Hexanôme 31

NGUEPIE NONO Menphis Jouves
ES-SOUFI Doha
ZHANG Lizhi
DELNESTE Etienne
HABIB Danial
SOW Amadou
WILSON Christopher
LEVRARD Thomas

Documentation Développeur – Compilateur IFCC

Le compilateur IFCC est capable d'analyser un programme en langage C et de générer le code en assembleur correspondant si le programme est correct. Il convient de noter que notre compilateur ne prend pas en charge l'ensemble des fonctionnalités offertes par gcc. Nous présenterons par la suite les principales différences entre le compilateur C standard et le nôtre, en mettant notamment en évidence les programmes que l'on ne prend pas en charge.

Ce projet est fourni avec une batterie de tests automatisés dont le fonctionnement est aussi expliqué par la suite.

Ce projet est réalisé en C++ avec ANTLR et cette documentation indique également les outils nécessaires ainsi que la procédure d'installation.

Le projet est aussi accessible sur GitHub :

https://github.com/danielhabib3/Compilateur

Sommaire

Documentation Développeur – Compilateur IFCC **1**
Sommaire **2**
Installation des outils nécessaires (pour linux) **3**
Architecture Générale **3**
Arborescence détaillée de compiler/ 3
Fonctionnement général de notre compilateur **4**
Grammaire acceptée **5**
Fonctionnalités implémentées **7**
Différences entre gcc et notre compilateur pour les fonctionnalités implémentées **9**
Autres remarques **10**
Compilation du projet **11**
Compilation avec le makefile 12
Utilisation de notre compilateur **12**
Tests Automatisés **13**
Pour exécuter tous les tests 13
Comportement 13
Pour exécuter un test ou un répertoire de tests en particulier 13
Résultats de l'exécution des tests automatisés 14
Affichage dans la sortie standard des statuts d'exécution des tests 14
Description des statuts de test 15
Création de tests 16
Affichage du CFG correspondant à un fichier .c **18**
Affichage de la structure en blocs {.} correspondant à un fichier .c **19**

Installation des outils nécessaires (pour linux)

```
sudo apt install g++ make python3
sudo apt install antlr4 libantlr4-runtime-dev default-jdk
```

Remarque : Nous avons testé notre compilateur sur le bureau virtuel linux de l'INSA et tout fonctionnait normalement

Architecture Générale

```
Compilateur/
├── _other_tests/      : Contient des tests valides qui génèrent des boucles infinies ou alors des
├                        Tests attendant des entrées de l'utilisateur (getchar)
├── compiler/          : Implémentation du compilateur, détaillée dans la suite
├── ifcc-test-output/  : Répertoire de sortie des résultats de tests
├── testfiles/         : Contient tous les fichiers de tests
├── clean_test.py      : Script python permettant de réinitialiser les numéros des fichiers tests
├── ifcc-test.py       : Script python permettant d'executer les tests
```

Arborescence détaillée de compiler/

```
compiler/
├── IR.h / IR.cpp              : Définitions des instructions IR, basicBlock, Cfg
├── IRVisitor.h / .cpp         : Génération de l'IR à partir de l'AST
├── VariableVisitor.h / .cpp   : Gère les variables/blocs (déclaration, définition, portée, erreurs)
├── FunctionVisitor.h / .cpp   : Gère les fonctions (déclaration, définition, erreurs)
├── ifcc.g4                    : Grammaire ANTLR de notre compilateur
├── generated/                 : Contient les fichiers ANTLR générés
├── main.cpp                   : Entrée principale (parsing → vérifications -> visites → asm)
└── Makefile / config.mk       : Compilation de notre projet
```

Fonctionnement général de notre compilateur

main.cpp :

- Lit le fichier source
- Parse le code via ANTLR pour générer l'AST
- Appelle les visiteurs dans l'ordre :
    1. **FunctionVisitor**
    2. **VariableVisitor**
    3. **IRVisitor**

FunctionVisitor :

- Vérifie la cohérence des définitions/appels de fonctions (déclaration vs définition, paramètres, etc.)
- Détecte la présence obligatoire d'un unique **main()**
- Stocke les informations sur les fonctions dans une map

VariableVisitor :

