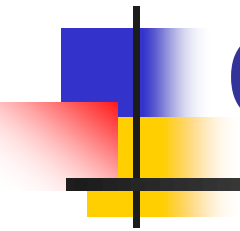
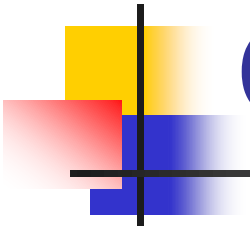


Compilation Avancée

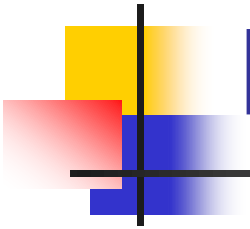
ENSIIE – S5

Cours 2 : Le Middle-End



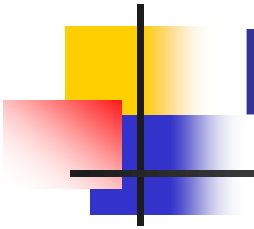
Organisation du cours (rappel)

- Responsable de cours
  - Patrick Carribault (patrick.carribault@cea.fr)
- Intervenants
  - Antoine Capra (antoine.capra@eviden.com)
  - Van Man Nguyen (van-man.nguyen@eviden.com)
- Evaluation
  - Projet en binôme



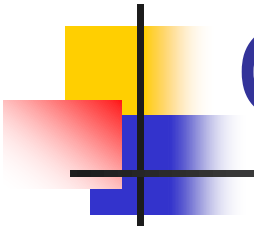
Projet

- Thème
  - Validation statique/dynamique de programmes MPI
- Evaluation en binôme
  - Rapport (dizaine de pages)
  - Soutenance (10 présentation + 5 minutes démonstration + questions)
- Dates clefs
  - **Liste des binômes : 02/10**
  - Rendu du rapport + code source : 06/11
  - Soutenance : 06/11
- Conseils
  - Les TPs suivent globalement le projet ➔ projet à travailler au fur et à mesure du déroulement du module !
  - Bien faire les parties obligatoires du projet
  - Vérifier et valider le code source avec plusieurs exemples



Plan du cours

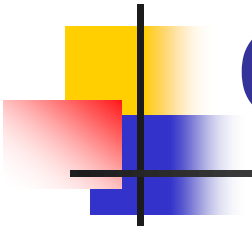
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG



Cours 2

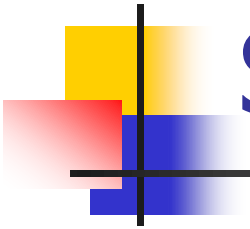
- Présentation de GCC
  - Introduction
  - Structuré générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG

Chaîne de compilation GNU

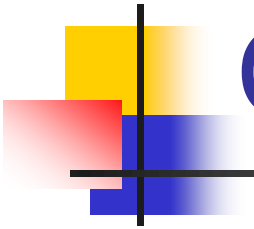


- GCC : GNU Compiler Collection
  - Historiquement GNU C Compiler
- Ensemble d'outils et de bibliothèque pour la compilation
  - Plusieurs langages, plusieurs architectures
  - Générateur de compilateurs !
- Disponible sous licence GPL
  - http://gcc.gnu.org
- Support principal des TDs/TPs !

Survol des fonctionnalités



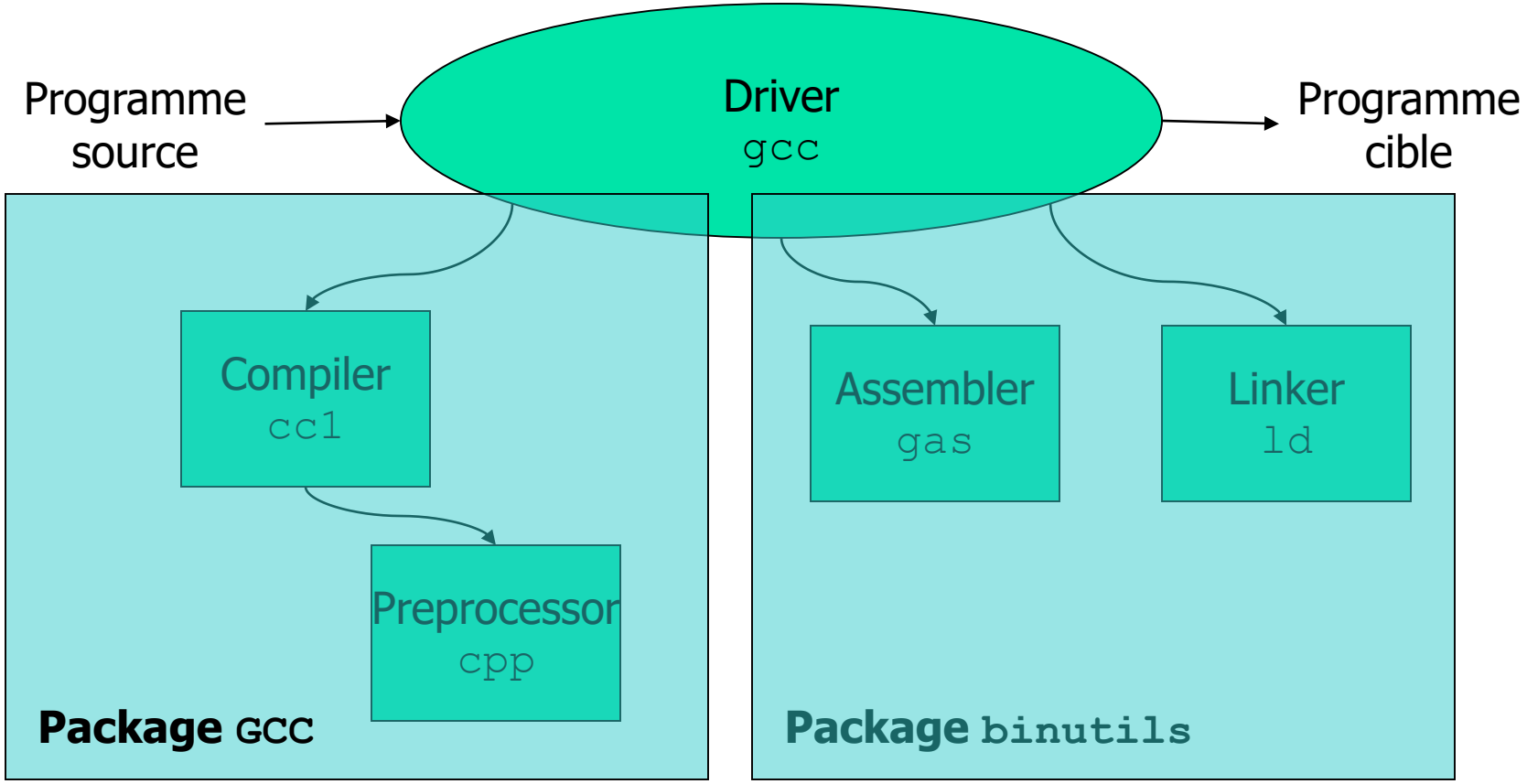
- Langages supportés
  - C, C++
  - Objective-C, Objective-C++
  - JAVA,
  - Fortran
  - ADA
- Processeurs supportés
  - ARM, IA-32 (x86), x86-64, IA-64, MIPS, SPARC, …
- Système de *plugins* pour ajouter/modifier des passes de compilation



