Auteurs : BOUTADGHART, CHAHINE, ELGHISSASSI, NOUKAM, NOURELLIL, RJIMATI, SAAD

Guide du Développeur

1. Architecture générale

Le compilateur est organisé en plusieurs modules spécialisés qui interagissent pour former une chaîne complète de compilation C vers assembleur (x86 ou ARM). Voici un aperçu des principaux composants du projet :

Liste des modules principaux

- **`main.cpp`**
  Point d'entrée du compilateur.
  Il gère la lecture du fichier source C, configure l'environnement de compilation (dont le backend ARM via l'option `--arm`), puis lance successivement :
  - L'analyse syntaxique avec ANTLR,
  - La construction de la table des symboles (`SymbolTableVisitor`),
  - La génération du graphe de contrôle (`CFG`) et de l'IR (`IRGenerator`),
  - Enfin, la génération du code assembleur.

- **`SymbolTableVisitor`**
  Implémente un visiteur ANTLR chargé de construire les tables de symboles pour chaque fonction.
  Il vérifie la validité des déclarations (existence, unicité, portée) et remplit une structure `Function` contenant les symboles, le type de retour, et d'autres métadonnées comme l'offset mémoire.

- **`IRGenerator`**
  C'est le second visiteur. Il parcourt l'arbre syntaxique pour transformer les instructions du C en une représentation intermédiaire (IR), sous forme d'instructions élémentaires.
  Chaque instruction IR est ajoutée à un bloc de base (`BasicBlock`), qui sera ensuite géré par le `CFG`.

- **`IRInstr`**, **`BasicBlock`**, **`CFG`**
  Ensemble des modules de génération intermédiaire :

  - `IRInstr` : représente une instruction IR unitaire (comme un `add`, `cmp`, `call`, etc.).
  - `BasicBlock` : séquence d'instructions IR sans branchement interne. Peut avoir un ou deux blocs successeurs.
  - `CFG` : structure principale d'une fonction. Organise les `BasicBlock`, les tables de symboles associées, les types, et génère le code assembleur final pour l'architecture cible.

2. Pipeline de compilation

Le compilateur suit un pipeline en plusieurs étapes, allant de l'analyse lexicale à la génération d'un code assembleur exécutable. Chaque étape est assurée par un ou plusieurs modules dédiés, facilitant la maintenance, l'extension et le débogage du projet.

Étape 1 : Parsing avec ANTLR (Analyse syntaxique)

Le fichier source `.c` est d'abord analysé avec **ANTLR4**, un générateur de parseurs à partir de grammaires.

- Le fichier `ifcc.g4` définit la grammaire du sous-ensemble C supporté.
- ANTLR génère automatiquement un parseur (`ifccParser`, `ifccLexer`, `ifccBaseVisitor`) utilisé dans `main.cpp`.
- Le parseur transforme le code source en un arbre de syntaxe abstraite (AST), qui capture la structure logique du programme (fonctions, blocs, instructions, expressions...).

Résultat : un AST prêt à être parcouru par les visiteurs.

Étape 2 : Construction de la table des symboles (`SymbolTableVisitor`)

Une fois l'AST obtenu, il est parcouru par le **`SymbolTableVisitor`**.

- Ce visiteur gère la **déclaration des variables**, la **portée** (avec gestion hiérarchique via des symbol tables imbriquées), et vérifie les erreurs statiques simples :
  - variable utilisée avant déclaration,
  - variable déclarée plusieurs fois dans un même scope,
  - variable déclarée mais jamais utilisée

Chaque fonction est associée à :

- un vecteur de symbol tables (une par bloc de code),
- un type de retour,
- un offset de pile pour les variables locales.

Résultat : une représentation sémantique valide du programme, prête à être traduite.

Étape 3 : Génération de l'IR (`IRGenerator`)

L'**IRGenerator** est un second visiteur qui convertit les instructions du C en une **représentation intermédiaire** (IR) manipulable et plus proche du langage machine.

- Chaque instruction est traduite en une ou plusieurs `IRInstr`.
- Ces instructions sont regroupées dans des `BasicBlock`, eux-mêmes rattachés à un `CFG` (Control Flow Graph).
- Les blocs permettent de gérer les structures conditionnelles (if, while, etc.) et la génération propre de branches conditionnelles.

Exemples d'instructions IR : `add`, `mul`, `cmp_eq`, `jump`, `ret`, etc.

Résultat : une version linéarisée et bas-niveau du programme, encore indépendante de l'architecture cible.

