Le langage et ses traductions
Axiomatique impérative

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 22 mai 2018

Table des matières

alg - Le langage et ses traductions

Objectif

Ce module donne la syntaxe et la description sémantique de l'axiomatique impérative dans le langage algorithmique et ses traductions dans divers langages de programmation.

1 Le langage

1.1 Le langage

Règles, Alphabet du langage, Éléments lexicaux

Tout **langage** de programmation possède :

- Des **règles de présentation** (aspect lexical).
- Des **règles d'écriture** (une syntaxe).
- Des **règles d'interprétation** (une sémantique).

L'**alphabet du langage** permet de construire les mots et les expressions. Comme toute phrase d'un langage courant, il est composé de mots appelés **lexèmes** ou **éléments lexicaux** (*tokens*).

((alg)) **Le langage algorithmique**

Langage simplifié **indépendant** de tout langage de programmation.

((alg)) **Alphabet du langage**

C'est celui de la langue française (majuscules et minuscules, avec ou sans accents), les chiffres 0-9 et quelques symboles spéciaux (opérateurs et ponctuation). Un caractère accentué est différent de son équivalent sans accent. Donc `a`, `à` et `â` sont des mots différents.

((alg)) **La casse**

Le langage **n'est pas sensible** à la casse[1].
Ceci signifie que `cout`, `Cout` et `COUT` réfèrent un même mot.

1.2 Les mots-clés

Mots-clés (*keywords*)

Dits aussi **mots réservés**, ils constituent le vocabulaire du langage. Ils forment l'ossature d'un algorithme en définissant la structure de ses données ou de ses instructions.

Propriété

Les mots-clés ont une **signification inchangeable et prédéfinie**, c.-à-d. qu'ils ne peuvent pas être redéfinis.

((alg)) **Liste des mots-clés**

(1) Procédurale (2) Objet

```
(1) action afficher afficherss ajout algorithme alors autre à booléen caractère cas
    chaîne constante D DR début écriture entier et faire faux fermer fichier fin
    findefichier finitérer finpour finrépéter finselon finsi fintantque fois fonction
    itérer jusqu'à lecture mettre non ou ouvrir pas pour prendre R remarque retourner
    répéter réel saisir selon si sinon tantque type variable vrai
```

1. Il ne fait pas de distinction entre les minuscules et les majuscules.

```
(2) classe enum finclasse privé protégé public typedef
```

Environnements IDE
Les mots-clés y sont en gras (ou en couleurs).

1.3 Les identifiants

Identifiant
Dit aussi **identificateur**, c'est un **nom** choisi par le concepteur pour désigner les entités : variable, fonction, type, etc. Cette définition reste valable dans son domaine de validité, à condition qu'elle ne soit pas recouverte par une autre définition. Il **ne doit pas** correspondre à un mot-clé.

Identifiant
Séquence de lettres (A...Z, a...z) avec ou sans accents, de chiffres (0...9) ou du caractère souligné (_, *underscore* en terme anglo-saxon). Il doit commencer par une lettre ou un souligné.

Le souligné est significatif
Ainsi `a`, `a_` et `_a` sont des identifiants différents.

1.4 Séparateurs et espaces blancs

Espace blanc
Suite de un ou plusieurs caractères choisis parmi : espace, tabulation horizontale, tabulation verticale, changement de ligne ou changement de page.

Séparateurs
Comprend les opérateurs (`+`, `-`, `*`, `/`...) et les caractères de « ponctuation ».

Quelques séparateurs

	Symbole	Rôle
:	deux-points	sépare un identifiant de son type dans une liste
()	parenthèses	encadrent des expressions
[]	crochets	symboles du composant tableau
.	point	accès à un composant structure
,	virgule	sépare les éléments d'une liste
;	point-virgule	sépare les éléments de déclaration
' '	quotes	encadrent les caractères
" "	guillemets	encadrent les chaînes de caractères

1.5 Types intégrés

Types intégrés (*builtins*)
Dits aussi **types de base**, **types fondamentaux** ou encore **types primitifs**, ils correspondent aux données qui peuvent être traitées directement par le langage.

((alg)) **Types intégrés**

- `Entier` : Pour les entiers relatifs $\mathbb{Z}$
- `Réel` : Pour les nombres réels (approchés) $\mathbb{R}$
- `Booléen` : Le domaine $\mathbb{B}$ des booléens (vrai / faux)
- `Caractère` : Le domaine $\mathbb{A}$ des caractères (alphanumériques et ponctuations)
- `Chaîne` : Le domaine $\mathbb{T}$ des textes (suite de caractères)

Variables booléennes
On veillera à ne pas utiliser les valeurs 0 et 1 pour les variables booléennes, même si leur emploi est correct dans beaucoup de langages de programmation.

1.6 Littéraux

Littéral
Valeur explicite au sein d'un algorithme.
Il possède *un type* déterminé par sa valeur (entier, réel, caractère...).

Notation scientifique d'un réel
Expression « m e p » qui signifie $m \cdot 10^p$ (où `m` est une suite de chiffres décimaux et `p` un entier). Ainsi, $12.4e-5 \equiv 12.4 \cdot 10^{-5} \equiv 0.000124$. Elle permet d'exprimer de très petits nombres (ex : 2.5e-201) et de très grands nombres (ex : 5e156).

Point décimal
Un réel se distingue d'un entier par l'ajout d'un point (.) à la fin.
Pour la lisibilité, préférez .0 (ex. 5.0 (réel) mais 5 (entier)).

((alg)) **Littéraux**

- `Entier` : Suite de chiffres éventuellement préfixé par un signe (+ ou −).
- `Réel` : S'écrit en notation décimale ou en notation scientifique.
- `Booléen` : Identifie le `Vrai` et le `Faux`.
- `Caractère` : Se place entre quotes (').
- `Chaîne` : Se place entre guillemets (").

2 Structure d'un algorithme

2.1 Structure générale

((alg)) **Structure générale**

```
Algorithme nomAlgo
| déclaration_des_variables_et_constantes
Début
| saisie_des_données
| instructions_utilisant_les_données_lues
| communication_des_résultats
Fin
```

Explication

Un algorithme est constitué par :

- Un **en-tête** qui donne un **nom** à l'algorithme. Ce nom `nomAlgo` n'a pas de signification particulière mais doit respecter les règles de formation des identifiants.
- Un **corps** formé de deux parties :
  - Encadrée par les mots `Algorithme` et `Début`, la **première** (facultative) a un **rôle descriptif**. Elle est aussi appelée **dictionnaire des données**.
  - Encadrée par les mots `Début` et `Fin`, la **deuxième** a un **rôle constructif**. Elle décrit les traitements à réaliser pour aboutir au résultat recherché.

