# Documentation Développeur

Compilateur IFCC (IF-C-Compiler)

Hexanome 4211 : DALAOUI Riad, AUBUT Antoine, CHIKHI Djalil, HANADER Rayan, TAIDER Sami, CHAOUKI Youssef, and LOUVET Samuel

INSA Lyon - Département Informatique - 4IF-PLD-COMP

Février - Avril 2025

## Table des matières

# 1 Architecture du compilateur

## 1.1 Fonctionnalités principales

Le compilateur réalise l'analyse lexicale et syntaxique, la vérification sémantique ainsi que la génération de l'équivalent assembleur (`x86`) pour la liste de fonctionnalitées suivante :

- Un seul fichier source sans pré-processing. Les directives du pré-processeur sont autorisées par la grammaire, mais ignorées, ce afin de garantir que la compilation par un autre compilateur soit possible (exemple : inclusion de `stdio.h`)
- Les commentaires sont ignorés
- Type de données de base `int` (un type 32 bits) et `char` (avec simple quote)
- Variables
- Constantes
- Opérations arithmétiques de base : `+`, `-`, `*`
- Division et modulo
- Opérations logiques bit-à-bit : `|`, `&`, `^`
- Opérations de comparaison : `==`, `!=`, `<`, `>`
- Opérations unaires : `!` et `-`
- Déclaration de variables n'importe où
- Affectation (qui, en `C`, retourne aussi une valeur)
- Utilisation des fonctions standard `putchar` et `getchar` pour les entrées-sorties
- Définition de **fonctions** avec paramètres, et type de retour `int` ou `void`
- Vérification de la cohérence des appels de fonctions et leurs paramètres
- Structure de blocs grâce à `{` et `}`
- Support des portées de variables et du shadowing
- Les structures de contrôle `if`, `else`, `while`
- Support du `return` expression n'importe où
- Vérification qu'une variable utilisée a été déclarée
- Vérification qu'une variable déclarée est utilisée
- Possibilité d'initialiser une variable lors de sa déclaration
- Opérateur d'affectation += -= *=
- Incrémentation ++ et Decrementation –

La grammaire à employer pour rédiger le code qui sera accepté par le compilateur est la même que la grammaire `C` usuelle, dans la limite des fonctionnalités énoncées ci-dessus.

## 1.2 Structure du projet

Le compilateur IFCC suit une architecture modulaire classique en plusieurs passes :

```
/
 antlr/
    include/
    jar/
    lib/
 docs/
    user_doc.pdf
    dev_doc.pdf
 tests/
    testfiles/
    new_tests/
    ifcc-test-output/
    ifcc-wrapper.sh
    ifcc-test.py
 compiler/
    configs/
      config.mk
      DI.mk
      ubuntu.mk
      fedora.mk
    BasicBloc.cpp
    BasicBloc.h
    CFG.cpp
    CFG.h
    CodeGenVisitor.cpp
    CodeGenVisitor.h
    ErrorListenerVisitor.cpp
    ErrorListenerVisitor.h
    ifcc.g4
    IR.cpp
    IR.h
    main.cpp
    Makefile
    Symbol.h
    Symbol.cpp
    SymbolTableVisitor.cpp
    SymboleTableVisitor.h
    Type.cpp
    Type.h
 .gitignore
```

`README.md`

### 1.3 Flux de compilation

1. Analyse lexicale par ANTLR4
2. Analyse syntaxique par ANTLR4
3. Construction de l'AST par ANTLR4
4. Vérification sémantique
5. Génération de code intermédiaire (IR)
6. Génération de code x86

## 2 API Interne

### 2.1 Classes et fonctions principales

Nous ne presenterons ici que les classes relatives à la vérificaton sémantique, à la génération de l'IR et à la génération du code x86 puisque les étapes précédentes du flux de compilation sont gérés par ANTLR4.

#### 2.1.1 Parcours de l'AST et vérification sémantique

Classe de visiteur utilisée pour parcourir l'AST généré par ANTLR et pour produire le code intermediaire (IR). Hérite de ifccBaseVisitor (généré automatiquement par ANTLR à partir de la grammaire).

```
class CodeGenVisitor : public ifccBaseVisitor
{
//Constructeur :
CodeGenVisitor();

// Acces a la liste des CFG generes pour chaque fonction
const vector<shared_ptr<CFG>> &getCfgList() const

// Entree principale du programme
virtual antlrcpp::Any visitAxiom(ifccParser::AxiomContext
*ctx) override;

...
};
```