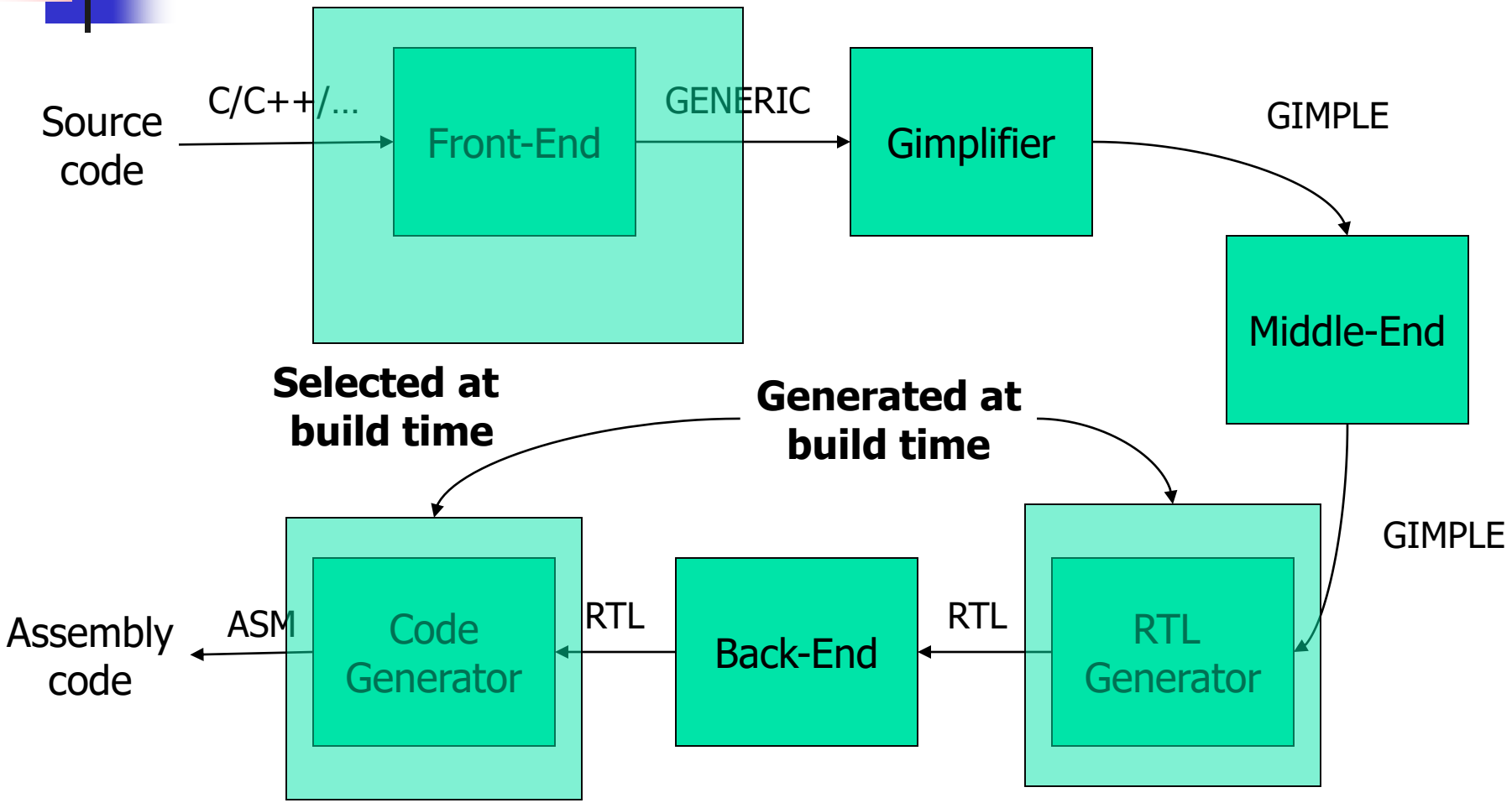
Cours 2

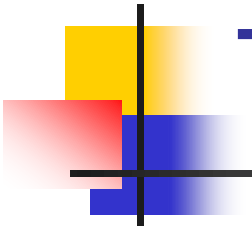
- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG

Architecture de GCC



Architecture de GCC

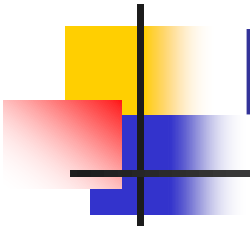




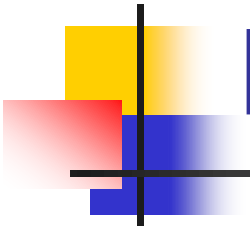
Transformations dans GCC

- GCC possède un total de 203 passes de transformations
- Le nombre total de passes effectuées lors d'une compilation est 239
  - Certaines transformations sont appelées plusieurs fois
- Pour l'enchainement des transformations sur les représentations intermédiaires, GCC utilise un *pass manager*
  - Situé dans les fichiers `${SOURCE}/gcc/passes.c` et `${SOURCE}/gcc/passes.def`

Historique de GCC



- 0.9 : 22 Mars 1987
  - Première version beta
- GCC 1.0 : 23 Mai 1987
- GCC 3.0 : 18 Juin 2001
  - Ajout du support du langage JAVA
- GCC 4.0 : 20 Avril 2005
  - Ajout de la branche *tree-ssa*
  - Ajout de l'algorithme de pipeline logiciel *Swing Modulo Scheduling (SMS)*
  - Représentation intermédiaire GIMPLE
- GCC 4.2.0 : 13 Mai 2007
  - Support de OpenMP pour C, C++ et Fortran
- GCC 4.5.0 : 14 Avril 2010
  - Optimisations au *link* (LTO)
- GCC 4.6.0 : 25 Mars 2011
  - Réduction de l'empreinte mémoire / meilleure exploitation du cache
  - Ajout de nouveau langages : CAF et GO
- GCC 4.7.0 : 22 Mars 2012
  - OpenMP 3.1
  - Standard C++11
- GCC 4.8.0 : 22 Mars 2013
  - Programmation en partie en C++ 2003
  - Support intégral du standard C++11



Historique de GCC

- GCC 4.9.0 : 22 Avril 2014
  - OpenMP 4.0
  - Amélioration des diagnostiques (incluant de la couleur)
  - Support expérimental pour C++14
  - Go 1.2.1
  - Support AVX-512
- GCC 5.1 : 22 Avril 2015
  - Amélioration du support C++ 14
  - OpenMP 4.0 offloading
  - Implémentation préliminaire pour OpenACC 2.0
  - Support spécifiques pour les architectures Intel Xeon Phi
  - Go 1.4.2
- GCC 5.2 : 16 Juillet 2015
  - Support du mot clé « vector »
  - Support amélioré pour les instructions AMD
  - Support du processeur IBM z13
- GCC 5.3 : 4 Décembre 2015
  - Support du processeur Intel Skylake avec AVX-512
  - Support des processeurs IBM z pour le langage GO
- GCC 6.1 : 27 Avril 2016
  - OpenMP 4.5
  - Amélioriation du support de OpenACC 2.0
  - Support expérimental pour C++ 17
- GCC 6.2 : 22 Août 2016
  - Support SPARC

Historique de GCC

- GCC 7.1 : 2 Mail 2017
- GCC 7.2 : 14 Août 2017
- GCC 7.3 : 25 Janvier 2018
  - Amélioration des avertissements
  - Proposition de noms dans le cas d'une typo (changement au niveau des front-ens)
  - Support expérimental du c++17
  - Ajout de la génération de code pour plusieurs processeurs ARM (e.g., Cavium ThunderX)
  - Possibilité d'utiliser les GPU Nvidia avec OpenMP 4
  - Ajout du jeu d'instruction RISC V
- GCC 8.1 : 2 Mai 2018
- GCC 8.2 : 14 Juillet 2018
  - Support expérimental du C++2a
  - Amélioration du C++17
  - Support du jeu d'instruction vectoriel ARM SVE
- GCC 9.1: 5 Mai 2019
- GCC 9.2: 12 Août 2019
  - Amélioration des diagnostiques
  - Amélioration de la génération de code
  - Support partiel d'OpenMP 5.0
  - Implémentation C++17 mature
- GCC 10.1 : 7 Mai 2020
- GCC 10.2 : 23 Juillet 2020
- GCC 10.3 : 8 Avril 2021
- GCC 11.1 : 27 Avril 2021
- GCC 12.1 : 6 Mai 2022
- GCC 12.2 : 19 Aout 2022

Support des TDs

Evolution de la taille de GCC

Count		GCC 4.3.0	GCC 4.4.2	GCC 4.5.0
Lines	Main source	2,029,115	2,187,216	2,320,963
	Libraries	1,546,826	1,633,558	1,671,501
	Subdirectories	3,527	3,794	4,055
Files	Number of files	57,660	62,301	77,782
	C source files	15,477	18,225	20,024
	Header files	9,646	9,213	9,389
	C++ files	3,708	4,232	4,801
	Machine description	186	206	229

(Line counts estimated by David A. Wheeler's sloccount program)

Taille de GCC 4.6.2

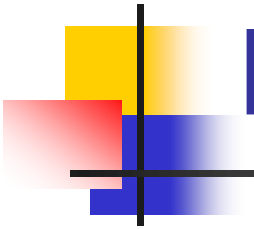
Language	Files	Code	Comment	Comment %	Blank	Total
c	18624	2106311	445288	17.5%	419325	2970924
cpp	22206	989098	230376	18.9%	215739	1435213
java	6342	681938	645505	48.6%	169046	1496489
ada	4616	680251	316021	31.7%	234551	1230823
autoconf	91	405517	509	0.1%	62919	468945
html	457	168378	5669	3.3%	38146	212193
make	98	121136	3658	2.9%	15555	140349
fortranfixed	2989	100688	1950	1.9%	13894	116532
shell	148	48032	10451	17.9%	6586	65069
assembler	208	46750	10227	17.9%	7854	64831
xml	75	36178	282	0.8%	3827	40287
objective_c	869	28049	5023	15.2%	8124	41196
fortranfree	831	13996	3204	18.6%	1728	18928
tex	2	11060	5776	34.3%	1433	18269
scheme	6	11023	1010	8.4%	1205	13238
automake	67	9442	1039	9.9%	1457	11938
perl	28	4445	1316	22.8%	837	6598
ocaml	6	2814	576	17.0%	378	3768
xslt	20	2805	436	13.5%	563	3804
awk	11	1740	396	18.5%	257	2393
python	10	1725	322	15.7%	383	2430
css	24	1589	143	8.3%	332	2064
pascal	4	1044	141	11.9%	218	1403
csharp	9	879	506	36.5%	230	1615
dcl	2	402	84	17.3%	13	499
tcl	1	392	113	22.4%	72	577
javascript	4	341	87	20.3%	35	463
haskell	49	153	0	0.0%	17	170
bat	3	7	0	0.0%	0	7
matlab	1	5	0	0.0%	0	5
Total	57801	5476188	1690108	23.6%	1204724	8371020