Étape 4 : Organisation en blocs et graphe de flot (`CFG`, `BasicBlock`)

Le `CFG` (Control Flow Graph) est responsable de :

- Structurer les **blocs de base** (`BasicBlock`) en un graphe dirigé,
- Gérer les entrées et sorties de chaque fonction,
- Maintenir les liaisons entre les blocs (`exit_true`, `exit_false` pour les branches),
- Associer correctement les variables à leurs offsets (via les symbol tables).

Le `CFG` assure aussi la gestion de la pile (`stack_allocation`) et l'ordonnancement correct de l'exécution.

Résultat : une structure claire et modulaire, prête pour la génération de code assembleur.

Étape 5 : Génération de l'assembleur (`gen_asm`, `gen_asm_arm`)

Enfin, le `CFG` appelle ses méthodes `gen_asm()` ou `gen_asm_arm()` pour produire le code assembleur final.

- Le backend peut cibler l'architecture **x86** (par défaut),ou **ARM** si l'option `--arm` est fournie.
- Chaque instruction IR est traduite en instruction assembleur (avec le bon suffixe selon le type : `l`, `b`, `sd`, etc.).
- Les blocs sont traduits dans l'ordre, en respectant le graphe de contrôle.

Des fonctions utilitaires gèrent les conventions d'appel (ABI), les appels à `putchar`, `getchar`, etc.

Résultat : un code assembleur valide, qui peut être compilé par `gcc` pour produire un exécutable.

3. Détail des fichiers principaux

Cette section détaille les fichiers C++ principaux du compilateur. Pour chacun, on précise son rôle, ses responsabilités, les structures de données impliquées, ainsi que les fonctions essentielles. L'objectif est de permettre à un nouveau développeur de s'orienter rapidement dans le code pour le modifier ou l'étendre.

`main.cpp`

Rôle : Point d'entrée de l'exécutable `ifcc`, il orchestre l'ensemble du processus de compilation.

Fonctionnement :

- Lit le fichier source passé en paramètre.
- Configure le mode ARM si l'option `--arm` est activée.
- Utilise ANTLR pour analyser le code source et produire un arbre de syntaxe (AST).
- Applique un premier visiteur `SymbolTableVisitor` pour construire les tables de symboles.
- Applique ensuite le visiteur `IRGenerator` pour générer l'IR.
- Crée une instance de `CFG`, y intègre les blocs de base (BasicBlock), et lance la génération de code assembleur.
- Affiche l'assembleur sur la sortie standard.

Spécificité : Gère aussi bien les erreurs de parsing que les erreurs sémantiques détectées lors de la visite de l'AST.

`SymbolTableVisitor.h` / `SymbolTableVisitor.cpp`

Rôle : Ce visiteur est responsable de la construction des symbol tables pour chaque fonction ou bloc de code.

Principales responsabilités :

- Lors de la visite des nœuds de déclaration, affectation ou fonction, il enregistre les variables avec leur nom, type et offset mémoire.
- Gère l'imbrication des blocs via des pointeurs vers les tables parentes, permettant le support du scope lexical (avec shadowing).
- Vérifie la validité des déclarations (pas de redéclaration, existence avant usage).
- Vérifie que chaque fonction a bien une instruction `return`.
- Identifie les variables déclarées mais non utilisées (pour émettre des avertissements).

Structures utilisées :

- `SymbolTable` : structure principale d'une portée.
- `Function` (struct locale) : regroupe le type de retour d'une fonction, son vecteur de symbol tables, les variables utilisées et l'offset courant sur la pile.

`SymbolTable.h`

Rôle : Implémente une structure hiérarchique de tables de symboles.

Fonctionnement :

Contient une map associant des noms de variables à des objets `Symbol` (nom, type, offset). Supporte l'insertion de nouvelles variables via `insert()` et permet la recherche dans le scope courant avec `contains()` et la recherche récursive dans les scopes parents avec `get()`.

Spécificité : La classe est conçue pour supporter des blocs imbriqués grâce au pointeur `parent` permet d'implémenter une gestion correcte des portées en C, y compris le shadowing.

`Type.h`

Rôle : Définit les types de données supportés par le compilateur (int, char, double, void).

Fonctionnement :

Énumération `Type` pour les types de base. Contient la fonction `get_type(std::string)` pour convertir une chaîne (ex. `"int"`) en type et la fonction `getTypeSize(Type)` pour retourner la taille mémoire d'un type (ex. : `int` → 4 octets).

Utilisation :

Dans la table des symboles (pour affecter un type à chaque variable). Dans les IRInstr (pour choisir le suffixe assembleur correct). Dans la gestion de la pile mémoire (savoir combien d'octets allouer).