L'algorithme commence son exécution sur le mot `Début`, se déroule séquentiellement et se termine sur le mot `Fin`.

Conseil

On veillera à ce qu'un algorithme tienne sur une vingtaine de lignes (donc, en pratique, sur un écran de 40 x 80 caractères ou une page). Ceci implique que, si votre algorithme devait être plus long, il faudra le découper, comme nous le verrons plus loin.

2.2 Commentaire

Commentaire (narratif)

Texte qui n'est **ni lu, ni exécuté** par la machine. Il est essentiel pour rendre plus lisible et surtout plus compréhensible un algorithme par un être humain.

((alg)) **Commentaire orienté ligne**

```
# ...
```

2.3 Inclure un fichier

alg : Inclure un fichier

```
Inclure "NomFichier"
```

Explication

Insère le fichier `NomFichier` à la place de la primitive `Inclure`. Par défaut, l'extension du fichier est ".alg".

3 Déclarations

3.1 Déclaration de variables

Déclaration de variables

Consiste à associer un type de données à une ou un groupe de variables. Toute variable doit impérativement avoir été déclarée avant de pouvoir figurer dans une instruction exécutable.

((alg)) **Déclaration de variables**

```
Variable nomVar : TypeVar
Variable nomVar1, nomVar2, ... : TypeVar
```

Explication

Déclare des variables d'identifiants `nomVarI` (le nom) de type `TypeVar`.

3.2 Définition de constante

Constante

Littéral à lequel est associé un **identifiant** (par convention, écrit en MAJUSCULES) afin de rendre plus lisible et simplifier la maintenance d'un algorithme. C'est donc une information pour laquelle nom, type et valeur sont figés.

((alg)) **Définition de constante**

```
Constante nomConst <- expression
```

Explication

Définit la constante d'identifiant `nomConst` de type `TypeConst` et lui affecte une valeur (littéral ou expression) spécifiée. Le type `TypeConst` est défini implicitement par la valeur de l'expression. Les types de constante autorisés sont basés sur les types intégrés du langage.

Valeur immuable fixée à la déclaration

Toute tentative de modification est rejetée par tout compilateur qui signalera une erreur (ou un avertissement).

3.3 Définition d'une énumération

Type énuméré

Définit un ensemble de constantes entières associées une à une à des identifiants ou *énumérateurs*. Deux avantages :

- Une indication claire des possibilités de la variable lors de la déclaration.
- Une lisibilité du code grâce à l'utilisation des valeurs explicites.

((alg)) **Définition d'une énumération**

```
Énumération NomType { nomVal1, nomVal2... }
```

Explication

Introduit `NomType` dont les valeurs discrètes sont `nomVal1`, `nomVal2`, etc.

Lien avec les entiers

Nous adoptons la syntaxe suivante :

```
NomType(n) // donne l'énumération de nomType numéro n (on commence à 1)
position(nomValI) // donne l'entier associé à nomType
```

Quid des langages de programmation ?

Chaque langage de programmation propose sa propre technique de conversion de valeurs.

- Certains langages (comme Java) proposent un type énuméré complet.
- D'autres (comme C et C++) proposent un type énuméré incomplet mais qui permet néanmoins une écriture comme celle ci-dessus.
- D'autres langages ne proposent rien. Pour ces derniers, l'astuce est de définir des constantes entières qui vont permettre une écriture proche de celle ci-dessus (mais sans une déclaration explicite).

3.4 Synonyme de type

Synonyme de type

Alias d'un type existant (lorsqu'un nom de type est trop long ou est difficile à manipuler).

((alg)) **Synonyme de type**

```
Typedef TypeAbrege = TypeExistant
```

Explication

Désigne l'identifiant `TypeAbrege` comme étant un synonyme du type `TypeExistant`.

Typedef = Définition

`Typedef` N'introduit pas de nouveau type mais un **nouveau nom** pour le type.

4 Instructions élémentaires

4.1 Primitives

Bibliothèque
Ensemble de fonctionnalités ajoutées à un langage de programmation.
Chaque bibliothèque décrit un thème.

Primitives
Noms de fonctions (`abs`, `log`, `sin`...), d'opérateurs (`div`, `mod`...) ou de traitement (`afficher`, `saisir`...). Elles acceptent un ou plusieurs paramètres et jouent le même rôle syntaxique qu'un identifiant.

Appel d'une primitive

```
P(x,...) # procédure
r <- F(x,...) # fonction
```

Explication
Appelle (on dit aussi **invoque**) la procédure `P` ou la fonction `F` avec les arguments `x`... Dans le cas de fonction, la valeur retournée peut être utilisée en tant que macro-expression.

4.2 Saisie de données

Saisie de données

```
Saisir(nomVar1, nomVar2, ..., nomVarN)
```

Explication
Ordonne à la machine de lire des valeurs `valI` depuis le clavier et de les stocker dans les variables `nomVarI` (qui doivent exister c.-à-d. déclarées).

Remarque
Par défaut, ce qui est tapé au clavier est envoyé à l'écran et temporairement placé dans un tampon pour permettre la correction d'erreurs de frappe. On peut donc se servir des touches [←] et [Suppr] pour effacer un caractère erroné ainsi que les flèches [<] et [>] pour se déplacer dans le texte.

alg : Nature de la donnée saisie
Une erreur se déclenche si elle ne correspond pas à celle de la variable affectée. Les langages de programmation proposent des moyens pour palier le « plantage » du programme en cas d'erreur de saisie. L'algorithmique considère que l'utilisateur saisit toujours des données ayant le type souhaité.

4.3 Affichage de résultats

((alg)) **Affichage de résultats**

```
AfficherSS(expr1,expr2,...,exprN) # SANS retour de ligne
Afficher(expr1,expr2,...,exprN)   # AVEC retour de ligne
```

Explication
Ordonne à la machine d'afficher les **valeurs** des expressions `exprI`. Par défaut, elles ne sont pas séparées par des espaces. Ajoutez le(s) délimiteur(s) si nécessaire.

Variables devant être initialisées
On ne peut *afficher* que des expressions dont les variables qui la composent ont été affectées préalablement.

4.4 Formats d'édition

Format d'édition d'une expression
Indique de quelle façon doit être cadrée l'expression à afficher. Il s'applique aux valeurs de type `Chaîne`, `Entier` ou `Réel`.

((alg)) **Formats d'édition**

```
exprChaine : largeur1
exprEntier : largeur1
exprReel : largeur1 : largeur2
```

Explication
Définit le format d'édition. L'entier `largeur1` indique sur combien de caractères doit être écrite l'expression. L'entier `largeur2` précise le nombre de chiffres après le point décimal des réels.

