# CS110 – Fiche de référence du langage algorithmique – Première partie

Le langage algorithmique que nous utilisons est un langage dit *impératif* : l'algorithme est exprimé comme une suite d'actions, réalisées par des *instructions*. Sa syntaxe est proche de celles de langages de programmation classiques comme Pascal et Ada. Il y a trois catégories d'objets dans le langage :

**les variables** peuvent être pensées comme des "boites" portant un nom et contenant une valeur. La valeur contenue dans la boite peut être utilisée ou modifiée par l'algorithme. L'ensemble des variables constituent l'*environnement* de l'algorithme. Une variable dont on n'a pas fixé la valeur est *indéterminée*.

**les instructions** représentent une action, elle ont un effet comme afficher un message à l'écran ou modifier l'environnement (changer la valeur d'une variable).

**les expressions** représentent une valeur. Cette valeur n'est pas forcément immédiatement apparente, elle est calculée par *évaluation* de l'expression.

## Structure d'un algorithme

La spécification doit indiquer clairement et précisément, sans ambiguïté, ce que fait l'algorithme (mais pas *comment* il le fait).

```
Algorithme son_nom
(* spécification *)
déclarations
début
   instructions
fin
```

## Identificateurs

Divers objets du langage, tels que les variables, sont désignés par leur *nom*. Ce nom, ou *identificateur*, est un mot constitué des caractères alphabétiques courants, des chiffres et des symboles "`-`" et "`_`". Le premier caractère d'un identificateur est forcément une lettre. Les identificateurs doivent être choisis judicieusement pour faciliter la compréhension de l'algorithme.

Exemples

`a, toto12, tri-sélection, ~~ouvert/fermé~~, ~~2ieme_compteur~~`

## Valeurs et types

Les valeurs qu'un algorithme peut manipuler sont classées en *types*. La notion de type recouvre plus ou moins la notion mathématique d'ensemble.

**entier** type des valeurs entières (sous-ensemble de $\mathbb{Z}$)

**réel** type des valeurs numériques non forcément entières (représentation approchée des nombres réels)

**booléen** type des valeurs logiques, notées `vrai` et `faux`

**car** type des caractères alphabétiques, numériques et de ponctuation.

| nom | description | littéraux |
|---|---|---|
| entier | sous-ens. de $\mathbb{Z}$ | 1 2 –5 9999 |
| réel | valeurs approchées d'une partie de $\mathbb{R}$ | 1.0 3.14 –2.67 |
| car | caractères | 'a' '6' ' ?' |
| booléen | valeurs logiques | vrai faux |
| chaîne | chaîne de caractères | "coucou !" |

TAB. 1 – Types de données élémentaires

| opérateur | type des opérandes | type du résultat | opération |
|---|---|---|---|
| $-$ | 1 entier ou réel | idem opérande | négation |
| $+$<br>$-$<br>$\times$<br>$/$ | 2 entiers ou 2 réels ou 1 entier et 1 réel | entier si les 2 opérandes le sont, réel sinon | addition<br>soustraction<br>multiplication |
| | | réel | division réelle |
| div<br>mod | 2 entiers | entier | quotient entier<br>reste de la division |
| $<$<br>$\leq$<br>$>$<br>$\geq$<br>$=$<br>$\neq$ | 2 nombres, 2 caractères ou 2 chaînes | booléen | inférieur<br>inférieur ou égal<br>supérieur<br>supérieur ou égal<br>égal<br>différent |
| non | 1 booléen | | négation logique |
| et<br>ou | 2 booléens | | conjonction<br>disjonction |

TAB. 2 – Opérateurs du langage

| | |
|---|---|
| Plus forte classe 1 | non $-$ (négation) |
| classe 2 | et $\times$ / div mod |
| classe 3 | ou $+$ $-$ |
| Plus faible classe 4 | $= \neq < \leq > \geq$ |

TAB. 3 – Priorité des opérateurs

### Expressions

Une expression représente une valeur d'un certain type. Une expression est construite à partir de littéraux du type, de variables du type et d'opérateurs qui produisent une valeur de ce type. L'évaluation de l'expression se fait suivant les priorités des opérateurs. Les parenthèses peuvent être utilisées pour changer l'ordre d'évaluation.

