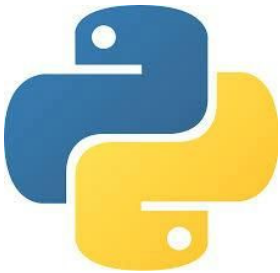
POLYTECH GRENOBLE

RICM4

Rapport de Projet de 4eme année

SPEEDING SIMPLIFIED SCRIPT LANGUAGE (SSSL)



Go

Florian POPEK
Adèle BERTRAND-DALECHAMPS
Wei WEI

Tuteurs : Olivier RICHARD et Didier DONSEZ

Sommaire

1. Présentation

CONTEXTE

Ce projet est la continuité d'un projet de l'année 2015, « Python sur ESP8266 » développé par deux étudiants Polytech : Luc LIBRALESSO et Olivier SOLDANO. L'objectif était d'intégrer le support des cartes STM32F4-Discovery à l'outil Shedskin, qui permet la traduction du langage Python vers le langage C++.

NOTRE PROJET

L'objectif de ce projet est de concevoir un langage script simple traductible en un langage compilable tel que C++ (ici le langage Go). Ce langage sera uniquement voué à être traduit, et ne sera donc jamais exécuté en soit.

TECHNOLOGIES UTILISEES

L'ensemble de ce projet est réalisé en Python (Python 3), à l'aide d'un générateur de parser. Pour ce dernier, nous avons choisi d'utiliser GRAKO, un module python permettant de générer un parser d'expressions selon une grammaire EBNF donnée.

AXES MAJEURS

Ce projet peut être divisé en plusieurs axes majeurs:

- Construction de la grammaire (fichier **SSSL.ebnf**)
- Analyse syntaxique de notre programme avec GRAKO (fichier **test_SSSL.py**)
- Nettoyage de l'arbre syntaxique généré par GRAKO (fichier **AST_cleaner.py**)
- Parsing de l'AST généré par GRAKO (contenu du dossier **AST_parser/**)
- Traduction en langage Go (fichier **Go.py**)

TESTS

Le dossier **progs_tests.sssl/** contient des tests mettant en oeuvre le projet (un fichier **readMe** complémentaire est également présent)

2. Réalisation

2.1. Grammaire et langage

Voici un programme écrit en langage SSSL:

```
class MaClasse {                          // déclaration de classe
        int a                             // déclaration de variable
        float b = 1.2                     // déclaration + affectation
        MaClasse(int arg1, int arg2) {    // constructeur
                int temp = arg1 + arg2
                if (temp > 5) {           // test
                        a = 5             // accès à l'attribut de classe
                }
                else {
                        a = temp
                }
        }
        func int getA() {                 // déclaration de méthode
                return this.a             // this possible
        }
}

func int maFonction() {                   // déclaration de fonction
        return 5
}

void Main() {
        int a = maFonction()              // appel de fonction
        MaClasse object = MaClasse(1, a)  // construction d'objet
        echo(object.getA())               // accès
}
```

Notre langage s'apparente donc à du C légèrement simplifié à la façon Python.

Le langage que nous avons mis en oeuvre est régi par une grammaire de style EBNF compréhensible pour Grako, par exemple:

```
(* __Main__ *)

START = {INSTR}* $ ;

INSTR = AFF | DECLAFF | DECL ;
```

```
(* __Instructions__ *)

DECL = TYPE nom ‘;’ ;

DECLAFF = TYPE nom '=' EXPR ';' ;

AFF = nom '=' EXPR ';' ;

nom = /[a-zA-Z_][a-zA-Z0-9_]*/

TYPE = “int” | “float | “string” ;

...
```

L’écriture de cette grammaire est très importante. Elle ne définit pas seulement le langage, mais aussi la priorité des opérateurs et le contenu de l’arbre syntaxique renvoyé.

Grâce à Grako, la grammaire peut être plus complexe que le langage lui-même et peut permettre la reconnaissance de méthodes de classe (en soit, des fonctions dans des classes) mais en indiquant qu’il s’agit de fonctions (la syntaxe pouvant être la même).

On peut donc jouer avec la grammaire afin de produire un arbre syntaxique plus facilement parsable.