Règle d'affichage
Si le format est :

- **Égal** à la longueur nécessaire à l'édition de la valeur : la valeur est écrite telle quelle.
- **Inférieur** à la longueur : il est ignoré et la valeur est écrite sur la longueur nécessaire.
- **Supérieur** à la longueur : le système effectue un cadrage de la valeur à afficher à l'intérieur du format qui lui a été spécifié. Les données numériques sont cadrées à droite sur le point décimal, et les données alphanumériques cadrées à gauche.

4.5 Affectation interne

Affectation interne
Opération qui **fixe une valeur** à une variable.

Affectation interne

```
nomVar <-- expression
```

Explication
Place la valeur de l'`expression` dans la zone mémoire de la variable de nom `nomVar`. En algorithmique, le symbole <- (qui se lit « devient ») indique le sens du mouvement : de l'expression située à droite **vers** la variable à gauche.

Rappel
Toutes les variables apparaissant dans l'`expression` doivent avoir été affectés préalablement. Le contraire provoquerait un arrêt de l'algorithme.

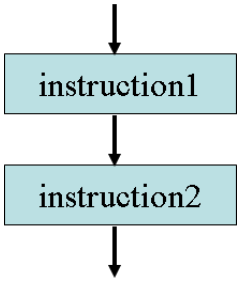
Conversions implicites
Il est de règle que le résultat de l'expression à droite du signe d'affectation soit de même type que la variable à sa gauche.

4.6 Instruction composée

Instruction
Ordre donné à l'ordinateur qui a pour effet de changer l'état de la mémoire ou le déroulement du programme ou bien de communiquer avec les unités périphériques (clavier, écran, imprimante, etc.).

Instruction composée ou Bloc
Regroupement syntaxique de 0, 1 ou plusieurs instructions (et déclarations) comme une unique instruction.

Séquentialité
Les algorithmes et programmes présentés sont exclusivement séquentiels : l'`instruction2` ne sera traité qu'une fois l'exécution de l'`instruction1` achevée.



Bloc

```
Début
  |  instruction1
  |  instruction2
  |  ...
Fin
```

Conventions usuelles

Savoir présenter un algorithme, c'est montrer que l'on a compris son exécution.

- Chaque ligne comporte une seule instruction.

  alg Une instruction se termine à la fin d'une ligne.

- Les indentations sont nécessaires à sa bonne lisibilité. Ainsi :

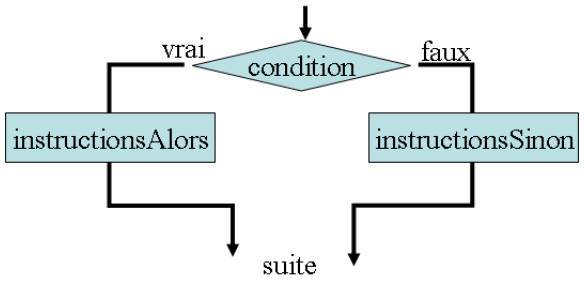
  alg Alignez les mots `Début` et `Fin` l'un sous l'autre.

5 Structures conditionnelles

5.1 Sélective Si

Sélective Si

Elle traduit : `Si` la `condition` est vraie, exécuter les `instructionsAlors`, `Sinon` exécuter les `instructionsSinon`. Il s'agit d'un choix binaire : **une et une seule** des deux séquences est exécutée.



La `condition` peut être simple ou complexe (avec des parenthèses et/ou des opérateurs logiques `Et`, `Ou`, `Non`).

((alg)) **Sélective Si**

```
Si condition Alors
  |  instructionsAlors
Sinon
  |  instructionsSinon
FinSi
```

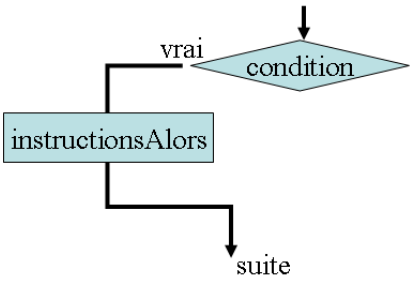
Remarque

Le corps de la clause « alors » commence après le mot `Alors` et se termine sur le mot `Sinon`. Celui de la clause « sinon » commence après le mot `Sinon` et se termine sur le mot `FinSi`.

5.2 Sélective Si-Alors

Sélective Si-Alors

Forme restreinte de la structure `Si` (sans clause `Sinon`).



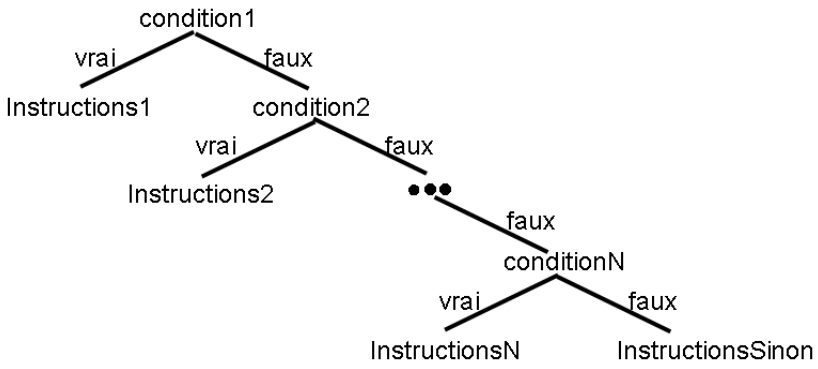
((alg)) **Sélective Si-Alors**

```
Si condition Alors
  |  instructionsAlors
FinSi
```

5.3 Sélective Si-Sinon-Si

Sélective Si-Sinon-Si

Elle évalue successivement la `conditionI` et exécute les `instructionsI` si elle est vérifiée. En cas d'échec des `n` conditions, exécute les `instructionsSinon`.



((alg)) **Sélective Si-Sinon-Si**

```
Si condition1 Alors
  |  instructions1
Sinon Si condition2 Alors
  |  instructions2
Sinon Si...
  |  ...
Sinon Si conditionN Alors
  |  instructionsN
Sinon
  |  instructionsSinon
FinSi
```

((alg)) **Si-cascade versus Si-Sinon-Si**

Question Comment le langage algorithmique fait-il la distinction entre une structure `Si-Sinon-Si` et des structures `Si` en cascade ?

