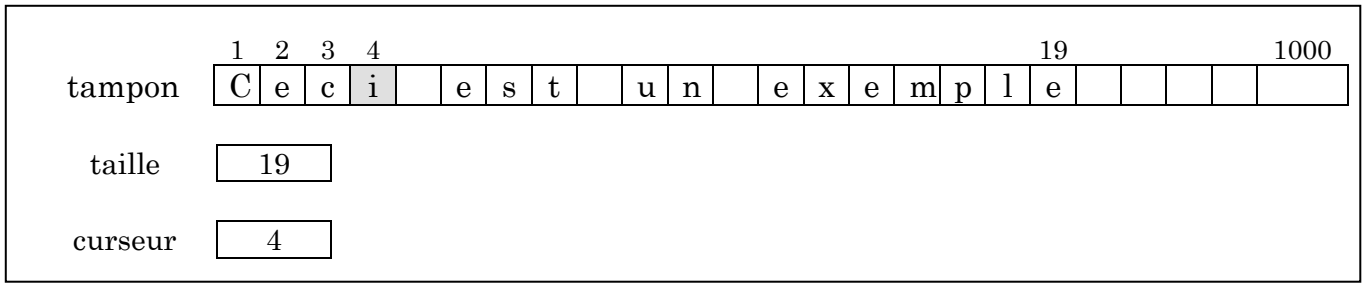
2) L'ensemble des valeurs possibles d'un enregistrement se définit comme l'union des valeurs possibles de chacun de ses composants. Parmi les opérations applicables à un tableau, détaillées au chapitre 5 (Corps d'un sous-programme), figure l'accès en consultation et en modification à un de ses composants.

3) En pratique, la donnée mémorisant le texte de l'éditeur ne se limite pas un tableau de caractères. Il s'agit plutôt d'un enregistrement mémorisant aussi la taille courante du texte en nombre de caractères (au plus égale à 1000) et la position du curseur c'est-à-dire ici un entier (compris entre 1 et 1000) qui repère un caractère du texte. Implicitement, la commande agit à la position courante du curseur. Avec cette convention, il vient :

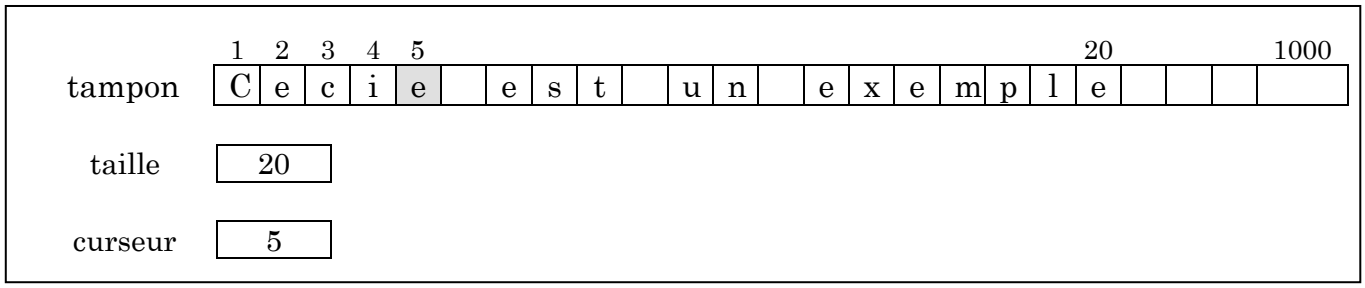
```
-- définition du type EnrTexte
type EnrTexte : enregistrement
        tampon <TabCaractères>,
        taille <Entier>,
        curseur <Entier> ;
```

```
-- définition du type TabCaractères
constante TAILLE_MAX <Entier> = 1000 ;
type TabCaractères : tableau [1 à TAILLE_MAX] de <Caractère> ;

glossaire
    -- déclaration de la variable leTexte
    leTexte <EnrTexte> ;
```

Voici un exemple de représentation en mémoire de la variable *leTexte* telle que sa zone tampon mémorise le texte *"Ceci est un exemple"* composé de 19 caractères. On suppose de plus que le curseur est positionné au 4ème caractère du texte.



Après insertion du caractère 'e' à droite du curseur, on obtient un nouvel état mémoire conforme au schéma ci-dessous où la zone tampon mémorise *"Cecie est un exemple"*, soit un texte de 20 caractères. De plus, le curseur référence le caractère inséré, soit le 5ème caractère de la zone tampon.



Noter également que le rang d'insertion est ici fourni par la valeur du champ *curseur* de l'enregistrement *leTexte*. L'appel *insérerCaractère*, contrairement à la version du chapitre précédent, peut dès lors se limiter à deux paramètres : *leTexte* et le caractère à insérer.

3.3. Typage des paramètres formels

Comme le langage algorithmique est typé, un type sera désormais attaché à tout paramètre formel d'un en-tête de sous-programme. La syntaxe utilisée sera *v <T>* où *v* désigne un paramètre formel et *T* son type. De plus, on précisera le type du résultat délivré par une fonction par le mot-clé ***retourne*** dans l'en-tête suivi d'une marque de type.

Exemples

1) Pour exécuter une commande, on précisera les types des paramètres formels (et donc les types des paramètres effectifs attendus) en écrivant :

```
-- exécute une commande de l'utilisateur
procédure exécuterCommande
      (entrée uneCommande <Caractère>, màj unTexte <EnrTexte>) ;
```

2) Pour le calcul du nombre de mots du texte, on spécifiera :

```
-- calcule le nombre de mots d'un texte
fonction nombreDeMots (entrée unTexte <EnrTexte>)
retourne <Entier> ;
```

4. INTERET DES TYPES

L'intérêt du typage en programmation concerne au moins trois points : la lisibilité du code, sa vérification et son efficacité. Les deux premiers sont importants en algorithmique ; le dernier est en rapport avec la compilation (voir Cours de Traduction des Langages).

4.1. La lisibilité du code

La notion de type permet de mieux comprendre ce que l'on fait. Typer, c'est déjà spécifier.

Exemples

1) Soient *test* de type *Booléen* et *a* de type *Entier*, l'expression *test + a* n'a pas de sens, car l'opération d'addition entre les booléens et les entiers n'est pas définie. En revanche, si *a* et *b* sont deux variables de type *Entier*, on sait donner une sémantique précise à l'expression *a + b* (c'est celle que l'on connaît depuis l'école maternelle...). D'où une meilleure interprétation du code grâce au typage.

2) Si l'on a défini une fonction *f* munie d'un paramètre formel de type *Entier* et retournant un *Réel*, l'on sait que pour tout appel *f (x), x* est de type *Entier* et *f (x)* est de type *Réel*. C'est donc une information importante qui spécifie le calcul *f (x).*

4.2. La vérification du code

Avec le typage, on peut s'assurer qu'une instruction, une procédure ou une fonction ne peut s'appliquer qu'à des paramètres d'un type autorisé.

Exemples

1) L'addition *a + b* avec *a* et *b* de type *Entier* sera acceptée, alors que *test + a* avec *test* de type *Booléen* et *a* de type *Entier* sera rejetée. Sans la notion de type, cette dernière écriture conduirait à une erreur détectée à l'exécution.

2) Si l'on a spécifié une procédure *échanger* avec deux paramètres formels *x* et *y* de type *Caractère*, on pourra vérifier que les paramètres effectifs sont de type *Caractère* (et pas d'un autre type).

Remarques

1) Un contrôle de type est dit statique s'il est effectué à la compilation du programme. Pouvoir effectuer un contrôle de type statique permet de détecter des erreurs de programmation au plus tôt, plus facilement, car on n'exécute pas le programme.

2) Un langage est dit fortement typé si son système de types permet de vérifier le bon usage (du point de vue des types) de toutes les variables et de toutes les opérations d'un programme. Par exemple, l'addition à des entiers, la conjonction à des booléens, l'association d'un paramètre formel et d'un paramètre effectif de même type lors d'un appel de procédure, ...

L'assembleur, Lisp et Prolog sont des exemples de langages non typés. Parmi les langages typés, on trouve Pascal et C (voir Cours de Programmation structurée en C++) ; les langages réellement fortement typés sont rares : Ada et ML par exemple. C n'est pas fortement typé, car il autorise notamment le mélange de variables de types différents dans un même calcul d'expression et effectue implicitement des conversions de type.

4.3. L'efficacité du code

En indiquant par un type l'ensemble des valeurs possibles que peut prendre une variable, on indique aussi la quantité d'information nécessaire pour coder une valeur. On peut optimiser l'espace mémoire nécessaire et les opérations de codage et de décodage des valeurs de cet ensemble. Avec un typage fort, on peut aussi produire un code plus efficace (voir Cours de Traduction des Langages) car certaines opérations de conversion ne sont plus nécessaires à l'exécution.

Exemples

1) En C (voir Cours de Programmation structurée en C++), on associe 2 octets à une variable de type ***short***, 4 octets pour un type ***long*** et 8 octets pour un type ***double***.

2) Sans typage fort, l'expression *a + b* avec *a* de type *Réel* et *b* de type *Entier* oblige souvent à convertir (accidentellement ou volontairement) à l'exécution *b* en type *Réel* pour que la valeur calculée soit un réel.

Chapitre 5 : Corps d'un sous-programme

Dans ce chapitre, on s'intéresse à l'écriture des actions avec l'acteur ordinateur qui mettent en œuvre la spécification d'un sous-programme. Cette description du sous-programme est appelée corps du sous-programme (*comment ?*) ; un corps de sous-programme doit respecter la spécification correspondante (*quoi ?*).

Définition

Un corps de sous-programme est une mise en œuvre, par l'acteur ordinateur, de l'en-tête du sous-programme. Lorsqu'on écrit un corps de sous-programme, on dit aussi que l'on donne une définition de ce sous-programme.

Exemple

Ayant spécifié le sous-programme *échanger* comme ci-dessous, il s'agit d'en donner une mise en œuvre, ce qui nous conduit à décrire dans ce chapitre les instructions élémentaires de l'acteur ordinateur.

```
-- échange le contenu des deux caractères c1 et c2
-- entraîne c1'=c2 et c2'=c1
procédure échanger (màj c1 <Caractère>, màj c2 <Caractère>) ;
```

1. VARIABLES ELEMENTAIRES ET AFFECTATION

La principale instruction élémentaire de l'acteur ordinateur est l'affectation.

Définition

L'affectation donne une nouvelle valeur à une variable.

Syntaxe

La syntaxe de l'affectation est *v* <- *e* où *v* désigne une variable et *e* une expression contenant des valeurs et/ou des variables. Une expression *e* est soit :

- une constante,
- une variable élémentaire,
- un appel de fonction,
- un calcul formé d'opérandes (constante, variable élémentaire ou appel de fonction) et d'opérateurs.

Les principaux opérateurs sont :

- les opérateurs arithmétiques (+, -, *, /, ...),
- les opérateurs relationnels (=, /=, <, >, <=, ...),
- les opérateurs booléens (non, et, ou).

Sémantique

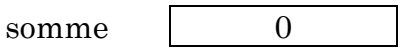
La sémantique d'une affectation *v* <- *e* consiste à calculer la valeur de l'expression *e* du membre droit de l'affectation, puis à modifier la valeur de la variable *v* spécifiée en membre gauche par la valeur de l'expression ainsi calculée.

