FrLang

Spécification - Guide Utilisateur

RedsTom

2022

Table des matières

* * *

Préambule

Ce document décrit les caractéristiques de la syntaxe de FrLang. Il est soumis à la license GNU Gpl v3, disponible sur le site https://www.gnu.org/licenses/quick-guide-gplv3.fr.html, tout comme l'intégralité du code source de ce même langage.

* * *

1 Structure basique

1.1 Structure d'une ligne

Toutes les instructions de FrLang peuvent être séparées sur plusieurs lignes sous réserve qu'aucun mot-clé ou identifiant ne soit coupé par ce saut de ligne. Elles doivent toutes être séparées par un `;` (habituellement en fin de ligne).

1.2 Structure d'un bloc

Les blocs sont une structure de code contenant des instructions. Ils commencent par `{` et se terminent par `}`. Toutes les instructions d'un block doivent être indentées d'un niveau au dessus du niveau précédent. Ils ont la particularité de créer une portée de variables qui n'est accessible que dans le bloc courant, et dans les éventuels blocs sous-jacent.

* * *

2 Les variables

Une variable est un moyen de stocker une valeur sous un certain nom. Elle est définie par un identifiant unique à sa portée et peut être utilisée dans les instructions suivant sa déclaration.

2.1 Déclaration d'une variable

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

Les variables sont déclarées avec la syntaxe `var nom;`. Il peut être suivi d'un symbole `=` pour définir une valeur à la variable comme suit : `var nom = valeur;`.

```
var a; // Creation de la variable "a" sans valeur
var b = 5; // Creation de la variable "b" avec la valeur 5
```

Plusieurs déclarations de variables peuvent se faire sur une même instruction, ainsi, chaque déclaration doit être séparée par une virgule.

```
var a, b; // Creation de la variable "a" et "b" sans valeur
var c = 3, d; // Creation de la variable "c" avec la valeur 3, et "d" sans valeur
```

2.2 Assignation d'une variable à une nouvelle valeur

2.2.1 Assignation simple

Une variable peut prendre une nouvelle valeur à l'aide de la syntaxe `identifier = valeur;`. L'instruction retourne alors la nouvelle valeur de la variable, permettant de chainer les assignations.

```
var a = 5; // Creation de la variable "a" avec la valeur 5
a = 6; // Assignation de la valeur 6 a la variable "a"

var b; // Creation de la variable b sans valeur
a = b = 2; // Assignation de la valeur 2 a la variable "a" et "b"
```

2.2.2 Assignation complexe

D'autres opérateurs que le `=` permettent d'agir sur la valeur d'une variable :

```
a += 6; // Ajoute 6 a la variable "a"
a -= 3; // Soustrait 3 a la variable "a"
a *= 2; // Multiplie "a" par 2
a /= 2; // Divise "a" par 2
a %= 2; // Modulo "a" par 2
```

2.3 Création de constantes

Une constante est une variable qui, une fois assignée, ne peut plus être modifiée. L'assignation de constantes se fait comme suit : `const nom = valeur;`.
Une constante ne peut pas être définie sans valeur, et ne peut pas prendre la valeur `nul`.

```
const a = 5; // Creation de la constante "a" avec la valeur 5
```

Le code suivant est donc incorrect :

```
const a = 5; // Creation de la constante "a" avec la valeur 5
a = 6; // Erreur : la constante "a" ne peut pas etre modifiee
```

* * *

3 Les types de données

En FrLang, les données sont typées fortement et statiquement, c'est-à-dire que chaque variable est associée à un type de valeur, et qu'une fois que cette variable a pris un type, celle-ci ne peut plus en changer.
Les types de valeurs disponibles en FrLang sont :

Type	Description	Exemple
`entier`	Représente un nombre entier qu'il soit positif ou négatif	`5`
`decimal`	Représente un nombre décimal qu'il soit positif ou négatif	`3.2`
`booleen`	Représente un booléen (vrai ou faux)	`vrai`
`texte`	Représente du texte	`"bonjour"`
`caractere`	Représente un caractère	`'a'`
`?`	Représente un type inconnu	`?`

3.1 Types absolus

Les types `entier` et `decimal` peuvent perdre leur capacité à devenir négatif s'ils sont suivis du mot clé `absolu`, créant ainsi les types `entier absolu` et `decimal absolu`.

3.2 Préciser un type de données à une variable

Il est possible, au moment de la création d'une variable, de préciser son type. Cela permet de déjà connaître, et avant même son initialisation, le type de la variable. En cas d'initialisation au moment de la déclaration de la variable, le type de la variable est déduit à partir de sa valeur. Il n'est donc jamais utile de préciser le type d'une constante.
La syntaxe pour préciser le type d'une variable est : `var nom: type`.

```
var a: entier; // Creation d'une variable "a" de type entier
var b: decimal absolu, c; // Creation d'une variable "b" de type decimal absolu, et d'une variable "c" de type inconnu
```

3.3 Type inconnu et pointeur nul

Le type `?` est un type inconnu, il est utilisé pour désigner un type qui n'est pas encore défini, par exemple le type d'une variable non assignée, et sans type précisé. C'est le seul type de variable qui peut changer après sa déclaration.

```
var a: ?; // Creation d'une variable "a" de type inconnu
a = 5; // On assigne la valeur 5 a la variable "a". Le type de la variable "a" est maintenant entier.
```

Le pointeur nul quant à lui est une valeur qui peut être donnée à une variable de n'importe quel type et qui permet de dire que la variable n'a pas de valeur. C'est la valeur qui est utilisée lorsque la variable n'a pas encore été initialisée. Le pointeur nul est donc la seule instance du type `?`.