- Analyse les déclarations, initialisations et portées (scopes)
- Construit une hiérarchie de blocs
- Vérifie les utilisations de variables (non-déclarées, non-initialisées, inutilisées)

IRVisitor :

- Transforme l'AST en IR (Création d'instructions IR qui possèdent chacune leur gen_asm)
- Génère des CFGs (Control Flow Graphs) par fonction
- Gère les registres/variables

CFG::gen_asm() (dans IR.cpp) :

- Parcourt les CFGs pour produire l'assembleur

Grammaire acceptée

```
grammar ifcc;

axiom : prog EOF ;

prog : (function_definition | function_declaration)+ ;

function_definition : type ID '(' (type ID (',' type ID)*)? ')' block;

function_declaration : type ID '(' (type (ID)? (',' type (ID)?)*)? ')' ';' ;

block : '{' (instruction)* '}' ;

instruction : declaration | declarationTable | return_stmt | block | test | switch_case | boucle_while | break | continue | expr ';' | ';' ;

switch_case : SWITCH '(' expr ')' '{' (CASE expr ':' (block | instruction)*)* (DEFAULT ':' (block | instruction)*)? '}' ;

test : IF '(' expr ')' block (ELSE block)? ;

boucle_while : WHILE '(' expr ')' block ;

declarationTable : type affectationDeclarationTable (',' affectationDeclarationTable )* ';' ;

affectationDeclarationTable : ID'['CONST']' ('=' '{' (expr (',' expr)*)? '}')? ;

break : (BREAK ';') ;

continue : (CONTINUE ';') ;

declaration : type affectationDeclaration (',' affectationDeclaration )* ';' ;

affectationDeclaration : ID ('=' expr)? ;

return_stmt: RETURN expr ';' ;

function_call : ID '(' (expr (',' expr)*)? ')' ;
```

```
expr : CONST                                                # exprConst
     | CHAR                                                 # exprChar
     | ID                                                   # exprID
     | OP=('++' | '--') ID                                  # exprPrefixIncDec
     | ID '[' expr ']'                                      # exprTable
     | ID OP=('++' | '--')                                  # exprPostfixIncDec
     | function_call                                        # exprFunctionCall
     | '(' expr ')'                                         # exprParenthesis
     | OP=('-' | '!') expr                                  # exprUnary
     | expr OP=('*' | '/' | '%') expr                       # exprMulDivMod
     | expr OP=('+' | '-') expr                             # exprAddSub
     | expr OP=('<' | '>') expr                             # exprCompSupInf
     | expr OP=('!=' | '==') expr                           # exprCompEqual
     | expr '&' expr                                        # exprAndBit
     | expr '^' expr                                        # exprXorBit
     | expr '|' expr                                        # exprOrBit
     | expr '&&' expr                                       # exprLogicalAndLazy
     | expr '||' expr                                       # exprLogicalOrLazy
     | ID '[' expr ']' '=' expr                             # exprAffectationTable
     | ID '=' expr                                          # exprAffectation
     | ID OP=('+=' | '-=') expr                             # exprAffectationComposee
     ;
```

```
type : 'int' | 'char' ;

WHILE : 'while' ;
IF: 'if' ;
ELSE : 'else' ;
RETURN : 'return' ;
BREAK : 'break' ;
CONTINUE : 'continue' ;
SWITCH : 'switch' ;
CASE : 'case' ;
DEFAULT :'default' ;
ID : [a-zA-Z_][a-zA-Z_0-9]* ;
CHAR : '\'' . '\'' ;
CONST : [0-9]+ ;
COMMENT : '/*' .*? '*/' -> skip ;
DIRECTIVE : '#' .*? '\n' -> skip ;
WS    : [ \t\r\n] -> channel(HIDDEN);
```

Fonctionnalités implémentées

Voici le tableau récapitulatif des fonctionnalités que nous avons implémentées :