Transformations GIMPLE

Pass Group	Number of passes
Lowering	12
Interprocedural optimizations	49
Intraprocedural optimizations	42
Loop optimizations	27
Remaining intraprocedural optimizations	23
Generating RTL	01
Total	154

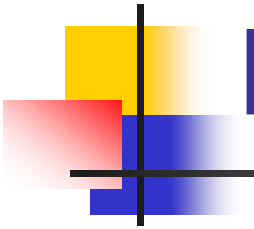
Transformations RTL

Pass Group	Number of passes
Intraprocedural Optimizations	21
Loop optimizations	7
Machine Dependent Optimizations	54
Assembly Emission and Finishing	03
Total	85



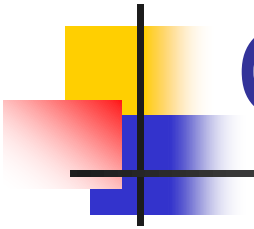
Préprocesseur

- CPP : Gestion des directives de précompilation
- Syntaxe des directives
  - #keyword
- Exemple de directives
  - #ifdef
  - #include
  - #warning
  - #error
- Explosion de la taille du code après *preprocessing*
- Attention #pragma n'est pas traité par le préprocesseur



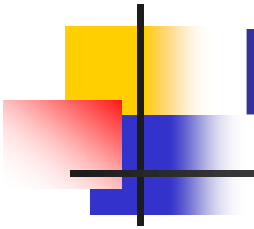
Front-end

- Lecture du fichier source en entrée
  - C, C++, Fortran, Java, C#, …
- Vérification de la validité du code
  - Analyse lexicale
  - Analyse syntaxique
  - Analyse sémantique
  - Cf. CPA cours 1
- Chaque front-end est dans un répertoire différent :
  - C, ObjectiveC → ${SOURCE}/gcc/c/, ${SOURCE}/gcc/c-family/
  - C++ → ${SOURCE}/gcc/cp/, ${SOURCE}/gcc/c-family/
  - Fortran → ${SOURCE}/gcc/fortran/
- En sortie, le code est représenté en GENERIC
  - Sauf pour C/C++ qui génère directement du GIMPLE



GENERIC

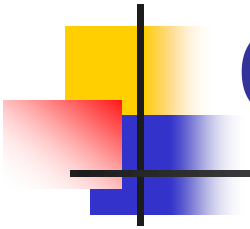
- Représentation intermédiaire sous forme d'arbre
- Indépendant du langage source
- Processus de création d'une représentation GENERIC
  - Génération de l'arbre de syntaxe abstraite par le *parser*
  - Le parser peut garder cette représentation
  - Suppression des constructions spécifiques au langage
  - Emission de l'arbre GENERIC à la fin de la phase de *parsing*
- Tous les noeuds sont définis dans `$(SOURCE)/gcc/tree.def`
  - Notion de *tree codes*



Middle-end

- Optimisation haut niveau
  - Indépendante de l'architecture
- Granularités
  - Optimisation par fonction
  - Optimisation par boucle
  - Optimisation inter-procédurale
- Ordre des transformations géré par le *pass manager* de GCC
- Travail sur une représentation intermédiaire nommée GIMPLE
  - En conjonction avec d'autre RIs (par exemple CFG)

  → Détails dans le prochain cours



GIMPLE

- Représentation intermédiaire de haut niveau
  - Introduite dans GCC 4.4
  - Basée sur une représentation avec un arbre
  - Nœud avec une sémantique
- Sous-ensemble simplifié de GENERIC
  - Représentation 3-adresses
  - Aplatissement du flot de contrôle
  - Simplifications et nettoyage (la grammaire est restreinte)
  - Transformation de GENERIC vers GIMPLE
    - `gimplify_function_tree()` dans le fichier `gimplify.c`
- Deux niveaux de GIMPLE
  - *High GIMPLE*
  - *Low GIMPLE*

GIMPLE – Exemple en C

- Exemple simple
  - Langage C
  - Une seule fonction main
- Compilation avec sortie des fichiers intermédiaires :
  - `gcc -fdump-tree-all test.c`
  - Génération de la représentation GIMPLE entre les transformations

```
int main() {
   int x = 10 ;
   if (x) {
        int y = 5 ;
        x = x*y+15 ;
   }
}
```

GIMPLE – Exemple en C

Fichier test.c:

```
int main() {
   int x = 10 ;
   if (x) {
        int y = 5 ;
        x = x*y+15 ;
   }
}
```


Fichier test.c.004t.gimple:

```
main() {
   int D.2720;
   int x;
   x = 10 ;
   if (x!=0) goto <D.2718>;
   else goto <D.2719>;
   <D.2718>:
   {
        int y;
        y=5;
        D.2720 = x*y;
        x = D.2720+15
   }
   <D.2719>:
}
```

- Déclaration de temporaires
  - `D.2720`
- Simplification pour le code 3 adresses
  - `D.2720 = x*y`
- Flot de contrôle avec `goto`

GIMPLE – Exemple en C

- Génération du code GIMPLE
  - `gcc -fdump-tree-all-raw test.c`

Fichier test.c.004t.gimple:

```
main() {
    int D.2720;
    int x;
    x = 10 ;
    if (x!=0) goto <D.2718>;
    else goto <D.2719>;
    <D.2718>:
    {
          int y;
          y=5;
          D.2720 = x*y;
          x = D.2720+15
    }
    <D.2719>:
}
```

Fichier test.c.004t.gimple:

```
main()
gimple_bind <
    int D.2720;
    int x;
    gimple_assign<integer_cst,x,10,NULL
    >
    gimple_cond <ne_expr, x, 0,
    <D.2718>, <D.2719> >
    gimple_label <<D.2718>>
    gimple_bind <
          int y;
          gimple_assign<integer_cst, y,
    5, NULL>
           gimple_assign<mult_expr,
    D.2720, x, y>
          gimple_assign<plus_expr,x,
    D.2720,15>
    >
    gimple_label<<D.2719>>
>
```

GIMPLE – Exemple en C

Fichier test.c.004t.gimple:

```
main() {
    int D.2720;
    int x;
    x = 10 ;
    if (x!=0) goto <D.2718>;
    else goto <D.2719>;
    <D.2718>:
    {
          int y;
          y=5;
          D.2720 = x*y;
          x = D.2720+15
    }
    <D.2719>:
}
```

Fichier test.c.011t.cfg

```
main() {
    int y;
    int x;
    int D.2720;

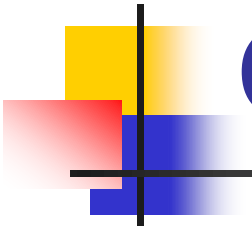
<bb2>:
    x=10;
    if (x!=0) goto <bb 3>;
    else goto <bb 4>;

<bb 3>:
    y=5;
    D.2720 = x*y;
    x=D.2720+15;

<bb 4>:
    return ;
}
```

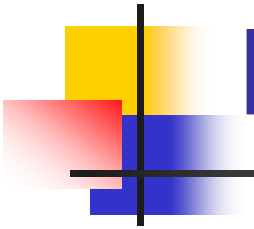
GIMPLE - *tree code*

- Tous les *tree code* de GCC (152) sont listés dans `$(SOURCE)/gcc/tree.def`
- Binary Operator
  - MAX EXPR
- Comparison
  - EQ EXPR, LT EXPR
- Constants
  - INTEGER CST, STRING CST
- Declaration
  - FUNCTION DECL, LABEL DECL , VAR DECL
- Expression
  - PLUS EXPR, ADDR EXPR
- Reference
  - COMPONENT REF, ARRAY RANGE REF
- Statement
  - GIMPLE MODIFY STMT, RETURN EXPR, COND EXPR, INIT EXPR
- Type
  - BOOLEAN TYPE, INTEGER TYPE
- Unary
  - ABS EXPR, NEGATE EXPR



GIMPLE - Transformations

- Un compilateur comporte un grand ensemble de transformations de haut niveau
  - Notion de middle-end
- On peut citer quelques exemples :
  - Déroulage de boucle
  - Vectorisation
  - Factorisation de code
  - …
- Les compilateurs introduisent des options pour définir des ensembles de transformations
  - -02, -03, …
- Dans quel ordre utiliser ces transformations ?