`IRGenerator.h` / `IRGenerator.cpp`

Rôle : Visiteur ANTLR chargé de transformer l'AST en instructions intermédiaires (IR).

Fonctionnement :

Chaque instruction ou expression du langage est traduite en une ou plusieurs instructions IR (`IRInstr`). Les instructions sont organisées dans des blocs de base (`BasicBlock`), eux-mêmes gérés par un `CFG`. Il gère les structures de contrôle (if, while), les appels de fonction, les opérateurs arithmétiques, logiques, bit à bit et les comparaisons et utilise un champ `lastExprType` pour propager le type d'une expression au fur et à mesure de la visite.

Spécificité : Le code IR généré est indépendant de l'architecture cible, ce qui permet une génération backend spécifique par la suite.

`IRInstr.h` / `IRInstr.cpp`

Rôle : Définit les instructions de la représentation intermédiaire (IR).

Fonctionnement :

Une instruction IR est définie par une opération (`Operation`), un type (`Type`) et une liste de paramètres (`vector<string>`).
Les méthodes `gen_asm()` et `gen_asm_arm()` permettent de traduire l'instruction IR en assembleur x86 ou ARM.

Les opérations couvrent :

- Le calcul (`add`, `sub`, `mul`, `div`, `mod`)
- Le transfert (`ldconst`, `copy`)
- Les comparaisons (`cmp_eq`, `cmp_lt`, etc.)
- Les branchements (`jump`, `cond_jump`)
- Les appels de fonctions (`call`, `ret`)

- Les conversions de types (`int_to_double`, `double_to_int`)

Spécificité : L'instruction IR est fortement typée. Le type influence directement la syntaxe assembleur (suffixes comme `l`, `b`, `sd`).

`BasicBlock.h` / `BasicBlock.cpp`

Rôle : Un bloc de base contient une séquence linéaire d'instructions IR sans branchement intermédiaire.

Fonctionnement :

Un basicblock Contient une liste ordonnée d'objets `IRInstr`. Les méthodes d'ajout d'instruction : `add_IRInstr()` (fin), `add_IRInstrAtTop()` (début) et les méthodes `gen_asm()` et `gen_asm_arm()` génèrent l'assembleur du bloc courant.. Les champs `exit_true`, `exit_false`, `exit` pointent vers les blocs successeurs (pour if, while, etc.).

Spécificité : Chaque bloc possède un label unique pour le marquage des points d'entrée lors des branchements. Il est utilisé par le `CFG` pour organiser le flot de contrôle complet d'une fonction.

`CFG.h` / `CFG.cpp`

Rôle : Gère l'ensemble des blocs de base d'une fonction et pilote la génération de code assembleur.

Fonctionnement :

Il reçoit l'AST et la table des symboles en entrée. Gère un vecteur de `BasicBlock`, dont l'un est le bloc courant (`current_bb`) et produit le code assembleur à l'aide de :

- `gen_asm_prologue()` : empile les registres, réserve de la mémoire.
- `gen_asm_epilogue()` : restaure la pile et retourne à l'appelant.
- `gen_asm()` : génère l'assembleur complet de la fonction.

Autres responsabilités :

- Traduction des noms de variables en offset mémoire via `get_var_index()`.
- Prise en charge des deux architectures (x86 ou ARM) via un booléen `is_arm`.
- Stockage des constantes flottantes (`double_constants`) pour les architectures qui nécessitent des sections de données.

Spécificité : Le `CFG` agit comme une interface entre la logique IR et le backend d'assembleur. Il centralise l'ensemble des décisions liées au contexte de compilation (taille de la pile, architecture, etc.).

4. Fonctionnement du CFG et IR

La représentation intermédiaire (IR) et le graphe de contrôle de flot (CFG) sont au cœur de l'architecture du compilateur. Ils permettent de séparer proprement l'analyse du code source (front-end) de la génération de code assembleur (back-end). Cette séparation facilite la modularité, le support multi-architecture et les futures optimisations.

Représentation intermédiaire (IR)

L'IR constitue un langage de plus bas niveau que le C, mais encore indépendant du processeur cible. Il est composé d'instructions simples, de type trois-adresses, regroupées dans des blocs de base.

Structure d'une instruction IR

Chaque instruction IR (`IRInstr`) est définie par :

- Une opération (`Operation`) : exemple `add`, `cmp_eq`, `call`, `ret`.
- Un type (`Type`) : `int`, `double`, etc., utilisé pour générer l'assembleur adapté.
- Une liste de paramètres (`std::vector<std::string>`) : ce sont des noms de variables temporaires ou offsets mémoire.