Exemples

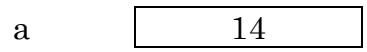
Soient les variables *somme*, *a*, *b*, *c*, *d* de type *Entier* et la variable *test* de type *Booléen*, et considérons les actions :

```
somme <- 0 ;
a <- b + c * d ;
test <- a = b ;
somme <- somme + 1 ;
nbMots <- nombreDeMots (leTexte) ;
```

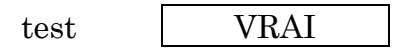
L'affectation *somme <- 0* se traduit par le schéma suivant, quelle que soit la valeur antérieure de la variable *somme* :



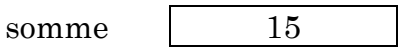
Pour l'affectation *a <- b + c * d*, l'expression *b + c * d* est évaluée, puis sa valeur devient la nouvelle valeur de la variable *a*. Par exemple, si *b = 2*, *c = 3* et *d = 4*, on a :



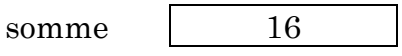
L'affectation *test <- a = b* commence par évaluer l'expression *a = b*. Le résultat de cette évaluation est ensuite affecté à la variable *test*, qui peut donc prendre pour valeur *VRAI* ou *FAUX*. Si *a = b*, *test* reçoit la valeur *VRAI*, sinon, *test* reçoit la valeur *FAUX*. Par exemple, si *a = b = 3*, alors :



L'affectation *somme <- somme + 1*, on commence par considérer la valeur de la variable *somme*, on lui ajoute ensuite la valeur entière 1 ; puis, le résultat devient la nouvelle valeur de la variable *somme*. En supposant l'état initial suivant :



on obtient l'état final :



Dans le dernier exemple *nbMots <- nombreDeMots (leTexte)*, le membre droit de l'affectation est un appel à la fonction *nombreDeMots* avec le paramètre effectif

leTexte. Cet appel fournit un résultat qui devient la nouvelle valeur de la variable *nb*. Ainsi, si le texte se compose de 513 mots, on obtient :

nbMots	513

On remarque donc que l'affectation <- peut aussi se lire « reçoit » ou « devient ».

Remarques

1) Lorsqu'une variable apparaît en membre droit d'une affectation, c'est que l'on suppose qu'elle contient déjà une valeur ; sinon on dira que sa valeur est indéfinie, et l'affectation n'a pas de sens. Ceci n'est en général pas détecté par les langages de programmation.

2) Une affectation ne traduit pas une identité, comme l'égalité. De même, il n'y a aucune symétrie entre les deux membres d'une affectation. Toutefois, la variable cible et l'expression source doivent avoir le même type.

3) Le membre gauche d'une affectation est toujours une variable !

4) Le membre droit d'une affectation peut inclure un appel de fonction, comme dans le dernier exemple *nbMots <- nombreDeMots (leTexte)*.

5) A une variable est associé un emplacement de mémoire (une case, un tiroir) dans lequel on peut placer, au cours du déroulement du programme, des valeurs. Contrairement à son appellation, une variable ne change pas, seul son contenu peut évoluer.

2. CORPS D'UN SOUS-PROGRAMME

2.1. Corps d'une procédure

Définition

Le corps d'une procédure comprend :
- l'en-tête de la procédure (sa spécification),
- le glossaire des variables locales de la procédure,
- les instructions de la procédure, délimitées par les mots-clés ***début*** et ***fin***.

Exemple

L'en-tête du sous-programme *échanger* était :

```
-- échange le contenu des deux caractères c1 et c2
-- entraîne c1'=c2 et c2'=c1
procédure échanger (màj c1 <Caractère>, màj c2 <Caractère>) ;
```

Son corps est le suivant :

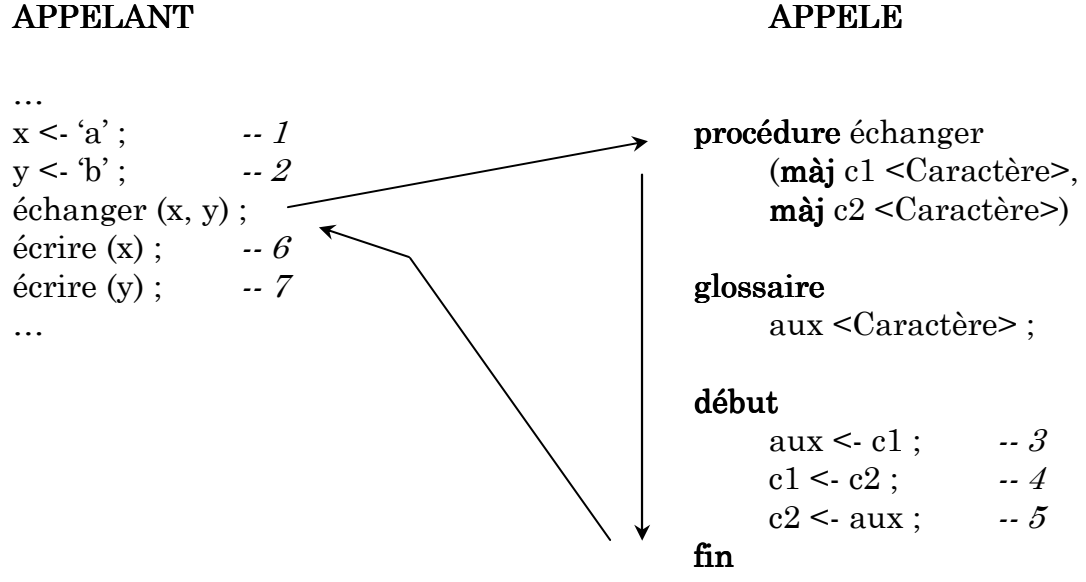
```
-- échange le contenu des deux caractères c1 et c2
-- entraîne c1'=c2 et c2'=c1
procédure échanger (màj c1 <Caractère>, màj c2 <Caractère>)

glossaire
        aux <Caractère> ;          -- variable locale pour réaliser l'échange

début
        aux <- c1 ;
        c1 <- c2 ;
        c2 <- aux ;
fin
```

Dans cette écriture, *c1* et *c2* sont des paramètres formels transmis en mode mise à jour. La variable *aux*, nécessaire pour réaliser l'échange, est dite locale au sous-programme car elle n'est accessible qu'au sein des instructions du sous-programme.

Ainsi, par exemple, si l'on considère le fragment de code suivant :

```
glossaire
        x <Caractère> ;            -- premier paramètre effectif de l'échange
        y <Caractère> ;            -- deuxième paramètre effectif de l'échange

début
        …
        x <- 'a' ;
        y <- 'b' ;
        échanger (x, y) ;
        écrire (x) ;
        écrire (y) ;
        …
fin
```

on obtient le schéma d'appel :



En considérant les valeurs des paramètres effectifs *x* et *y* définies par l'appelant :

x	a	-- *1*
y	b	-- *2*

la procédure *échanger* réalise leur échange sur leur copie d'entrée *c1* = 'a' et *c2* = 'b' :

c1	~~a~~ b	-- *4*
c2	~~b~~ a	-- *5*
aux	~~?~~ a	-- *3*

puis, les 2 appels à la procédure *écrire* écrivent les nouvelles valeurs de *x* et de *y,* après copie en sortie des valeurs des paramètres formels *c1* = 'b' et *c2* = 'a' dans les paramètres effectifs correspondants :

x	b	-- *6*
y	a	-- *7*

Dans cet exemple la procédure *écrire* est spécifiée comme suit :

```
-- écrit un caractère e
procédure écrire (entrée c <Caractère>) ;
```

2.2. Corps d'une fonction

Définition

Comme pour une procédure, le corps d'une fonction comprend :
- l'en-tête de la fonction (sa spécification),
- le glossaire des variables locales de la fonction,
- les instructions de la fonction, délimitées par les mots-clés ***début*** et ***fin***.

Le résultat retourné par la fonction est précisé par l'instruction ***retourner*** qui accepte en argument un nom de variable ou une expression. De plus, cette instruction restitue le contrôle à l'appelant.

Exemple

Soit la spécification du sous-programme *maximum* suivant de type fonction qui détermine le plus grand de deux entiers :

```
-- retourne le plus grand des deux entiers x et y
-- entraîne x > y ⇒ résultat = x et x ≤ y ⇒ résultat = y
fonction maximum (entrée x <Entier>, entrée y <Entier>)
retourne <Entier> ;
```

Son corps s'écrit :

```
-- retourne le plus grand entier des deux entiers x et y
-- entraîne x > y ⇒ résultat = x et x ≤ y ⇒ résultat = y
fonction maximum (entrée x <Entier>, entrée y <Entier>)
retourne <Entier>

début
    si x > y alors
        retourner (x) ;
    sinon
        retourner (y) ;
    fin si ;
fin
```

Pour cette fonction, l'expression retournée est soit le contenu du paramètre formel *x*, soit le contenu du paramètre formel *y*.

3. REGLES D'ACCES AUX VARIABLES

3.1. Règles d'accès aux paramètres formels

Dans le corps d'un sous-programme, un paramètre en mode entrée n'est accessible qu'en consultation (lecture seulement). Il peut donc être opérande d'une expression et n'apparaît qu'en membre droit d'un symbole affectation (<-). Par exemple, pour la fonction *maximum*, les paramètres formels *x* et *y* apparaissent dans l'expression *x > y*.

Dans le corps d'un sous-programme, un paramètre en mode sortie ou mise à jour est accessible à la fois en consultation et en modification (lecture et écriture). Il peut donc apparaître indifféremment de part et d'autre d'un symbole d'affectation (<-). Ainsi, pour la procédure *échanger*, les paramètres formels *c1* et *c2* sont accédés à la fois en consultation et en modification.

Remarque

Pour éviter les problèmes d'accès en lecture ou en écriture aux paramètres formels d'un sous-programme, ne pas transformer tous les modes de transmission en entrée en des modes mise à jour ! Par exemple, il est faux de remplacer :

```
-- détermine le nombre de lettres du mot délimité
-- par les rang i (début de mot) et j (fin de mot) du texte unTexte
fonction nbLettres
    (entrée unTexte <TabCaractères>,
    entrée i <Entier>, entrée j <Entier>)
retourne <Entier> ;
```

par :

```
-- détermine le nombre de lettres nbLettres du mot délimité
-- par les rang i (début de mot) et j (fin de mot) du texte unTexte
procédure calculerNbLettres
        (màj unTexte <TabCaractères>,
        màj i <Entier>, màj j <Entier>, sortie nbLettres <Entier>) ;
```

car le sous-programme *nbLettres* ne doit pas logiquement être autorisé à modifier le texte *unTexte*, ni les rang *i* et *j* qui délimitent un mot. Pire encore, ce sous-programme qui devait être spécifié par une fonction l'est maintenant par une procédure à cause de la transmission des paramètres en mode mise à jour !

3.2. Règle d'accès aux variables locales

Une variable locale est accessible en consultation et en modification, tout comme un paramètre en mode sortie ou mise à jour. De plus, au moment de l'invocation du sous-programme, les variables locales du sous-programme appelé ne sont pas initialisées. C'est donc au programmeur à gérer ces variables. La variable *aux* du sous-programme *échanger* est typique de ce cas de figure.