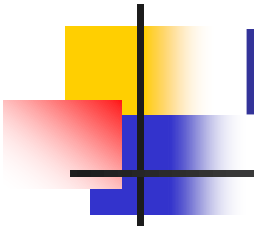
Pass Manager

- GCC utilise un *pass manager* pour enchainer les différentes transformations
- Dépendant du niveau d'optimisation
  - Ainsi que des options de compilation
- Depuis GCC 4.5
  - Souplesse du *pass manager*
  - Possibilité de créer des plugins pour ajouter une transformation
  - Détails dans le prochain cours

Pass Manager

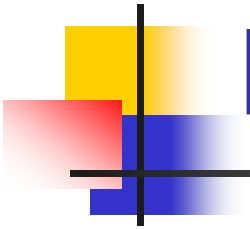
- Construction d'un arbre de transformations dans la fonction `init_optimization_passes()` dans le fichier `passes.c`
- Exemple : *lowering passes*

```
NEXT_PASS(pass_warn_unused_results)
NEXT_PASS(pass_diagnose_omp_blocks)
NEXT_PASS(pass_mudflap_1);
NEXT_PASS(pass_lower_omp);
NEXT_PASS(pass_lower_cf);
```



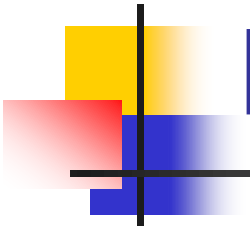
Back-end

- Rôles principaux
  - Optimisations dépendantes de l'architecture
  - Génération finale du code assembleur
- Travaille sur une représentation intermédiaire nommée RTL
  - *Register Transfer Language*
- Utilise une représentation de la machine
  - Notion de *machine description*



RTL

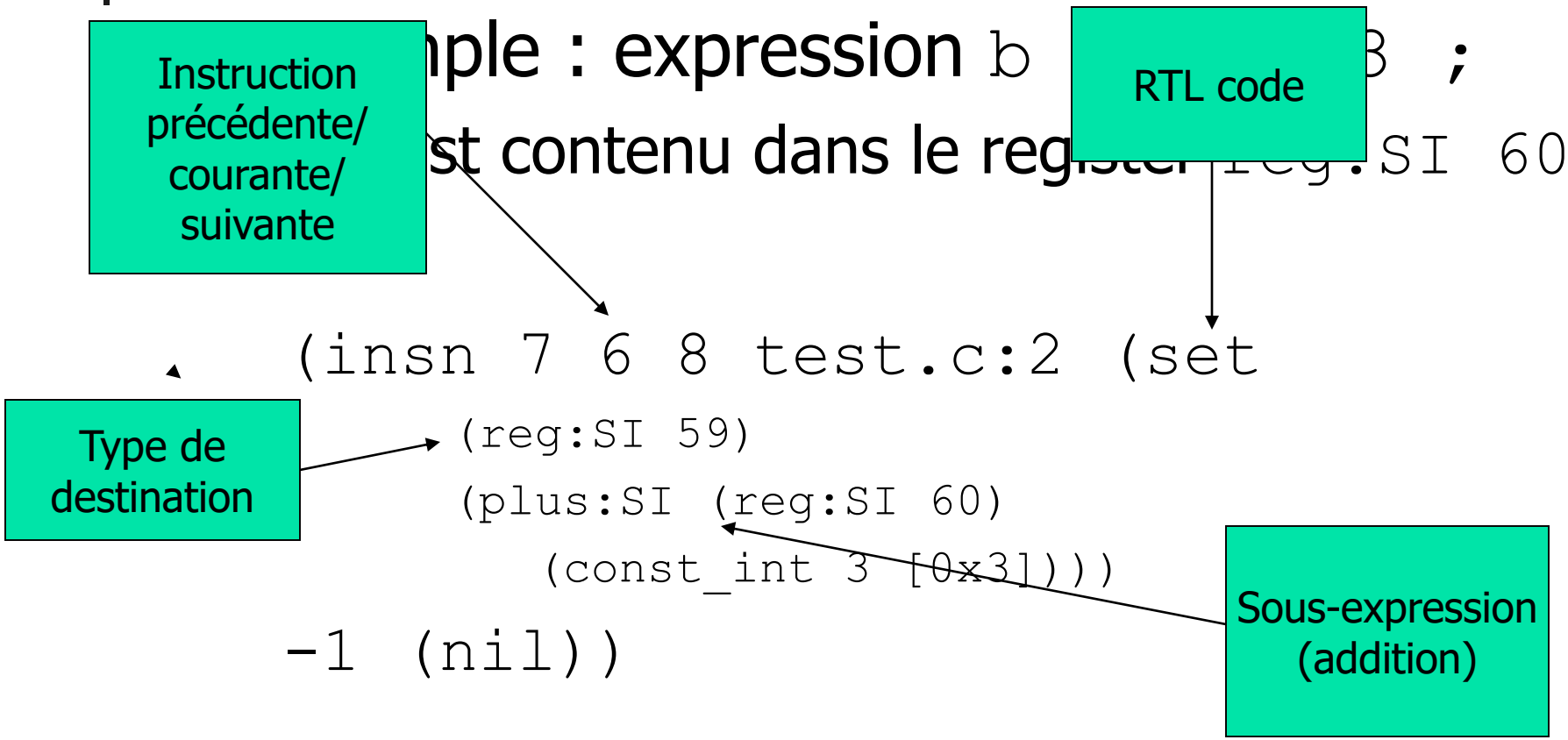
- Briques de base : object RTL
  - Expressions
  - Integers
  - Wide integers
  - Strings
  - Vectors
- Chaque expression a un code
  - La liste des codes est défini dans le fichier `rtl.def`
  - Macro pour connaître le code d'une expression : `GET_CODE(x)`

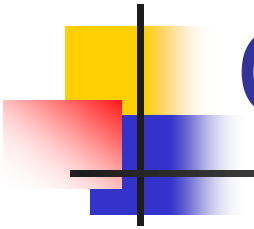


RTL

- Exemple d'affectation
  - `DEF_RTL_EXPR(SET, "set", "ee", RTX_EXTRA)`
- Deux opérandes
  1. Destination (registre, mémoire, ...)
  2. Valeur
- Macro
  1. Nom interne (majuscules par convention)
  2. Nom ASCII (minuscules par convention)
  3. Format d'affichage (documenté dans rtl.c)
     1. 'e' définit un pointer vers une expression

RTL


Instruction précédente/ courante/ suivante
RTL code
Type de destination
Sous-expression (addition)
(insn 7 6 8 test.c:2 (set
(reg:SI 59)
(plus:SI (reg:SI 60)
(const_int 3 [0x3])))
-1 (nil))



Cours 2

- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG

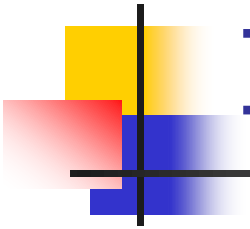
Installation de GCC

- Site web (documentation, téléchargement, …)
  - GCC : http ://gcc.gnu.org/
  - Version 12.2 actuellement
- Dépendances (bibliothèques)
  - GMP
  - MPFR
  - MPC
- Configuration
  - Création d'un sous-répertoire travail

```
./configure --prefix=chemin-vers-travail --enable-languages=c,c++ --
   enable-plugin
```

- Compilation
  - `make && make install`

Installation de GCC

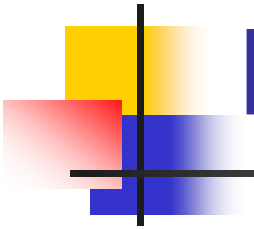


- Après l'étape `make install`
  - GCC est installé dans le répertoire donné avec l'option –prefix lors de la configuration
- Utilisation
  - Modification du PATH

```
export PATH=chemin-vers-travail/bin:$PATH
```

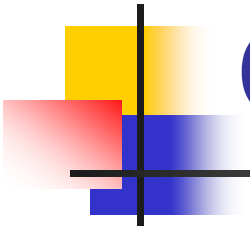
  - `gcc -v` devrait vous donner la version 11.1 et la ligne de configuration que vous avez mis
- Modification du compilateur
  - On modifie ce qu'on veut et ensuite

```
make && make install
```



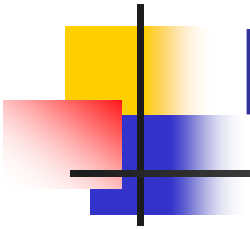
Documentation de GCC

- Documentation principale
  - Le code de GCC
- Important : il faut pouvoir lire le code de GCC pour comprendre comment cela fonctionne
  - Ne pas hésiter à parcourir les fichiers sources du cœur du compilateur
- Souvent la solution existe dans une autre partie de GCC
- Autre documentation de référence
  - The GCC internals
  - `http ://gcc.gnu.org/onlinedocs/gccint/Plugins.html#Plugins`
- Exemple du PDF…



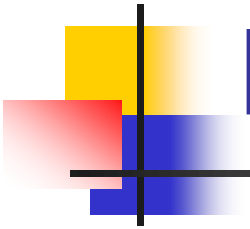
Cours 2

- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG



Modification du compilateur

- Catégories de modifications
  - Corrections de bugs
  - Ajout de fonctionnalités
- Ajout de fonctionnalités
  - Nouveau langage en entrée
  - Nouvelle architecture cible
  - Nouvelle passe (analyse/transformation/optimisation)
- Comment faire des modifications dans le compilateur GCC ?
  - Nouveau Front-end
  - Nouvelle description d'architecture (*machine description*)
  - Nouvelle passe
- Comment ajouter une nouvelle passe ?
  - Ajout direct dans le cœur du compilateur
  - Programmation d'un plugin externe



