I. Codage, Structures de Contrôle et fonctions

I.1 Algorithme : définition

Définition : *Algorithme*

Un algorithme est une description précise d'une suite d'opérations permettant d'obtenir, en un temps fini, la solution d'un problème.

Algorithme et spécification

Un algorithme ne peut être valide que si il est bien spécifié :

- les données sont clairement décrites : types et valeurs possibles pour chaque donnée de l'algorithme ;

- les résultats le sont aussi : types et valeurs possibles pour les résultats attendus.

Il faut donc que chaque algorithme commence par

- une ligne indiquant ce qu'il doit calculer

- Preconditions : la liste des préconditions sur chaque variable (type, domaine de valeurs, éventuellement liens entre données)

- Résultats : liste des résultats attendus, ainsi que le domaine des valeurs de chacun.

Exemple


```
// Algorithme Somme des carrés
// Precondition :
// - n : int, n > 0
// Résultat :
// - S : int, S >0, la somme des carrés des entiers
//                  inférieurs ou égaux à n
```

I.2 Notion de type

Définition : *Type de données*

Un type de donnée se définit par un *nom*, un *domaine de valeurs* et un *ensemble d'opérations* définies sur un sous-ensemble (qui peut être l'ensemble entier) des valeurs du domaine.

En précisant le type d'une donnée, on précise en fait :

- l'ensemble des valeurs que peut prendre cette donnée ;
- l'ensemble des opérations autorisées (sur les types de base, le plus souvent des opérateurs arithmétiques, logiques et de comparaison).

I.2.1 Types scalaires

Définition : *Type scalaire*

On appelle scalaire un type dont les valeurs sont atomiques et ordonnées.

- *Float* : ce type représente des données qui sont des nombres fractionnaires
- *Bool* : ce type représente des données qui sont des valeurs logiques

I.2.2 Types ordinaux

Définition : *Type ordinal*

On appelle *ordinal* un type scalaire dont chaque valeur (sauf la première) a un prédécesseur unique, et chaque valeur (sauf la dernière) a un successeur unique.

Seul le type *Float* n'est pas un type ordinal

En outre nous considèrerons que nous disposons du type de base suivant, même si ce n'est pas un type scalaire :

- *Text* : ce type représente les données de type texte

I.3 Variable, expression et affectation

Une *variable* sert à stocker de l'information.

Pour employer une image, une variable est une boîte qui sera repérée par une étiquette et l'on peut changer autant de fois que l'on veut le contenu de la boîte.

La valeur de la variable peut changer au cours de l'exécution de l'algorithme, contrairement aux littéraux des équations.

Chaque variable est un triplé *(nom, type, valeur)* où la valeur n'est pas fixe et peut évoluer au cours de l'algorithme.

Une variable doit toujours être *déclarée* afin de définir correctement ce triplé *(nom, type, valeur)* :

```
var nom : type = valeur
```

Expression

Une expression permet de désigner le calcul d'une certaine valeur à partir d'éléments pouvant être évalués et reliés par des opérateurs.

Définition : *Expression*

Une expression de type T est

- une ***valeur*** de type `T` ;
- le ***nom*** d'une variable de type `T` ;
- une ***combinaison*** faite à partir de valeurs, de noms de variables, d'expressions dont le résultat est de type T, et d'opérateurs.

Affectation

Définition : *Affectation*

L'affectation consiste à attribuer une valeur à une variable.

L'opérateur d'affectation se note =

Affectation

Dans une instruction d'affectation, il y a donc deux parties, la *partie gauche* constituée de l'identifiant de la variable et la *partie droite* représentant la valeur à affecter à la variable :

1. Une instruction d'affectation ne modifie que ce qui est situé à gauche de l'opérateur d'affectation `=`
2. Une instruction d'affectation est constituée à gauche d'une variable, à droite d'une expression, les deux parties étant reliées par l'opérateur d'affectation `=`
3. Les deux parties, la variable et l'expression doivent absolument être du même type
4. La variable doit avoir été déclarée
5. syntaxe générale d'une affectation est donc : `var=expression`

I.4 Structures de contrôle

I.4.1 Alternatives

Les *alternatives* ou *instructions de test* permettent de faire des choix en fonction des valeurs de données fournies au programme.

Ce choix dépend d'une condition évaluée comme une expression de type `bool` et aura donc une valeur `True` ou `False`. L'alternative s'écrit alors :

Pseudo-code

```
if condition then
    action 1
else
    action 2
endif
```

On peut étendre l'instruction précédente avec l'instruction `sinon si`. Cela nous donne par exemple pour une décisions entre trois choix :

```
if condition1 then
  action 1
elif condition2 then
  action 2
else
  action 3
endif
```

On peut étendre l'instruction précédente avec l'instruction `sinon si`. Cela nous donne par exemple pour une décisions entre trois choix :

```
if condition1 then
  action 1
elif condition2 then
  action 2
else
  action 3
endif
```

Les tests peuvent également s'emboîter pour exprimer des conditions de décisions elle-même conditionnelles :

```
if condition1 then
  if condition1.1 then
    action 1.1
  else
    action 1.2
  endif
elif condition2 then
  action 2
else
  action 3
endif
```

Les expressions booléennes

Pour construire des expressions permettant d'exprimer les conditions d'une alternative, ou d'une itération, on utilise des opérateurs booléens :

- et logique `&&` : `a&&b` est évaluée à vrai uniquement si les deux expressions `a` et `b` sont vraies

- ou logique `||` : `a||b` est évaluée à vrai si au moins l'une des deux expressions `a` et `b` est vraie

- non logique `!` : inverse la valeur booléenne

- l'égalité `a==b` : vrai si les deux valeurs a et b sont égales

- la différence `a!=b` : vrai les deux valeurs a et b sont différentes

- les comparaison $<$, $>$, $\leqslant$, $\geqslant$

I.4.2 Répétition ou itérations

Lorsqu'on écrit un algorithme, on a souvent besoin de répéter la même action plusieurs fois, avec éventuellement des paramètres de valeurs différentes. Pour gérer ces cas, on fait appel à des instructions permettant d'itérer plusieurs fois la même séquence d'instructions.