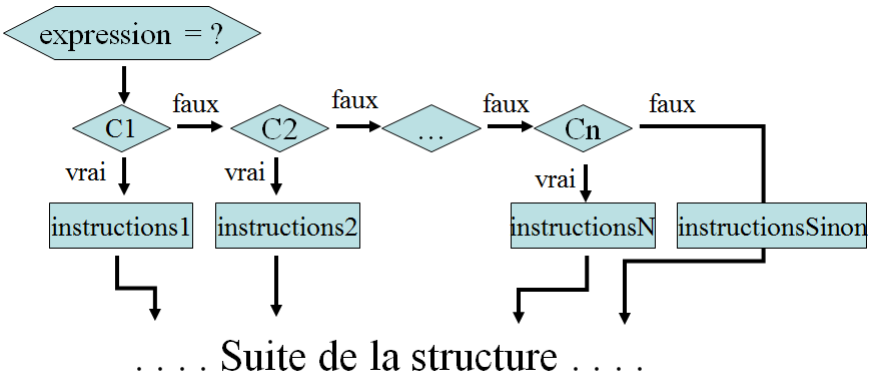
Réponse Lorsque les mots-clés `Sinon` et `Si` se retrouvent de suite sur une même ligne, le langage active l'interprétation d'une structure `Si-Sinon-Si`.

5.4 Sélective Selon (listes de valeurs)

Sélective Selon (listes de valeurs)

Elle évalue l'`expression` et n'exécute que les `instructionsI` qui correspondent à la valeur

ordinale `Ci` (c.-à-d. de type entier ou caractère). La clause `Cas Autre` est facultative et permet de traiter tous les cas non traités précédemment. Il s'agit de l'instruction multi-conditionnelle classique des langages.



Sélective Selon (listes de valeurs)

```
Selon expression
  |  Cas liste1 de valeurs séparées par des virgules
  |     |  instructions1
  |     |  ...
  |  Cas listeN de valeurs séparées par des virgules
  |     |  instructionsN
  |  Cas Autre
  |     |  instructionsSinon
FinSelon
```

Remarque

Veillez à ne pas faire apparaitre une même valeur dans plusieurs listes.

Selon v.s. Si

Le `Selon` est moins général que le `Si` :

- L'expression doit être une valeur discrète (`Entier` ou `Caractère`).
- Les cas doivent être des *constantes* (pas de variables).

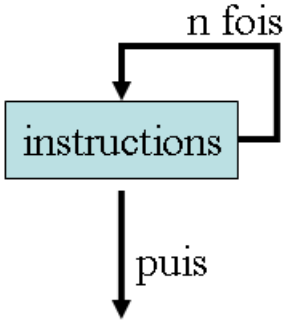
Si ces règles sont vérifiées, le `Selon` est plus efficace qu'une série de `Si` en cascade (car l'expression du `Selon` n'est évaluée qu'une seule fois et non en chacun des `Si`).

6 Structures répétitives

6.1 Répétitive Itérer

Répétitive Itérer

Elle traduit : Exécuter `n` fois les `instructions`, avec `n` un entier positif.
Finitude assurée.



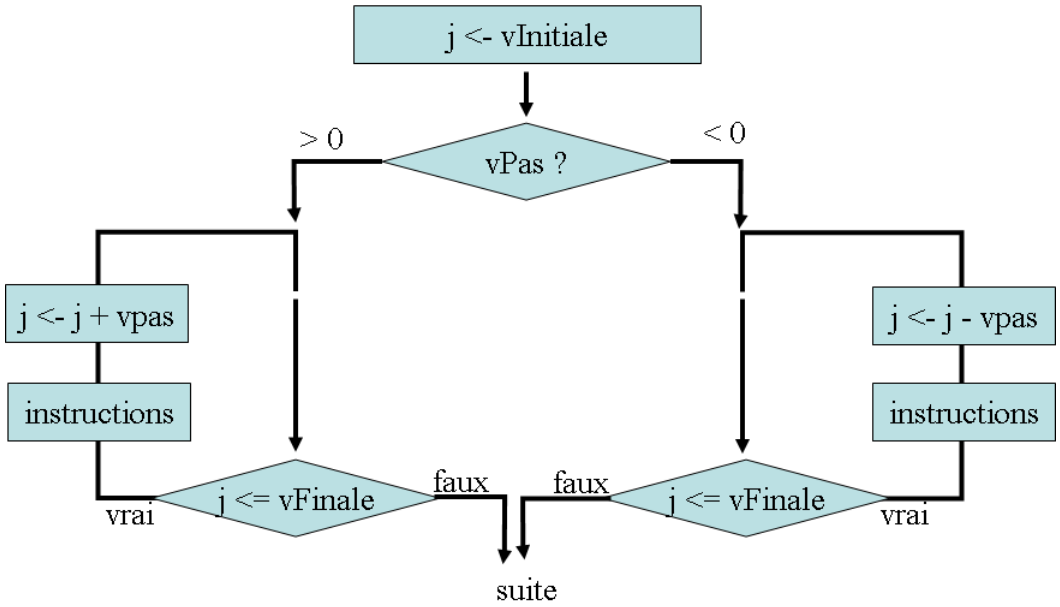
Répétitive Itérer

```
Itérer n Fois
|  instructions
FinItérer
```

6.2 Répétitive Pour

Répétitive Pour

Elle traduit : Exécuter les `instructions`, `Pour` une variable de boucle `nomVar` (entière ou réelle) dont le contenu varie de la valeur initiale `valDeb` à la valeur finale `valFin` par pas de `valPas` (par défaut de 1). Finitude assurée.



Terminologie
La variable utilisée dans la boucle `Pour` s'appelle la **variable de boucle**, **variable de contrôle**, **indice d'itération** ou **compteur de boucle**. En général, son nom se réduit simplement à une lettre, par exemple `j`.

Répétitive Pour

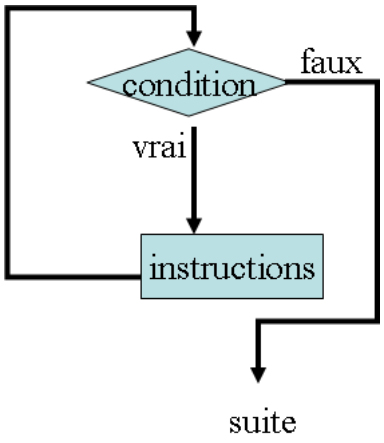
```
Pour nomVar <- valDeb à valFin [ Pas valPas ] Faire
  |  instructions
FinPour
```

Les valeurs initiale, finale et de pas (cas de variables ou d'expressions de calcul) ne sont évaluées qu'une unique fois avant le premier tour de boucle.

Remarque
Il peut n'y avoir aucune répétition (si la valeur initiale est supérieure à la valeur finale pour un pas positif, ou l'inverse pour un pas négatif). Le pas, par défaut de 1, détermine ce qui est ajouté à la variable après chaque itération.

6.3 Répétitive TantQue

Répétitive TantQue (répétition a-priori)
Elle traduit : `TantQue` la `condition` est vraie, exécuter les `instructions`.
Si la `condition` est fausse dès le début, la tâche n'est jamais exécutée.