Description d'un plugin

- Plugin
  - Bout de code chargé par le compilateur au moment de la compilation d'un fichier
  - Sous forme de bibliothèque dynamique
  - Interaction avec le cœur du compilateur

- Contenu minimal d'un plugin
  - Initialisation : fonction prédéfinie qui retourne 0 si tout se passe bien
  - Licence GPL : déclaration d'une variable globale prédéfinie

```
int plugin_is_GPL_compatible ;
```

- Etapes
  1. Compilation du plugin en une bibliothèque dynamique
  2. Exécution
     - Lors de la compilation d'un fichier : renseigné l'utilisation d'un plugin
     - Possibilité de mettre des arguments

Initialisation d'un plugin

- Fonction d'initialisation du plugin
  - ```
    int plugin_init (
      struct plugin_name_args *plugin_info,
      struct plugin_gcc_version *version
    ) ;
    ```
- **Chaque plugin doit implémenter cette fonction**
- Point d'entrée
  - Fonction `main` dans le cœur du compilateur
  - Lors de la phase d'initialisation des plugins, le compilateur appelle la fonction `plugin_init` de tous les plugins (de façon séquentielle)
- Où est définit ce prototype ?
  - Dans le *header* `gcc-plugin.h`
    ```

Plugin minimal

```
#include <gcc-plugin.h>

/* Global variable required for plugin to execute */
int plugin_is_GPL_compatible;

/* Main entry point for plugin */
int
plugin_init(struct plugin_name_args * plugin_info,
                  struct plugin_gcc_version * version)
{
    printf( "plugin_init: Entering...\n" ) ;
    return 0;
}
```

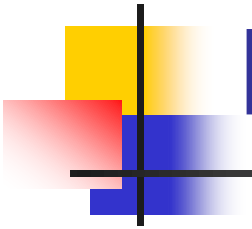
Compilation d'un plugin

- Etape 1 : compilation séparée des fichiers appartenant au plugins
  - Besoin du chemin où sont stocker les *headers* servant au plugin (comme `gcc-plugin.h`)
  - Commande : `gcc -print-file-name=plugin`
    - Donne le répertoire de base pour les fichies qui concernent les plugins
    - Besoin d'ajouter le sous-répertoire `include` pour la recherche de header (option `-I` pour le compilateur)
- Etape 2 : *link* de ces fichiers pour créer une bibliothèque dynamique
  - Utilisation de l'option `-shared`
  - Extension par convention : `.so`

Compilation d'un plugin

```
$ gcc -print-file-name=plugin
/home/patrick/ENSIIE/GCC/gcc910
_install/lib/gcc/x86_64-pc-
linux-gnu/9.1.0/plugin


$ g++ -I`gcc -print-file-
name=plugin`/include -g -Wall -fno-rtti -
shared -fPIC -o libplugin.so plugin.cpp
```



Exécution d'un plugin

- Pas d'exécution directe d'un plugin
  - Fonction `main` dans le compilateur
  - Plugin contrôlé par le compilateur
  - Le compilateur connaît un point d'entrée pour le plugin (fonction d'initialisation avec un prototype forcé)
- Option pour renseigner un plugin à utiliser lors de la compilation
  - `-fplugin=name`
  - L'argument `name` est le nom de la bibliothèque dynamique contenant le plugin (e.g., `plugin.so`)
  - Possibilité d'utiliser plusieurs plugins !

Exécution d'un plugin

```
#include <gcc-plugin.h>

/* Global variable required for plugin to execute */
int plugin_is_GPL_compatible;

/* Main entry point for plugin */
int
plugin_init(struct plugin_name_args * plugin_info,
                  struct plugin_gcc_version * version)
{
    printf( "plugin_init: Entering...\n" ) ;
    return 0;
}
```

Exécution d'un plugin

```
$ cat test.c
#include <stdio.h>

void f() {
      printf( "In f\n" ) ;
}

void g() {
      printf( "In g\n" ) ;
}

$ mpicc -c test.c -fplugin=./libplugin.so
plugin_init: Entering...
```

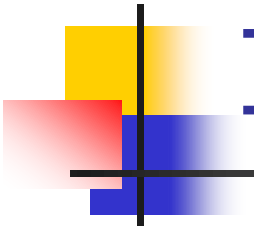
Structures d'initialisation

- Rappel : fonction d'initialisation du plugin

```
int plugin_init (
    struct plugin_name_args *plugin_info,
    struct plugin_gcc_version *version
  ) ;
```

- Deux arguments en entrée de la fonction
  - Arguments fournis par le compilateur
  - Correspond à deux pointeurs sur des structures
  - Information sur le contexte d'exécution
  - Définition dans le *header* `plugin.h`

Information sur GCC



- Second argument : informations sur le compilateur qui exécute ce plugin

- Détail de cette structure

```
struct plugin_gcc_version
{
    const char *basever;
    const char *datestamp;
    const char *devphase;
    const char *revision;
    const char *configuration_arguments;
};
```

Information sur GCC

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible;

int
plugin_init(struct plugin_name_args * plugin_info,
                struct plugin_gcc_version * version)
{
  printf( "Plugin initialization:\n" ) ;
  printf( "\tbasever = %s\n", version->basever ) ;
  printf( "\tdatestamp = %s\n", version->datestamp ) ;
  printf( "\tdevphase = %s\n", version->devphase ) ;
  printf( "\trevision = %s\n", version->revision ) ;
  printf( "\tconfig = %s\n", version->configuration_arguments ) ;
  return 0;
}
```

Informations sur GCC

```
Plugin initialization:
        basever = 9.1.0
        datestamp = 20190503
        devphase =
        revision =
        config =
/home/patrick.carribault/LOCAL/GCC/gcc-9.1.0/configure
--program-suffix=_910 --enable-languages=c,c++,fortran
--disable-bootstrap --disable-multilib --enable-plugin
--
prefix=/home/patrick.carribault/LOCAL/GCC/gcc910_instal
l --with-
gmp=/home/patrick.carribault/LOCAL/GMP/gmp612_install -
-with-
mpfr=/home/patrick.carribault/LOCAL/MPFR/mpfr401_instal
l --with-
mpc=/home/patrick.carribault/LOCAL/MPC/mpc110_install
```

Information sur le plugin

- Premier argument : informations sur le contexte d'exécution du plugin
- Détail de cette structure

```
struct plugin_name_args {
    char *base_name; /* Short name of the plugin
                        (filename without .so suffix). */
    const char *full_name; /* Path to the plugin as specified
                        with -fplugin=. */
    int argc; /* Number of arguments specified with
                        -fplugin-arg-.... */
    struct plugin_argument *argv; /* Array of ARGC
                        key-value pairs. */
    const char *version; /* Version string provided by
                        plugin. */
    const char *help; /* Help string provided by plugin. */
}
```

Information sur le plugin

- Le champ `argv` représente les arguments donnés au plugin lors de l'exécution de la compilation
  - Option : `-fplugin-arg-name-key1[=value1]`
  - Nom du plugin (sans le chemin, ni l'extension `.so`) : `name`
  - Nom de l'argument : `key1`
  - Valeur de l'argument (optionnelle) : `value1`

- Structure pour accéder aux arguments

```
struct plugin_argument {
  char *key;    /* key of the argument.  */
  char *value;  /* value is optional and
                   can be NULL.  */
};
```

Information sur le plugin

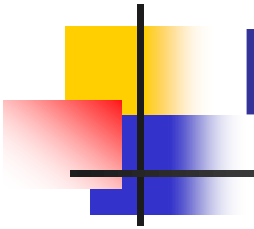
```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible ;

int plugin_init (struct plugin_name_args *plugin_info,
                 struct plugin_gcc_version *version) {
        int i ;
        printf( "Plugin initialization:\n" ) ;
        printf( "\tbase_name = %s\n", plugin_info->base_name ) ;
        printf( "\tfull_name = %s\n", plugin_info->full_name ) ;
        printf( "\targc = %d\n", plugin_info->argc ) ;
        for ( i = 0 ; i < plugin_info->argc ; i++ ) {
                printf( "\t\tArg %d: %s = %s\n", i,
                                plugin_info->argv[i].key,
                                plugin_info->argv[i].value ) ;
        }
        printf( "\tversion = %s\n", plugin_info->version ) ;
        printf( "\thelp = %s\n", plugin_info->help ) ;
        return 0 ;
}
```