Les instructions sont créées dans le visiteur `IRGenerator`, au fur et à mesure de la visite des nœuds de l'AST.

Elles sont ajoutées à un `BasicBlock` via les méthodes : `add_IRInstr(op, type, params)` : ajoute une instruction à la fin du bloc et `add_IRInstrAtTop(...)` : insère au début (utile pour les prologues).

Chaque instruction est ensuite traduite en assembleur via les méthodes : `gen_asm(std::ostream&)` pour x86 et `gen_asm_arm(std::ostream&)` pour ARM

BasicBlock : unité de code linéaire

Un bloc de base (`BasicBlock`) est une suite d'instructions IR sans branchements internes. Les blocs sont liés entre eux par des branchements explicites. Chaque bloc possède :

- Un `label` unique pour le marquage des sauts.
- Une liste d'instructions `instrs`.
- Des pointeurs vers les blocs successeurs : `exit_true` et `exit_false` pour les branches conditionnelles (ex : if), `exit` pour les branches inconditionnelles (ex : fin de bloc sans condition).
- Un champ `test_var_name`, qui contient la variable utilisée pour les conditions de branchement.

Le `BasicBlock` joue un rôle fondamental pour les structures de contrôle comme `if`, `else`, et `while`, où plusieurs chemins d'exécution sont possibles.

CFG : Control Flow Graph

Le `CFG` (classe `CFG`) gère l'ensemble des blocs de base d'une fonction. Il contient :

- Un vecteur `bbs` de tous les `BasicBlock` créés pour la fonction.
- Un pointeur `current_bb` vers le bloc de base en cours de construction.
- Une map `functions` qui stocke les informations symboliques de toutes les fonctions.
- Les indices `currentST_index` et `last_ST_index` pour accéder à la bonne symbol table selon la profondeur du bloc.

Ajout de blocs

Le CFG permet d'ajouter dynamiquement des blocs avec `add_bb(BasicBlock*)`, par exemple pour :

- un nouveau corps de `if`,
- le bloc qui suit un `else`
- le corps d'une boucle `while`.

Génération d'assembleur

Une fois tous les blocs construits et remplis avec les instructions IR, le `CFG` génère le code assembleur final.

Les méthodes utilisées sont :

- `gen_asm(std::ostream&)` : point d'entrée principal pour générer l'ensemble des blocs.
- `gen_asm_prologue(std::ostream&)` : insère le prologue de fonction (sauvegarde des registres, allocation de la pile).
- `gen_asm_epilogue(std::ostream&)` : insère l'épilogue (restauration de la pile, retour).

Architecture multi-cible

Le champ `is_arm` dans `CFG` permet de générer du code spécifique pour ARM plutôt que pour x86 : `IRInstr::gen_asm()` génère du code x86 avec la syntaxe AT&T et `IRInstr::gen_asm_arm()` utilise la syntaxe ARM.

L'IR étant commun aux deux architectures, il suffit de basculer le back-end en modifiant `is_arm`.

Points importants

- Les instructions IR sont indépendantes de l'ordre des opérations dans le code source : elles sont générées dans un ordre linéaire optimisé pour l'exécution. Le découpage en blocs permet une modularité très forte : chaque bloc peut être traité indépendamment lors de l'optimisation ou de la génération d'assembleur.
- Le `CFG` peut être enrichi à l'avenir pour supporter : des optimisations (ex : propagation de constantes), l'analyse statique (ex : data-flow), d'autres architectures cibles (ex : bytecode Java, MSP430).

5. Gestion ARM vs x86

Le compilateur a été conçu dès le départ pour pouvoir **générer du code assembleur pour plusieurs architectures**. Le reciblage s'appuie sur une stratégie à deux couches :

1. Une **représentation intermédiaire indépendante de l'architecture (IR)**
2. Un **back-end spécialisé** qui traduit les instructions IR en code assembleur spécifique à l'architecture choisie.

Le choix entre x86 et ARM se fait dynamiquement à l'exécution via l'option `--arm`.

5.1 Sélection de l'architecture : le mode `--arm`

Dans `main.cpp`, l'argument de ligne de commande `--arm` est analysé :

```
bool use_arm = false;

// ...

if (string(argv[i]) == "--arm") {
    use_arm = true;
}
```

Ce booléen est ensuite passé au `CFG` :

```
cfg.is_arm = use_arm;
```

Le champ `is_arm` conditionne les appels à `gen_asm()` ou `gen_asm_arm()` dans les blocs IR.