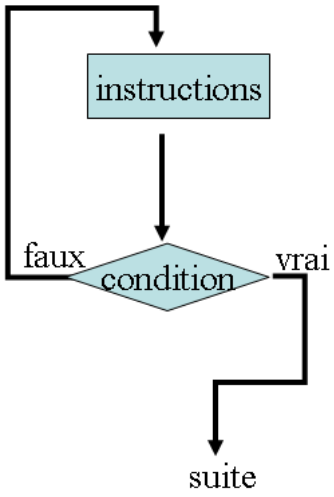
Boucle infinie
La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **fausse**. Dans le cas contraire, la boucle va tourner sans fin (condition indéfiniment vraie) : c'est ce qu'on appelle une **boucle infinie**.

Répétitive TantQue

```
TantQue condition Faire
  |  instructions
FinTantQue
```

6.4 Répétitive Répéter

Répétitive Répéter (répétition a-posteriori)
Elle traduit : Exécuter les `instructions` `Jusqu'à` ce que la `condition` est vraie.



Boucle infinie
La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **vraie** pour arrêter l'itération.

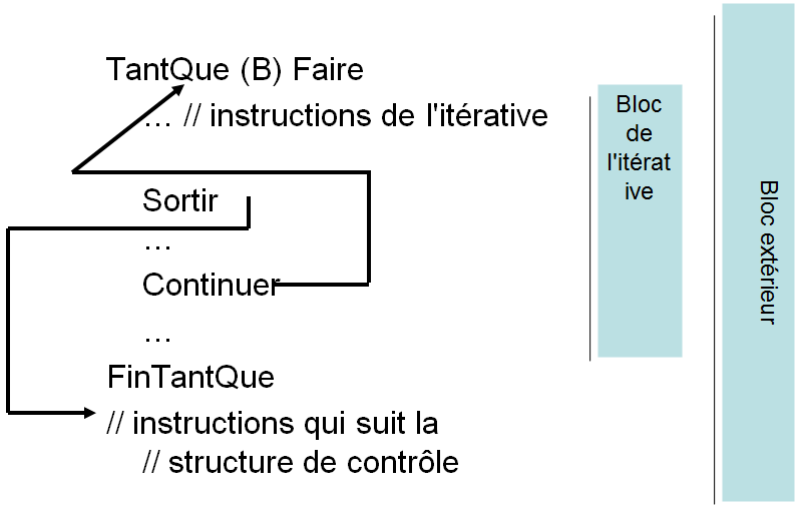
((alg)) **Répétitive Répéter**

```
Répéter
  |  instructions
Jusqu'à condition
```

6.5 Ruptures de séquence (de bloc)

Ruptures de séquence (de bloc)
On en distingue deux :

- `Sortir` : Interrompt l'exécution de la structure de contrôle en provoquant un saut vers l'instruction qui suit la structure de contrôle.
- `Continuer` : Interrompt l'exécution des instructions du bloc d'une **répétitive** et provoque la ré-évaluation de la condition de continuation afin de déterminer si l'exécution de l'itérative doit être poursuivie (avec une nouvelle itération).



Utilisez-les avec parcinomie

Car elles ne permettent pas de réaliser une preuve formelle d'un algorithme (cf. @[Preuve et Notations asymptotiques]).

((alg)) **Rupture Sortir**

```
SortirSi
```

((alg)) **Rupture Continuer**

```
Continuer
```

7 Fonction

7.1 Profil de fonction

Profil de fonction
Constitué par le nom de la fonction, la liste des types des paramètres d'entrée et le type du résultat.

((alg)) **Profil de fonction**

```
Fonction nomFcn( param1 : T1 ;...; paramN : Tn ) : TypeRes
```

Explication
Définit le **profil** de la fonction d'identifiant `nomFcn` ayant pour paramètres formels les `paramI` de type correspondants `Ti`. Le **type** de la valeur renvoyée est `TypeRes`. La liste est vide si la fonction n'a pas besoin de paramètres.

Pour déclarer plusieurs paramètres ayant le même type, séparez les noms de variables par une virgule.

Remarque
En théorie, le type de la valeur retournée peut être un type simple (entier, réel, booléen...), un type structuré, un tableau ou même un objet (ces types seront vus dans les modules suivants). En pratique il conviendra de s'en tenir aux limitations du langage utilisé.

Remarque
Les paramètres formels deviennent automatiquement des variables locales (cf. plus bas, @[Variable locale]) du module.

7.2 Instruction de retour

((alg)) **Instruction de retour**

```
Retourner( expression )
```

Explication
Renvoie (retourne) au module appelant le résultat de l'`expression` placée à la suite du mot-clé.

alg : Retourner
La primitive ne met pas fin à la fonction comme cela peut être le cas dans certains langages de programmation comme le C/C++ ou Java. S'il y a plusieurs instructions `Retourner` dans la même fonction, cette dernière renvoie à l'appelant le résultat de l'expression de la dernière instruction `Retourner` exécutée.

Dans une fonction

Il doit **toujours** y avoir l'exécution d'une primitive `Retourner`, et ceci quelles que soient les situations (conditions).

En effet, si dans un cas particulier, la fonction s'exécute sans être passée par cette primitive, ceci révèle une incohérence dans la conception de votre fonction car celle-ci aura une valeur inconnue et aléatoire.

7.3 Schéma d'une fonction

((alg)) **Schéma d'une fonction**

```
Fonction nomFcn( paramètresFormels ) : TypeRes
Variable resultat : TypeRes
Début
  |  calcul_du_resultat
  |  Retourner ( resultat )
Fin
```

Explication

Pour des questions de lisibilité et de preuve de programme, il vous est fortement recommandé d'adopter le schéma ci-dessus.

((alg)) **Expression fonctionnelle**

```
Fonction nomFcn( parametresFormels ) : TypeRes
Début
  |  Retourner ( expression )
Fin
```

Explication

Dans le cas d'une expression calculable directement, on peut regrouper le tout : on parle alors d'**expression fonctionnelle**.

7.4 Appel d'une fonction

((alg)) **Appel d'une fonction**

```
v <- nomFcn( a1, ..., aN )
```

Explication

Appelle (on dit aussi *invoque*) la fonction `nomFcn` avec les (éventuels) arguments `aI`. La valeur retournée peut être utilisée en tant que macro-expression.

Fonction = macro-expression
Une fonction retourne **toujours** une information à l'algorithme appelant. C'est pourquoi l'appel d'une fonction **ne se fait jamais** à gauche du signe d'affectation.

8 Procédure

8.1 Profil de procédure

((alg)) **Profil de procédure**

```
Action nomSsp(parametres)
```

Explication
Définit le **profil** de la procédure de nom `nomSsp` ayant pour paramètres formels les `parametres` lesquels décrivent pour chaque paramètre, son nom, son type et sa caractéristique.