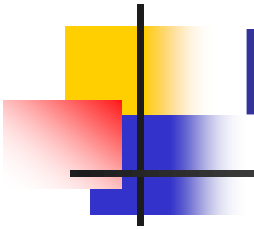
Information sur le plugin

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
Plugin initialization:
        base_name = plugin
        full_name = ./plugin.so
        argc = 0
        version = (null)
        help = (null)
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugn-mon_arg1=toto -c test.c
cc1: error: plugin plugn should be specified before -fplugin-arg-plugn-mon_arg1=toto in the command line
Plugin initialization:
        base_name = plugin
        full_name = ./plugin.so
        argc = 0
        version = (null)
        help = (null)
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugin-mon_arg1=toto -c test.c
Plugin initialization:
        base_name = plugin
        full_name = ./plugin.so
        argc = 1
                Arg 0: mon_arg1 = toto
        version = (null)
        help = (null)
carribaultp$ gcc -fplugin=./plugin.so -fplugin-arg-plugin-mon_arg1=toto -fplugin-arg-plugin-val=2 -c test.c
Plugin initialization:
        base_name = plugin
        full_name = ./plugin.so
        argc = 2
                Arg 0: mon_arg1 = toto
                Arg 1: val = 2
        version = (null)
        help = (null)
```



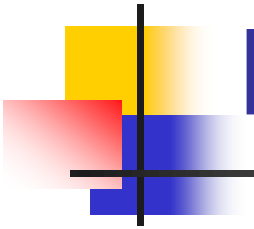
Plan du cours

- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - *Pass manager*
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG



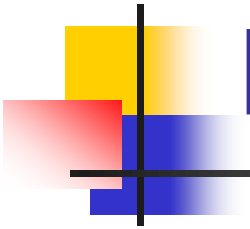
Evènement d'un plugin

- Pour le moment, le plugin s'initialise
  - Dans cette fonction, il faut donner des infos au compilateur sur le comportement de notre plugin
- Programmation évènementielle avec des *callbacks*
  - Enregistrement d'évènements à capturer par le plugin
  - Appel d'une fonction pour l'enregistrement avec un type d'évènement
  - Ajout d'un pointeur de fonction pour désigner la fonction que le compilateur doit appeler lorsque l'évènement se produit
- Liste des évènements dans le fichier `plugin.def`



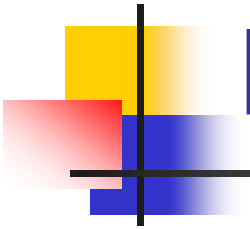
Liste exhaustive

- PLUGIN_START_PARSE_FUNCTION
- PLUGIN_FINISH_PARSE_FUNCTION
- PLUGIN_PASS_MANAGER_SETUP
- PLUGIN_FINISH_TYPE
- PLUGIN_FINISH_DECL
- PLUGIN_FINISH_UNIT
- PLUGIN_PRE_GENERICIZE
- PLUGIN_FINISH
- PLUGIN_INFO
- PLUGIN_GGC_START
- PLUGIN_GGC_MARKING
- PLUGIN_GGC_END
- PLUGIN_REGISTER_GGC_ROOTS
- PLUGIN_ATTRIBUTES
- PLUGIN_START_UNIT
- PLUGIN_PRAGMAS
- PLUGIN_ALL_PASSES_START
- PLUGIN_ALL_PASSES_END
- PLUGIN_ALL_IPA_PASSES_START
- PLUGIN_ALL_IPA_PASSES_END
- PLUGIN_OVERRIDE_GATE
- PLUGIN_PASS_EXECUTION
- PLUGIN_EARLY_GIMPLE_PASSES_START
- PLUGIN_EARLY_GIMPLE_PASSES_END
- PLUGIN_NEW_PASS
- PLUGIN_INCLUDE_FILE



Evènements intéressants

- PLUGIN_PASS_MANAGER_SETUP
  - Permet d'interagir avec le *pass manager* pour ajouter une nouvelle passe
- PLUGIN_START_UNIT
  - Utile pour initialiser des données (e.g., ouverture de fichiers) au début de la compilation d'un fichier
- PLUGIN_FINISH ou PLUGIN_FINISH_UNIT
  - Utile pour finaliser des données (e.g., fermeture de fichiers) à la fin de la compilation d'un fichier
- PLUGIN_PRAGMAS
  - Ajout de la reconnaissance d'une directive (`#pragma` en C/C++)



Enregistrement

- Fonction pour enregister un évènement :

```
void register_callback (const char *plugin_name,
                        int event,
                        plugin_callback_func callback,
                        void *user_data);
```

- Arguments
  - `plugin_name` : nom du plugin sans le chemin ni l'extension
    - Utilisation de `plugin_info->base_name` pour le plugin courant
  - `event` : évènement à enregistrer
  - `callback` : fonction appelée lorsque cet évènement apparait
  - `user_data` : données utilisateurs utiles pour le callback
- Selon les évènements,
  - `callback` peut être NULL
  - `user_data` peut être NULL
- Prototype de la fonction de call back

```
void (*plugin_callback_func) (void *gcc_data, void *user_data);
```

Exemple de *callback*

```
#include <gcc-plugin.h>

int plugin_is_GPL_compatible;

void callback_start_unit(
  void * gcc_data, void * user_data)
{
  printf( "Callback start unit\n" ) ;
}

void callback_finish_unit(
  void * gcc_data, void * user_data)
{
  printf( "Callback finish unit\n" ) ;
}

void callback_finish(
  void * gcc_data, void * user_data)
{
  printf( "Callback finish\n" ) ;
}
```

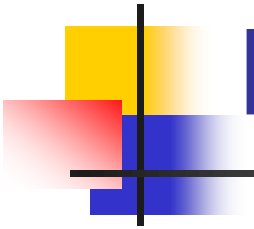
```
 int
plugin_init(struct plugin_name_args * plugin_info,
                struct plugin_gcc_version * version)
{
        register_callback(plugin_info->base_name,
                          PLUGIN_START_UNIT,
                          callback_start_unit,
                          NULL);

        register_callback(plugin_info->base_name,
                          PLUGIN_FINISH_UNIT,
                          callback_finish_unit,
                          NULL);

        register_callback(plugin_info->base_name,
                          PLUGIN_FINISH,
                          callback_finish,
                          NULL);

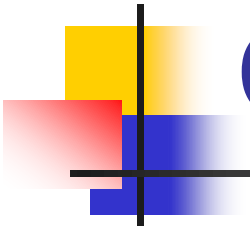
        return 0;
}
```

```
$ gcc_910 test.c -g -O3 -fplugin=./libplugin_callback1.so
Callback start unit
Callback finish unit
Callback finish
```



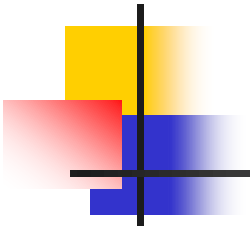
Plan du cours

- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - Pass manager
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG



Création d'une nouvelle passe

- Etapes pour ajouter une nouvelle passe
  1. Définition d'une nouvelle passe
  2. Insertion dans le *pass manager*
  3. Enregistrement de l'évènement associé pour le plugin



1 - Définition d'une passe

- Structure d'une passe
  - Définie dans `tree-pass.h`
  - Localisée dans répertoire `include` des plugins
  - `class opt_pass : public pass_data`
- Champs intéressants hérités de `pass_data` :
  - Type de passes (voir ci-après)
    `enum opt_pass_type type`
  - Nom de la passe
    `const char *name;`
- Champs intéressants ajoutés dans cet objet :
  - Fonction pour la décision d'exécution
    `virtual bool (*gate) (function *fun) ;`
  - Fonction d'exécution de la passe
    `virtual unsigned int (*execute) (function *fun) ;`

1 - Définition d'une passe

- Type de passe (différences ?)

```
enum opt_pass_type {
  GIMPLE_PASS,
  RTL_PASS,
  SIMPLE_IPA_PASS,
  IPA_PASS
};
```

- Fonction pour la décision d'exécution
  - Retourne un booléen
  - Permet d'activer la passe seulement dans un contexte particulier (niveau d'optimisation, option, …)
- Fonction d'exécution de la passe
  - Corps de la passe proprement dite
  - Cette fonction n'est appelée que si la *gate* a répondue VRAI

2 – Insertion de la passe

- Interaction avec le pass manager
  - Une fois notre passe définie, besoin de l'insérer dans le processus de compilation
  - Besoin également de donner plusieurs infos (e.g., la fréquence de décisions)

- Structure permettant de renseigner les informations sur la passe et son insertion

```
struct register_pass_info {
     opt_pass *pass; /* New pass provided by the plugin. */
     const char *reference_pass_name; /* Name of the reference pass
                                         for hooking up the new pass. */
     int ref_pass_instance_number; /* Insert the pass at the specified
                                      instance number of the reference pass. */
                                      /* Do it for every instance if it is 0. */
     enum pass_positioning_ops pos_op; /* how to insert the new pass. */
};
```

- Positionnement de la passe

```
enum pass_positioning_ops {
     PASS_POS_INSERT_AFTER, // Insert after the reference pass.
     PASS_POS_INSERT_BEFORE, // Insert before the reference pass.
     PASS_POS_REPLACE // Replace the reference pass.
};
```

3 - Enregistrement

- Utilisation de l'évènement `PLUGIN_PASS_MANAGER_SETUP`

- Appel à la fonction d'enregistrement `register_callback`
  - Fonction de `callback` → NULL
    - Fonctions nécessaires pour décider et exécuter la passe sont contenues dans l'instance de la structure `opt_pass`
  - Pointeur `user_data` → pointeur sur une structure pour renseigner les informations sur la passe (adresse sur instance de `register_pass_info`)

Exemple