4. CORPS AVEC VARIABLES STRUCTUREES

4.1. Opérations du type *tableau*

Soit l'en-tête :

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
fonction estUnMot
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen> ;
```

où le type *TabCaractères* est défini comme suit :

```
constante TAILLE_MAX <Entier> = 10000 ;
type TabCaractères : tableau [1 à TAILLE_MAX] de <Caractère> ;
```

Ecrivons maintenant le corps de la fonction *estUnMot*. Comme *TabCaractères* définit un modèle de tableau, nous devons connaître les opérations applicables à un tableau. Par abus de langage, on appelle opération d'un type, toute opération applicable à une variable du type.

En algorithmique, les opérations sur une variable de type tableau sont l'accès à un élément (notation []), et les opérations globales d'affectation (<-) et de comparaison (= pour l'égalité et /= pour la différence).

Soient *tab* un tableau et *i* appartenant au domaine d'indice de *tab*. La notation *tab[i]* fournit la valeur de l'élément d'indice *i* du tableau *tab*, et est appelée fonction d'accès à un élément du tableau *tab*.

L'affectation (respectivement la comparaison) de deux tableaux *tab1* et *tab2* de même type implique une affectation (respectivement une comparaison) des éléments respectifs des deux tableaux. Ces opérations globales n'existent pas nécessairement dans tous les langages de programmation, en particulier en C.

Plus précisément, les opérations applicables à un tableau peuvent se résumer à :

Opérations	Syntaxe	Sémantique
Accès à un élément	tab[i]	Accès à l'élément au rang *i* (évalué) du tableau *tab*
Affectation	tab1 <- tab2	Recopie les éléments du tableau *tab2* dans les éléments correspondants du tableau *tab1*
Egalité	tab1 = tab2	VRAI si tous les éléments du tableau *tab1* sont égaux aux éléments correspondants du tableau *tab2*, et FAUX sinon
Différence	tab1 /= tab2	VRAI si au moins un élément du tableau *tab1* est différent de son correspondant dans le tableau *tab2*, et FAUX sinon

Exemples

1) En supposant déclarée la variable *leTexte* de type *TabCaractères*, l'écriture *leTexte[1]* désigne le premier caractère du texte, *leTexte[2]* le deuxième, et ainsi de suite, jusqu'à *leTexte[TAILLE_MAX]* pour le dernier. Les notations *leTexte[0]* et *leTexte[TAILLE_MAX + 1]* n'ont pas de sens puisque l'indice d'accès à un élément de *leTexte* n'appartient pas à l'intervalle compris entre 1 et *TAILLE_MAX*. On pourra appliquer à chaque élément du tableau *leTexte* une opération du type *Caractère*.

2) En supposant que les deux tableaux *monTexte* et *tonTexte* soient de même type *TabCaractères*, l'affectation *monTexte <- tonTexte* recopie alors élément par élément le contenu du tableau *tonTexte* dans le tableau *monTexte*.

Application

Pour écrire le corps de la fonction *estUnMot*, il suffit de balayer séquentiellement tous les caractères du texte depuis le rang *i* (début) jusqu'au rang *j* (fin) et d'examiner si le caractère courant est une lettre. On définit donc :

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
fonction estUnMot
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>

glossaire
        k <Entier> ;            -- indice de parcours du tableau unTexte
```

```
début
        k <- i ;
        tantque k <= j faire
                -- examiner si le caractère courant est une lettre
                si non estUneLettre (unTexte[k]) alors
                        retourner (FAUX) ;
                fin si ;
                -- passer au caractère suivant du texte
                k <- k + 1 ;
        fin tantque ;
        retourner (VRAI) ;
fin
```

Dans cette écriture, la fonction *estUneLettre* est spécifiée comme suit :

```
-- indique si le caractère c est une lettre
fonction estUneLettre (entrée c <Caractère>)
retourne <Booléen> ;
```

Remarques

1) On peut aussi définir le corps de la fonction *estUnMot* en utilisant une conjonction qui teste, dans cet ordre, la validité de l'indice de parcours *k* (*k <= j*) et la nature du caractère de rang *k* (*estUneLettre (unTexte[k])*). En fin de répétition, si *k* a pu « dépasser » l'indice de fin *j*, *unTexte[i..j]* est un mot.

Dans cette version, l'instruction ***retourner*** constitue la dernière instruction du corps de la fonction, ce qui est en général préconisé.

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
fonction estUnMot
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>

glossaire
        k <Entier> ;            -- indice de parcours du tableau unTexte

début
        k <- i ;
        tantque k <= j et estUneLettre (unTexte[k]) faire
                k <- k + 1 ;
        fin tantque ;
        retourner (k > j) ;
fin
```

2) Quelle que soit sa version, la fonction *estUnMot* met en évidence le schéma général d'une répétition **tantque** … **faire** … **fin tantque**. Nous savons qu'un tel schéma

se compose de quatre composantes : l'initialisation, la condition d'arrêt, la progression et le traitement. On le décline comme suit :

```
initialisation ;
tantque condition faire
    traitement ;
    progression ;
fin tantque ;
```

Les trois premières composantes, essentielles pour l'écriture de toute répétition, s'instancient comme suit pour chacune des deux versions de la fonction *estUnMot* :

Composante	Version 1	Version 2
Initialisation	k <- i	k <- i
Condition	k <= j	k <= j **et** estUneLettre (unTexte[k])
Progression	k <- k + 1	k <- k + 1

On notera sur cet exemple qu'une partie du traitement de la première version a été déportée en condition de la deuxième version.

3) Bien que disposant de la comparaison par égalité de deux tableaux, il est simple de la (re)définir en visitant leurs éléments respectifs. En reprenant le schéma général d'une répétition, on peut écrire comme suit l'égalité de deux tableaux de type *TabCaractères* :

```
-- indique si les deux tableaux unTexte1 et unTexte2 sont égaux
fonction égalitéDeDeuxTableaux
    (entrée unTexte1 <TabCaractères>,
    entrée unTexte2 <TabCaractères>)
retourne <Booléen>

glossaire
    i <Entier> ;          -- indice de parcours des deux tableaux
                          -- unTexte1 et unTexte2
début
    i<- 1 ;
    tantque i <= TAILLE_MAX faire
        -- examiner si les caractères respectifs des deux tableaux
        -- unTexte1 et unTexte 2 sont égaux
        si unTexte1[i] /= unTexte2[i] alors
            retourner (FAUX) ;
        fin si ;
        -- passer au caractère suivant des deux tableaux
        i <- i + 1 ;
    fin tantque ;
    retourner (VRAI) ;
fin
```

4.2. Opérations du type *enregistrement*

En algorithmique, les opérations sur une variable de type enregistrement sont l'accès à un champ et les opérations globales d'affectation (<-) et de comparaison (= pour l'égalité et /= pour la différence).

La fonction d'accès à un champ d'enregistrement utilise l'opérateur '.' des enregistrements. Si *enr* est une variable désignant un enregistrement, la fonction d'accès à un composant de *enr* est réalisée par la notation pointée *enr.champ* où *champ* représente le composant sélectionné de *enr*.

Comme pour les tableaux, on peut affecter (respectivement comparer) globalement deux enregistrements *enr1* et *enr2* ; on affecte (respectivement compare) alors les champs respectifs des deux enregistrements. Noter aussi que d'autres comparaisons comme <, >, <=, ... ne sont pas prédéfinies puisqu'il faut alors expliciter la relation d'ordre.

Plus précisément, les opérations applicables à un enregistrement peuvent se résumer à :

Opérations	Syntaxe	Sémantique
Accès à un champ	enr.champ	Accès au *champ* de l'enregistrement *enr*
Affectation	enr1 <- enr2	Recopie les champs de l'enregistrement *enr2* dans les champs correspondants de l'enregistrement *enr1*
Egalité	enr1 = enr2	VRAI si tous les champs de l'enregistrement *enr1* sont égaux aux champs correspondants de l'enregistrement *enr2*, et FAUX sinon
Différence	enr1 /= enr2	VRAI si au moins un champ de l'enregistrement *enr1* est différent de son homologue dans l'enregistrement *enr2*, et FAUX sinon

Exemples

1) En supposant les définitions suivantes :

```
-- définition du type EnrTexte
type EnrTexte : enregistrement
        tampon <TabCaractères>,
        taille <Entier>,
        curseur <Entier> ;

-- définition du type TabCaractères
constante TAILLE_MAX <Entier> = 1000 ;
type TabCaractères : tableau [1 à TAILLE_MAX] de <Caractère> ;
```

```
glossaire
    -- déclaration de la variable leTexte
    leTexte <EnrTexte> ;
```

on pourra écrire *leTexte.taille* pour connaître la taille du texte et *leTexte.tampon[1]* pour accéder à son premier caractère. Notons dans ce dernier exemple la combinaison des fonctions d'accès des tableaux ([ ]) et des enregistrements (.).

2) Soient deux variables *monTexte* et *tonTexte* de type *EnrTexte*, l'affectation *monTexte* <- *tonTexte* est autorisée ; elle recopie les trois champs *tampon*, *taille* et *curseur* de l'enregistrement *tonTexte* dans les champs correspondants *tampon*, *taille* et *curseur* de l'enregistrement *monTexte*. On notera sur cet exemple la recopie de tableau évoquée précédemment.

Applications

1) Pour le corps de la procédure *créerTexte* dans cette représentation, on écrira :

```
-- crée un texte initialement vide
procédure créerTexte (sortie unTexte <EnrTexte>)

début
    unTexte.taille <- 0 ;
    unTexte.curseur <- 1 ;
fin
```

2) Pour savoir si les 10 premiers caractères du texte *leTexte* constituent un mot, il suffit d'invoquer la fonction *estUnMot* comme suit où la notation pointée *leTexte.tampon* désigne le tableau *tampon* de l'éditeur.

```
testUnMot <- estUnMot (leTexte.tampon, 1, 10) ;
```

3) La procédure de recherche d'un caractère dans le texte *rechercherCaractère* ci-dessous accède globalement au tableau mémorisant les caractères par l'expression *unTexte.tampon* (de type *TabCaractères*) et à chacun des caractères par *unTexte.tampon[k]* (de type *Caractère*) où *k* est un indice du tableau *unTexte.tampon*.

```
-- recherche la première occurrence du caractère c dans le texte unTexte
-- à partir de la position courante du curseur
-- si l'occurrence existe, trouvé est positionné à VRAI
-- et le curseur se positionne sur cette occurrence
-- sinon, trouvé est positionné à FAUX et la position du curseur
-- est inchangée
procédure rechercherCaractère
    (màj unTexte <EnrTexte>, entrée c <Caractère>,
    sortie trouvé <Booléen>)
```

```
glossaire
        k <Entier> ;            -- indice de parcours du tableau unTexte.tampon

début
        -- rechercher le caractère c dans le texte
        k <- unTexte.curseur ;
        tantque k <= unTexte.taille et unTexte.tampon[k] /= c faire
                k <- k + 1 ;
        fin tantque ;
        -- fournir le résultat de la recherche
        trouvé <- k <= unTexte.taille ;
        si trouvé alors
                unTexte.curseur <- k ;
        fin si ;
fin
```

Remarques

1) L'écriture de la condition composée *k <= unTexte.taille* ***et*** *unTexte.tampon[k] /= c* mérite attention. Elle suppose que l'opérateur ***et*** du langage algorithmique n'est pas commutatif. En effet, la condition « *p* ***et*** *q* » est évaluée comme « ***si non*** *p* ***alors*** *faux ;* ***sinon*** *q ;* ***fin si*** *; ».* Dans cette formulation, on commence par évaluer *p*, et l'évaluation de *p* ***et*** *q* est stoppée si *p* n'est pas vérifié. L'ordre des deux conditions dans une condition ***et*** est donc essentiel : ainsi, lorsque *k > unTexte.taille*, la condition *unTexte.tampon[k] /= c* n'est pas évaluée , ce qui semble raisonnable si l'on songe au cas où *unTexte.taille = TAILLE_MAX* se traduisant par un accès au caractère *unTexte.tampon[TAILLE_MAX + 1]* qui n'existe pas !

2) On peut formuler la même remarque vis-à-vis de l'évaluation de la condition « *p* ***ou*** *q* », équivalente à la formulation « ***si*** *p* ***alors*** *vrai ;* ***sinon*** *q ;* ***fin si*** *; ».*

3) Bien que disposant de la comparaison par égalité de deux enregistrements, il est simple de la (re)définir en invoquant l'égalité de chacun de leurs composants respectifs. Noter dans l'écriture ci-dessous l'usage de l'égalité de deux tableaux, prédéfinie par le langage algorithmique : *unTexte1.tampon = unTexte2.tampon*.