Fonctionnalité	Priorité	Commentaires
Un seul fichier source sans pré-processing. Les directives du préprocesseur sont autorisées par la grammaire mais ignorées	DI	
Les commentaires sont ignorés	DI	
Type de données de base int (un type 32 bits)	O	
Variables	O	
Constantes entières et caractère (avec simple quote)	O	
Opérations arithmétiques de base : +,-, * avec support des parenthèses	O	
Division et modulo	O	
Opérations logiques bit-à-bit : |, &,^	O	
Opérations de comparaison : ==, !=, <, >	O	
Opérations unaires : ! et -	O	
Déclaration de variables n'importe où	O	
Affectation (qui, en C, retourne aussi une valeur)	O	
Utilisation des fonctions standard putchar et getchar pour les entrées-sorties	O	
Définition de fonctions avec paramètres, et type de retour int	O	**Sans Void et maximum 6 paramètres et la taille allouée pour les variables locales est définie statiquement**
Vérification de la cohérence des appels de fonctions et leurs paramètres	O	
Structure de blocs grâce à { et }	O	
Support des portées de variables et du shadowing	O	
Les structures de contrôle if, else, while	O	
Support du return expression n'importe où	O	**Pareil que gcc si y a pas un return dans un main qui retourne un int on retourne 0**

Vérification qu'une variable utilisée a été déclarée	**O**	
Vérification qu'une variable n'est pas déclarée plusieurs fois	**O**	
Vérification qu'une variable déclarée est utilisée	**O**	
Reciblage vers plusieurs architectures : x86, MSP430, ARM	**F**	**x86 + ARM**
Support des double avec toutes les conversions implicites	**F**	
Propagation de constantes simple	**F**	
Propagation de variables constantes (avec analyse du data-flow)	**F**	
Tableaux (à une dimension)	**F**	**seulement les tableaux à une dimension**
Pointeurs	**F**	
break et continue	**F**	
Les chaînes de caractères représentées par des tableaux de char	**F**	
Possibilité d'initialiser une variable lors de sa déclaration	**F**	
switch...case	**F**	**Uniquement dans la grammaire mais rien de plus**
Les opérateurs logiques paresseux ||, &&	**F**	
Opérateurs d'affectation +=, -= etc., d'incrémentation ++ et décrémentation - -	**F**	**Uniquement (+= ; -=) et (++ ; --)**
Les variables globales	**NP**	
Les autres types de inttypes.h, les float	**NP**	
Le support dans les moindres détails de tous les autres opérateurs arith- métiques et logiques : <=, >=, << et >> etc.	**NP**	
Les autres structures de contrôle : for, do...while	**NP**	
La possibilité de séparer dans des fichiers distincts les déclarations et les définitions	**D**	
Le support du préprocesseur (#define, #include, #if, etc.)	**D**	
Les structures et unions	**D**	

Support en largeur du type de données char (entier 8 bits)	**D**	

Différences entre gcc et notre compilateur pour les fonctionnalités implémentées

Fonctions :

- Nous n'avons pas généralisé l'allocation de mémoire pour les variables locales d'une fonction, nous avons fixé l'allocation à 64 octets par fonction ce qui était suffisant pour la plupart de nos tests

Types de données :

- int et char sont gérés de la même manière, ils sont synonymes, seulement on tronque pour ne récupérer que le premier octet lorsque c'est un char.

Tableaux :

- Seuls les tableaux de tailles fixes sont acceptés (donc pas de variable), d'ailleurs il faut même que la taille indiquée lors de la déclaration soit directement un nombre entier et non pas une expression (ex : int t[2*10]; pas accepté mais int t[20]; oui)

Messages d'erreurs :

- Ifcc ne spécifie pas de messages d'erreurs très détaillés comme peut le faire gcc parfois mais souvent nous indiquons tout de même la ligne concernée, la colonne concernée et le type d'erreur.

Gestion des erreurs :

- Certaines erreurs complexes ne sont pas gérées, comme par exemple, des programmes contenant : return a+5=10;

Priorité opérateurs :

- Certains opérateurs devraient être au même niveau (si c'était identique au C) mais nous les avons séparés

  Exemple : ID '[' expr ']' devrait être au même niveau que exprPrefixIncDec

Autres remarques

Retour void :

- Nous n'avons pas implémenté le fait de pouvoir mettre void en tant que type de retour

Reciblage vers ARM :

- Nous avons implémenté le reciblage vers ARM mais pas pour toutes les fonctionnalités. Globalement, toutes les fonctionnalités que nous avions implémentées pour le rendu intermédiaire fonctionnent en ARM, c'est à dire :