#### 2.1.2 Definition de l'IR

```
// Represente le Control Flow Graph d'une fonction.
class CFG
{
// Constructeur :
CFG(Type type, const string &name, int argCount,
CodeGenVisitor *visitor);

// Ajoute un bloc
void add_bb(BasicBlock *bb);

// Traduit les instructions IR du CFG en assembleur
void gen_asm_prologue(ostream &o);
void gen_asm(ostream &o);
void gen_asm_epilogue(ostream &o);

...
};

//Represente un bloc de base dans le CFG.
class BasicBlock
{
// Constructeur :
BasicBlock(CFG *cfg, string entry_label);

// Genere l'assembleur du bloc
void gen_asm(ostream &o);

// Ajoute une instruction IR au bloc courant
shared_ptr<Symbol> add_IRInstr(IRInstr::Operation operation,
Type type, vector<Parameter> parameters);

...
};
```

Toutes les instructions de l'`IR` sont contenues dans l'`enum` suivant :

```
typedef enum
  {
    var_assign,
    ldconst,
    ldvar,
    add,
    sub,
    mul,
    div,
```

```
    mod,
    b_and,
    b_or,
    b_xor,
    cmpNZ,
    ret,
    leq,
    lt,
    geq,
    gt,
    eq,
    neq,
    neg,
    not_,
    lnot,
    inc,
    dec,
    nothing,
    call,
    param,
    param_decl
  } Operation;
```

### 2.1.3 Génération du code assembleur

```
// Represente une instruction intermediaire (IR) dans un basic block
class IRInstr
{
// Constructeur :
IRInstr(BasicBlock *basicBlock, Operation operation,
Type type, const vector<Parameter> &parameters);

// Genere le code assembleur pour cette instruction
void genAsm(ostream &os, CFG *cfg);

// Fonctions utilitaires pour l'allocation de registres
set<shared_ptr<Symbol>> getUsedVariables();
set<shared_ptr<Symbol>> getDeclaredVariable();

...
};
```

### 2.1.4 Gestion de la Table des Symboles

La Table des Symboles est partagée entre l'AST et le CFG.

1. Structure representant un Symbol et ses proprietes :

```
struct Symbol
{
// Memory offset (in bytes)
int offset;
// Wheter or not this variable has been used
bool used;
// Wheter the variable was initialized
bool initialized;
// In which line the symbol was declared
int line;
// The type of the symbol
Type type;
// The identifier name corresponding to this symbol
string identifierName;

Symbol(Type type, const string &identifierName)
: type(type), identifierName(identifierName),
used(false), initialized(false), offset(0), line(1) {}

Symbol(Type type, const string &identifierName, int line)
: type(type), identifierName(identifierName),
used(false), initialized(false), offset(0), line(line) {}

};
```

2. Visiteur de la Table des Symboles :

```
class SymbolTableVisitor : public ifccBaseVisitor
{

private:
  unordered_map<string, int> symbolTable;
  set<string> usedVariables;
  int currentOffset = -4;

public:
  unordered_map<string, int> getSymbolTable()
  {return symbolTable;}

  antlrcpp::Any visitFuncCall
  (ifccParser::FuncCallContext *ctx) override;
```

...

}

## 2.2 Fonctionnalités

### 2.2.1 Opérations arithmétiques : +, -, *, /, %

Cette fonctionnalité permet d'appliquer des opérations arithmétiques sur des expressions, et renvoie le resultat de l'opération.

- La grammaire associée est contenue dans `expr` :
  `expr op=('*' | '/' | '%') expr #multdiv`
  `expr op=('+' | '-') expr #addsub`
  au sein du fichier `ifcc.g4`.
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitAddsub(ifccParser::AddsubContext *ctx)`
  et dans :
  `CodeGenVisitor::visitMultdiv(ifccParser::MultdivContext *ctx)`
  au sein du fichier `CodeGenVisitor.cpp`
- La génération du code x86 associé est réalisée par l'appel de l'instruction IR suivante : `IRInstr::add, IRInstr::sub, IRInstr::mul, IRInstr::div, IRInstr::mod`

### 2.2.2 Opérations logiques bit-à-bit : |, &, ^

Cette fonctionnalité permet d'appliquer des opérations logiques bit-à-bit sur des expressions, et renvoie le resultat de l'opération.