```
-- indique si les deux enregistrements unTexte1 et unTexte2 sont égaux
fonction égalitéDeDeuxEnregistrements
        (entrée unTexte1 <EnrTexte>, entrée unTexte2 <EnrTexte>)
retourne <Booléen>

début
        retourner
                (unTexte1.tampon = unTexte2.tampon
                et unTexte1.taille = unTexte2.taille
                et unTexte1.curseur = unTexte2.curseur) ;
fin
```

Bien évidemment, comparer les deux enregistrements *unTexte1* et *unTexte2* s'écrira plus directement : *unTexte1 = unTexte2*.

Attention, la comparaison des deux tableaux *unTexte1.tampon* et *unTexte2.tampon* implique la comparaison des 1000 caractères de chacun des deux tableaux. Nous évoquons ici la comparaison de deux enregistrements de type *EnrTexte* et non la comparaison de deux textes au sens strict du terme, où seuls les caractères compris entre les bornes 1 et *taille* du tableau *tampon* sont significatifs ! Mais qui peut le plus, peut le moins.

De plus, le connecteur ***et*** n'étant pas commutatif, on aurait tout intérêt à comparer initialement les deux champs *taille* et *curseur*. En effet, deux textes sont différents si les tailles sont distinctes ; on peut formuler la même remarque concernant la position courante du curseur. Ce qui conduit à l'ordonnancement suivant pour le calcul de la valeur retournée :

```
unTexte1.taille = unTexte2.taille
et unTexte1.curseur = unTexte2.curseur
et unTexte1.tampon = unTexte2.tampon
```

Chapitre 6 : Compléments sur les sous-programmes

Ce chapitre complète les notions déjà vues concernant les sous-programmes. Il traite notamment de la distinction entre procédures et fonctions, des règles de transmission et d'association des paramètres et des appels récursifs.

1. DISTINCTION ENTRE PROCEDURE ET FONCTION

Il n'y a pas de règle générale pour choisir entre procédure et fonction. Un appel de procédure correspond à un traitement et s'assimile à une action élémentaire ne retournant pas de résultat (sauf dans les paramètres en mode sortie) ; un appel de fonction réalise un calcul et produit toujours un résultat que le code doit exploiter.

Exemple

Le sous-programme *maximum* suivant de type fonction détermine le plus grand de deux nombres :

```
-- calcule le maximum des 2 entiers x et y
fonction maximum (entrée x <Entier>, entrée y <Entier>)
retourne <Entier> ;
```

Considérons 3 nombres *x1*, *x2* et *x3*. L'instruction ci-dessous calcule dans la variable *max* le plus grand des trois nombres *x1*, *x2* et *x3*.

```
-- calculer le maximum des 3 entiers x1, x2 et x3
max <- maximum (x1, maximum (x2, x3)) ;
```

Une écriture équivalente, mais très maladroite, consiste à spécifier *maximum* par une procédure, qu'on rebaptise *calculerMaximum* (car il s'agit maintenant d'une action...) :

```
-- affecte max par le maximum des 2 entiers x et y
procédure calculerMaximum
        (entrée x <Entier>, entrée y <Entier>, sortie max <Entier>) ;
```

L'appel équivalent au calcul *maximum (x1, maximum (x2, x3))* se traduit alors par la séquence de deux appels à la procédure *calculerMaximum*, le premier pour calculer le maximum de *x2* et de *x3* (soit *max23*), et le second pour calculer le maximum de *x1* et de *max23* (soit *max*). Notons également que l'on est contraint d'introduire un nouveau paramètre effectif *max23*. Clairement, on spécifiera donc le calcul du maximum par une fonction.

```
-- calculer le maximum des 3 entiers x1, x2 et x3
calculerMaximum (x2, x3, max23) ;
calculerMaximum (x1, max23, max) ;
```

Remarques

1) L'appel d'une fonction ne peut apparaître qu'au sein d'une expression. Ainsi, l'écriture « isolée » *maximum (x1, x2)* n'a pas de sens. Il conviendra d'exploiter le résultat de cet appel :

- en le mémorisant dans une variable soit *max <- maximum (x1, x2)*,
- en l'utilisant au sein d'une expression comme *maximum (x1, x2) > 10*,
- ou encore en le transmettant en tant que paramètre d'entrée à un sous-programme comme dans l'appel *écrire (maximum (x1, x2))*.

On dit ainsi que le résultat de la fonction prend la place de l'appel.

2) Il est facile de transformer une fonction en une procédure, comme l'illustre l'exemple précédent du calcul du maximum de deux entiers ; la réciproque est bien évidemment fausse. La différence se situe essentiellement au niveau du style de programmation qui en découlera dans les appels : fonctionnel dans le cas des fonctions (expressions) et impératif dans le cas des procédures (actions).

3) Considérant le postulat qu'une fonction n'admet que des paramètres en mode entrée, il serait faux d'en conclure qu'il n'est pas possible de définir des procédures munies uniquement de paramètres en mode entrée ! En imaginant une mise en page du texte en cours d'édition, on peut définir une procédure chargée de laisser un certain nombre de lignes blanches entre deux paragraphes, ce que l'on pourra facilement réaliser en invoquant la procédure *sauterLignes* suivante :

```
-- saute n lignes blanches à l'affichage
procédure sauterLignes (entrée n <Entier>) ;
```

2. LES PROCEDURES D'ENTREE/SORTIE

Définition

On appelle procédure d'entrée/sortie toute procédure permettant à l'acteur ordinateur :

- de demander une valeur à son environnement et de la mémoriser dans une de ses variables (lecture de données ou primitive d'entrée),
- de fournir à son environnement la valeur d'une de ses variables (écriture de résultats ou primitive de sortie).

On appelle environnement d'un algorithme toute entité susceptible, sur ordre de l'acteur, de fournir (lecture de données) ou de recevoir (écriture de résultats) des valeurs.

Exemple

Pour l'éditeur de texte, l'utilisateur fournit, sur demande du programme, une commande, munie éventuellement de paramètres (lecture de données). Il récupère ensuite sur son écran et sur ordre du processeur, le texte éventuellement modifié par la dite commande (écriture de résultats).

2.1. La lecture de données

La lecture de données est une action permettant de demander en cours de calcul une valeur qui n'est pas connue au moment où l'on définit l'algorithme. Elle permet d'appliquer un même algorithme à des situations différentes ; un des paramètres est alors la valeur demandée. On obtient ainsi une plus grande généralité de l'algorithme.

Cas général

La demande d'une valeur *d* est en général incarnée par le sous-programme *lire* d'en-tête générique :

```
-- lit une donnée d (de type T quelconque)
procédure lire (sortie d <T>) ;
```

Exemple

La procédure de saisie d'une commande de l'utilisateur est une procédure de lecture spécifiée comme suit :

```
-- lit une commande uneCommande de l'utilisateur
procédure lireCommande (sortie uneCommande <Caractère>) ;
```

Remarques

1) Cette opération d'entrée *lire* doit être considérée du point de vue de l'acteur ordinateur qui exécute l'algorithme, et non de l'utilisateur qui fournit la donnée. Ceci explique le mode de transmission en sortie pour le paramètre *d*.

2) On ne confondra pas cette opération de lecture externe (notée *lire (d)*) avec l'accès au contenu d'une variable qui fait référence à une lecture en mémoire (notée simplement *d*). En pratique, les données externes ne sont lues qu'une fois.

3) La valeur lue est communiquée par l'environnement à l'algorithme. Elle est mise en correspondance et affectée au paramètre effectif. La lecture effectue donc l'affectation à une variable d'une valeur saisie sur un organe périphérique adéquat (clavier, disque, …). L'instruction *lire (d)* peut ainsi s'assimiler à l'affectation *d <- valeur lue*.

2.2. L'écriture de résultats

L'écriture est une action permettant de fournir à l'environnement un certain nombre de valeurs sélectionnées parmi l'ensemble des valeurs calculées par l'algorithme.

Cas général

Dans le cas général, l'écriture d'une valeur *r* s'incarne par le sous-programme *écrire* d'en-tête :

```
-- écrit un résultat r (de type T quelconque)
procédure écrire (entrée r <T>) ;
```

Exemple

L'affichage du texte à l'écran est une procédure d'écriture spécifiée comme suit :

```
-- affiche le texte unTexte
procédure afficherTexte (entrée unTexte <EnrTexte>) ;
```

Remarque

Comme pour la lecture, l'opération de sortie *écrire* doit être considérée du point de l'ordinateur qui exécute l'algorithme, et non de l'utilisateur qui prend connaissance du résultat. Ceci explique le mode de transmission en entrée pour le paramètre *r*.

3. REGLES DE TRANSMISSION DES PARAMETRES

Dans un corps de sous-programme, un paramètre formel en mode entrée doit nécessairement continuer d'être propagé aux sous-programmes appelés en mode entrée. Sinon, il y aurait violation de la propriété d'un paramètre en entrée qui stipule qu'une entrée est une constante pour l'appelé.

Par contre, un paramètre formel en mode sortie ou mise à jour peut être propagé aux sous-programmes appelés indifféremment en mode entrée, sortie ou mise à jour. Toutefois, lorsqu'un paramètre est transmis en mode sortie, il faudra s'assurer que sa transmission en mode entrée ou mise à jour (qui inclut par définition le mode entrée) à un autre sous-programme soit précédée d'un calcul de valeur.

Exemple

Considérons le calcul du plus grand de 3 entiers. Si l'on dispose déjà de maximum de 2 entiers, il est simple d'écrire une fonction *maximumPour3* pour 3 entiers, comme vu précédemment. Les paramètres *x*, *y* et *z*, transmis en mode entrée à la fonction *maximumPour3* sont propagés à la fonction *maximum*, toujours en mode entrée :

```
-- calcule le maximum des 3 entiers x, y et z
fonction maximumPour3
        (entrée x <Entier>, entrée y <Entier>, entrée z <Entier>)
retourne <Entier>

début
        retourner (maximum (x, maximum (y, z)) ;
fin
```

4. REGLES D'ASSOCIATION DES PARAMETRES

Lors d'un appel de procédure ou de fonction, les paramètres effectifs sont mis en correspondance avec les paramètres formels, un à un et de gauche à droite.

Pour un paramètre formel en mode entrée, le paramètre effectif correspondant peut être un nom de variable ou une expression. Si le paramètre effectif est une expression, l'expression est évaluée avant l'appel ; c'est donc la valeur de l'expression qui est transmise à l'appelé.

Pour un paramètre formel en mode sortie ou mise à jour, le paramètre effectif correspondant est obligatoirement une variable, car il faut bien que le sous-programme rende la valeur du paramètre modifié dans quelque chose (c'est-à-dire une variable)... ce ne peut donc être une expression. L'identité d'une telle variable est déterminée à l'appel du sous-programme.

Exemples

1) Soit l'en-tête du sous-programme *échanger* :

```
-- échange le contenu des 2 entiers x et y
procédure échanger (màj x <Entier>, màj y <Entier>) ;
```

A partir de ce sous-programme, on peut réaliser l'appel :

```
échanger (a, b) ;
```

A l'inverse, les deux appels suivants sont rejetés par le compilateur car on associe des expressions aux paramètres formels *x* et *y* définis en mode mise à jour dans le sous-programme *échanger* :

```
échanger (5, 12) ;
échanger (a + b, c) ;
```

Comment, par exemple, récupérer après l'appel à *échanger (a + b, c)*, le contenu de l'entier mémorisé dans *c* dans l'expression *a + b* ? Ceci n'a guère de sens.

2) Soit l'en-tête :

```
-- calcule le maximum des 2 entiers x et y
fonction maximum (entrée x <Entier>, entrée y <Entier>)
retourne <Entier> ;
```

Pour cet en-tête, les expressions suivantes sont valides :

```
maximum (a, b)
maximum (a + 12, 5)
maximum (a + b, maximum (c, d + 5))
```

Remarque

On peut également transmettre un élément de tableau ou un champ d'enregistrement lors d'un appel de sous-programme ; l'élément ou le champ est identifié avant l'appel. On peut ainsi écrire, en supposant *unTexte* de type *EnrTexte* :

