Free Basic to C compiler

Jérémy Bardon

24 mars 2015

1 Introduction

Le but de ce projet est de créer un compilateur capable de convertir un code écrit en Free Basic dans le langage C.

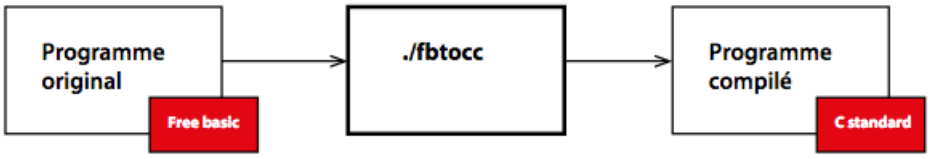

FIGURE 1 – Rôle du compilateur

Le programme en C ainsi généré sera compilable avec *gcc* et effectuera les mêmes opérations que le programme original.

2 Adaptations

Par défaut, un programme écrit en free basic dispose comme en C d'un certain nombre de fonctions que l'on peut utiliser sans inclure des programmes externes. Cependant, ces fonctions « basiques » ne sont pas équivalentes dans les deux langages.

L'exemple le plus simple est la fonction qui permet d'afficher un message : en free basic il s'agit de la fonction *print* qui est incluse par défaut mais en C il est nécessaire d'inclure *stdio.h*.

Une autre différence de taille est la présence – en C – d'une routine principale (main) qui doit être présente dans tout programme écrit en C. Ce n'est pas le cas du free basic qui propose une structure plus libre du programme.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
```

```
    // Converted free basic code
    // comes here

    return EXIT_SUCCESS;
}
```

Listing 1 – Programme C englobant

Ce sont ces différences qui amènent à devoir proposer une structure englobante d'un programme dans laquelle on va ensuite insérer le code free basic converti.

3 Grammaire

L'organisation des fichiers sources du compilateur permet de facilement identifier quelles sont les règles de grammaire ainsi que les caractères reconnus. En effet, le fichier *scanner.mll* regroupe la partie analyse syntaxique tandis que le fichier *parser.mly* s'occupe de l'analyse syntaxique et donc de la grammaire.

Tout les exemples utilisés lors du développement du compilateur sont disponibles dans le répertoire *test* – à la racine du projet – et il contient notamment le fichier *full.fba* qui peut donner un aperçu global de ce que le compilateur est capable de reconnaitre et de traduire.

3.1 Variables

Il existe de nombreux types de variables en free basic mais le compilateur reconnait seulement les types *Integer* et *String* qui sont les plus utilisés.

Hormis la déclaration de variables et de constantes, il est possible de faire des affectation de valeurs simples ou d'expressions plus complexes comme des formules de math.

3.2 Fonctions

Le compilateur gère de manière transparente l'appel à n'importe quelle fonction avec un ou plusieurs arguments qu'il soient donnés en dur ou via une variable.

Cela dit, pour assurer un minimum de compatibilité avec le langage c, la fonction *Print* est automatiquement remplacée par *printf* pour des paramètres passés en dur. Il n'est donc pas possible d'afficher plusieurs variables car cela aurait nécessiter de construire le pattern demandé par la fonction *printf*. L'implémentation de cette fonctionnalité est tout de même possible mais aurait pris plus de temps.

3.3 Commentaires

Afin de se concentrer sur des éléments considérés plus importants – conditions, boucles, ... –, seul les commentaires mono-lignes sont reconnus. Cela permet tout de même d'insérer des commentaires sans donner la souplesse des commentaires multi-lignes.

3.4 Conditions

L'implémentation des structures conditionnelles est fonctionnelle dans une forme simple et n'intègre pas l'utilisation de plusieurs clauses *Else if* sous une même condition.

Il reste tout à fait possible d'imbriquer une condition dans une clause *else* d'un première condition, même si cela rend la syntaxe plus complexe ce type d'opérations reste possible.

```
// Forme simple (implementee)
If inPass = password Then
    Print "Login success"
Else
    If inPass = "admin" Then
        Print "Login success (admin mode)"
    End If
End If