```
const pass_data my_pass_data = {
    GIMPLE_PASS, /* type */
    "NEW_PASS", /* name */
    OPTGROUP_NONE, /* optinfo_flags */
    TV_OPTIMIZE, /* tv_id */
    0, /* properties_required */
    0, /* properties_provided */
    0, /* properties_destroyed */
    0, /* todo_flags_start */
    0, /* todo_flags_finish */
};

class my_pass : public gimple_opt_pass {
    public:
        my_pass (gcc::context *ctxt)
                : gimple_opt_pass (my_pass_data, ctxt)
        {}

        my_pass *clone () { return new my_pass(g); }

        bool gate (function *fun) { return true; }

        unsigned int execute (function *fun) {
            printf("Executing my_pass with function %s\n",
                function_name(fun) );
            return 0;
        }
};
```

```
int
plugin_init(
    struct plugin_name_args * plugin_info,
    struct plugin_gcc_version * version)
{
    struct register_pass_info pass_info;

    my_pass p(g);

    pass_info.pass = &p;
    pass_info.reference_pass_name =
        "omplower";
    pass_info.ref_pass_instance_number = 0;
    pass_info.pos_op = PASS_POS_INSERT_BEFORE;

    register_callback(
            plugin_info->base_name,
            PLUGIN_PASS_MANAGER_SETUP,
            NULL,
            &pass_info);
    return 0;
}
```

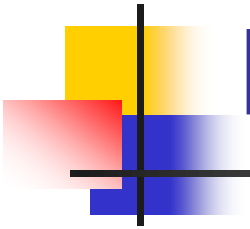
Exemple

```
$ cat test.c
#include <stdio.h>

void f() {
  printf( "In f()\n" ) ;
}

int main() {
  printf( "Hello\n" ) ;
  f() ;
  return 0 ;
}

$ gcc test.c -c  -fplugin=./libplugin.so
Executing my_pass with function f
Executing my_pass with function main
```



Pass Manager

- Nécessité de connaître l'ordre des passes exécutées par GCC !
- Besoin de regarder le code du *pass manager*
  - Dans les sources du compilateur
  - Pas disponible dans les *headers* relatifs aux plugins
- *Pass manager*
  - Sous-répertoire `gcc`, fichier `passes.c`
  - Constructeur de `pass_manager`
  - Inclusion du fichier pass-instances.def généré à la compilation à partir de passes.def
- Plusieurs types de passes
  - *Lowering*, *IPA*, *All passes*, …

Pass Manager

```
/* All passes needed to lower the function into shape optimizers can
    operate on.  These passes are always run first on the function, but
    backend might produce already lowered functions that are not processed
    by these passes.  */
  INSERT_PASSES_AFTER (all_lowering_passes)
  NEXT_PASS (pass_warn_unused_result);
  NEXT_PASS (pass_diagnose_omp_blocks);
  NEXT_PASS (pass_diagnose_tm_blocks);
  NEXT_PASS (pass_lower_omp);
  NEXT_PASS (pass_lower_cf);
  NEXT_PASS (pass_lower_tm);
  NEXT_PASS (pass_refactor_eh);
  NEXT_PASS (pass_lower_eh);
  NEXT_PASS (pass_build_cfg);
  NEXT_PASS (pass_warn_function_return);
  NEXT_PASS (pass_expand_omp);
  NEXT_PASS (pass_sprintf_length, false);
  NEXT_PASS (pass_walloca, /*strict_mode_p=*/true);
  NEXT_PASS (pass_build_cgraph_edges);
  TERMINATE_PASS_LIST (all_lowering_passes)
```

Notre plugin a été inséré une passe à cet endroit

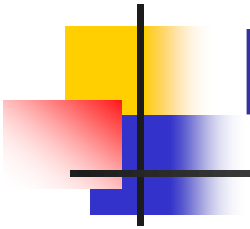
Nom des passes

- Une fois la position trouvée, il manque une information
  - Nom de la passe de référence
- Comment trouver ce nom ?
  - Pas de solution immédiate simple
  - Besoin de regarder la structure qui définit cette passe
  - Si vous trouvez une meilleure solution…
- Exemple : fichier `omp-low.c`

```
const pass_data pass_data_lower_omp =
{
  GIMPLE_PASS, /* type */
  "omplower", /* name */
  OPTGROUP_OMP, /* optinfo_flags */
  TV_NONE, /* tv_id */
  PROP_gimple_any, /* properties_required */
  PROP_gimple_lomp | PROP_gimple_lomp_dev, /*
properties_provided */
  0, /* properties_destroyed */
  0, /* todo_flags_start */
  0, /* todo_flags_finish */
};

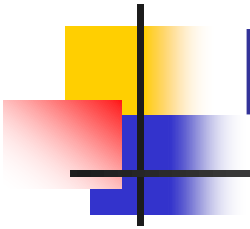
class pass_lower_omp : public gimple_opt_pass
{
public:
  pass_lower_omp (gcc::context *ctxt)
    : gimple_opt_pass (pass_data_lower_omp,
ctxt) {}

  virtual unsigned int execute (function *) {
return execute_lower_omp (); }
}; // class pass_lower_omp
```



Plan du cours

- Présentation de GCC
  - Introduction
  - Structure générale
  - Installation
- Modification du compilateur
  - Plugin
  - Evénement
  - Pass manager
- Manipulation du code
  - Structures GIMPLE
  - Structures CFG



Manipulation du code

- Focalisation sur les passes en GIMPLE
  - Code source du fichier à compiler représenté en GIMPLE
  - GIMPLE est notre représentation intermédiaire
- Gestion du code en GIMPLE
  - Deux étapes : avant et après la création du graphe de flot de contrôle (CFG)
- Avant la création du CFG :
  - Tout peut être fait grâce à un traitement itératif en GIMPLE (récursif)
  - Documentation : `gimple.def gimple.h gimple.c`
- Après la création du CFG :
  - Accès au code à travers le graphe de flot de contrôle

Fichier source en GIMPLE

```c
#include <stdio.h>

int f( int a, int * m ) {
        int i ;
        int b = a ;
        if ( a ) {
                b++ ;
        }
        else {
                for ( i = 0 ; i < a ; i++ ) {
                        b += m[i] ;
                }
        }
        return b ;
}
```

```
f (int a, int * m)
gimple_bind <
  unsigned int i.0;
  unsigned int D.1823;
  int * D.1824;
  int D.1825;
  int D.1826;
  int i;
  int b;

  gimple_assign <parm_decl, b, a, NULL>
  gimple_cond <ne_expr, a, 0, <D.1819>, <D.1820>>
  gimple_label <<D.1819>>
  gimple_assign <plus_expr, b, b, 1>
  gimple_goto <<D.1821>>
  gimple_label <<D.1820>>
  gimple_assign <integer_cst, i, 0, NULL>
  gimple_goto <<D.1816>>
  gimple_label <<D.1815>>
  gimple_assign <nop_expr, i.0, i, NULL>
  gimple_assign <mult_expr, D.1823, i.0, 4>
  gimple_assign <pointer_plus_expr, D.1824, m, D.1823>
  gimple_assign <mem_ref, D.1825, *D.1824, NULL>
  gimple_assign <plus_expr, b, D.1825, b>
  gimple_assign <plus_expr, i, i, 1>
  gimple_label <<D.1816>>
  gimple_cond <lt_expr, i, a, <D.1815>, <D.1817>>
  gimple_label <<D.1817>>
  gimple_label <<D.1821>>
  gimple_assign <var_decl, D.1826, b, NULL>
  gimple_return <D.1826>
>
```

Fichier source avec le CFG

```
;; Function f (f, funcdef_no=0, decl_uid=1811, cgraph_uid=0)

f (int a, int * m)
{
  int b;
  int i;
  int D.1826;
  int D.1825;
  int * D.1824;
  unsigned int D.1823;
  unsigned int i.0;

<bb 2>:
  gimple_assign <parm_decl, b, a, NULL>
  gimple_cond <ne_expr, a, 0, NULL, NULL>
    goto <bb 3>;
  else
    goto <bb 4>;

<bb 3>:
  gimple_assign <plus_expr, b, b, 1>
  goto <bb 7>;

<bb 4>:
  gimple_assign <integer_cst, i, 0, NULL>
  goto <bb 6>;
```

```
<bb 5>:
  gimple_assign <nop_expr, i.0, i, NULL>
  gimple_assign <mult_expr, D.1823, i.0, 4>
  gimple_assign <pointer_plus_expr, D.1824, m, D.1823>
  gimple_assign <mem_ref, D.1825, *D.1824, NULL>
  gimple_assign <plus_expr, b, D.1825, b>
  gimple_assign <plus_expr, i, i, 1>

<bb 6>:
  gimple_cond <lt_expr, i, a, NULL, NULL>
    goto <bb 5>;
  else
    goto <bb 7>;