```
position <- maximum (unTexte.curseur + 30, unTexte.taille) ;
```

5. LE MECANISME DES EXCEPTIONS

Définition

Une exception désigne un événement correspondant à une situation anormale pouvant survenir au cours de l'exécution d'un programme ou d'un sous-programme. On l'associe le plus souvent au non-respect d'une précondition (clause ***nécessite***) du contrat entre appelant et appelé.

Syntaxe

On signale une exception par l'instruction ***déclencher*** suivie d'un nom d'exception *e* (qu'on peut interpréter comme un signal d'anomalie). Cette exception *e* pourra être interceptée et traitée par un récupérateur d'exception (mots clés ***traite-exception*** et ***lorsque***).

Sémantique

L'instruction ***déclencher*** *e* provoque l'abandon de la séquence en cours et la propagation de l'exception *e* à travers les différents niveaux d'appels du programme.

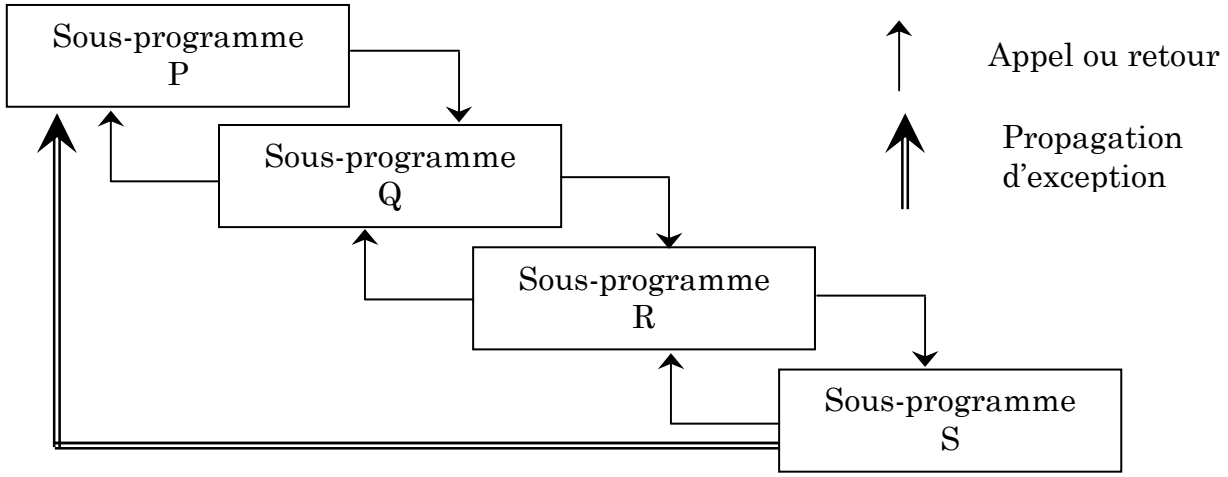
La propagation de *e* interrompt l'exécution des sous-programmes qu'elle traverse tant qu'elle n'est pas interceptée par son ***traite-exception***. C'est le traitement associé à l'interception, décrit par un récupérateur d'exception ***lorsque*** … ***faire*** de même nom d'exception *e*, qui lui donnera un sens. Le code du récupérateur remplace la séquence en cours d'exécution.

Sous-programme	Syntaxe	Sémantique
Appelé	**déclencher** *e*	Propage l'exception *e* vers l'appelant
Appelant	**traite-exception**	Introduit un récupérateur d'exception
	lorsque e **faire** … **fin lorsque** ;	Intercepte l'exception *e* et exécute la séquence d'instructions délimitée par les mots clés ***lorsque*** et ***fin lorsque***

Exemple

Dans l'exemple ci-dessous, nous supposons les appels en cascade suivants : *P* invoque *Q*, *Q* invoque *R*, et *R* appelle *S*. Nous supposons de plus que le sous-programme *P* est doté d'un traite-exception capable d'intercepter et de traiter une

exception *e*. Si le sous-programme *S* déclenche l'exception *e*, cette exception traversera les corps des sous-programmes *R* et *Q* avant d'être interceptée par *P*.



Remarque

La propagation d'un signalement d'exception fait partie intégrante du contrat d'utilisation d'un sous-programme ; elle s'exprime par la clause ***déclenche*** dans son en-tête.

Si le sous-programme appelant intercepte une exception *e* des niveaux appelés par un traite-exception associé, il n'y pas lieu de signaler par une clause ***déclenche*** la propagation. En revanche, si l'appelant laisse se propager *e* vers les niveaux appelants, il convient de spécifier la propagation dans l'en-tête.

Application

1) Soit l'en-tête de la procédure *estUnMot* :

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
fonction estUnMot
      (entrée unTexte <TabCaractères>,
      entrée i <Entier>, entrée j <Entier>)
retourne <Booléen> ;
```

Conformément à l'en-tête, l'appelant transmet les rangs *i* et *j* du texte. Qu'advient-il si *i* > *j* ? Dans ce cas, la fonction *estUnMot* signale à l'appelant l'anomalie *intervalleInvalide* et l'en-tête du sous-programme spécifie par le mot-clé ***déclenche*** l'exception levée.

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
-- nécessite i ≤ j
fonction estUnMot
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>
déclenche intervalleInvalide ;
```

L'instruction ***déclencher*** *(intervalleInvalide)* signale l'anomalie à l'appelant :

```
-- indique si tous les caractères compris entre les rangs i (début)
-- et j (fin) du texte unTexte sont des lettres
-- nécessite i ≤ j
fonction estUnMot
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>
déclenche intervalleInvalide

glossaire
        …

début
        -- intercepter l'anomalie i > j
        si i > j alors
                déclencher (intervalleInvalide) ;
        fin si ;
        -- effectuer le traitement demandé (respect de la précondition)
        …
fin
```

2) La commande d'insertion d'un caractère *c* dans le texte *unTexte* est également susceptible de déclencher une exception, ici nommée ***débordement***. Cette situation intervient chaque fois que le texte n'est plus en mesure de mémoriser un nouveau caractère, soit *unTexte.taille = TAILLE_MAX*.

```
-- insère le caractère c dans le texte unTexte à la position du curseur
-- nécessite unTexte.taille < TAILLE_MAX
procédure insérerCaractère
        (màj unTexte <EnrTexte>, entrée c <Caractère>)
déclenche débordement

glossaire
        …
```

```
début
        -- intercepter le débordement de la zone tampon
        si unTexte.taille = TAILLE_MAX alors
                déclencher (débordement) ;
        fin si ;
        …
fin
```

Pour intercepter l'exception ***débordement***, on écrira :

```
        -- exécute une commande de l'utilisateur
        procédure exécuterCommande
                (entrée uneCommande <Caractère>, màj unTexte <EnrTexte>)

        glossaire
                c <Caractère> ;          -- le caractère à insérer de la commande 'i'

        début
                si uneCommande = 'i' alors
                        -- appeler le sous-programme d'insertion
                        lire (c) ;
                        insérerCaractère (unTexte, c) ;
                        …
                sinon …
                        …
                fin si ;
        traite-exception
                lorsque débordement faire
                        écrire ("tampon plein !") ;
                        écrire ("veuillez saisir une nouvelle commande") ;
                fin lorsque ;
        fin
```

Remarques

1) Ne pas utiliser abusivement les exceptions et les réserver à des situations anormales (division par 0, calcul de la racine carrée d'un nombre négatif, débordement d'un indice de tableau, ...).

2) Le traite-exception d'une fonction devra comporter une instruction ***retourner*** pour fournir un résultat à l'appelant, sauf si le traitement d'exception consiste à propager à nouveau l'exception par une instruction ***déclencher***.

3) En général, l'interception d'une exception doit s'accompagner d'une correction de la situation anormale identifiée.

6. RECURSIVITE

Un corps de sous-programme peut comprendre une instruction d'appel à ce même sous-programme. On parle alors d'appel récursif. Plus généralement, le concept sous-jacent à cette possibilité est appelé récursivité.

Définition

Une définition récursive d'une entité est telle que cette entité est définie en fonction d'elle-même. En programmation, on parle de données récursives et d'algorithmes récursifs.

Exemple

On peut affirmer que la tranche du texte *unTexte[i..j]* est un palindrome si et seulement si les caractères *unTexte[i]* et *unTexte[j]* sont égaux et si la tranche de texte *unTexte [i + 1..j - 1]* est un palindrome. De cela, on en déduit directement l'écriture récursive du sous-programme correspondant :

```
-- retourne VRAI si unTexte[i..j]  est un palindrome
-- et FAUX sinon
fonction palindrome
        (entrée unTexte<TabCaractères>,
         entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>

début
        si i > j alors
                retourner (VRAI) ;
        sinon
                si unTexte[i] /= unTexte[j] alors
                        retourner (FAUX) ;
                sinon
                        retourner (palindrome (unTexte, i + 1, j - 1)) ;
                fin si ;
        fin si ;
fin
```

On peut aussi s'appuyer sur la sémantique de l'opérateur ***et*** du langage algorithmique qui n'évalue pas son deuxième argument lorsque le premier a pour valeur FAUX. Dans ce cas, le processus récursif s'arrête dès la première inégalité *unTexte[i]* ≠ *unTexte[j]* rencontrée.

```
-- retourne VRAI si unTexte[i..j] est un palindrome
-- et FAUX sinon
fonction palindrome
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>

début
        si i > j alors
                retourner (VRAI) ;
        sinon
                retourner
                (unTexte[i] = unTexte [j] et palindrome (unTexte, i + 1, j - 1)) ;
        fin si ;
fin
```

Remarques

1) La répétition (ou itération), la récurrence et la récursivité sont des concepts fondamentaux qui apparaissent sous de multiples formes dans les modèles de données et les algorithmes.

2) Les algorithmes qui n'utilisent pas cette propriété de récursivité et qui expriment malgré tout des répétitions sont appelés algorithmes itératifs. Par exemple, on peut exprimer itérativement la fonction *palindrome* comme suit :

```
-- retourne VRAI si unTexte[i..j] est un palindrome
-- et FAUX sinon
fonction palindrome
        (entrée unTexte <TabCaractères>,
        entrée i <Entier>, entrée j <Entier>)
retourne <Booléen>

glossaire
        k1 <Entier> ;   -- indice de parcours croissant du tableau unTexte
        k2 <Entier> ;   -- indice de parcours décroissant du tableau unTexte