Langage SSSL en détail

Sont présents actuellement dans notre langage:

- Typage statique
- Fonctions, objets (méthodes, attributs, constructeurs, pas d’héritage)
- Présence de “;” en fin d’instruction optionnelle
- Accès aux éléments d’un objet via l’opérateur “.”
- Plusieurs types: bool, int, float, string, void
- Plusieurs opérateurs: ==, !=, >, >=, <, <=, +, -, *, /, or, and
- Instructions conditionnelles: if, else, elif, while, do … while
- Plusieurs mots-clés: func, class, return, break, echo, echoln, Main
    - func: déclaration de fonction
    - class: déclaration de classe
    - return: retour de fonction
    - break: retour de boucle
    - echo: fonction d’affichage
    - echoln: fonction d’affichage avec retour à la ligne
    - Main: entrée du programme
- Possibilité de déclarer des fonctions ou classes dans n’importe quel bloc

Grammaire
Ce langage est encore à discuter et est très maléable (changer la grammaire suffit pour changer le langage).
Peu importe sa syntaxe, notre grammaire contient les primitives suivantes:

- Déclaration de variable — int a
- Déclaration + affectation de variable — int a = 2
- Affectation de variable — a = 2
- Déclaration de fonction — func int maFonction(int arg)
- Appel de fonction — maFonction(2)
- Arguments de fonction (à l'appel)
- Paramètres de fonction (à la déclaration)
- Déclaration de classe — class MaClasse
- Déclaration de méthode
- Déclaration de constructeur
- Main (entrée du programme)
- Instructions conditionnelles
- Opérateurs
- Types
- Noms de variable
- Valeurs immédiates — 4.2, 8, "chaine_de_caractère"
- Bloc d'instructions
- Plus quelques mots clés… — return, break, ...

Cette grammaire se veut la plus large possible. Plus la grammaire possède de primitives, plus il sera facile de traduire vers un langage quelconque. Par exemple, Python possède le mot-clé **elif** (concaténation de **else** et **if**) tout comme notre langage. Si le langage destination ne possède pas ce mot-clé, il suffira simplement de traduire **elif** en **else if**. Par contre, si l'on inverse les rôles (SSSL ne possède pas **elif**, le langage destination oui), il faudra préalablement fusionner les 2 noeuds **else** et **if** en "**elif**" avant de traduire.

Qui peut le plus peut le moins, mais pas inversement. Il s'agit donc d'identifier les concepts généraux des langages afin d'être le plus indépendant possible du langage destination, sans pour entrer dans des détails trop élitistes réservés à des langages bien particuliers. Il est facile de passer d'un langage typé à un langage non typé, d'où cette motivation.

2.2. Arbre syntaxique abstrait (AST)

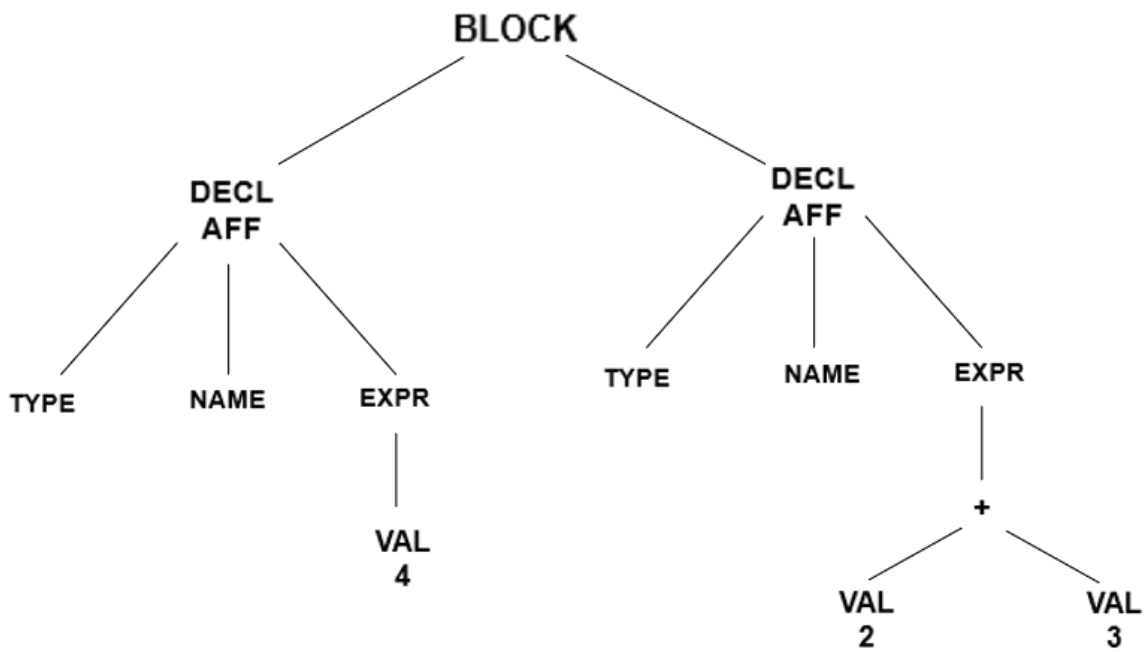
Le module Grako nous permet de vérifier la syntaxe des programmes que nous écrivons, et génère un arbre syntaxique abstrait (abrégé AST) desdits programmes.