5.2 Traduction conditionnelle des instructions IR

Chaque instruction IR (`IRInstr`) dispose de deux méthodes de génération :

- `void gen_asm(std::ostream &o)` : pour x86 (Intel AT&T syntaxe, utilisée avec GCC).
- `void gen_asm_arm(std::ostream &o)` : pour ARM (syntaxe adaptée à ARMv7 ou ARM64).

La fonction appelante dans `CFG` choisit dynamiquement laquelle utiliser en fonction de `is_arm`.

Exemple :

```
if (cfg->is_arm) {
    instr->gen_asm_arm(output);
} else {
    instr->gen_asm(output);
}
```

5.3 Fonctions de génération spécifiques à ARM

`IRInstr::gen_asm_arm(std::ostream&)`

Chaque type d'instruction IR (addition, saut, retour, appel de fonction...) possède une implémentation ARM spécifique. Elle respecte la convention ARM pour le nommage des registres (`r0` à `r12`, `sp`, `lr`, `pc`), et le passage de paramètres via `r0` à `r3`.

Par exemple :

- Les arguments de fonction sont placés dans les registres `r0` à `r3`.
- Le retour est effectué dans `r0`.
- Les variables locales sont accédées via des offsets relatifs à `sp` (stack pointer), par exemple `[sp, #8]`.

Particularité ARM :

Contrairement au x86 où les offsets sont négatifs par rapport à `%rbp`, ARM utilise des **offsets positifs** depuis le registre `sp`. Il faut donc **adapter la syntaxe et le signe des offsets** lors de la génération.

5.4 Fonctions utilitaires dans le CFG

Certaines fonctions du `CFG` aident à convertir les variables en adresses assembleur, selon l'architecture.

`get_var_index(name)`

Cette fonction récupère l'offset mémoire d'une variable depuis la table des symboles. Elle est utilisée dans la génération IR, indépendamment de l'architecture.

`IR_reg_to_asm(string param)` ***(commentée dans le code)***

Cette fonction convertit un nom de variable ou un offset en adresse mémoire assembleur. Elle adapte la syntaxe selon la cible : Pour x86 : `"8(%rbp)"` et pour ARM : `"[sp, #8]"`. Même si cette fonction est actuellement commentée dans le code, elle montre bien le principe d'abstraction entre IR et assembleur.

5.5 Génération du prologue et de l'épilogue

Dans les fonctions `gen_asm_prologue()` et `gen_asm_epilogue()`, on insère manuellement les instructions de sauvegarde des registres, l'allocation/désallocation de la mémoire pour la pile (`stack_allocation`), et le respect de l'**ABI** de l'architecture cible.

Par exemple, sur x86 :

```
pushq %rbp
movq %rsp, %rbp
subq $offset, %rsp
```

Sur ARM (équivalent possible) :

```
push {fp, lr}
mov fp, sp
sub sp, sp, #offset
```

Chaque instruction IR n'a pas nécessairement un équivalent direct entre x86 et ARM, donc des ajustements sont faits dans chaque méthode `gen_asm` selon les contraintes de l'architecture.

5.6 Fonctions liées au passage de paramètres

Deux instructions IR sont utilisées pour les appels de fonctions : `assign_param` : prépare les arguments avant l'appel et `call` : effectue l'appel de fonction.
Sur x86 : les six premiers arguments sont passés via registres (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`) selon l'ABI System V. Au-delà, les arguments sont passés sur la pile (non encore géré dans ce projet).

Sur ARM : les quatre premiers arguments sont passés via registres (`r0` à `r3`). Là aussi, les suivants sont passés sur la pile.

5.7 Conclusion

Le compilateur repose sur un **IR indépendant de l'architecture**, ce qui permet une génération de code spécifique pour différentes cibles sans modifier les étapes de parsing et de construction sémantique.

Grâce à l'abstraction apportée par le `CFG` et les méthodes spécifiques `gen_asm`/`gen_asm_arm`, il est facile d'ajouter d'autres cibles à l'avenir (comme MSP430, Java bytecode, etc.), à condition de respecter la convention d'appel et la syntaxe propre à l'architecture.

6. Ajout d'une fonctionnalité

Le compilateur est conçu de manière modulaire, ce qui permet d'ajouter de nouvelles fonctionnalités en intervenant à des points précis de la chaîne de compilation : la grammaire ANTLR, le visiteur sémantique, le générateur IR, et les instructions IR elles-mêmes.