- La grammaire associée est contenue dans `expr` :
  `expr '&' expr #b_and`
  `expr '^' expr #b_xor`
  `expr '|' expr #b_or`
  au sein du fichier `ifcc.g4`.
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitB_and(ifccParser::B_andContext *ctx)` et dans :
  `CodeGenVisitor::visitB_or(ifccParser::B_orContext *ctx)` et dans :
  `CodeGenVisitor::visitB_xor(ifccParser::B_xorContext *ctx)` au sein du fichier `CodeGenVisitor.cpp`
- La génération du code x86 associé est réalisée par l'appel de l'instruction IR suivante : `IRInstr::b_and, IRInstr::b_or, IRInstr::b_xor`

### 2.2.3 Opérations unaires : !, ~, ++, - -, + et -

Cette fonctionnalité permet d'appliquer des opérations unaires sur une expression, et renvoie le resultat de l'opération.

- La grammaire associée est contenue dans `expr` :
  `op=('-'|'~'|'!'|'++'|'-'|'+') expr #unaryOp`
  au sein du fichier `ifcc.g4`.
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitUnaryOp(ifccParser::UnaryOpContext *ctx)`
  au sein du fichier `CodeGenVisitor.cpp`
- La génération du code x86 associé est réalisée par l'appel de l'instruction IR suivante : `IRInstr::neg, IRInstr::not_, IRInstr::lnot, IRInstr::inc, IRInstr::dec`

### 2.2.4 Opérations de comparaison : ==, !=, <=, >=

Cette fonctionnalité permet d'appliquer une opération de comparaison entre deux expressions, et renvoie le resultat de l'opération.

- La grammaire associée est contenue dans `expr` :
  `expr op=('==' | '!=') expr #eq`
  `expr op=('<' | '<=' | '>' | '>=') expr #cmp`
  au sein du fichier `ifcc.g4`.
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitEq(ifccParser::EqContext *ctx)` et dans :
  `CodeGenVisitor::visitCmp(ifccParser::CmpContext *ctx)` au sein du fichier `CodeGenVisitor.cpp`
- La génération du code x86 associé est réalisée par l'appel de l'instruction IR suivante : `IRInstr::lt, IRInstr::leq, IRInstr::gt, IRInstr::geq`

### 2.2.5 Entrée/sortie de caractères : getchar, putchar

Cette fonctionnalité permet de lire un caractère depuis l'entrée standard (`getchar`) et d'écrire un caractère vers la sortie standard (`putchar`).

- La grammaire associée est contenue dans `expr` :
  `expr : ID '(' (expr (',' expr)*)? ')' #func_call`
  au sein du fichier `ifcc.g4`.
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitFunc_call(ifccParser::Func_callContext *ctx)` au sein du fichier `CodeGenVisitor.cpp`
- La génération du code IR correspondant est réalisée par les instructions `IRInstr::param` (pour empiler les arguments) et `IRInstr::call`

(pour l'appel de la fonction)

### 2.2.6 Définition et appel de fonctions : `func`, `func_call`

Cette fonctionnalité permet de définir des fonctions avec paramètres et un type de retour `int` ou `void`, et de vérifier la cohérence des appels (nombre et types des paramètres).

- La grammaire associée est contenue dans :
  - `func` :
    `TYPE ID '(' (TYPE ID (',' TYPE ID)*)? ')' #func`
  - `expr` pour l'appel :
    `expr : ID '(' (expr (',' expr)*)? ')' #func_call`
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitFunc(ifccParser::FuncContext *ctx)` et
  `CodeGenVisitor::visitFunc_call(ifccParser::Func_callContext *ctx)`
  au sein du fichier `CodeGenVisitor.cpp`
- La génération du code IR correspondant est réalisée par les instructions :
  `IRInstr::param_decl` (déclaration des paramètres),
  `IRInstr::param` (empilement des arguments),
  `IRInstr::call` (appel de la fonction) et
  `IRInstr::ret` (instruction de retour)

### 2.2.7 Structures de contrôle : `if`, `else`, `while`

Cette fonctionnalité permet de gérer les structures de contrôle conditionnel (`if`/`else`) et les boucles `while`.

- La grammaire associée est contenue dans `stmt` :
  ```
  if_stmt : IF '(' expr ')' block #if
  | IF '(' expr ')' if_block=block ELSE else_block=block #if_else
  while_stmt : WHILE '(' expr ')' block #while_stmt
  ```
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitIf(ifccParser::IfContext *ctx)`,
  `CodeGenVisitor::visitIf_else(ifccParser::If_elseContext *ctx)`
  et
  `CodeGenVisitor::visitWhile_stmt(ifccParser::While_stmtContext *ctx)`
  au sein du fichier `CodeGenVisitor.cpp`
- La génération du code IR correspondant est réalisée par l'instruction :
  `IRInstr::cmpNZ` (pour évaluer la condition et séparer les BasicBlocks)

### 2.2.8 Instruction de retour : `return`

Cette fonctionnalité permet de retourner une expression (pour les fonctions non-void) ou de faire un retour sans valeur (pour les fonctions void) n'importe où dans un bloc, tout en vérifiant la cohérence avec le type de retour de la fonction.