Il existe deux schémas de répétition (on dira le plus souvent boucles) :

- l'itération à nombre d'opérations déterminé ou *itération inconditionnelle* ;
- l'*itération à condition d'arrêt*, c'est à dire l'itération dont l'arrêt dépend de l'évaluation d'une expression au cours de l'exécution de l'algorithme.

I.4.2.1 Itération inconditionnelle

Lorsque les deux conditions suivantes sont réunies :

- il existe une suite d'opérations similaires (modulo certains paramètres) qui se répètent,
- le nombre d'itérations est connu « à l'avance », c'est à dire déterminé avant le début de l'itération,

il faut utiliser un schéma itératif à *nombre d'itérations borné*, autrement appelé « *boucle pour* ».

Définition : « *boucle pour* »

Une boucle *pour* est une itération à *nombre d'itérations borné*. Ce nombre d'itérations doit être connu avant le début de la boucle. Le schéma d'utilisation de la boucle pour est le suivant :

```
for var in Ensemble do
  liste d'instructions
endfor
```

`in` indique dans quel ensemble vont être prises les valeurs qui vont être affectées successivement à `var`. *Ensemble* doit donc être un ensemble ordonné de valeurs.

Lorsque l'on veut itérer sur un ensemble d'entier, par exemple les indices d'un tableau, on utilisera la notation suivante qui permet de définir des intervalles de valeurs d'entier :

```
for i in 0…n do // 0…n définit l'intervalle [0,n]
  something
endfor

for i in 0..<n do // 0..<n définit l'intervalle [0,n-1]
  something
endfor
```

Lorsque l'on veut itérer sur un ensemble d'entier, par exemple les indices d'un tableau, on utilisera la notation suivante qui permet de définir des intervalles de valeurs d'entier :

```
for i in 0…n do // 0…n définit l'intervalle [0,n]
  something
endfor

for i in 0..<n do // 0..<n définit l'intervalle [0,n-1]
  something
endfor
```

en swift :

```
for i in 0…n{
  something
}
```

```
for i in 0..<n {
  something
}
```

I.4.2.2 Itération à condition d'arrêt

Lorsque les trois conditions suivantes sont réunies :

- il existe une suite d'opérations similaires (modulo certains paramètres) qui se répètent,
- le nombre d'itérations n'est pas connu « à l'avance »,
- l'arrêt de l'itération s'exprime sous forme d'expression booléenne (i.e. sous forme de test),

il faut utiliser un schéma itératif à condition d'arrêt, autrement appelé « *boucle tant que* ».

Définition : « *boucle tant que* »

Une boucle *tant que* est une itération à condition d'arrêt. Cette instruction a une condition de poursuite sous forme d'expression booléenne et une liste d'instructions qui est répétée tant que la valeur de la condition de poursuite est vraie : la liste d'instructions est répétée autant de fois que la condition de poursuite a la valeur vraie. Le schéma d'utilisation de la boucle *tant que* est le suivant :

```
while condition do
  liste d'instructions
endw
```

Il existe une variante de la boucle tant que, la boucle *répéter jusqu'à*. C'est donc aussi une boucle à condition d'arrêt. La principale différence est que la condition d'arrêt est testée à la fin de la boucle. Il y aura donc toujours au moins une itération exécutée !

Définition : *« boucle répéter jusqu'à »*

Une boucle *répéter jusqu'à* est une itération à condition d'arrêt. Cette instruction a une condition d'arrêt dont la valeur est de type booléen et une liste d'instructions qui est répétée si la valeur de la condition d'arrêt est Fausse.

Le schéma d'utilisation est le suivant :

```
répéter
  liste d'instructions
jusqu'à condition d'arrêt
```

I.4.2.3 Boucles imbriquées

Les instructions de boucle contiennent une liste d'instructions. Une boucle étant une instruction, une instruction de boucle peut contenir une boucle. On parle alors de *boucles imbriquées*.

Dans le cas de boucles à itérations bornées, les compteurs de boucles **doivent** être différents.

```
for i in 0..<n do
  for j in 0..<p do
    liste d'instruction
  endfor
endfor
```

I.5 Langage support : Swift

Swift est un langage compilé, développé en libre par Apple. Disponible nativement sur Apple (et Ipad), mais aussi sur Linux

La syntaxe est moderne et le langage permet d'écrire rapidement des programmes rigoureux, entre autres parce le *typage* est fort et les *déclarations* sont *dynamiques*.

Swift

```
if condition {
    action 1
}
else{
    action 2
}
```

Pseudo-code

```
if condition then
    action 1
else
    action 2
endif
```

Alternatives et itérations

Swift

```
for i in 0…n{
  something
}
for i in 0..<n {
  something
```

Swift

```
while !stop {
  something
}
// condition d'arrêt : stop
```

Swift

```
repeat {
  something
}
while !stop
```

```
}
```

Pseudo-code

```
for i in 0…n do
    something
end-for
for i in 0..<n do
```

Pseudo-code

```
while !stop do
    something
endw
// condition d'arrêt : stop
// condition d'arrêt : stop
```

Pseudo-code

```
repeat
    something
until !stop
```