Cette section décrit les étapes nécessaires pour ajouter une **nouvelle opération : l'opérateur `&&` (ET logique paresseux)**, qui évalue le deuxième opérande seulement si le premier est vrai. Ce cas illustre bien l'ensemble du processus de modification.

Étape 1 : Modifier la grammaire (`ifcc.g4`)

Il faut d'abord étendre la grammaire pour reconnaître l'opérateur `&&`. Cela se fait en ajoutant une règle `LogicalAndExpr` dans la règle `expr`.

Exemple :

```
Expr
     : expr '&&' expr                 # LogicalAndExpr
     | ...                           // les autres alternatives
```

Cela permet à ANTLR de produire un nœud `LogicalAndExprContext` lorsqu'un `&&` est rencontré.

Étape 2 : Étendre le générateur d'IR (IRGenerator)

Une fois que la grammaire reconnaît le nouveau nœud, on doit ajouter une méthode dans `IRGenerator` pour le traiter. Ajouter dans la classe `IRGenerator` (dans `IRGenerator.h`) :

```
Virtual antlrcpp::Any
visitLogicalAndExpr(ifccParser::LogicalAndExprContext* ctx) override;
```

Et implémenter la méthode correspondante dans `IRGenerator.cpp` :

```
antlrcpp::Any
IRGenerator::visitLogicalAndExpr(ifccParser::LogicalAndExprContext*
ctx) {

    // Étape 1 : créer les blocs nécessaire
BasicBlock* bb_condition = cfg->current_bb;
BasicBlock* bb_second = new BasicBlock(cfg, "logical_and_second");
BasicBlock* bb_merge = new BasicBlock(cfg, "logical_and_merge");

cfg->add_bb(bb_second);
cfg->add_bb(bb_merge);

// Étape 2 : évaluer la première expression

string left = visit(ctx->expr(0));
string tmp_result = cfg->new_temp();

// Générer saut conditionnel : si false → fin (merge)

bb_condition->exit_true = bb_second;
bb_condition->exit_false = bb_merge;
bb_condition->test_var_name = left;

// Étape 3 : évaluer la seconde expression si nécessaire

cfg->current_bb = bb_second;
string right = visit(ctx->expr(1));
bb_second->add_IRInstr(IRInstr::copy, INT, {tmp_result, right});
bb_second->exit = bb_merge;

// Étape 4 : créer le bloc de fusion
```

```
cfg->current_bb = bb_merge;
     return tmp_result;
}
```

Ce code met en œuvre l'évaluation paresseuse : le deuxième opérande n'est évalué que si le premier est vrai.

Étape 3 : Ajouter une opération IR si nécessaire

Dans ce cas précis, on n'ajoute pas une instruction IR `logical_and` propre, car l'opérateur `&&` est directement modélisé par des branchements (`jump, cond_jump`) et une évaluation conditionnelle. Cependant, dans d'autres cas (comme `^` ou `<<`), une instruction IR dédiée pourrait être requise.

Pour une extension future, on pourrait créer une opération `IRInstr::logical_and` dans `IRInstr::Operation` et la traiter dans `IRInstr::gen_asm` et `gen_asm_arm`.

Étape 4 : Générer l'assembleur

La gestion de l'opérateur `&&` repose principalement sur des blocs de branchement, ce qui signifie que le travail est déjà accompli en grande partie via les `BasicBlock`. Il suffit que le bloc `merge` soit généré normalement, et le compilateur se chargera de produire le bon code assembleur.

Si on souhaite rendre le traitement plus explicite, il est possible d'ajouter une variable temporaire qui contiendra `0` ou `1` selon le résultat de l'opération.

Étape 5 : Écrire des tests

Créer un fichier de test `logical_and.c` dans le dossier `testfiles` avec un programme comme :

```
int main() {
   int a = 1;
   int b = 0;
    return a && b;  // Devrait renvoyer 0
}
```

Puis un autre avec :

```
int main() {
   int a = 1;
   int b = 1;
   return a && b;  // Devrait renvoyer 1
}
```

Ces tests peuvent être ajoutés à l'infrastructure existante et comparés au résultat de GCC.

Étape 6 : Vérifications

- L'opérateur `&&` est bien reconnu par la grammaire ?
- Le visiteur `IRGenerator` est appelé correctement ?
- Le comportement paresseux est respecté (le second opérande n'est pas évalué si le premier est faux) ?
- Le résultat final est correct, et l'assembleur produit est valide sur les deux architectures (x86 et ARM si activé) ?

