# EZ Language

# I. Instructions conditionnelles

## A. If

La première forme d'exécution conditionnelle en EZ Language respecte la syntaxe suivante :

**If** *condition* **then** *instruction(s)* **end if**

Le bloc d'instructions est exécuté uniquement si le bloc condition (évaluation booléenne) est vrai.

Par exemple, l'instruction conditionnelle suivante permet l'affichage d'une variable entière :

| EZ | C++ |
|---|---|
| variable Var is integer<br>Var = 5<br>If Var > 3 then<br>  Print Var<br>end if | Int var = 5;<br>if(var > 3){<br>  Std::cout << var << std::endl;<br>} |

Il est également possible de préciser le comportement attendu du programme lorsque la condition n'est pas vérifiée via l'utilisation du mot-clé **else** :

**If** *condition* **then** *instruction(s)* **else** *instruction(s)* **end if**

| EZ | C++ |
|---|---|
| variable var is integer<br>var = 5<br>If var > 3 then<br>  print "variable supérieur à 3"<br>else<br>  print "variable inférieur à 3"<br>end if | Int var = 5;<br>if(var > 3){<br>  Std::cout << "variable supérieur à 3" << std::endl;<br>}else{<br>  Std::cout << "variable inférieur à 3" << std::endl;<br>} |

Enfin on peut concaténer plusieurs conditions afin de tester plusieurs cas possibles :

| EZ | C++ |
|---|---|
| variable var is integer<br>var = 5<br>If var > 3 then<br>  print “variable supérieur à 3”<br>else<br>  If var< 0 then<br>  print “variable négative”<br>  else<br>  print “variable positive inférieur à 3”<br>  end if<br>end if | Int var = 5;<br>if(var > 3){<br>  Std::cout << “variable supérieur à 3” << std::endl;<br>}else if(var < 0){<br>  Std::cout << “variable négative” << std::endl;<br>}else{<br>  Std::cout << “variable positive inférieur à 3” << std::endl;<br>} |

# B.when

L’instruction **when**, permet de tester la valeur d’une et une seule variable et d’exécuter un bloc d’instructions selon celle-ci. Il est toutefois nécessaire de rappeler que le mot-clé **end case** est obligatoire après chaque bloc d’instruction, exception faite du case **default** qui doit toujours être le dernier cas possible et être suivis du mot-clé **end when**. Bien que pouvant toujours être représentée via plusieurs instructions **if**, le **when** permet d’obtenir un code plus clair et plus léger. Il respecte la syntaxe suivante :

**when** expression **is**
  **case** constant1
    bloc1
  **end case**
  **case** constant2
    bloc2
  **end case**
  **default**
    blocdefault
  **end case**
**end when**

Voici un petit exemple d’utilisation de l’instruction when :

| EZ | C++ |
|---|---|
| variable x, y are<br>integer<br>x = 0<br><br>when  y is<br>  case 1<br>  x = x +1<br>  end case<br>  case 2<br>  x = x +2<br>  end case<br>  default<br>  x = x + 3<br>  end case<br>end when | Int x = 0, y;<br><br>switch(y){<br>  case 1 : x = x +1;<br>  break;<br>  case 2 : x = x +2;<br>  break;<br>  case default : x = x + 3;<br>  break;<br>} |

# II. Instructions itératives

Les instructions itératives permettent de répéter une ou plusieurs instructions selon plusieurs critères. L'EZ Language offre trois instructions itératives : tant que(while), jusqu'à(repeat … until) et pour(for).

## A.while

L'instruction  **while**  répète simplement le ou les instruction(s) tant que la condition est vraie. De fait, elle effectue dans un premier temps le test de la condition puis effectue le bloc d'instructions si le test est un succès.
Elle a pour syntaxe :

**while** condition **do**  instruction(s) **end while**

| EZ | C++ |
|---|---|
| variable x is integer<br>x = 0<br><br>while x < 10 do<br>  x = x +1<br>end while | Int x = 0;<br><br>while(x < 10){<br>  x = x +1;<br>} |

Ici la variable x qui est initialisée à 0 avant la boucle, est incrémentée tant qu'elle est inférieure à 10.

# B.repeat .. until

L'instruction **repeat...until** est similaire à l'instruction while dans le sens où elle répète le bloc d'instruction(s) tant que la condition n'est pas vérifiée. La syntaxe est la suivant :

**repeat** instruction(s) **until** condition

Il est **important** de noter ici, que le programme effectue un passage dans la boucle avant de vérifier la condition.

| EZ | C++ |
|---|---|
| variable x is integer<br>x = 0<br><br>repeat<br>  x = x +1<br>until x < 10 | int x = 0;<br><br>do{<br>  x = x +1<br>}while(x < 10) |