* * *

4 La gestion des E/S

Les entrées & sorties sont une partie importante d'un langage de programmation. En effet, c'est un moyen de communiquer entre le programmeur et l'utilisateur. Il est donc important que la syntaxe des entrées et sorties soit minimale, afin de ne pas perdre de temps et de place inutilement. C'est pourquoi, dans le langage Fr-Lang, les entrées et sorties sont définies par une syntaxe simple et concise :

4.1 Sorties

Les sorties sont définies par le symbole `.` suivi de l'expression à afficher.

```
. "Bonjour"; // Affichage du texte "Bonjour"
.a; // Affichage de la valeur de la variable "a"
.5; // Affichage de valeur "5"
.0.4; // Affichage de "0.4"
```

4.2 Entrées

Pour demander une valeur à l'utilisateur, il suffit d'écrire `???` dans le code. Cela demandera alors à l'utilisateur de saisir une valeur. Les variables résultant de cette saisie sont par défaut des variables de type `texte`, mais il est possible d'automatiquement les transformer en variables d'un autre type primitif, en spécifiant explicitement leur type.

```
var a = ???; // Saisie de la valeur de la variable "a" (texte)
var b: entier = ???; // Saisie de la valeur de la variable "b" (entier)
```

* * *

5 Les instructions conditionnelles

```
si condition
    instruction | bloc
[sinon
    instruction | bloc]
```

Les instructions conditionnelles sont une grande partie des langages de programmation impératifs. Elles permettent de définir des conditions dans lesquelles un bloc de code sera exécuté. La syntaxe de ces instructions est la suivante :

```
si condition1 {
    // code a exécuter si la condition1 est vraie
} sinon si condition2 {
    // code a exécuter si la condition1 est fausse, mais si la condition2 est vraie
} sinon {
    // code a exécuter si aucune condition n'est vraie
}
```

* * *

6 Les instructions répétées

Les instructions répétitives, ou plus couramment appelées boucles sont des instructions destinées à être exécutées plusieurs fois.
Il existe quatre types principaux de boucles :

— `tant que` : Permet de répéter une instruction tant que la condition est vraie.
— `faire tant que` : Permet de répéter une instruction une fois ou plus tant que la condition est vraie.
— `pour` : Permet de créer une boucle qui s'exécute d'une certaine quantité de fois.
— `pour chaque` : Permet de parcourir un tableau ou une liste et d'exécuter une instruction à chaque itération.

6.1 Les boucles `tant que` et `faire tant que`

```
tant que condition
    instruction | bloc
```

```
faire
    instruction | bloc
tant que condition;
```

La boucle `tant que` permet d'exécuter une série d'instructions tant que la condition qui lui est donnée est vraie.
Elle vérifie avant chaque itération si la condition est vraie et exécute les instructions contenues dans la boucle si c'est le cas.

```
var a = "";
tant que taille(a) < 10 {
    a += "a";
}
.a;
```

Cette boucle exécute l'instruction `a += "a"` tant que la longueur de tu texte `a` est inférieure à 10, et affiche le contenu de `a` à la fin.
La boucle `faire tant que` permet elle aussi d'exécuter une série d'instructions tant que la condition qui lui est donnée est vraie. Cependant, elle vérifie si la condition est vraie après chaque itération, permettant donc de s'assurer que le contenu de la boucle est exécuté au moins une fois.

```
var a: texte;
faire {
    a = "a";
} tant que faux;
.a;
```

```
var a: texte;
tant que faux {
    a = "a";
}
.a;
```

Dans le premier cas, l'instruction `.a;` à la fin affichera `a`, alors que dans le second cas, cette même instruction affichera `nul`.

6.2 Les boucles pour

Il existe trois types de boucles pour :
— `pour` : Permet de créer une boucle avec un compteur qui s'incrémente d'une certaine quantité un certain nombre de fois.
— `pour var` : Permet de créer une boucle `pour` avec un comportement personnalisé.
— `pour chaque` : Permet de parcourir un tableau ou une liste et d'exécuter une instruction à chaque itération.

6.2.1 La boucle `pour`

```
pour nom = début -> fin [(pas)]
    instruction | bloc
```

Cette boucle permet de créer une variable qui ira d'un nombre à un autre, avec un certain interval. L'interval est optionnel et vaut 1 par défaut. Si le nom de la variable donné existe déjà, la variable existante sera écrasée, et la valeur de la variable après l'itération sera la valeur d'arrivée de la boucle.

```
pour i = 1 -> 10 {
    .i;
}
```

```
pour i = 1 -> 10 (2) {
    .i;
}
```

Le premier exemple affiche les entiers de 1 à 10, et le second exemple affiche les entiers de 1 à 10 de 2 en 2.

6.2.2 La boucle `pour var`

```
pour [initialisation]; [condition]; [
    ↪ etape]
    instruction | bloc
```

La boucle `pour var` permet de créer un variable qui sera modifiée et vérifiée à chaque itération. Cette boucle est séparée en trois parties, séparées par un point-virgule :
— `initialisation` : Création de la variable unique à la boucle.
— `condition` : Condition vérifiée avant chaque itération.
— `etape` : Modification apportée à la variable après chaque itération.

```
pour var i = 0; i <= 5; i += 1 {
    .i;
}
```

Cette boucle est équivalente à la boucle `pour` suivante :

```
pour i = 0 -> 5 {
    .i;
}
```

6.2.3 La boucle `pour chaque`

```
pour chaque [nom] dans [tableau]
    instruction | bloc
```

La boucle `pour chaque` permet de parcourir un tableau ou une liste et d'exécuter une action à chaque itération.

```
const eleves = ["Tom", "Laura", "Bob",
    ↪ "John"];
pour chaque eleve dans eleves {
    .eleve;
}
```

* * *