Cas d'autres fonctionnalités

Le processus est similaire pour les extensions suivantes :

- Opérateurs binaires : `||`, `^`, `<<`, etc.
- Instructions : `break`, `continue`, `for`
- Types : `double`, `char`, tableaux
- Structures : `switch`, `struct`, etc.

Chaque fois, les étapes sont :

1. Étendre la grammaire,
2. Ajouter un `visitXXX` dans `IRGenerator`,
3. Créer ou réutiliser des instructions IR,
4. Gérer leur traduction en assembleur,
5. Tester le tout avec rigueur.

7. Tests

Le projet de compilateur adopte une approche de développement dirigée par les tests (TDD – Test Driven Development). Cela signifie que chaque nouvelle fonctionnalité ou modification du code doit idéalement être accompagnée d'un ou plusieurs tests, avant même sa mise en œuvre complète.

L'infrastructure de test fournie permet de :

- comparer le comportement du compilateur `ifcc` avec celui de `gcc`,
- détecter les régressions lors des évolutions du code,
- valider les fonctionnalités supportées et identifier celles qui ne le sont pas encore.

7.1 Organisation des tests

Les tests sont regroupés dans des répertoires structurés, par défaut dans le dossier `testfiles`. Une bonne pratique est de les classer selon leur statut fonctionnel :

- `TestsOK/` : contient des programmes corrects qui doivent être compilés avec succès et produire le bon résultat.
- `NotImplementedYet/` : programmes valides en C, mais dont certaines constructions ne sont pas encore supportées par `ifcc`.
- `KnownBugs/` : tests qui exposent un comportement incorrect connu, servant de référence pour de futures corrections.
- `TestsInvalid/` : programmes contenant des erreurs (syntaxe, sémantique) et qui doivent être refusés par `ifcc`.

Chaque fichier de test est un programme C valide ou volontairement incorrect, selon l'objectif du test.

7.2 Lancement des tests

Les tests sont automatisés à l'aide d'un script Python fourni dans le projet : `ifcc-test.py`.

Nous avons mis tous nos tests dans des dossiers `/testfiles_[Categorie_test]`. Exécuter `make test` ou `make test_arm` depuis le dossier `/compiler` permet d'exécuter tous ces tests les uns après les autres.

Commande de base :

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

Ce script parcourt récursivement les fichiers contenus dans `testfiles/`, compile chaque fichier : avec `gcc` pour obtenir le comportement de référence, avec `ifcc` suivi de `gcc` pour assembler le fichier `.s` généré.

Il exécute ensuite les deux programmes et compare la **valeur de retour** (valeur retournée par `main`) à la **sortie standard** (par exemple, les caractères affichés avec `putchar`).

Options utiles :

```
python3 ifcc-test.py --help
```

Permet d'afficher les options disponibles : filtrer certains dossiers, afficher uniquement les échecs, tester un fichier précis, etc.

7.3 Ajout de nouveaux tests

L'ajout d'un test se fait simplement en créant un nouveau fichier `.c` dans l'un des dossiers de tests.

Exemple :

```
// testfiles_addition/addition_simple.c
int main() {
    return 3 + 4;
}
```

On peut ensuite exécuter les tests pour vérifier que ce nouveau fichier est pris en compte. Pour tester une nouvelle fonctionnalité (comme un nouvel opérateur ou une structure de contrôle), on recommande d'écrire au moins :

- un test de fonctionnement correct
- un test avec une erreur (sémantique ou syntaxique),
- un test avec des cas limites (par exemple : division par zéro, grandes valeurs, etc.).

7.4 Comportement attendu en cas d'erreur

Le test est considéré comme **réussi** dans deux cas : `ifcc` compile et exécute un programme, avec les **mêmes résultats que** `gcc` ou `ifcc` refuse de compiler un programme invalide **que** `gcc` **refuse également**.

Le test est **échoué** si : `ifcc` plante ou segfault à la compilation ou `ifcc` compile, mais produit un programme au comportement incorrect, si `ifcc` accepte un code invalide, ou rejette un code valide.

7.5 Tests négatifs et couverture

Il est important d'écrire des **tests négatifs** pour s'assurer que les erreurs sont bien détectées (ex : variable non déclarée) et pour vérifier la robustesse du compilateur en cas d'entrées incorrectes.

En fin de projet, une **bonne couverture de tests** permet de s'assurer que toutes les constructions du langage sont gérées et que chaque fonctionnalité fonctionne sur un ensemble représentatif de cas.