début
        k1 <- i ;
        k2 <- j ;
        tantque k1 < k2 faire
                si unTexte[k1] /= unTexte[k2] alors
                        retourner (FAUX) ;
                fin si ;
                k1 <- k1 + 1 ;
                k2 <- k2 - 1 ;
        fin tantque ;
        retourner (VRAI) ;
fin
```

Remarquer ainsi qu'à un même en-tête de sous-programme (*quoi* ?) peut correspondre des corps différents (*comment* ?).

7. STRUCTURE GENERALE D'UN PROGRAMME

Un programme en langage algorithmique est composé de trois sections distinctes, dans l'ordre :

- la section de définition des constantes et des types,
- la section de définition des sous-programmes,
- la section des instructions du programme.

La section associée aux définitions de constantes et de types introduit des valeurs constantes d'un type élémentaire et des types spécifiques au problème traité. Une constante s'identifie par le mot-clé ***constante*** et se définit par un identificateur de constante, un type et sa valeur ; la valeur de la constante doit être conforme au type spécifié. Comme indiqué précédemment, le mot-clé ***type*** fait correspondre à un nom de type une définition de type ***tableau*** ou ***enregistrement***. Un type faisant référence à une constante doit être défini après ladite constante.

Suivent ensuite les diverses définitions des sous-programmes avec pour chaque définition, son en-tête de type ***procédure*** ou ***fonction***, son glossaire de variables introduit par ***glossaire*** et son corps d'instructions délimité par ***début*** et ***fin***.

La section des instructions est composé de l'en-tête du programme, du glossaire de ses variables et de sa séquence d'instructions. L'en-tête d'un programme en langage algorithmique fournit le moyen de nommer un programme ; elle est formée du mot-clé ***programme*** suivi d'un identificateur correspondant au nom du programme. On introduit ensuite le ***glossaire*** des variables du programme, conformément à la syntaxe précisée. La dernière partie de cette section précise les instructions à exécuter ; elles sont précédées du mot-clé ***début*** et suivies du mot-clé ***fin***.

Remarques

1) Un programme comporte toujours une section décrivant ses instructions. Tout programme possède donc au moins un nom fourni par son en-tête et une séquence d'instructions à exécuter.

2) Par la directive ***importer*** en début de texte, un programme peut introduire en son sein des déclarations de constantes et de types et des définitions de sous-programmes. Cette directive permet de rendre visible les entités importées au sein de l'unité qui les importe.

8. NEUF REGLES POUR CONCLURE

Pour terminer, voici neuf règles de bonne gestion des variables. Elles participent à la cohérence syntaxique et sémantique des textes algorithmiques et doivent à terme devenir des réflexes de programmation.

1. Toute variable en membre droit d'une affectation est soit paramètre d'entrée, soit membre gauche d'une affectation antérieure, soit paramètre transmis en mode sortie d'un appel antérieur.

2. Toute variable locale est initialement membre droit d'une affectation ou paramètre de sortie d'une procédure appelée.

3. Un paramètre en mode entrée ne peut pas apparaître en membre gauche d'une affectation.

4. Un paramètre en mode entrée ne peut pas se transmettre en mode sortie ou mise à jour.

5. Une fonction ne peut en aucun cas modifier ses paramètres (corollaire des règles 3 et 4).

6. Un paramètre en mode sortie doit obligatoirement soit apparaître au moins une fois en membre gauche d'une affectation, soit être un paramètre de sortie d'un appel du corps du sous-programme.

7. Un paramètre en mode sortie transmis en mode entrée ou mise à jour à une autre procédure doit obligatoirement soit apparaître au moins une fois en membre gauche d'une affectation antérieure, soit être un paramètre de sortie d'un appel antérieur.

8. Un paramètre en mode mise à jour doit obligatoirement soit apparaître au moins une fois en membres gauche et droit d'une affectation, soit être en mise à jour d'une procédure appelée.

9. La variable de contrôle d'une itération doit toujours apparaître au moins une fois en membre gauche d'une affectation du corps de la répétition.

Glossaire

Acteur : entité qui exécute l'algorithme.

Action : *action élémentaire* ou *action complexe*.

Action complexe : action décomposable en termes d'actions plus simples.

Action élémentaire : action non décomposable d'un acteur.