### Variables (déclaration, affectation)

Une variable est un objet contenant une valeur d'un certain type, cette valeur pouvant changer pendant l'exécution de l'algorithme. Toute variable doit être *déclarée* (avec le mot clé `var`) avant d'être utilisée. La déclaration fixe le type de la variable mais pas sa valeur, qui est *indéterminée* (c'est à dire n'importe quoi) avant d'avoir été affectée. Il est donc nécessaire de fixer la valeur d'une variable avant de l'utiliser dans une expression.
Le nom d'une variable représente sa valeur à l'instant considéré sauf s'il apparaît à gauche du symbole de l'affectation $\leftarrow$, qui permet d'affecter une valeur à la variable.
Dans l'instruction `i←i+1` (où `i` est une variable de type `entier`) le deuxième `i` représente la valeur de la variable au moment où l'expression `i+1` est évaluée. Le premier `i` indique que la valeur de l'expression doit être affectée à la variable `i`. Cette instruction a donc pour effet d'augmenter de 1 la valeur de la variable `i`.

### Instructions d'entrée/sortie

`lire` permet de lire au clavier une valeur et de l'affecter à la variable passée en paramètre.
`écrire` permet d'afficher à l'écran les valeurs des expressions passées en paramètres.

Exemples

```
...
var i : entier
début
    lire(i)                 ⇐ 12
    écrire("i+1=", i+1)     ⇒ i+1=13
    ...
```

### Constructions conditionnelle et alternative

*condition* est une expression booléenne. Ces constructions peuvent être imbriquées, il est alors judicieux de bien utiliser l'indentation.

Instruction conditionnelle

```
si condition alors
        instructions
fin si
```

*condition* est évaluée. Si elle est vraie, les *instructions* sont exécutées avant de passer à la suite. Si elle est fausse, on passe directement à la suite.

Instruction alternative

```
si condition alors
        instructions1
sinon
        instructions2
fin si
```

*condition* est évaluée. Si elle est vraie, les *instructions1* sont exécutées avant de passer à la suite. Si elle est fausse, les *instructions2* sont exécutées avant de passer à la suite.

### Construction de choix

*expression* est une expression de type `entier` ou `car`.

```
selon expression faire
    valeur1 : instructions1
    valeur2 : instructions2
    ...
    autrement : instructionsn
fin selon
...
```

L'expression est tout d'abord évaluée. Si sa valeur est une des valeurs prévues, alors les instructions correspondantes sont exécutées. Sinon les instructions associées au cas `autrement` sont exécutées. Le cas `autrement` est facultatif.

### Constructions itératives ("boucles")

Les boucles permettent de répéter une liste d'instructions tant qu'une certaine condition est vraie, ou jusqu'à ce qu'une certaine condition devienne vraie, ou un certain nombre de fois. Aucune de ces boucles ne peut être interrompue autrement que par le fonctionnement normal (condition devenant fausse dans `tant que`, devenant vraie dans `répéter`, nombre d'itérations déterminé dans `pour`).
On dispose de trois constructions itératives :

```
tant que condition faire
    instructions
fin tant que
```

La condition, qui est une expression booléenne, est évaluée. Si elle est fausse, on passe à l'instruction suivant le `fin tant que`, sinon les *instructions* sont exécutées, la condition est de nouveau évaluée etc.

```
répéter
    instructions
jusqu'à condition
```

Les *instructions* sont exécutées. La *condition* est évaluée. Si elle est vraie, on passe à la suite, sinon les *instructions* sont de nouveau exécutées, la *condition* évaluée etc.

```
pour cpt de début à fin faire
    instructions
fin pour
```

*cpt* (la *variable de boucle*) est une variable de type `entier`. Les expressions entières *début* et *fin* sont évaluées. Les *instructions* sont exécutées *fin* − *début* + 1 fois (ou 0 si négatif), avec *cpt* valant *début*, *début*+1, ... *fin*. Les *instructions* ne doivent pas modifier la valeur de *cpt*. Pour la variante montrée en exemple ci-dessous la valeur de *cpt* est décrémentée à chaque itération.