    1. Type de données de base int (un type 32 bits)
    2. Variables
    3. Constantes entières
    4. Opérations arithmétiques de base : +,-, *
    5. Division et modulo
    6. Opérations logiques bit-à-bit : |, &,ˆ
    7. Opérations de comparaison : ==, !=, <, >
    8. Opérations unaires : ! et -
    9. Déclaration de variables n'importe où
    10. Affectation
    11. Vérification qu'une variable utilisée a été déclarée
    12. Vérification qu'une variable n'est pas déclarée plusieurs fois
    13. Vérification qu'une variable déclarée est utilisée
    14. Vérification qu'une variable utilisée a été initialisée

Compilation du projet

Choix de l'assembleur cible

Nous avons implémenté le reciblage vers ARM en plus du reciblage vers x86.

Pour choisir le reciblage que vous souhaitez utiliser, il suffit de modifier le nom du fichier IR.cpp. En effet, dans la version fournie, le fichier IR.cpp contient de l'assembleur x86 donc si vous utilisez cette version sans rien changer, le reciblage choisi sera par défaut x86.

Cependant, si vous souhaitez tester le reciblage vers ARM, il suffit de renommer le fichier IR.cpp actuel en IR_x86.cpp puis de renommer le fichier IR_ARM.cpp actuel en IR.cpp.

Ainsi, vous pourrez tester le reciblage ARM.

Si vous souhaitez, dans la suite, repasser en assembleur x86, il vous suffira, d'une manière analogue, de renommer le fichier IR.cpp actuel en IR_ARM.cpp et de renommer le fichier IR_86.cpp en IR.cpp.

Remarques :

- Pour tester le reciblage vers ARM, il faut que vous ayez un processeur qui prenne en charge l'assembleur ARM (souvent les MAC), et de même pour le x86.

- Nous sommes conscients que la manière de choisir le reciblage n'est clairement pas "propre"/pratique mais cela devrait tout de même fonctionner

Compilation avec le makefile

Depuis `compiler/`

- Avec la commande : `make`
- Utiliser les tests automatisés (cf. suite) compile automatiquement le compilateur ifcc, il est juste nécessaire de faire un `make clean` au préalable si vous possédez une ancienne version de l'exécutable

Utilisation de notre compilateur

Depuis `compiler/`

Il est possible de compiler un programme C à l'aide du compilateur IFCC à l'aide de la commande suivante (après compilation de ce dernier)

```
./ifcc path/to/file.c
```

Les résultats de cette exécution sont les suivants :

- Affiche de l'ASM généré dans la sortie standard.

En cas d'erreur, les messages sont affichés sur `stderr`.

Remarque:
Dans le code on a utilisé la méthode anycast de std::any et on l'a testé sur le bv et ça marche

Tests Automatisés

Pour exécuter tous les tests

Commande

Depuis `compiler/`

```
make test
```

Ou depuis `compilateur/` :

```
python3 ifcc-test.py testfiles/
```

Comportement

- Teste tous les fichiers de tests dans `testfiles/`.
- Pour chaque fichier :
    - Compile avec **GCC** et **IFCC**.
    - Exécute les deux binaires si le programme compile
    - Compare les résultats d'exécution si le programme compile
- Stocke les résultats dans `ifcc-test-output/`
- Affiche les résultats des tests dans la sortie standard

Pour exécuter un test ou un répertoire de tests en particulier

Commande

Depuis `compiler/`

```
make test FILE="path/to/file.c"
```

Ou depuis `compilateur/` :

```
python3 ifcc-test.py "path/to/file.c"
```

Ainsi, vous pouvez rajouter, si vous le souhaitez, vos propres fichiers .c afin de les tester avec notre compilateur.

remarque : Vous pouvez tester un répertoire de tests en particulier et non pas seulement un fichier, pour cela il suffit de mettre un chemin de répertoire et non de fichier .c

Résultats de l'exécution des tests automatisés

Fichiers de résultat dans `ifcc-test-output/`

Voici ci-dessous la liste des fichiers stockés dans `ifcc-test-output/` pour chaque fichier de test .c :