Par exemple, le programme...

```
int a = 4
int b = 2+3
```

… aura comme AST

```
"BLOCK": [
        "DECLAFF":    // ici DECLAFF signifie “Déclaration + Affectation”
        "TYPE": "int"
        "NAME": "a"
        "EXPR":
                "VAL": "4"
        "DECLAFF":
        "TYPE": "int"
        "NAME": "b"
        "EXPR": [
                "VAL": "2"
                "+"
                "VAL": "3"
        ]
]
```



Cet AST sera voué à être parsé par la suite afin de procéder à la génération du code. Pour obtenir un tel AST il faut simplement fournir à Grako la grammaire du langage.

Dans un terminal :

```
>       python3 -m grako -m SSSL -o parser_SSSL.py SSSL.ebnf
```

Avec:	SSSL	nom du parser python (SSSLParser)
	parser_SSSL.py	nom du fichier qui contiendra le parser
	SSSL.ebnf	nom de la grammaire

Une fois le parser généré, on peut ensuite l'utiliser dans un programme python...

```
parser = SSSLParser(trace=True, nameguard=False)
ast = parser.parse(program_file.read(), rule_name='START')
print(json.dumps(ast, indent=2))
```

… afin de générer l'AST.

2.3. Nettoyage de l'AST

Toutefois, il faut préalablement nettoyer l'AST. Ceci est un défaut que l'on pourrait reprocher à Grako: lorsque celui se retrouve face à plusieurs possibilités (est-ce une déclaration, une affectation ?) il ne se contente pas d'indiquer la solution, il indique également les possibilités qui ne se sont pas réalisées, L'AST brut :


```
"BLOCK": [
        {
        "DECLAFF": {
        "TYPE": "int",
        "NAME": "a",
        "EXPR": {
                "VAL": "4",
                "PARENTH": null,
                "CFUNC": null,
                "NAME": null
        }
        },
        "DOBJT": null,
        "DMAIN": null,
        "DFUNC": null,
        "DECL": null
        },
```


On procède donc à une passe (fichier **AST_cleaner.py**) permettant d'éliminer ces champs inutiles afin de faciliter le parsing de l'AST. L'AST est en général réduit de moitié, le gain sera davantage important que la grammaire complexe.

2.4. Parsing

Parser l'AST a plusieurs objectifs:

- Recréer l'AST au niveau python (gestion des environments, encapsulation, …)
- Effectuer une analyse sémantique
- Traduire le code

Principe

À chaque noeud est associé une classe dans lesquels se trouve les fils du noeud :

```
class Declaration(Node):
    def __init__(self, data):
        Node.__init__(self, data)

        self.type = Type(data)
        self.name = Name(data, False)

        self.type.fill()
        self.name.fill()
```

```
class Declaffectation(Node):
    def __init__(self, data):
        Node.__init__(self, data)

        self.type = Type(data)
        self.name = Name(data, False)
        self.expr = Expression(data)

        self.type.fill()
        self.name.fill()
        self.expr.fill()
```

Parser l'AST revient à effectuer un parcours en profondeur sur celui-ci, où les différents fils se créent au fur et à mesure, récursivement. L'encapsulation des noeuds constitue des environnements (et sous environnements) qui permettent l'analyse sémantique du programme.