<bb 7>:
  gimple_assign <var_decl, D.1826, b, NULL>

gimple_label <<L6>>
  gimple_return <D.1826>

}
```

Exemple de passe GIMPLE

- Première passe de *lowering* : `warn_unused_result`
  - Affiche un warning lorsque le retour d'un appel de fonction n'est pas capturé et que cette fonction a un attribut warn_unused_result
- Fonction
  - `do_warn_unused_result (gimple_body (current_function_decl));`
- `current_function_decl`
  - Pointeur vers la racine de la déclaration de la fonction courante
  - Tout le corps de la fonction (ainsi que les arguments, le retour et les variables locales) est accessible à partir de ce pointeur
- Accès à la représentation gimple de la fonction
  - `gimple_seq gimple_body(tree t)`

```
const pass_data pass_data_warn_unused_result =
{
  GIMPLE_PASS, /* type */
  "*warn_unused_result", /* name */
  OPTGROUP_NONE, /* optinfo_flags */
  TV_NONE, /* tv_id */
  PROP_gimple_any, /* properties_required */
  0, /* properties_provided */
  0, /* properties_destroyed */
  0, /* todo_flags_start */
  0, /* todo_flags_finish */
};

class pass_warn_unused_result : public gimple_opt_pass
{
public:
  pass_warn_unused_result (gcc::context *ctxt)
    : gimple_opt_pass (pass_data_warn_unused_result, ctxt)
  {}
virtual bool gate (function *) { return
flag_warn_unused_result; }
  virtual unsigned int execute (function *)
    {
      do_warn_unused_result (gimple_body
(current_function_decl));
      return 0;
    }
}; // class pass_warn_unused_result
```

Exemple de passe GIMPLE

Boucle d'itération sur les statements

Itérateur de *statement*

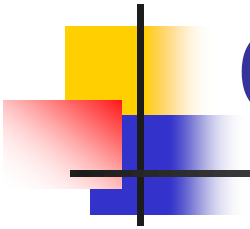
Accès à la représentation GIMPLE du statement

Accès au type d'instruction

Type de nœud possédant un ensemble de statements

Appel récursif

```
static void
do_warn_unused_result (gimple_seq seq)
{
  tree fdecl, ftype;
  gimple_stmt_iterator i;

  for (i = gsi_start (seq); !gsi_end_p (i); gsi_next (&i))
    {
      gimple g = gsi_stmt (i);

      switch (gimple_code (g))
        {
        case GIMPLE_BIND:
          do_warn_unused_result (gimple_bind_body (g));
          break;
        case GIMPLE_TRY:
          do_warn_unused_result (gimple_try_eval (g));
          do_warn_unused_result (gimple_try_cleanup (g));
          break;
        case GIMPLE_CATCH:
          do_warn_unused_result (gimple_catch_handler (g));
          break;
        case GIMPLE_EH_FILTER:
          do_warn_unused_result (gimple_eh_filter_failure (g));
          break;

        case GIMPLE_CALL:
          if (gimple_call_lhs (g))
            break;
          if (gimple_call_internal_p (g))
            break;
```



Construction du CFG

- Une fois le graphe de flot de contrôle (CFG) construit
  - Passage par la structure du CFG pour la manipulation du code
  - Ensemble de nœuds et d'arcs
- Passe qui construit le CFG
  - Nom dans le *pass manager* : `pass_build_cfg`
  - Nom de la passe : `cfg`
- Structures utilisées pour le CFG
  - `struct control_flow_graph`
  - CFG de la fonction courante :
    `cfun->cfg`

Structure principale du CFG

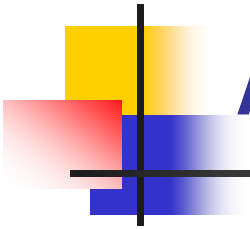
```
struct GTY(()) control_flow_graph {
  /* Block pointers for the exit and entry of a function.
     These are always the head and tail of the basic block list.  */
  basic_block x_entry_block_ptr;
  basic_block x_exit_block_ptr;

  /* Index by basic block number, get basic block struct info.  */
  VEC(basic_block,gc) *x_basic_block_info;

  /* Number of basic blocks in this flow graph.  */
  int x_n_basic_blocks;

  /* Number of edges in this flow graph.  */
  int x_n_edges;

  /* The first free basic block number.  */
  int x_last_basic_block;

  /* UIDs for LABEL_DECLs.  */
  int last_label_uid;

  /* Mapping of labels to their associated blocks.  At present
     only used for the gimple CFG.  */
  VEC(basic_block,gc) *x_label_to_block_map;

  enum profile_status_d x_profile_status;

  /* Whether the dominators and the postdominators are available.  */
  enum dom_state x_dom_computed[2];

  /* Number of basic blocks in the dominance tree.  */
  unsigned x_n_bbs_in_dom_tree[2];

  /* Maximal number of entities in the single jumptable.  Used to estimate
     final flowgraph size.  */
  int max_jumptable_ents;
};
```

Noeuds du CFG

- Basic block

  ```
  typedef struct basic_block_def *basic_block;
  ```

- Fichiers concernées :
  - `coretypes.h`, `basic-block.h`
- Champs
  - Vecteurs d'arc (edge) entrant et sortant : `preds`, `succs`
  - Double liste chaînée : `prev_bb`, `next_bb`
  - Indice dans le vecteur des BBs : `index`
  - Ensemble de flags…
- Notion de vecteurs : cf. `vec.h` pour plus d'infos…
- Deux BB spéciaux (source et puits)
  - `ENTRY_BLOCK_PTR`
  - `EXIT_BLOCK_PTR`



Arcs du CFG

- Edge

```
typedef struct edge_def *edge;
```

- Fichiers concernées :
  - `coretypes.h`, `basic-block.h`
- Champs
  - Source de l'arc : src
  - Destination de l'arc : `dest`
  - Indice dans le vecteur de destination : `dest_idx`
  - Ensemble de flags…

Parcours du CFG

- Parcours du corps de la fonction à travers le CFG
  - Possibilité d'itérer sur les nœuds
  - Ensuite, sur les arcs
- Plusieurs solutions pour le parcours des nœuds
  - Tous les bloc de base sauf source et puits (peu importe leur ordre)

```
basic_block bb;
FOR_EACH_BB (bb) { /* … */ }
```

  - Tous les bloc de base (peu importe leur ordre)

```
basic_block bb;
FOR_ALL_BB (bb) { /* … */ }
```

  - Commencer au premier BB

```
basick_block bb = ENTRY_BLOCK_PTR ;
```

Parcours du CFG

- Exemple simple du parcours du CFG
  - Itération sur tous les BBs (sauf source et puits)
  - Ordre non défini

```
gimple_stmt_iterator gsi;
gimple stmt;

FOR_EACH_BB (b)
{
    for (gsi = gsi_start_bb (b); !gsi_end_p (gsi); gsi_next
       (&gsi))
    {
        stmt = gsi_stmt (gsi);
        /* … */
    }
}
```

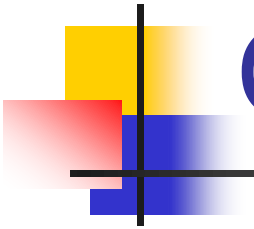
Exemple

```
unsigned int execute_my_pass (void) {
  basic_block bb ;
  gimple_stmt_iterator gsi;
  gimple stmt;

  printf( "Function %s w/ %d BB(s)\n",
      get_name (current_function_decl),
      n_basic_blocks ) ;

  FOR_EACH_BB (bb)
  {
    printf( "BB #%d\n", bb->index ) ;
    for (gsi = gsi_start_bb (bb); !gsi_end_p (gsi); gsi_next (&gsi)) {
      printf( "\tStatement\n" ) ;
      stmt = gsi_stmt (gsi);
      debug_gimple_stmt( stmt ) ;
    }
  }
  return 0 ;
}
```

Exemple

```
carribaultp$ cat test.c
#include <stdio.h>

int f( int a ) {
        int b = a ;
        if ( a ) {
                printf( "A is not 0\n" ) ;
                {
                        b = a + 1 ;
                }
        }
        return b ;
}
```

```
carribaultp$ gcc -fplugin=./plugin.so -c test.c
Function f w/ 6 BB(s)
BB #2
        Statement
b = a;

        Statement
if (a != 0)

BB #3
        Statement
__builtin_puts (&"A is not 0"[0]);

        Statement
b = a + 1;

BB #4
        Statement
D.1816 = b;

BB #5
        Statement
<L2>:

        Statement
return D.1816;
```



Conclusion

- Modification du compilateur possible grâce à un plugin
  - Avantage : en dehors des sources du cœur du compilateur
  - Inconvénient : modifications limitées
- Documentation
  - Manuel *internals* et transparents disponibles
  - Documentation la plus efficace : code source
- Représentation intermédiaire
  - Attention au type de représentation utilisée (Arbre GIMPLE, CFG, les deux, …)
  - Certaines données ne sont construites que plus tard (par exemple boucles)