8.2 Schéma d'une procédure

((alg)) **Schéma d'une procédure**

```
Action nomSsp(
  D d1 : D1; ...; // les données
  R r1 : R1; ...; // les résultats
 DR m1 : M1; ...) // les modifiés
Début
  # Corps de la procédure:
  # toute action (lecture/écriture) sur une donnée D
  # n'est pas visible à l'extérieur, tandis que toute
  # action sur un résultat R ou un modifié DR s'effectue
  # sur les entités "extérieurs" à la procédure
Fin
```

Explication
Définit la procédure de nom `nomSsp`.

8.3 Paramètres formels

Paramètres Entrants/Sortants/Mixtes
Les paramètres **entrants** ou *données* :

- Ont une **valeur à l'entrée** du module.
- Et seront **consultés à l'intérieur** du module.

Les paramètres **sortants** ou *résultats* :

- Ont une **valeur indéterminée à l'entrée** du module.
- Et seront **utilisables après l'appel** du module.

Les paramètres **mixtes** ou *modifiés* :

- Ont une **valeur à l'entrée** du module.
- Et seront éventuellement **modifiés à l'intérieur** de celui-ci.

((alg)) **Paramètres formels**

```
Action nomSsp( [D|R|DR] nomParam : TypeParam)
Fonction nomFcn( [D|R|DR] nomParam : TypeParam ) : TypeRes
```

Explication
Définit le paramètre d'identifiant `nomParam` et de type `TypeParam`.
Un paramètre :

- **Entrant** est accompagné d'une flèche vers le bas (↓) ou du mot-clé `D` ou rien.
- **Sortant** d'une flèche vers le haut (↑) ou du mot-clé `R`.
- **Mixtes** de la double flèche (↕) ou du mot-clé `DR`.

`D`, `R`, `DR` signifient respectivement « en Donnée », « en Résultat », « en Donnée Résultat ».

alg : Rappel
Pour déclarer plusieurs paramètres :

- Ayant même type et même mode : séparez les noms de paramètres par une virgule.
- Avec des types ou modes différents : placez des points-virgules entre chaque groupe de paramètres.

8.4 Appel d'une procédure

((alg)) **Appel d'une procédure**

```
nomSsp(d1, ..., r1, ..., m1, ...)
```

Explication
Appelle la procédure `nomSsp` avec les (éventuels) arguments figurant entre les parenthèses.

Procédure = macro-instruction
Une procédure étant une macro-instruction, un appel de procédure se fait obligatoirement **en dehors de toute expression de calcul**.

9 Tableaux

9.1 Déclaration et initialisation d'un tableau

((alg)) **Déclaration d'un tableau**

```
Variable nomTab : TypeElement [taille] # nombre d'éléments
Variable nomTab : TypeElement [borneMin..borneMax] # bornes explicites
```

Explication

Déclare une variable dimensionnée. Avec : `TypeElement` le type (simple ou non) des éléments constitutifs du tableau, `nomTab` l'identifiant et `taille` son nombre d'éléments. La taille doit être une valeur entière positive (littéraux ou expressions constantes). Les `borneMin` et `borneMax` désignent les bornes inférieure et supérieure (valeurs entières).

((alg)) **Initialisation d'un tableau**

```
nomTab <- {val1, val2, ...}
```

Explication

Initialise une variable dimensionnée. Les `valI` sont des valeurs littérales ou expressions constantes de type compatible `TypeElement` initialisant séquentiellement chacun des éléments du tableau.

9.2 Accès indiciel

((alg)) **Accès indiciel**

```
tab[k]
```

Explication

Accède à la case d'indice `k` d'un tableau `tab`.
Le temps d'accès à l'élément est fixe.

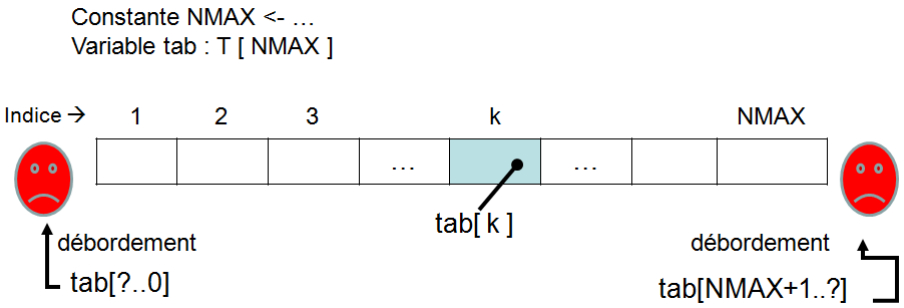
Numérotation des cases

Chaque langage de programmation possède sa propre convention.

- **alg :** Les cases sont numérotées de 1 (par défaut) à `TMAX` (taille du tableau).
- **C/C++, Java, Python :** Ils commencent à indicer un tableau à partir de 0. Ce principe est dit l'**indexation en base 0**.
- **Basic :** Il débute la numérotation à partir de 1 ou 0.
- **Ada :** Il permet de numéroter les cases à partir d'une valeur quelconque.

Dépassement des bornes

Les langages contrôlent le débordement des bornes d'un tableau et déclenchent une erreur qui généralement arrête le programme.



9.3 Déclaration et initialisation d'un k-tableau

((alg)) **Déclaration d'un k-tableau**

```
Variable nomTab : T[bMin1..bMax1, ..., bMinK..bMaxK]
Variable nomTab : T[taille1 , ..., tailleK]
```

Explication

Déclare un tableau k-dimensionnel de nom `nomTab` d'éléments de type `T`. Les `tailleI` sont des valeurs entières positives (littérales ou expressions constantes). Chaque paire de bornes `bMinI` et `bMaxI` limite l'indice correspondant à la i-ème dimension du tableau. Si les `tailleI`s sont indiquées, la borne inférieure équivaut à 1. Les bornes sont également des valeurs entières (littérales ou expressions constantes).

Cas particulier d'un tableau bidimensionnel

```
Variable nomTab : T[ligMin..ligMax, colMin..colMax]
Variable nomTab : T[nligs, ncols] # ligMin=1,colMin=1
```

((alg)) **Initialisation d'un k-tableau**

```
nomTab <- { { {v111, ..., v11k}, ... }
```

Explication

Initialise un tableau k-dimensionnel. Avec : Les `valI` sont des valeurs littérales ou expressions constantes de type `T` initialisant **séquentiellement** chacun des éléments du tableau.

9.4 Accès indiciel

((alg)) **Accès indiciel**

```
tab[k1, k2, ...] # syntaxe normale
tab[k1][k2][...] # autre écriture
```

Explication

Accède à la case indice `(k1, k2, ...)` d'un tableau multidimensionnel `tab`. Il faut spécifier autant d'indices qu'il y a de dimensions dans le tableau.