Exemples

```
tant que i<n faire
    i ← i+1
    écrire("i=",i)
fin tant que
```

```
répéter
    lire(n)
jusqu'à n ≥ 0
```

```
pour i décroissant de n à 1 faire
    f ← f×i
fin pour
```

## Commentaires

Les *commentaires* sont des textes libres encadrés par `(*` et `*)`. Ils ne sont pas destinés au processeur, qui les ignore, mais à une personne qui lit l'algorithme. On les utilise pour donner la *spécification* de l'algorithme, mais aussi pour en faciliter la lecture et la compréhension. Ils peuvent être placés au début de l'algorithme (spécification), entre les instructions (pour donner une propriété vraie à cet endroit, voir exemple ci-dessous), à côté des déclarations de variables (pour indiquer à quoi va servir une variable). Un commentaire ne doit pas paraphraser le texte de l'algorithme (c'est à dire répéter ce qui est déjà évident à la lecture) auquel cas il alourdit le texte de l'algorithme au lieu de le clarifier.

Exemples

```
...
p ← 0
pour i de 1 à n faire
    p ← p + m
    (* p = m × i *)
fin pour
(* p = m × n *)
...
```

# CS110 – Fiche de référence du langage algorithmique – Deuxième partie

## Procédures et fonctions

Les procédures et fonctions sont des sous-algorithmes qui, comme tout algorithme, ont un nom, une *spécification*, un environnement et une liste d'instructions à exécuter. Leur environnement comprend éventuellement des *paramètres* en plus des *variables locales*.
Les procédures et fonctions permettent d'*abstraire* une séquence d'instructions. Elles sont définies dans la partie déclaration d'un algorithme, après les déclarations de variables. Une procédure s'utilise comme une instruction, alors qu'une fonction produit une valeur, et s'utilise donc dans une expression. On dit qu'on *appelle* une procédure ou fonction quand on l'utilise.

**Définition.** La *définition* d'une procédure ou fonction comprend :
- une en-tête comprenant le nom, la liste des *paramètres formels*, chacun avec leur type et leur mode de passage, puis dans le cas d'une fonction le type de la valeur produite. Suit un commentaire donnant la spécification de la procédure ou fonction. Dans les déclarations de paramètres, le mot clé `var` indique que le paramètre est passé par variable[1], son absence qu'il est passé par valeur. Les paramètres peuvent être :
  - d'entrée : ils fournissent une donnée à la procédure,
  - de sortie : ils permettent à la procédure de fournir un résultat,
  - d'entrée/sortie : ils jouent les deux rôles.
- éventuellement des déclarations (indiquées par le mot-clé `var`) de *variables locales* qui n'existent que pendant l'exécution de la procédure.
- une séquence d'instructions (le *corps* de la procédure) encadrée par les mot-clés `début` et `fin`. Ces instructions peuvent manipuler les paramètres formels et les variables locales, qui constituent l'environnement local de la procédure.

Une fonction produit une valeur qui doit être spécifiée par l'instruction `retour(`*`expression`*`)`. Cette instruction doit être la dernière d'une séquence d'exécution de la fonction[2]. Elle ne peut donc en aucun cas apparaître à l'intérieur d'une boucle.