Affectation : action qui consiste à donner une nouvelle *valeur* à une variable.

Affinage : processus de décomposition d'une action complexe en actions plus simples.

Algorithme : spécification d'un schéma de calcul sous la forme d'un enchaînement d'actions.

Appel de sous-programme : action qui invoque l'exécution du sous-programme.

Assertion : proposition que l'on énonce et que l'on soutient comme vraie.

Condition : expression logique (prédicat).

Corps de sous-programme : mise en œuvre algorithmique correspondant à l'en-tête du sous-programme.

Ecriture : action qui consiste à fournir une *valeur* à l'environnement extérieur à l'acteur.

Enregistrement : regroupement fini de composants accessibles par une sélection.

En-tête de sous-programme : interface entre le sous-programme et le monde extérieur.

Exception : événement associé à une situation anormale à laquelle un programme ou un sous-programme ne peut répondre.

Expression : constante, variable, appel de fonction ou calcul formé d'opérandes (constante, variable, appel de fonction) et d'opérateurs.

Fonction : sous-programme décrivant l'abstraction fonctionnelle d'un calcul.

Glossaire : suite de déclarations de variables.

Lecture : action qui consiste à demander une *valeur* à l'environnement extérieur à l'acteur.

Paramètre : donnée spécifiant le contexte de calcul d'un sous-programme.

Paramètre effectif : *paramètre* d'un appel de sous-programme.

Paramètre en entrée : *paramètre* dont la valeur est définie par l'appelant.

Paramètre en mise à jour : *paramètre* dont la valeur d'entrée définie par l'appelant est modifiée par l'appelé.

Paramètre en sortie : *paramètre* dont la valeur est calculée par l'appelé.

Paramètre formel : *paramètre* d'un en-tête de sous-programme.

Paramètre réel : voir *paramètre effectif*.

Précondition : assertion décrivant l'état du calcul avant l'exécution d'une action.

Postcondition : assertion décrivant l'état du calcul après l'exécution d'une action.

Procédure : sous-programme décrivant l'abstraction fonctionnelle d'une action.

Répétition : enchaînement qui met en jeu plusieurs fois la même action.

Sélection : enchaînement lié au choix d'une action parmi deux (en général).

Séquence : enchaînement inconditionnel d'actions.

Sous-programme : unité modulaire indépendante associée à une abstraction fonctionnelle de l'algorithme.

Spécification de sous-programme : voir *en-tête de sous-programme*.

Structure de contrôle : construction spécifiant un enchaînement d'actions.

Tableau : séquence d'éléments de longueur fixe accessibles par un indice.

Type : ensemble de valeurs et d'opérations.

Valeur : contenu d'une variable.

Variable : triplet (identificateur, *valeur*, *type*).

Variable locale : variable accessible uniquement au sein des actions d'un sous-programme.

Variable simple : variable qui ne mémorise qu'une seule valeur.

Variable structurée : association de variables simples.

Vingt Proverbes de Programmation[8]

Etude du problème à résoudre

1. Définir le problème avec soin.
2. Si nécessaire, limiter le programme en simplifiant le problème.
3. S'intéresser uniquement à ce que doit calculer le programme.

Ecriture de l'algorithme

4. Développer l'algorithme de façon descendante.
5. Valider l'algorithme à chaque étape d'affinage.
6. Ne résoudre qu'une seule difficulté à la fois.
7. Ne pas commencer par un point de détail, puis s'occuper de l'ensemble.
8. Ecrire l'algorithme indépendamment du langage de programmation.

Ecriture du programme

9. Soigner la présentation du programme.
10. Décomposer le programme en sous-programmes.
11. Ne pas ajouter les commentaires lorsque le programme est terminé !
12. Ne jamais supposer que la machine suppose quelque chose...
13. Bannir les astuces de programmation.

Mise au point du programme

14. Avant toute exécution, tester le programme à la main.
15. Limiter à 4 ou 5 le nombre de compilations pour la mise au point.
16. En cas d'erreur, reprendre les niveaux d'affinage.
17. Attendre qu'un programme marche pour l'améliorer.

Amélioration et Exploitation du programme

18. Ne pas croire que tout est terminé lorsque le programme marche...
19. Ne pas hésiter à tout recommencer.
20. Utiliser les particularités du langage de programmation.

[8] Inspirés de l'ouvrage de H. F. Ledgard, Proverbes de programmation, Dunod, 1975

Bibliographie

A. Aho, J. Hopcroft, J. Ullman
Structures de données et algorithmes
InterEditions, 1989

J. Barnes