// Raccourci de syntaxe (non implemente)
If inPass = password Then
    Print "Login success"
Else If inPass = "admin" Then
    Print "Login success (admin mode)"
End If
```

Listing 2 – Raccourci de syntaxe pour les conditions

[1]

Le compilateur supporte les comparateurs les plus utilisés :

=, >, <, >= et <=

Il supporte la comparaison entre des variables et/ou les types *Integer* et *String* de manière avancée puisque dans la cas de comparaison entre deux chaines de caractère, celui-ci utilise la fonction générique du c *strcmp* de manière automatique.

Cela dit, il ne permet pas de comparer le résultat d'une fonction ou encore une expression avec une autre et il ne déduit pas que l'absence de comparateur est similaire à *== 0*.

1. Commentaires en c, LaTeX refuse l'apostrophe

3.5 Boucles

L'ensemble des boucles – pour et tant que – sont implémentés et le compilateur donne un peu plus de souplesse lors de leur utilisation.

En effet, il est possible en free basic de définir une boucle infinie – qui n'a pas de condition – et l'équivalent en c est de donner la condition *true*. Ce type de boucle est tout à fait reconnu par le compilateur qui converti automatique une boucle infinie en une boucle avec la condition *true* en c.

Dans certaines version du c, il est impossible de définir une variable directement dans la clause *for*. C'est pourquoi le compilateur va se charger de créer de manière automatique la variable utilisée dans la construction de la boucle si elle n'a pas été déclarée auparavant.

3.6 Erreurs

Le compilateur est capable de founir le numéro de ligne et de caractère auquel il s'est arrêté si la phase d'analyse lexicale à échoué. De plus, l'utilisation de variables non déclarés renvoie une erreur qui spécifie quelle est cette variable qui a été utilisé alors qu'elle n'existe pas.

4 Reconnaissance des erreurs

Dans sa première version – qui est l'actuelle version –, le compilateur ne gère que un type d'erreur de logique – utilisation d'une variable non déclarée – dans le code en free basic mais seulement les défauts d'ordre syntaxiques.

On fait donc l'hypothèse que le code en entrée est compilable et exécutable sans erreurs ce qui permet de se concentrer d'avantage sur la reconnaissance – et la conversion – de nombreux éléments du langage.

L'architecture interne du compilateur – décrite plus bas – a été pensée pour permettre l'intégration de cette fonctionnalité. En effet, tout les éléments sont stockés sous forme d'un arbre syntaxique et les variables sont sauvegardées dans un registre.

5 Fonctionnement interne

Au delà de l'analyse lexicale plutôt générique, la partie intéressante se situe au niveau de l'analyse syntaxique. En effet, lors de cette phase on vérifie pas le sens que avoir une ligne de code mais seulement si elle est bien organisé.

Le compilateur ne se contente pas transformer chaque ligne en une autre sous forme des chaine de caractères mais il construit un arbre syntaxique qui permet

d'identifier clairement comment s'enchainent les instructions et quel sont leur sens (définition de variable, condition, boucle..).

En parallèle de cet arbre, il tient à jour un registre des variables – sous forme de table de hashage – qui sont déclarés dans le programme afin de pouvoir afficher différemment un entier et une chaine de caractères. Ce registre retient donc le nom et le type de chaque variable ce qui rend facilement possible la traduction d'une comparaison entres des entiers ou entre des chaines de caractères.

Une fois l'arbre construit et le registre rempli, un algorithme parcours l'arbre et s'occupe de transformer chaque élément sous forme de chaîne de caractère correspondant à un code C valide.

6 Exemple concret

6.1 Présentation et code free basic

Pour illustrer de manière plus explicite les fonctionnement du compilateur, prenons un exemple de code en free basic que l'on veux convertir en C.

```
Dim chaine As String
chaine = "password"