Exemples

```
procédure bienvenue(nom:chaîne; sexe:car)
(* souhaite la bienvenue *)
début
    si sexe='M' alors
        écrire("Soit le bienvenu, cher ", nom)
    sinon
        écrire("Soit la bienvenue, chère ", nom)
    fin si
fin
```

```
fonction max(a,b:entier):entier
(* retourne le maximum de a et b *)
début
    si a<b alors retour(b)
    sinon retour(a)
    fin si
fin
```

```
fonction puissance(a:réel; n:entier):réel
(* calcule la valeur de a puissance n *)
var p:réel
    i:entier
début
    p ← 1
    pour i de 1 à n faire
        p ← p × a
    fin pour
    retour(p)
fin
```

```
procédure échange(var a,b:entier)
(* échange les valeurs des variables a et b *)
var tmp:entier
début
    tmp ← a
    a ← b
    b ← tmp
fin
```

[1] On dit aussi par référence ou par adresse.

[2] Elle peut donc apparaître plusieurs fois si il y a plusieurs séquences d'exécution possibles, comme c'est le cas pour la fonction `max` de l'exemple.

**Utilisation.** L'*appel* d'une procédure/fonction se fait en fournissant un *paramètre effectif* avec le bon type pour chaque *paramètre formel*. Dans le cas d'un paramètre passé par valeur, le paramètre formel reçoit une copie du paramètre effectif, qui doit être une expression. Une modification du paramètre formel n'a aucune incidence sur le paramètre effectif. Dans le cas d'un paramètre passé par variable, le paramètre formel doit être considéré comme un *alias*, un nouveau nom pour le paramètre effectif, qui doit être une variable. Toute manipulation du paramètre formel est donc en fait effectuée sur le paramètre effectif (cas de l'appel de `échange` ci-dessous). Les paramètres de sortie et d'entrée/sortie doivent donc toujours être passés par variable.

Exemples

```
...
i ← 3
j ← i+1
échange(i, j)
écrire("i=", i, "j=", j)      ⇒ i=4 j=3
```

### Types complexes

**tableau** un tableau est une séquence de variables de même type, chacune désignée par un indice entier. On précise le type des éléments et l'intervalle des indices.

Soit `t` une variable de type `tableau[`*a*`..`*b*`] de` *type* où *a* et *b* sont deux constantes entières. `t` est une séquence de *b-a+1* variables de type *type*, accessibles par les indices de *a* à *b* : `t[`*a*`], t[`*a*`+1], ...t[`*b*`]`.

Chaque élément d'un tableau peut lui même être un tableau, on obtient alors un tableau à plusieurs dimensions. Par exemple `tableau[1..10] de tableau[1..47] de réels` désigne un tableau de 10 tableaux chacun de 47 éléments, soient 470 éléments au total. Une notation raccourcie pour ce type est `tableau[1..10, 1..47] de réels`. Chaque élément peut alors être manipulé en indiquant les valeurs des deux indices.

**article** un article est un agrégat de variables de types non forcément égaux chacune désignée par un nom. Chaque composante est appelée un *champ* de la variable et est désignée par *nomarticle.nomchamp* .

Exemples

```
...
var cs110 : tableau[1..47] de réels
    notes : tableau[1..10, 1..47] de réels
    étudiant : article
                   nom : chaîne
                   prénom : chaîne
                   moyenne : réel
               fin article
début
    cs110[1] ← 12.5
    notes[2,5] ← 13.0
    étudiant.nom ← "Dupont"
    ...
```

### Définition de constantes et de types

Il est courant que l'on ait à utiliser une valeur constante dans un algorithme. Dans l'exemple ci-dessus, 47 représente le nombre d'élèves. Afin de le faire apparaître clairement et de faciliter la maintenance[3] on peut donner un nom, par exemple `nbélèves` à cette constante.
Les constantes se définissent dans la partie déclaration d'un algorithme (ou procédure ou fonction), avec le mot clé `const`. Comme leur nom l'indique, elles ne peuvent pas être modifiées. Elles représentent partout où elles apparaissent la valeur à laquelle elles sont égales.

[3] Imaginez que l'on veuille réutiliser l'algorithme l'année prochaine, alors que le nombre d'élèves a changé...