# C.for

L'instruction **for** permet de répéter un certain nombre de fois un bloc d'instructions. Les variables a et b représentent les bornes minimale et maximale de l'intervalle de valeurs que prend x au cours de l'exécution de la boucle. Notons également le mot-clé **step**, permettant de définir les variations de la variable x entre deux itérations. Ce paramètre est facultatif et par défaut, la valeur de x est incrémentée de 1 par itération. La syntaxe est donc la suivante :

```
variable a, b are integer
variable var, k  are integer
for var in a..b step  k do
        instruction(s)
end for
```

qui est **équivalent** à la syntaxe suivante :

```
variable a, b are integer
for var is integer in a..b step  k is integer do
        instruction(s)
end for
```

Voici un exemple d'utilisation :

| EZ | C++ |
|---|---|
| variable x, y, k are integer<br>x = k = 0<br>y = 10<br><br>for a is integer in x..y do<br>  k = k + 1<br>end for | Int x = 0, y = 10, k = 0;<br><br>for(int a = x; a = y; a++){<br>  k = k + 1<br>} |

Il est également possible dans le cas d'un parcours de conteneur tel que le vecteur d'être simplifié :

| EZ | C++ |
|---|---|
| foreach élément e in nom_vecteur do<br>  ...<br>end each | for(auto e : nom_vecteur){<br>  ….<br>} |

Ici, on parcourt le vecteur, élément par élément.

# III. Fonctions / Procédures

Une fonction est un regroupement d'instructions dans un bloc identifié qui pourra être exécuté comme une instruction en appelant l'identifiant de la fonction dans le programme.

## A. Syntaxe

Ici, nous faisons la distinction entre fonction qui retourne un résultat et procédure qui n'en retourne pas.

| EZ | C++ |
|---|---|
| **Syntaxe fonction** | |
| **function** Name_Fonction **(**liste arguments**) return** type1<br>type1 variable;<br>instruction(s)<br>return variable;<br>**end function** | **type1** Name_Fonction **(**liste arguments**){**<br>type1 variable;<br>instruction(s)<br>return variable;<br>**}** |
| **Syntaxe procédure** | |
| **procedure** Name_Procedure **(**liste arguments**)**<br>instruction(s)<br>**end procedure** | **void** Name_Procedure **(**liste arguments**){**<br>instruction(s)<br>**}** |

L'utilisateur peut également fournir une valeur par défaut dans les arguments des fonctions et procédures.

| EZ | C++ |
|---|---|
| **Syntaxe fonction** | |
| **function** Name_Fonction **(**arg is type1 = valeur**) return** type2<br>type2 variable1;<br>instruction(s)<br>return variable1;<br>**end function** | **type2** Name_Fonction **(**type1 arg = valeur**){**<br>type2 variable1;<br>instruction(s)<br>return variable1;<br>**}** |
| **Syntaxe procédure** | |
| **procedure** Name_Procedure **(**arg is type1 = valeur**)**<br>instruction(s)<br>**end procedure** | **void** Name_Procedure **(**type1 arg = valeur**){**<br>instruction(s)<br>**}** |

Les appels des fonctions et procédures sont les suivants :

| EZ | C++ |
|---|---|
| **Syntaxe fonction** | |
| variable var is type = Name_Fonction (liste arguments); | type var = Name_Fonction(liste arguments); |
| **Syntaxe procédure** | |
| Name_Procedure (liste arguments); | Name_Procedure(liste arguments); |

# B.Surcharge

La surcharge est le principe de définir une fonction ou procédure plusieurs fois dans le cas où elle prendrait des paramètres différents.

| EZ | C++ |
|---|---|
| **Déclaration** | |
| procedure Name_Procedure (arg is type1)<br>  instruction(s)<br>end procedure<br><br>procedure Name_Procedure (arg is type2)<br>  instruction(s)<br>end procedure | void Name_Procedure(type1 arg){<br>  instruction(s)<br>}<br><br>void Name_Procedure(type2 arg){<br>  instruction(s)<br>} |
| **Appel** | |
| variable A is type1<br>variable B is type2<br>//type1 différent de type2<br><br>Name_Procedure(A)<br>Name_Procedure(B) | type1 A;<br>type2 B;<br>//type1 différent de type2<br><br>Name_Procedure(A)<br>Name_Procedure(B) |

A l'appel de la fonction, la version utilisée sera en fonction de l'argument passé en paramètre par l'utilisateur.

# IV. Flux d'entrée / sortie

Pour utiliser la sortie standard et afficher à l'écran, on utilise l'instruction **print**. Dans le but de conserver une simplicité d'utilisation, on écrira d'abord l'instruction d'affichage puis on écrira à la suite les variables à afficher en utilisant le séparateur "**,**". L'instruction affichera les valeurs des différentes variables sur la sortie correspondante.

```
print "valeur de la variable" , variable
```

On définit une entrée standard qui est le clavier de l'utilisateur. Pour récupérer les saisies de l'utilisateur, on utilise l'instruction **read**. Celle-ci est bloquante, le programme est donc interrompu tant que l'utilisateur n'a pas appuyé sur entrée pour valider la saisie. Le résultat de la saisie est affecté à une variable passée en paramètre.

```
variable saisie is string
read saisie
```