Interprétation et Compilation

Projet Finale

Guillaume Magniadas

Année 2019 - 2020

Petite documentation du langage

Syntaxe générale :

Mon petit langage à une syntaxe assez similaire à du c et est structuré un peu comme en python. Pas de fonction main obligatoire, on peut écrire des instruction a la suite sans problèmes.

Un programme dans ce langage est une suite d'instruction.

Une instruction se termine par un point virgule.

Il y a une notion de bloque, un bloque commence par { et fini par } et contient une suite d'instruction (comme un mini programme). Chaque bloque possède son propre contexte, une variable déclaré dans un bloque sera oublié a sa sortie. Par-contre en entrant dans un bloque, les variable déjà présente restent connue. Il est possible d'avoir un bloque qui ne contient qu'une seule instruction, dans ce cas là, les accolades no sont pas obligatoires.
(Je vais revenir sur où est comment sont utilisé les bloques)

Variables :

Les variables se déclarent comme en C, un type, un nom et = une valeur, par-contre, il est obligatoire d'assigner une valeur directement avec la déclaration de la variable.
Il y à 3 type disponible :

- int (Entier)
- string (Chaine de caractères) (Entouré de guillemets)
-[int] (Tableau d'entiers) (Entouré de crochet et chaque valeur séparé de virgules)

Exemple :

```
int a = 10;
string b = “Ceci est une string“;
[int] c = [0, 1, 2, 3];
```

Calculs sur entier :

Les entier possède les opérateur de base pour effectuer des calculs basique :

- + la somme
- - la soustraction
- * la multiplication
- / la division

Booleans, test et comparaison :

Dans ce langage, les boolean n'existe pas vraiment mais toutes les comparaisons fonctionnes avec des entier classique. Un entier valant 0 sera faux et tout le reste sera vrai.

Il existe les opérateur de base pour comparer des valeur entre elles :

- == Égale
- > Supérieur
- < Inférieur

- >= Supérieur ou égale
- <= Inférieur ou égale

Bloque de contrôle :

Dans ce langage, il existe deux bloque de contrôle, le bloque if else, et le bloque while.

IF ELSE :

Ce bloque de contrôle contient un test, et deux bloque. Le premier bloque est exécuté si le test est vrai, ou sinon si le test est faux, le deuxième bloque est exécuté.

WHILE :

Ce bloque de contrôle contient un test et un bloque. Le bloque est exécuté tant que le test est vrai.

Fonctions :

Il est possible de déclarer des fonctions dans ce langage.
Une fonction possède un type de retour, un nom, des paramètres (qui peuvent être vide) et un bloque de retour (Qui est une variante des bloques classique, qui a pour différence de devoir se terminer par une instruction return. L'instruction return peut être vide, dans le cas d'une fonction de type void.)
Lorsqu'une fonction est appelé, un nouveau contexte est créer pour cette dernière, qui n'est pas au courant des variables précédemment déclaré au cours de l'execution du programme. Les seul variable présente dans ce dernier sont les fonctions précédemment déclaré, elle même, les variables passé en paramètre et les variable déclaré au cours de l'execution de la fonction.
On en déclare en indiquant son type de retour, le met clef def, le nom de la fonction, ses arguments entre parenthèse (type et nom de variable) séparé par des virgules, et suivi d'un bloque.

Exemple :

```
int def sum(int a, int b){
        return a + b;
}
```

(Note : Ce bloque pourrait s'écrire sans les accolades)

Une fonction une fois déclaré peut être appelé en écrivant son nom et en lui passant ses arguments entre parenthèse et séparé par des virgules.

Exemple :

```
int a = sum(3, 4);
```

Tableau d'entier :

Comme vu dans les variable, il est possible de déclarer des tableaux d'entier.
Un tableau est enfaite une suite de pair.
Il existe quelques fonctionnalité native pour travailler sur ces dernier :

- pair(int a, [int] b) : Cette fonction retourne un nouveau tableau avec comme première valeur a et qui est suivi de b.
- head([int] a) : Cette fonction retourne la première valeur du tableau a.
- tail([int] a) : Cette fonction retourne la suite du tableau a après la première valeur.
- pair_empty([int] a) : Cette fonction retourne 1 si le tableau a est vide, 0 sinon.

Autre fonctionnalité :

Il est aussi possible d'affiche sur le terminale certaines valeur tel que des entier ou des chaines de caractères :

- print_num(int a) : affiche l'entier a sur le terminale.
- print_str(string a) : affiche la chaine de caractère a sur le terminale.
- print_nl() : affiche un saute de ligne sur le terminale.

On peut aussi demander une valeur à un utilisateur :

- read_num() : Récupère un nombre sur l'entrée standard et le renvoie.

Égalité des pointeur de tableau d'entier avec l'opérateur ===.

Et il existe une fonction pour récupérer la taille d'une chaine de caractère :

- str_len(string a) : Renvoie la longueur de la chaine de caractère a.

Commentaires

Je suis assez satisfait du résultat, mais il y quand même quelques détail que j'aurais bien aimé corriger / ajouter :

- Une meilleur optimisation (Sur la liste d'instructions mips généré a la fin après la passe du compiler, j'ai déjà rajouté une petite optimisation sur les addi mais j'aurais aimé en faire plus, ou aussi sur les calcul avec des valeur immédiates, les calculé directement plutôt que tout calculé en mips)

- L'instruction break (J'avais commencé à en écrire une qui fonctionnais seulement dans des bloque d'une seul profondeur, le problème étant qu'a la sortie de plusieurs bloque en même temps, il faut a chaque fois récupérer le bon ancienne fp qui se cache sur la pile donc il aurait fallu trouver un système différent pour se souvenir du fp de sortie de bloque sachant que fp et sp sont modifié a chaque profondeur de bloque (Peut être en sauvegardant chaque profondeur supplémentaire dans une variable sur la pile, mais celé deviendrait un peu lourd))

- Un meilleur traitement des bloques d'une seule instruction, pour le moment quand un bloque d'une seule instruction est créer j'agis comme si c'était un bloque classique mais avec une liste d'une seule instruction mais du coup avec la même création de contexte qu'un bloque classique ce qui peut faire lourd pour une seule instruction. J'avais essayé quelque chose mais je l'ai supprimé car cela ne fonctionnait pas pour les fonction qui possède un nouveau contexte et rendent cet amélioration un peu plus complexe.

- Pouvoir mettre des return ou l'on veut dans une fonction. (Il était possible de faire cela dans une version antérieure de ce langage mais cela rendait trop compliqué la vérification de l'appel d'un return en cas de if else ou de while j'ai donc préféré simplifier le tout en forçant une seule instruction return en fin de programme)

J'ai reproduit vos 10 exemples avec ce langage dans le dossier myExemple. Il y a aussi le fichier test qui regroupe quelques test qui fonctionne sur ce langage, je vous l'ai laissé au cas ou.