Pour alléger l'écriture des algorithmes, on peut *nommer* un type complexe. On choisit un identificateur pour le type et on déclare l'association avec le mot clé `type` dans la partie déclarations de l'algorithme, avant les déclarations de variables.
Les déclarations comprennent donc dans l'ordre des déclarations de constantes, de types, de variables et enfin de procédures et fonctions.

Exemples

```
const nbélèves = 47
      nbmatières = 10
type  tnotes1A = tableau[1..nbmatières, 1..nbélèves] de réels
      tétudiant = article
                     nom : chaîne
                     prénom : chaîne
                     moyenne : réel
                  fin article

var   notes : tnotes1A
      étudiant : tétudiant
```

## Récursivité

Une fonction ou procédure peut être *récursive*, c'est à dire contenir un appel à elle même. La récursivité est une technique puissante pour écrire des algorithmes résolvant un problème pouvant s'exprimer sous la forme d'équations de récurrence. On peut souvent résoudre un problème soit sous une forme *itérative* (c'est à dire utilisant une ou des boucles) soit sous une forme *récursive* (c'est à dire utilisant une ou des fonctions/procédures récursives).
Un cas moins courant est celui de la *récursivité indirecte*, où une procédure A appelle une procédure B qui contient elle même un appel à A (ou un appel à C qui elle même contient un appel à A etc). Comme exemple, on montre ci-dessous une façon originale (et assez peu efficace...) de tester la parité d'un entier positif.

Exemples

```
fonction pgcd(a,b:entier):entier
(* calcule le pgcd de a et b *)
début
    si b = 0 alors
           retour(a)
    sinon
           retour(pgcd(b, a mod b))
    fin si
fin

fonction pair(n:entier):booléen
(* n pair ? (n positif) *)
début
    si n=0 alors retour(vrai)
    sinon retour(impair(n-1))
    fin si
fin

fonction impair(n:entier):booléen
(* n impair ? (n positif) *)
début
    si n=0 alors retour(faux)
    sinon retour(pair(n-1))
    fin si
fin
```

## Allocation de la mémoire

### Type pointeur

Une variable pointeur est une variable dont la valeur est l'adresse d'une autre variable. Si le pointeur a pour nom `p`, alors la deuxième est dite *pointée par* `p`. On dit aussi que `p` *pointe vers* la variable pointée.

### Manipulation des pointeurs

`nouveau(p)` prend en paramètre une variable `p` de type ↑*type* (que l'on lit "pointeur sur *type*"). Une variable de type *type* est créée et son adresse est stockée dans la variable `p` qui *pointe* alors vers la variable nouvellement créée. `p↑` (que l'on lit "`p` flèche") représente la variable pointée, qui peut être manipulée comme n'importe quelle variable.

`libère(p)` prend en paramètre une variable `p` de type ↑*type*. La variable *pointée* par `p` (`p↑`) est supprimée. La valeur de `p` devient indéterminée.

Un pointeur ne peut être modifié que par les instructions `nouveau` et `libère`, par l'affectation de la valeur `nil` (qui par convention est la valeur du pointeur que ne pointe sur rien) et par l'affectation entre pointeurs.

Exemples

```
...
var pa, pb:↑entier
début
   écrire(↑pa)     FAUX ! ↑pa n'existe pas, pa a une valeur indéterminée
   nouveau(pa)
   écrire(↑pa)     FAUX ! ↑pa a une valeur indéterminée
   ↑pa ← 12
   écrire(↑pa)     ⇒ 12
   pb ← pa
   ↑pa ← 13
   écrire(↑pb)     ⇒ 13
   ...
```

### Intérêt

Les pointeurs permettent de construire des structures de données dont la taille n'est pas connue à priori et n'est pas limitée (autrement que par la mémoire disponible). L'élément de base de ces structures est la *cellule*, article contenant un champ pour l'information stockée et un champ pointeur vers une (autre) cellule (on retrouve ici la notion de récursivité...)

Exemples

```
type cellule = article
                   val:élt
                   suiv:↑cellule
               fin article

     liste = ↑cellule
```