# Rapport du projet de Compilateur

# 1 But

Ce projet du cours de compilation du module de 3ème année de développement système a pour but de développer un interpréteur de c simplifié.

Cet interpréteur a été développé avec les librairies externes existantes Lex et Yacc. La première est consacrée à réaliser l’analyse lexicale d’un fichier texte. Cette analyse sert à savoir qu’est-ce qui est un nombre, un caractère ou un mot. La deuxième reçoit les lexèmes de la première analyse citée et réalise l’analyse syntaxique et sémantique de ce même fichier. Avec Yacc l’analyse syntaxique se produit construisant un arbre (Arbre Syntaxique Abstrait). La sémantique vérifie que l’ordre des lexèmes ait du sens. Ces deux étapes ne sont pas réellement itératives. Elles se mélangent les deux.

Pour savoir si le texte a du sens, ces analyseurs ont besoin de règles. Ces règles ont un nom : la grammaire. La grammaire utilisée est donc celle officielle du langage C un peu simplifiée.

# 2 Spécification

Notre interpréteur reconnait les fonctions et ce qui leur ait caractéristique :

* Le type de retour ;
* Un identifiant (le nom de a fonction) ;
* Les parenthèses ;
* Les accolades ;
* Les différentes instructions ;
* Les retours de fonction.

Les paramètres de fonction ne sont pas traités.

Il interprète aussi les types simples « int » et « float ».

## 2.1 Dépendances

Langage source : **C simplifier**

Langage intermédiaire : **Python 3.5.0** avec

• pydot 1.0.3

• PLY 3.8 (avec les modules lex et yacc)

## 2.2 Type acceptés

Void

Char

Bool

Int

Float

## 2.3 Déclarations acceptées

Int b ;

Int b = 0 ;

## 2.4 Fonctions acceptées

type\_de\_retour nom\_de\_fonction()

{

//Instruction(s) et return

}

## 2.5 Vérification des types

L’interpréteur génère un warning si :

* Une variable est déclarer mais pas utilisée ;
* Une variable est utilisée sans avoir été déclarée.

L’interpréteur génère une erreur sur la ligne courante si :

* Une variable est utilisée sans avoir été initialisée.

# 3 Règles lexicales

reserved\_words = (

'void',

'int',

'float',

'bool',

'char',

'return'

)

tokens = (

'NUMBER',

'SUM\_OP',

'MUL\_OP',

'ID'

) + tuple(map(lambda s:s.upper(),reserved\_words))

literals = '();=<>{}'

# 4 Grammaire

program : declaration\_list

declaration\_list : declaration\_list declaration | declaration

declaration : var\_declaration | fun\_declaration

var\_declaration : type\_specifier ID

fun\_declaration : type\_specifier ID ( ) statement

statement : expression\_stmt | compound\_stmt | return\_stmt

expression\_stmt : expression ; | ;

compound\_stmt : { expression }

return\_stmt : RETURN ; | RETURN expression ;

expression : mutable = simple\_expression| mutable = mutable | mutable = call | simple\_expression

simple\_expression : mutable SUM\_OP mutable | mutable MUL\_OP mutable | call

call : ID ( )

mutable : ID | NUMBER

type\_specifier : VOID | INT | FLOAT | BOOL | CHAR

statement : SUM\_OP statement %prec UMINUS

precedence = (

('left', 'SUM\_OP'),

('left', 'MUL\_OP'),

('right', 'UMINUS'),

)

# 5 Prise en main

Notre compilateur est composé de quatre modules distincts :

* Analyseur lexical (lex.py) ;
* Analyseur syntaxique (parser.py) ;
* Constructeur d’arbre syntaxique (AST.py) ;
* Analyseur sémantique (semantic.py) ;
* Interpréteur récursif (recInterpreter.py)