For iter = 1 To 5
    If chaine = "password" Then
        Print "Hello master"
    Else
        Print "Try again"
    End If
Next
```

Listing 3 – Exemple en free basic

Cet exemple ne montre pas toutes les fonctionnalités que le compilateur possède ni l'intégralité des syntaxes reconnus mais essaye d'en montrer une partie.

Ce programme demande un mot de passe et donne 5 essais pour y réussir. On a donc une variable – qui contient l'entrée utilisateur – puis une boucle – dont la variable n'est pas déclarée – dans laquelle un message différent est affiché selon une condition.

6.2 Structures générée par le compilateur

Arbre syntaxique

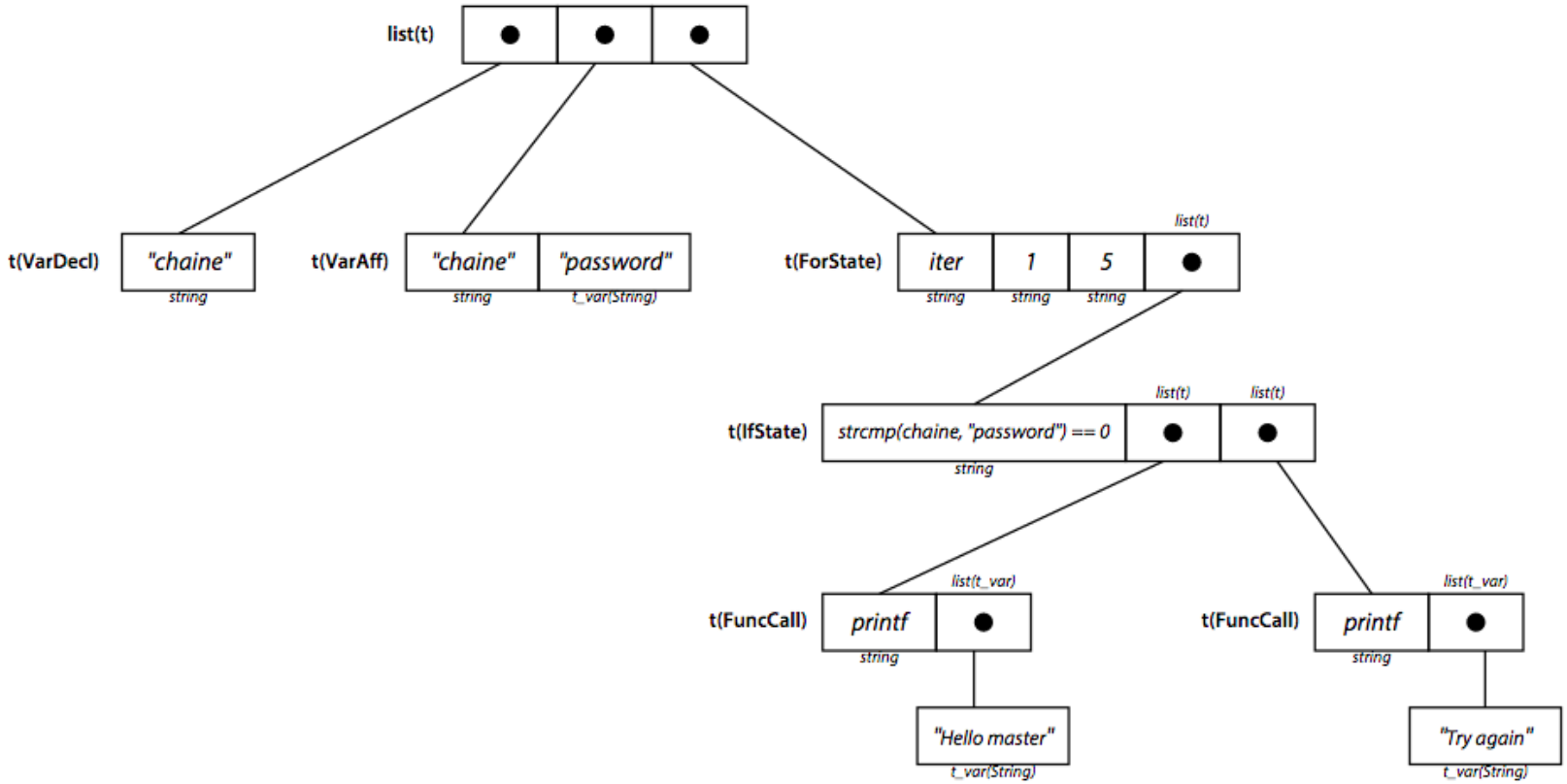

FIGURE 2 – Arbre syntaxique de l'exemple

Registre des variables

La variable utilisée dans la boucle n'apparait pas dans le registre car elle est ajoutée au moment où l'on traduit l'instruction en C.

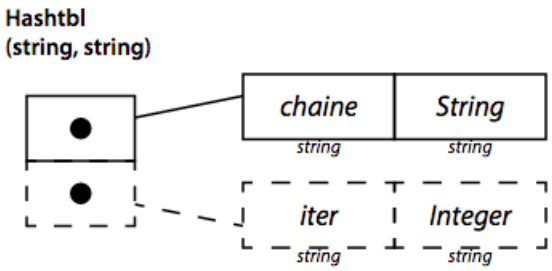

FIGURE 3 – Registre des variables

A partir du moment où cette étape est passée, la variable *iter* est dans le registre pour permettre à la suite du programme de la reconnaitre.

6.3 Code généré

Le code C généré par le compilateur se trouve dans le fichier *output.c* et il est possible de le compiler – et de l'exécuter – sans problème avec un compilateur c.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){

    char* chaine;
    chaine = "password";

    int iter;
    for (iter = 1; iter < 5; iter++) {
        if(strcmp(chaine, "password") == 0){
            printf("Hello master");
        } else {
            printf("Try again");
        }
    }

    return EXIT_SUCCESS;
}
```

Listing 4 – Code c de l'exemple

On peut remarquer que le tout est indenté correctement mais il s'agit d'un traitement à la main pour que se soit plus lisible dans le rapport.

En effet, le compilateur ne gère pas du tout l'indentation du code car cela est

aspect moins important que la traduction du code. De plus, lors de l'utilisation d'outils de conversion – comme se compilateur –, l'utilisateur ne s'intéresse que très rarement au code généré.