7.6 Intégration dans le développement

Voici quelques recommandations pour intégrer les tests dans votre cycle de développement :

- Lorsqu'une fonctionnalité est prévue : **écrire les tests en premier**.
- Avant chaque sprint : **valider tous les tests** pour détecter d'éventuelles régressions.
- Lors de la correction d'un bug : **ajouter un test de non-régression** correspondant au problème corrigé.
- Avant le rendu : **lancer tous les tests et corriger les cas échoués ou les classer explicitement** (par exemple dans `KnownBugs`).

8. TODOs et perspectives

Le compilateur `ifcc` mis en place constitue déjà une base solide capable de :

- parser un sous-ensemble du langage C,
- effectuer une vérification sémantique,
- générer une représentation intermédiaire structurée (IR + CFG),

- produire du code assembleur pour x86 ou ARM.

Cependant, plusieurs éléments restent à implémenter, améliorer ou corriger. Cette section regroupe ces points en quatre catégories : améliorations fonctionnelles, optimisations, bugs connus, et commentaires `TODO` dans le code.

8.1 Améliorations fonctionnelles prévues

Certaines fonctionnalités du langage C ne sont pas encore supportées, ou le sont partiellement. En voici une liste non exhaustive, ordonnée de manière indicative :

- **Support complet des types `double`**

Ajout du support dans les fonctions et gestion de la précision à l'arrondi. Génération d'instructions assembleur compatibles avec les FPU ou instructions SIMD (selon architecture).

Opérateurs logiques complets : `||` (OU logique paresseux) et `!` (négation logique)

- **Opérateurs d'affectation combinés :** `+=`, `-=`, `*=`, `/=`, etc.
- **Incrémentation et décrémentation :** `++` et `--`, en préfixe et suffixe.
- **Boucles supplémentaires :** `for`, `do...while`
- **Structures conditionnelles avancées :** `switch...case`
- **Support des tableaux :** Déclaration, affectation, et accès aux éléments (`a[i]`).
- **Fonctions à plusieurs paramètres (au-delà de 6) :** Passage par la pile, conformément à l'ABI.
- **Structures de données utilisateur :** `struct`, `union`, et éventuellement `enum`.
- **Préprocesseur :** Traitement minimal des directives comme `#include`, `#define`.
- **Appels de fonctions récursives complexes ! :** Gestion correcte de la pile pour des appels profonds et multiples fonctions récursives.

8.2 Optimisations envisageables

Certaines optimisations peuvent être ajoutées pour améliorer les performances des programmes générés :

- **Propagation de constantes dans les expressions :** Détection de constantes dans les arbres d'expressions et simplification au moment de la génération IR.

- **Propagation de variables constantes (data-flow) :** Remplacement d'une variable par sa valeur connue si elle est constante dans un chemin donné.
- **Élimination de code mort :** Suppression des blocs de code inaccessibles ou des instructions redondantes.
- **Fusion de blocs de base :** Réduction du nombre de blocs en fusionnant ceux qui n'ont qu'un seul successeur ou prédécesseur.
- **Analyse de portée des registres temporaires :** Réutilisation intelligente des noms de temporaires pour réduire la consommation de la pile ou des registres.

8.3 Bugs connus

Certains comportements ne sont pas encore gérés correctement. Voici une liste indicative à maintenir à jour :

- Les appels de fonctions avec plus de six arguments ne sont pas encore correctement empilés.
- L'évaluation des expressions logiques (`&&`, `||`) n'est que partiellement paresseuse ou absente.
- Certaines instructions IR peuvent produire du code assembleur incorrect si des conversions de types ne sont pas explicites.
- En ARM, la gestion des variables de type `double`, de `getchar/putchar`,ainsi que des fonctions, est absente.
- Le compilateur ne génère pas encore d'erreurs si une fonction `main` est absente ou mal typée.

Chaque bug corrigé devrait idéalement être associé à un test de non-régression.

8.4 Perspectives d'évolution

Enfin, plusieurs directions sont possibles pour enrichir ce projet, notamment dans le cadre d'un projet de recherche, d'un approfondissement ou d'un prolongement pédagogique :

- **Ajout d'un backend Java bytecode ou LLVM IR** pour explorer la compilation vers des machines virtuelles.
- **Support de la compilation croisée** sur des cartes embarquées (Raspberry Pi, MSP430, etc.).
- **Interface graphique de visualisation de l'AST et du CFG** pour l'analyse statique ou le debug pédagogique.
- **Intégration dans une chaîne CI/CD avec exécution automatique des tests** après chaque modification du code.

- **Traduction inverse : désassemblage et reconstruction partielle du C à partir de l'IR ou de l'ASM généré.**