Programmer en ADA
InterEditions, 1988

T. Cormen, C. Leiserson, R. Rivest
Introduction à l'algorithmique
Dunod, 1994

J. Courtin, I. Kowarski
Initiation à l'algorithmique et aux structures de données
tome 1 : programmation structurée et structures de données élémentaires
tome 2 : récursivité et structures de données avancées
tome 3 : problèmes, études de cas
Dunod, 1987

O. El kharki, J. Mechbouh, D. Ducrot
Algorithmique – Cours, Exercices & Examens Corrigés
2009 MO 2299, 2009

C. Froidevaux, M.-C. Gaudel, M. Soria
Types de Données et Algorithmes
Mc Graw-Hill, 1990

L. Goldschlager, A. Lister
Informatique et algorithmique
InterEditions, 1986

Grégoire (nom collectif)
Informatique-Programmation, CNAM cycle A
tome 1 : la programmation structurée
tome 2 : la spécification récursive et l'analyse des algorithmes
tome 3 : exercices et corrigés
Masson, 1990

M. Griffiths
Algorithmique et programmation
Hermès, 1992

J. Guyot, C. Vial
Arbres, Tables et Algorithmes
Eyrolles, 1988

H. F. Ledgard
Proverbes de programmation,
26 règles et une méthode pour mieux programmer
Dunod, 1975

J.-M. Léry
Algorithmique – Applications en C
Informatique – Synthèse de cours & exercices corrigés
Collection Synthex, Pearson Education, 2005

J. Maysonnave
Introduction à l'algorithmique générale et numérique (DEUG Sciences)
Résumés de cours, 193 exercices et problèmes avec solutions
Masson, 1996

C. Pair, R. Mohr, R. Schott
Construire les algorithmes - les améliorer, les connaître, les évaluer
Dunod, 1988

G. Pierra
Les bases de la PROGRAMMATION et du GENIE LOGICIEL
Dunod, 1991

R. Sedgewick
Algorithmes en langage C
InterEditions, 1991

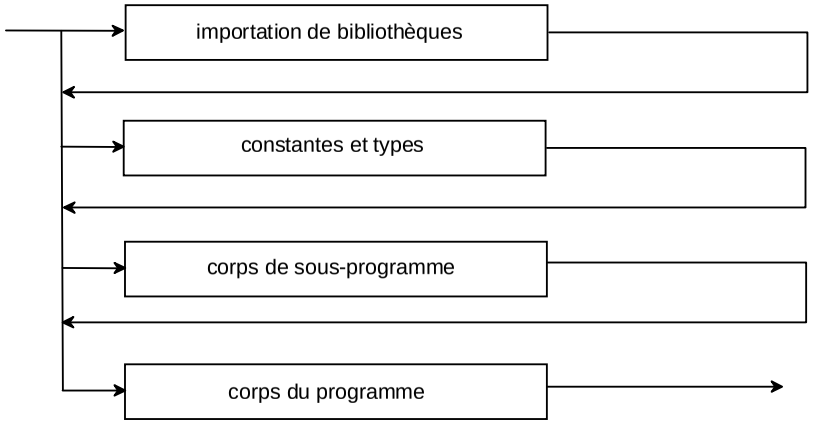
B. Warin
L'algorithmique, votre passeport informatique pour la programmation
Ellipses, 2002

N. Wirth
Systematic Programming: An Introduction
Prentice-Hall, 1973

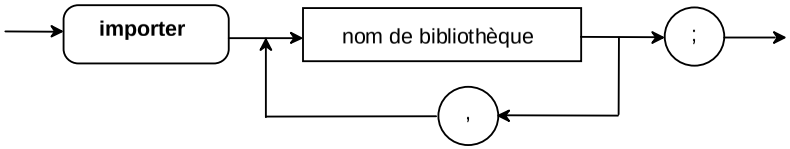
N. Wirth
Algorithms + Data Structures = Programs
Prentice-Hall, 1976

Diagrammes syntaxiques du langage algorithmique

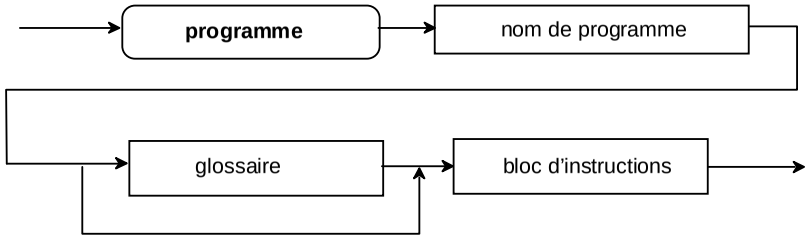
Programme



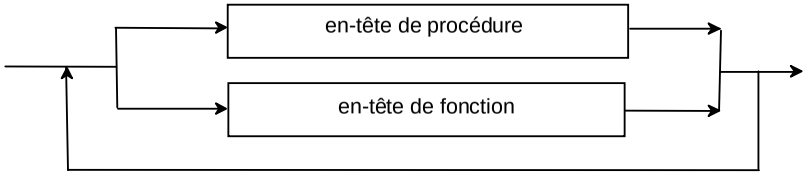
Importation de bibliothèques



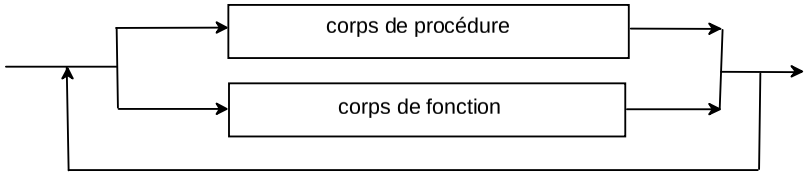
Corps du programme



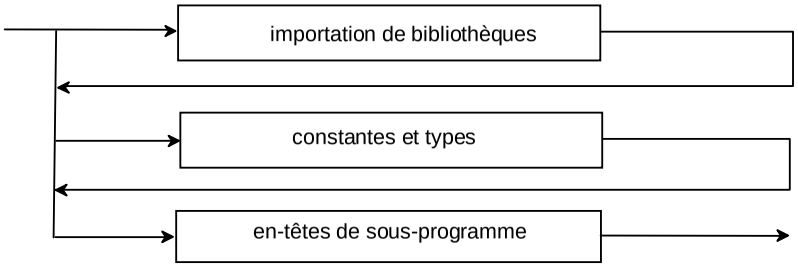
En-têtes de sous-programme



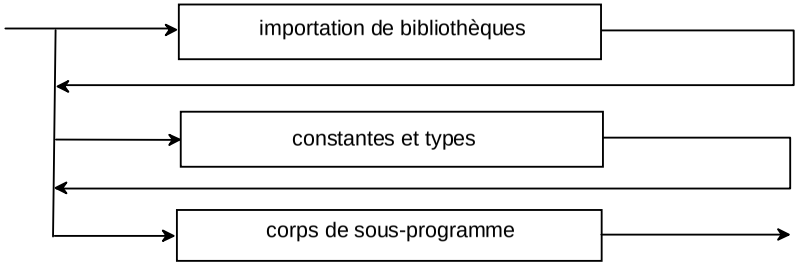
Corps de sous-programme



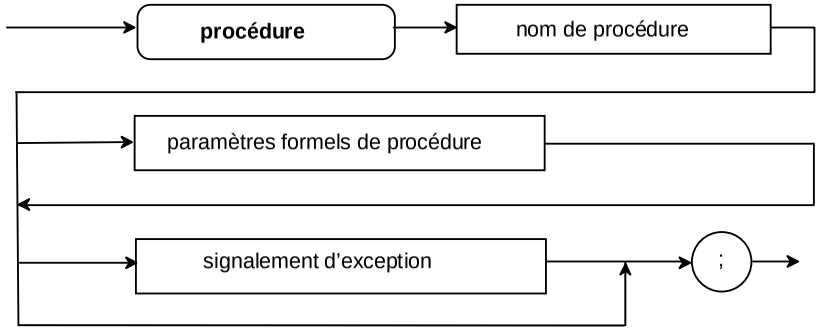
Unité de bibliothèque



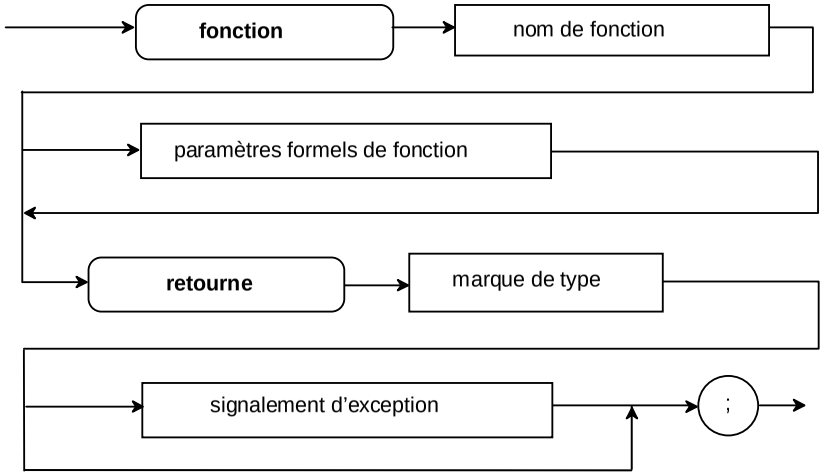
Corps de bibliothèque



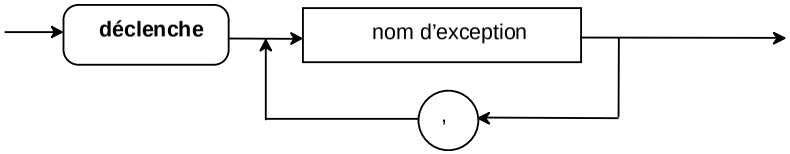
En-tête de procédure



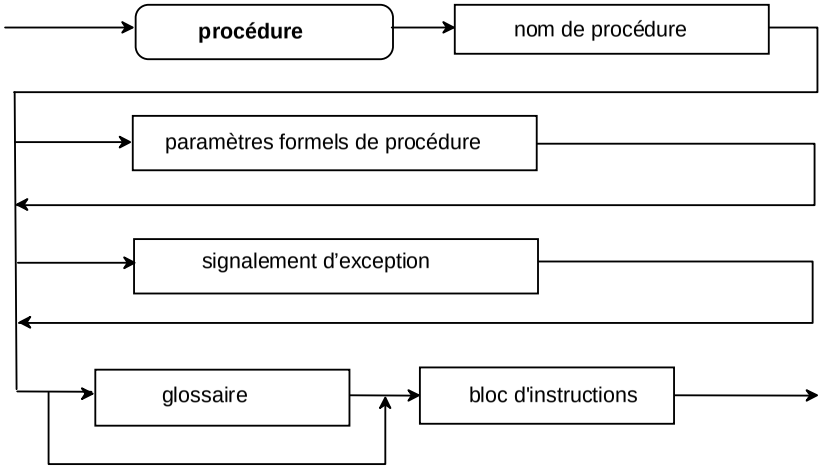
En-tête de fonction



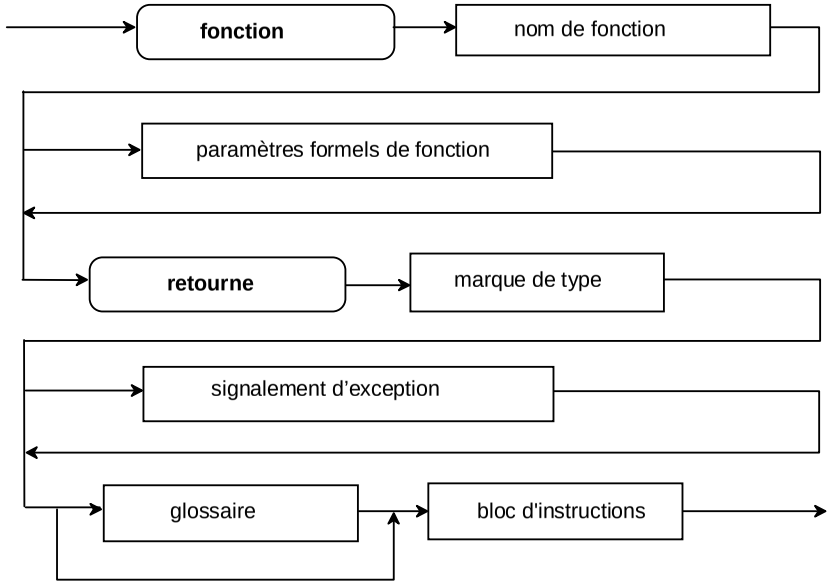
Signalement d'exception



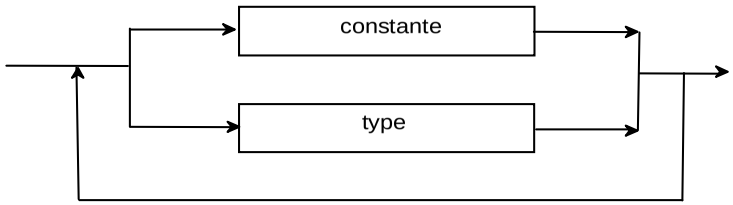
Corps de procédure



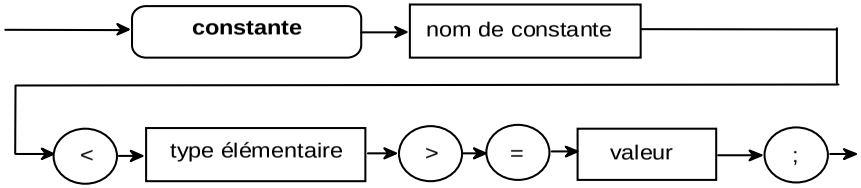
Corps de fonction



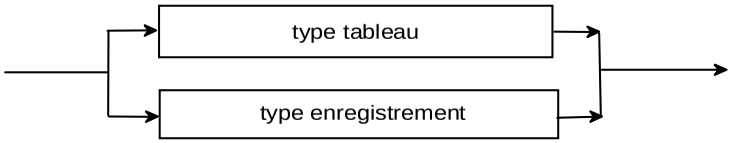
Constantes et types



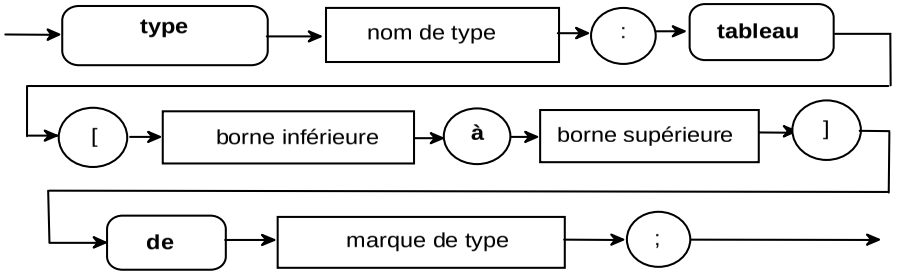
Constante



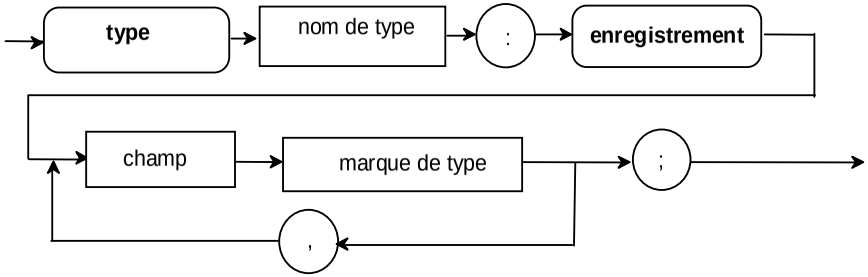
Type



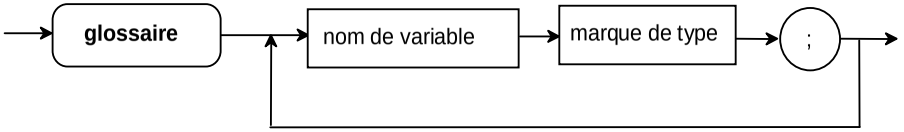
Type tableau



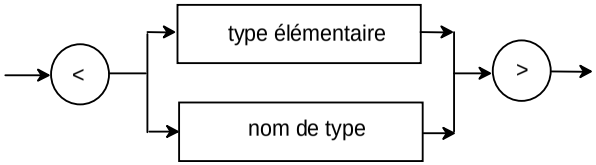
Type enregistrement



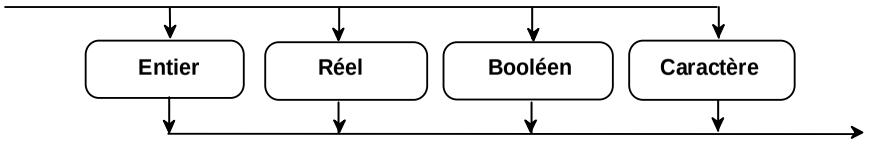
Glossaire



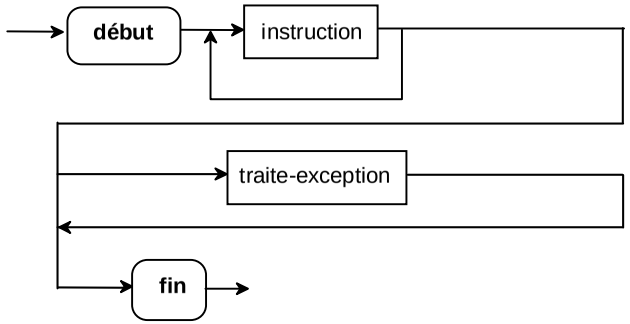
Marque de type



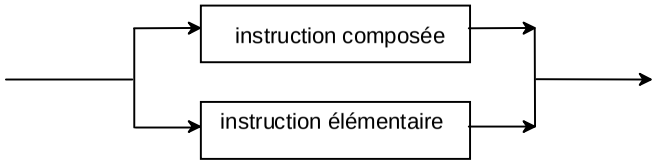
Type élémentaire



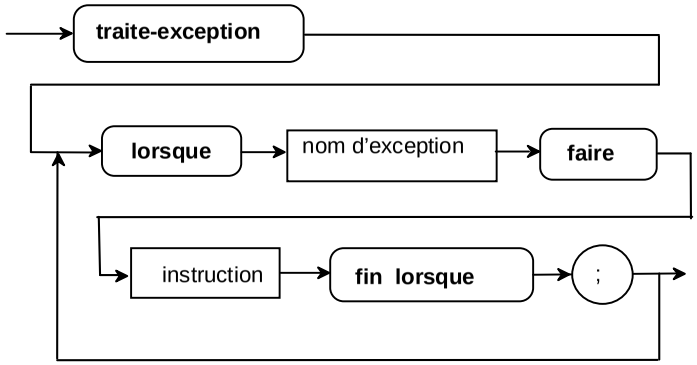
Bloc d'instructions



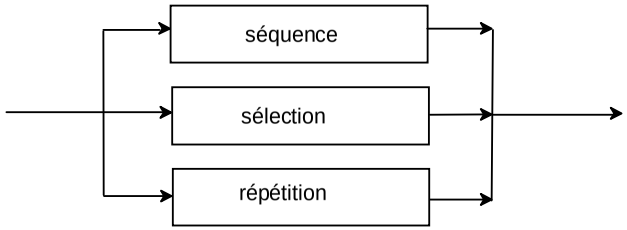
Instruction



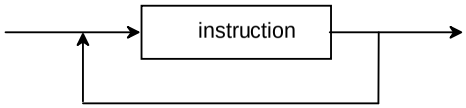
Traite-exception



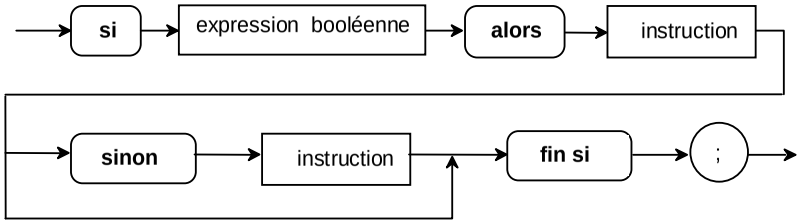
Instruction composée



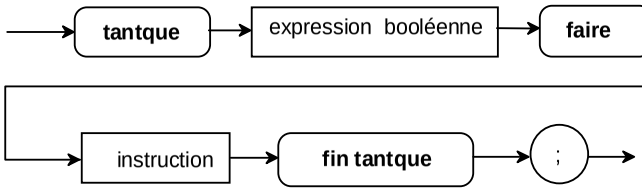
Séquence



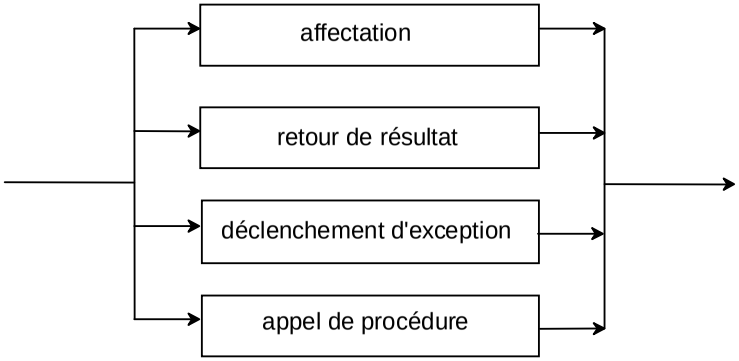
Sélection



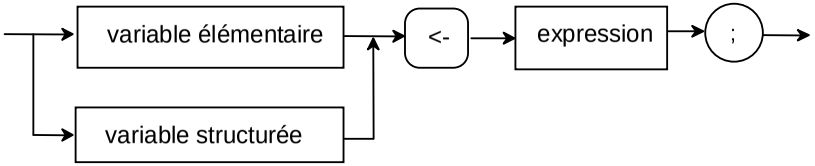
Répétition



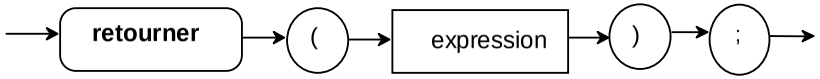
Instruction élémentaire



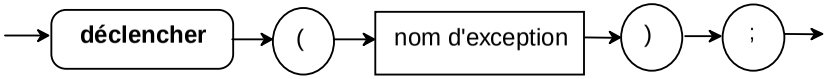
Affectation



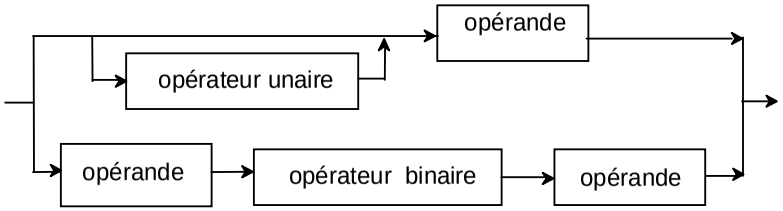
Retour de résultat



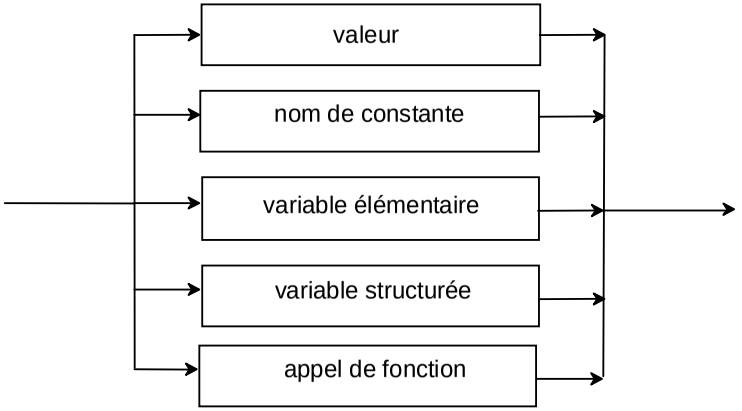
Déclenchement d'exception



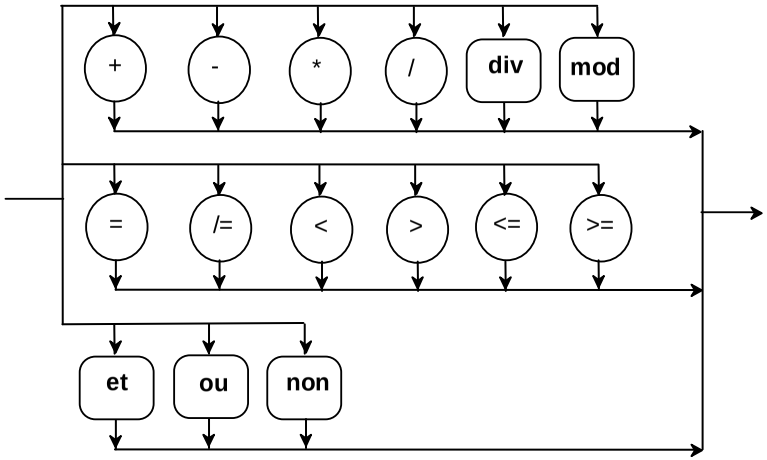
Expression



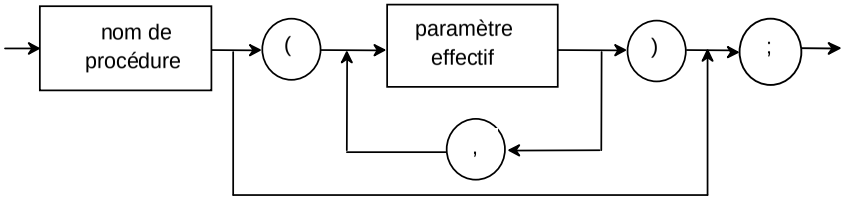
Opérande



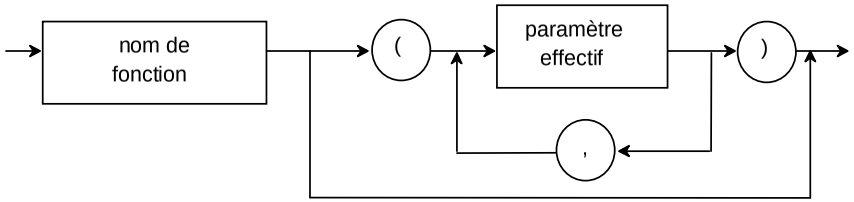
Opérateur



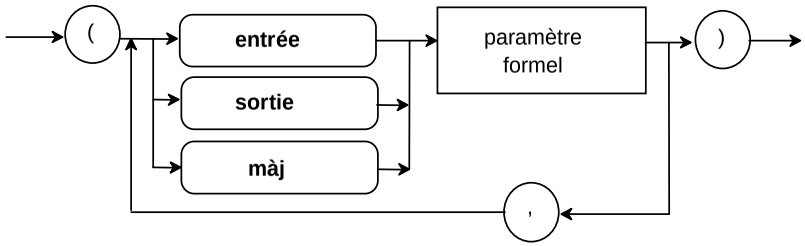
Appel de procédure



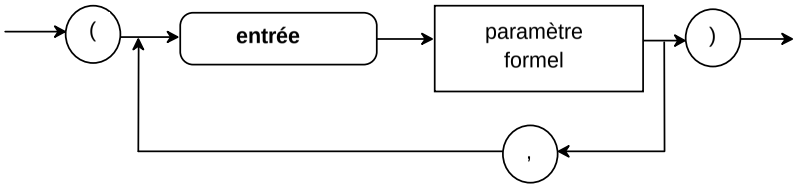
Appel de fonction



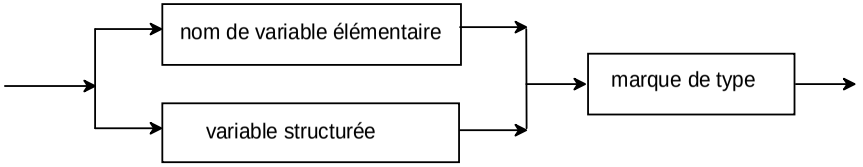
Paramètres formels de procédure



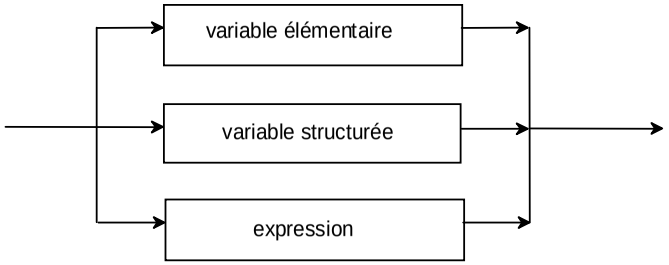
Paramètres formels de fonction



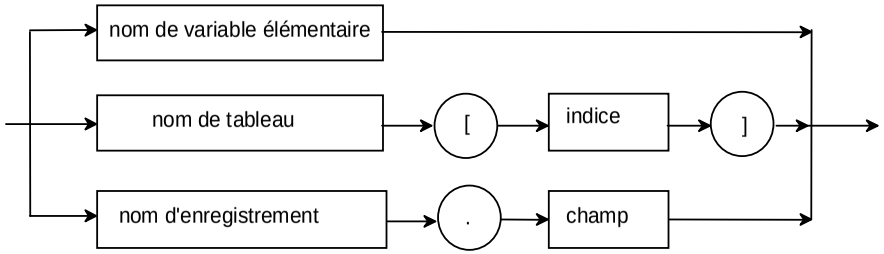
Paramètre formel



Paramètre effectif



Variable élémentaire



Variable structurée