Fichier	Description
`gcc-asm`	L'ASM généré par GCC
`ifcc-asm`	L'ASM généré par IFCC
`gcc-compiler`	Messages de compilation GCC
`gcc-execute`	Résultat de l'exécution GCC
`ifcc-compiler`	Messages de compilation IFCC
`ifcc-execute`	Résultat de l'exécution IFCC

Affichage dans la sortie standard des statuts d'exécution des tests

```
-------------------------------------------

Starting NotImplementedYet

TEST-CASE: 3_if_with_huge_constant                              TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 4_invalid_array_size                                 TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 5_pointeur_base                                      TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 6_print                                              TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 1_2d_array                                           TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 2_array_declaration_with_expression_size             TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 10_if_expr_complexe_sans_variable                    TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 11_if_prio_logique_ambigue                           TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 12_if_sans_else_plusieurs_if                         TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 7_dangling_else_ambiguity                            TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 8_if_direct_return_sans_accolades                    TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 9_if_expr_bitwise_priorite                           TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 13_BreakDansSwitch                                   TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 14_SwitchAvecCaseSansDeuxPoints                      TEST OK
TEST-CASE: 15_SwitchAvecDeuxDefaults                            TEST OK
TEST-CASE: 16_SwitchAvecExprNonValide                           TEST OK
TEST-CASE: 17_SwitchSansAccolades                               TEST OK
TEST-CASE: 18_SwitchAvecBlocDansCase                            TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 19_SwitchAvecDeuxCasesEtDefault                      TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 20_SwitchAvecInstructionsDansCase                    TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 21_SwitchAvecUnSeulCaseEtDefault                     TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 22_SwitchSansBreakDansUnCase                         TEST FAIL    (your compiler rejects a valid program)
TEST-CASE: 23_SwitchSansDefault                                 TEST FAIL    (your compiler rejects a valid program)

==== Summary of Tests for NotImplementedYet  ===
4 success, 19 failures

-------------------------------------------

Starting TestsOK

Starting TestsOK-Invalid

TEST-CASE: 24_invalid_program                                   TEST OK
TEST-CASE: 25_unknown_type                                      TEST OK
TEST-CASE: 26_if_var_declared_in_if_block                       TEST OK
TEST-CASE: 27_invalid_multiple_affectation_2                    TEST OK
TEST-CASE: 28_invalid_multiple_declaration                      TEST OK
```

Description des statuts de test :

`TEST OK` : Comportement identique entre IFCC et GCC

Le test est considéré comme réussi lorsque le compilateur IFCC et GCC produisent le même comportement.

Cela inclut :

- une sortie standard strictement identique (affichages avec putchar, etc.) ;
- un code de retour identique
- un comportement de compilation similaire, notamment le rejet d'un programme invalide par les deux compilateurs.

Ainsi, un test est **OK** si le programme compilé avec IFCC se comporte exactement comme celui compilé avec GCC, que ce soit dans le cas d'une exécution normale, d'une erreur détectée et gérée, ou d'un refus de compilation justifié.

`TEST FAIL` : Divergence entre IFCC et GCC

Le test est considéré comme échoué lorsque les compilateurs IFCC et GCC produisent un comportement différent. Cela inclut :

1. **IFCC accepte un programme invalide**
    - GCC échoue (rejette le code), mais IFCC le compile sans erreur.
    - ➜ Message : `TEST FAIL (your compiler accepts an invalid program)`
2. **IFCC rejette un programme valide**
    - GCC réussit à compiler, mais IFCC échoue.
    - ➜ Message : `TEST FAIL (your compiler rejects a valid program)`
3. **IFCC génère du code assembleur incorrect**
    - IFCC compile, mais l'assembleur généré par ifcc est incorrect
    - ➜ Message : `TEST FAIL (your compiler produces incorrect assembly)`

4. **Différences à l'exécution**
    - Les deux compilateurs produisent des exécutables, mais les résultats à l'exécution sont différents.
    - ➜ Message : `TEST FAIL (different results at execution)`
    -

Création de tests

Il est possible d'ajouter des tests, pour cela il suffit d'ajouter le programme C sur lequel vous voulez tester le compilateur IFCC, de préférence dans le répertoire `testfiles/`.