10 Types complexes

10.1 Définition d'un type structuré

Structure, Champ

Une **structure** permet de regrouper une ou plusieurs variables, de n'importe quel type (structure de données **hétérogène**), dans une entité unique et de la manipuler comme un tout. Chaque élément, appelé **champ** de la structure, possède un nom unique.

((alg)) **Définition d'un type structuré**

```
Type TypeStruct
  |  nomChamp1 : Type1
  |  nomChamp2 : Type2
  |  ...
  |  nomChampN : TypeN
FinType
```

Explication

Crée un nouveau type nommé `TypeStruct` à partir d'autres types élémentaires ou composés déjà définis `TypeI` et d'identifiants respectifs `nomChampI`.

10.2 Déclaration et initialisation d'une variable structurée

Propriété

La déclaration de variables d'un type structuré défini est identique à celle des variables simples.

((alg)) **Déclaration d'une variable structurée**

```
Variable nomVar : TypeStruct
```

Explication

Déclare une variable structurée `nomVar` de type `TypeStruct` (qui doit être défini).

((alg)) **Initialisation d'une variable structurée**

```
nomVar <- { valChamp1, valChamp2, ... }
```

Explication

Initialise une variable structurée `nomVar`. Chaque `valChampI` est la valeur de type `TypeI` définis dans le type `TypeStruct`.

10.3 Accès aux champs d'une structure

((alg)) **Accès aux champs d'une structure**

```
nomVar.nomChamp
```

Explication

Accède au champ `nomChamp` d'une variable structurée `nomVar` (notation « pointée »).

11 Fichiers

11.1 Déclaration de fichier

((alg)) **Déclaration de fichier**

```
Variable f : Fichier
```

Explication
Déclare une variable `f` de type `Fichier`.

11.2 Ouverture d'un canal d'entrée/sortie

((alg)) **Ouverture d'un canal**

```
Ouvrir(f, nomFich, utilisation)
```

Explication
Associe un canal d'entrées/sorties à un fichier et indique le mode d'accès (l'`Utilisation`) du canal d'entrée/sortie : `Lecture`, `Ecriture` ou `Ajout`. La variable `f` sera utilisée pour toutes les opérations sur ce fichier jusqu'à sa fermeture. Le `nomFich` est une chaîne de caractères contenant le nom du fichier à ouvrir avec éventuellement le chemin d'accès à savoir : le nom du disque et le chemin relatif ou absolu permettant d'atteindre le fichier. A défaut, le fichier doit être dans le dossier courant (habituellement le dossier où est sauvegardé le projet en exécution).

Erreur à l'ouverture
Le système peut être dans l'impossibilité d'ouvrir le fichier spécifié pour une ou l'autre des raisons suivantes :

- Tentative d'ouvrir un fichier inexistant en mode lecture.
- Tentative d'ouvrir un fichier qui est déjà ouvert.
- Tentative d'ouvrir un fichier sur un canal d'entrées/sorties invalide.
- Le nom du fichier est invalide : ceci peut être dû au dossier inexistant, au nom du fichier contenant des caractères interdits par le système d'exploitation ou à l'unité de stockage défectueuse ou non disponible.

Dans ce cas le système provoque l'interruption du programme et affiche un message d'erreur précisant la cause de l'erreur.

Fichier en mode écriture
L'ouverture efface automatiquement son contenu s'il existe déjà.

11.3 Fermeture d'un canal d'entrées/sorties

((alg)) **Fermeture d'un canal**

```
Fermer(f)
```

Explication
Ferme le canal d'entrées/sorties `f` précédemment ouvert et purge toutes les mémoires tampon. Dans le cas où le fichier a été ouvert en écriture, cette primitive place la marque spéciale de fin de fichier dans l'élément courant. Une fois le fichier fermé, il n'est plus permis de l'utiliser.

Remarque
Un fichier créé et non refermé risque de contenir des données aléatoires et invalides.

11.4 Lecture depuis un canal d'entrée

((alg)) **Lecture depuis un canal d'entrée**

```
Prendre(f, nomVar1, nomVar2...)
```

Explication
Lit dans le fichier référencé par la variable `f`, ouvert en lecture, des valeurs qui seront affectées aux variables `nomVarI`. Cette opération peut échouer si la fin de fichier est atteinte. Ceci est décelable grâce à la fonction `FinDeFichier`.

Remarque
Le canal d'entrées/sorties doit obligatoirement être associé à un fichier ouvert en mode `Lecture`. Toute tentative de lecture visant un canal d'entrées/sorties associé à un fichier ouvert en mode `Ecriture` ou `Ajout` cause l'arrêt d'exécution de l'algorithme.

11.5 Écriture sur un canal de sortie

((alg)) **Écriture sur un canal de sortie**

```
Mettre(f, expr1, expr2...)
```

Explication
Rajoute des données dans le fichier référencé par la variable `f`, ouvert en écriture, les valeurs des expressions `exprI`. Cette opération peut échouer si le support utilisé pour le fichier est plein.

Remarque
Le canal d'entrées/sorties doit obligatoirement être associé à un document ouvert en mode `Ecriture` ou `Ajout`. Toute tentative d'écriture visant un canal d'entrées/sorties associé à un document ouvert en mode `Lecture` provoque l'arrêt d'exécution de l'algorithme.

11.6 Détection de fin de contenu

Détection de fin de contenu

```
FinDeFichier(f)
```

Explication
Renvoie la valeur `Vrai` si le pointeur de lecture référencé par `f` détecte la **fin de fichier**, `Faux` sinon.

Attention
La primitive n'est applicable qu'aux canaux associés en mode `Lecture`. Toute invocation de la primitive sur un canal associé à un document ouvert en mode `Ecriture` ou `Ajout` cause l'arrêt d'exécution de l'algorithme.

12 Opérateurs

12.1 Opérateurs arithmétiques

Opérateurs arithmétiques

Dits aussi **opérateurs algébriques**, ils agissent sur des opérandes de type numérique.

Opérateurs arithmétiques

Opérateur Mathématique	Signification	Équivalent Algorithmique
+	(unaire) valeur	`+a`
-	(unaire) opposé	`-a`
+	addition	`a + b`
-	soustraction	`a - b`
*	multiplication	`a * b`
/	division décimale	`a / b`
div	division entière	`DivEnt(a,b)`
mod	modulo (reste de la division entière)	`Modulo(a,b)`
^	élévation à la puissance	`a ^ b`

Division euclidienne, cas des négatifs

Il n'y a pas unicité du quotient et du reste lorsque le dividende ou le diviseur sont négatifs. Si a et b sont deux entiers relatifs dont l'un au moins est négatif, il y a plusieurs couples (q, r) tels que $a = b \times q + r$ avec $|r| < |b|$. Par exemple, si $a = -17$ et $b = 5$ alors $(q = -3, r = -2)$ ou $(q = -4, r = 3)$ sont deux solutions possibles. Habituellement, q et r sont choisis comme le quotient et le reste de la division entière de $|a|$ et $|b|$ affectés du signe approprié (celui permettant de vérifier $a = b \times q + r$ avec $|r| < |b|$). Dans l'exemple ci-avant, la solution retenue serait $(q = -3, r = -2)$. Par contre, la norme impose que la valeur de $(a \text{ div } b) * b + a \bmod b$ soit égale à la valeur de a.