- La grammaire associée est contenue dans `return_stmt` :
  `return_stmt : RETURN (expr)? ';' #return_stmt`
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitReturn_stmt(ifccParser::Return_stmtContext *ctx)` au sein du fichier `CodeGenVisitor.cpp`
- La génération du code IR correspondant est réalisée par l'instruction :
  `IRInstr::ret`

### 2.2.9 Opérations d'affectation composées : `+=`, `-=`, `*=`

Cette fonctionnalité permet de combiner une opération arithmétique (addition, soustraction ou multiplication) avec l'affectation, en modifiant la variable de gauche et en lui assignant le résultat de l'opération.

- La grammaire associée est contenue dans `var_assign_stmt` :
  `var_assign_stmt : ID ('=' | '+=' | '-=' | '*=' | '/=') expr ';' #var_assign_stmt`
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitVar_assign_stmt(ifccParser::Var_assign_stmtContext *ctx)` au sein du fichier `CodeGenVisitor.cpp`
- La génération du code IR correspondant est réalisée par les instructions :
  `IRInstr::ldvar` (chargement de la variable),
  `IRInstr::add` / `IRInstr::sub` / `IRInstr::mul` (selon l'opérateur),
  et
  `IRInstr::var_assign` (réaffectation du résultat)

### 2.2.10 Incrémentation et décrémentation : `++`, `- -`

Cette fonctionnalité permet d'augmenter ou de diminuer la valeur d'une variable de un, soit avant (*pré-*) soit après (*post-*) son utilisation.

- La grammaire associée est contenue dans `expr` :
  `expr : ID ('++' | '- -') #postIncDec`
  `| ('++' | '- -') ID #preIncDec`
- L'analyse sémantique associée est contenue dans :
  `CodeGenVisitor::visitPostIncDec(ifccParser::PostIncDecContext *ctx)` et

`CodeGenVisitor::visitPreIncDec(ifccParser::PreIncDecContext *ctx)`
au sein du fichier `CodeGenVisitor.cpp`

— La génération du code IR correspondant est réalisée par les instructions :
`IRInstr::inc` et `IRInstr::dec`

# 3 Build et tests

## 3.1 Compilation

Les commandes pour compiler le compilateur `IFCC` sont les suivantes

```
cd pld-comp/compiler/
make clean
make
```

Elles présupposent que le fichier `/configs/config.mk` soit adapté à votre système d'exploitation, mais aussi et surtout à votre façon de télécharger ANTLR4 et son emplacement dans votre machine. N'oubliez donc pas **d'installer** ANTLR4 de la manière dont vous souhaitez, mais d'adapter ensuite les fichiers nécessaires.

## 3.2 Exécution des tests

```
cd ../tests
python3 ifcc-test.py [--verbose , --debug , --wrapper] testfiles/*.c
python3 ifcc-test.py [--verbose , --debug , --wrapper] new_tests/*.c
```

Cela présuppose que :

— Vous ayez un interpreteur Python d'installé et mis dans le PATH (si il est autre que `python3`, remplacez-le par la commande correcte).

— Les répertoires `pld-comp/tests/new_tests` et `pld-comp/tests/testfiles` ne contiennent que des fichiers à l'extension `.c`

— Que la compilation du compilateur `IFCC` se soit bien déroulée, et que le fichier `ifcc` soit présent dans `pld-comp/compiler`.

— Si vous ajoutez l'option de compilation `-wrapper` ou `-w`, vous ajoutiez à la suite le chemin vers le script shell que vous souhaitez utiliser à la place de celui par défaut (`ifcc-wrapper.sh`).

Si vous souhaitez ne compiler qu'un fichier `C`, par exemple : `exemple.c`, remplacez à la fin de la ligne de commande `testfiles/*.c` ou `new_tests/*.c` par `testfiles/exemple.c` ou `new_tests/exemple.c`

La compilation d'un nouveau fichier `C` n'est possible que depuis le repertoire /tests puisque c'est là que sont contenus les fichiers principalement

utilisés pour cela : `ifcc-test.py` et `ifcc-wrapper.sh`. L'execution depuis un autre repertoire mènera à une erreur. Le fichier `C` peut, lui, être situé n'importe où dans la machine à condition que le bon chemin soit spécifié dans la commande de compilation.

Le projet est disponible sur GitHub au repository suivant :
https ://github.com/Samsam19191/C-Compiler