Une classification des tests est déjà présente les tests présents dans `testfiles/` :

Il y a 3 sous répertoires principaux dans testfiles/

• `NotImplementedYet/` : Tests d'anticipation des fonctionnalités que nous comptons implémentées dans le futur

• `TestsOK/ :` Tests des fonctionnalités implémentées

• `WillNeverBeImplemented/` : Tests des fonctionnalités qu'on ne va jamais implémenter ou alors qui ne peuvent pas obtenir le même résultat que gcc car indéterminé (ex : retour d'une variable non définie)

Puis, pour améliorer l'organisation du répertoire `TestsOK`/ (qui contient la majorité des tests), nous avons créé 2 sous répertoires :

- Répertoire **Valid/**
    - Prérequis : le code est correct aussi bien syntaxiquement que sémantiquement, ainsi son exécution doit amener à la production d'un code assembleur
    - Statuts d'exécution :
        - TestOK
        - TestFAIL
- Catégorie **Invalid/**
    - Prérequis : le code est incorrect
    - Statuts d'exécution :
        - TestOK
        - TestFAIL

- Remarque: Pour les tests valid, il y a des tests qui font TEST FAIL de temps en temps (mais c'est rare) mais quand on les refait ils font des TEST OK, c'est peut être un bug du fichier python mais on n'a pas eu le temps de la régler.

Organisation des tests en sous répertoires propres à chaque fonctionnalité :

Puis, pour encore améliorer l'organisation de ses sous répertoires, nous avons créé des sous répertoires propres à chaque fonctionnalité (ex : Expressions/, Functions/, Break/, …)

Ainsi, si vous voulez rajouter des tests, il vaut mieux respecter cette organisation.

Nommage des tests :

Il est recommandé d'utiliser la numérotation automatique des tests, pour cela il suffit de nommer votre test à ajouter comme suit :

`_nomtest` où nomtest correspond au nom souhaité.

Affichage du CFG correspondant à un fichier .c

Prérequis

Installer graphviz et feh

```
sudo apt install graphviz
sudo apt install feh
```

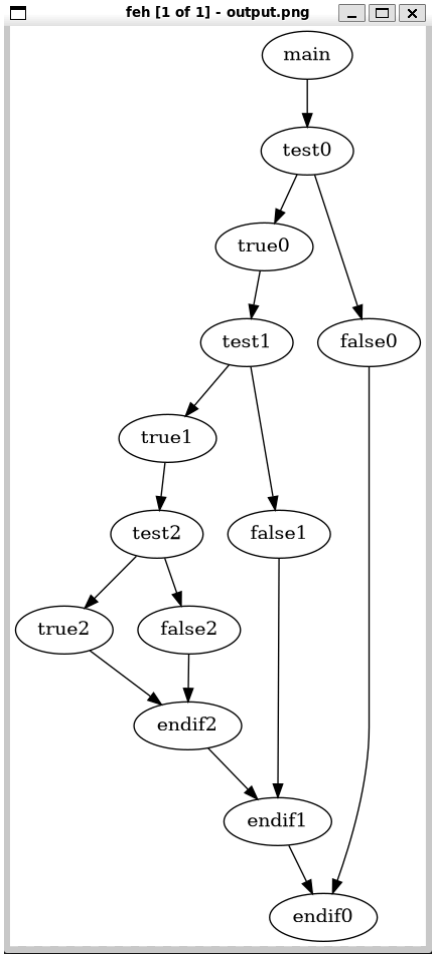
Utilisation

Depuis `compiler/`

```
make cfg FILE=”path/to/file.c”
```

Exemple :

```
make cfg FILE= ”../testfiles/TestsOK/Valid/IfElse/213_3_if_imbrique.c”
```



Affichage de la structure en blocs {.} correspondant à un fichier .c

Prérequis

Installer graphviz et feh

```
sudo apt install graphviz
sudo apt install feh
```

Utilisation

Depuis `compiler/`

```
make bloc FILE=”path/to/file.c”
```

Exemple :

```
make bloc
FILE= ”../testfiles/TestsOK/Valid/IfElse/213_3_if_imbrique.c”
```