Analyse sémantique

Ce parser permet actuellement de détecter les conflits suivants:

- Déclaration double dans un même bloc
- Utilisation d'une variable, fonction, classe, type n'existant pas dans l'environnement d'un bloc
- Utilisation d'un attribut ou d'une méthode n'existant pas dans un objet
- Incompatibilité de type (arguments de fonction / méthode, affectation, opérateur

2.5. Génération de code

Pour traduire le programme SSSL en langage Go, il suffit simplement d'appeler la méthode __str__() de l'objet racine de l'AST.

Chaque noeud doit définir la façon dont il se traduit et appelle ensuite ses fils à se traduire :

```
# Traduction d'une fonction en Go: func name(...) type {...}
def __str__(self):
    return = "func " + self.name.__str__()
                     + self.parm.__str__()
                     + " " + self.type.__str__()
```

Pour permettre de modifier ces classes facilement, on tire partie de Python pour ajouter dynamiquement ces méthodes aux classes, dans un même fichier:

```
def __go__(self):
    return "var " + self.name.__go__() + " " + self.type.__go__()
Declaration.__go__ = __go__
```

Ce procédé permet également d'utiliser des fonctions intermédiaires à souhait si la traduction devient difficile.

Il est aussi possible de faire coexister plusieurs traductions de différents langages (C++, Java, …).

3. Améliorations

Notre projet se basant sur la création d'un langage, il y aura toujours de la place pour de nouvelles fonctionnalités et améliorations.
Cependant si nous avions continué ce projet, les prochains outils développés auraient été:

- Les Objets : le problème réside dans la transformation en GO. En effet Go ne possède pas vraiment de classe mais l'équivalent de types structurés et la différence de syntaxe est de ce fait très importante
- Commentaires : la possibilité de mettre en commentaire certaines lignes (ou certains bloc ?) de code SSSL
- Lecture au clavier : nous avons implémenté les fonctions system echo et echoln (équivalent du printf avec ou sans retour à la ligne), la suite logique aurait été une fonction de lecture au clavier, très couramment présente sur d'autres langages
- Passage des arguments par valeur ou par référence

- Amélioration de la fonction echo : pour l'instant cette fonction supporte des variables et des chaines de caractères. Elle ne peut cependant pas prendre plus d'un argument
- Caractères spéciaux : implémenter la gestion des caractères tels que \n ou \t. Ils sont pour l'instant incompatibles avec SSSL
- Plus de types primitifs : hashmap, liste, tuple, ...
- Une surcouche permettant d'alléger les fichiers de traductions (=> langage de description adapté)
- Système de tests automatiques
- Héritage et polymorphisme, accesseurs et getters
- Gestion des imports
- ...

4. Problèmes rencontrés

Le plus dur a été de se familiariser avec Grako. Il y a très peu d'exemples et la documentation est assez mince. Obtenir un AST convenable fût difficile.

Python était également nouveau pour nous, le démarrage se fît doucement...

Le parser d'AST a subi de nombreux changements au fil du projet, mais est à présent stable et évolutif.

Le langage Go n'est pas objet. Enfin pas totalement. Cette différence fait que sa traduction en est difficile, mais pas impossible

5. À l'avenir

Le vrai but de SSSL n'a pas encore été défini. Il est extrèmement difficile de traduire un programme d'un langage à un autre lorsque les paradigmes sont différents.

Il est donc extrèmement simple de traduire SSSL en un langage orienté objet (C++, Java, Python, …). Mais si le langage destination ne possède pas d'objet, il ne sera pas possible de traduire un programme SSSL qui en contient, à moins de traitement complexes (ex: implémentation ou pseudo-implémentation des objets en C) les problématiques sont équivalentes à ce qui se fait en optimisation.

À défaut de pouvoir tout traduire, SSSL doit contenir les concepts les plus répandus...