Pas d'overflow sur les entiers

Si le résultat d'une opération dépasse la capacité d'un entier, le résultat est un entier négatif ou inférieur : il n'y a pas d'erreur de **dépassement de capacité**.

Division par zéro

Tout emploi de la division devra être accompagné d'une réflexion sur la valeur du dénominateur, une division par 0 entraînant toujours l'arrêt d'un algorithme.

12.2 Opérateurs de comparaison

Opérateurs de comparaison

Dits aussi **opérateurs relationnels** ou **comparateurs**, ils agissent généralement sur des

variables numériques ou des chaînes et donnent un résultat booléen. Pour les caractères et chaînes, c'est l'ordre alphabétique qui détermine le résultat.

((alg)) **Opérateurs de comparaison**

Opérateur Mathématique	Signification	Équivalent Algorithmique
$<$	(strictement) inférieur	`a < b`
$\leq$	inférieur ou égal	`a <= b`
$>$	(strictement) supérieur	`a > b`
$\geq$	supérieur ou égal	`a >= b`
$=$	égalité	`a = b`
$\neq$	différent de (ou inégalité)	`a <> b`

12.3 Opérateurs logiques

Opérateurs logiques

Dits aussi **connecteurs logiques** ou **opérateurs booléens**, ils agissent sur des expressions booléennes (variables ou expressions à valeurs booléennes) pour donner un résultat du même type. Ils peuvent être enchaînés.

((alg)) **Opérateurs logiques**

Opérateur Mathématique	Signification	Équivalent Algorithmique
$\neg$	négation (unaire)	`Non a)`
$\wedge$	conjonction logique	`a Et b)`
$\vee$	disjonction logique (ou inclusif)	`a Ou b)`

Opérateur Ou-exclusif

Il n'y a pas d'opérateur OU-exclusif (`xor`) logique.

12.4 Opérateur Si-expression

((alg)) **Fonction Si-expression**

```
Sii(exprBool, exprAlors, exprSinon)
```

Explication

Évalue l'expression logique `exprBool` et si elle est vérifiée, effectue l'expression `exprAlors`, sinon l'expression `exprSinon`. Les `exprAlors` et `exprSinon` doivent être du même type.

Remarque
Cette syntaxe très raccourcie doit être réservée à de petits tests.

12.5 Priorité des opérateurs

alg : Priorité des opérateurs
Les opérateurs de même priorité sont regroupés sur une même ligne.

Priorité	Opérateur	Signification
La plus élevée	- (unaire)	Négation algébrique
	^	Puissance
	* / div mod	Multiplication, division, division entière et modulo
	+ -	Addition et soustraction
	< <= > >=	Opérateurs de comparaison
	= <>	Opérateurs d'égalité
	Non	Négation logique
	Et	Et logique
La plus basse	Ou	Ou logique

Cas de combinaisons de Et et de Ou
Mettez des parenthèses :

```
(cond1 Et cond2) Ou cond3
```

est différent de

```
cond1 Et (cond2 Ou cond3)
```

En l'absence de parenthèses, le `Et` est prioritaire sur le `Ou`.

13 Fonctions prédéfinies

13.1 Fonctions mathématiques

Fonctions mathématiques
Elles agissent sur des paramètres à valeurs réelles et donnent un résultat réel.

((alg)) **Fonctions entières**

Fonctions Mathématiques	Signification	Équivalent Algorithmique
$\lvert x \rvert$	Valeur Absolue	`Abs(x)`
$\lfloor x \rfloor$	Partie Entière	`Ent(x)`
	Arrondi à l'entier le plus proche	`Arrondi(x)`

((alg)) **Fonctions mathématiques**

Fonctions Mathématiques	Signification	Équivalent Algorithmique
e^x	Exponentielle	`Exp(x)`
$\ln(x)$	Logarithme Népérien (naturel)	`Ln(x)`
$\sqrt{x}$	Racine carrée	`RacineCarrée(x)`
x^2	Carré	`Carré(x)`
$\sin(x)$	Sinus	`Sin(x)`
$\cos(x)$	Cosinus	`Cos(x)`

Racine carrée
Attention de ne l'utiliser qu'avec un radicant positif.

13.2 Fonctions caractère

((alg)) **Fonctions caractère**

```
CaractèreAscii(n) # caractère dont le code ASCII est l'entier n
CodeAscii(c) # code ASCII du caractère c
```

13.3 Opérations de chaînes

Longueur d'une chaîne
Nombre de caractères dans la chaîne.

Concaténation de deux chaînes
Consiste à prendre ces deux chaînes et à les coller bout-à-bout.

Sous-chaîne d'une chaîne

Suite consécutive de caractères de la chaîne : c'est une partie (un morceau) de cette chaîne.

Comparer deux chaînes

C'est déterminer laquelle des deux précède l'autre pour l'ordre alphabétique des dictionnaires (encore appelé **ordre lexicographique**) où la chaîne vide `""` est avant toutes les autres et où il faut tenir compte des lettres minuscules et majuscules ainsi que des caractères spéciaux. La comparaison de deux chaînes s'effectue **caractère par caractère de gauche à droite** jusqu'à rencontrer la fin d'une des deux chaînes ou une différence. La chaîne de caractères qui **précède** l'autre est celle qui, la première, a un caractère qui **précède** le caractère correspondant de l'autre chaîne. En cas d'égalité permanente, la chaîne la plus courte **précède** la chaîne la plus longue.

Comparaison de chaînes

Les opérateurs usuels `=`, `<>`, `<`, `>`, `<=` et `>=` servent à comparer deux chaînes.

Opérations de chaînes

```
chn1 & chn2 #concaténation de chaînes
```

Quelques fonctions

```
LgChaîne(chn) #longueur d'une chaîne
SousChaîne(chn,p,n) #sous-chaine à partir de p de longueur n
PositionChaîne(cible,chn) #position de cible dans une chaîne; 0 en cas d'échec
```

Remarque

La « concaténation » de chaînes n'est pas une opération commutative car la chaîne `x & y` n'est pas identique à la chaîne `y & x`.

Attention

Les positions dans les chaînes commencent à 1.