

Guide de l'utilisateur

1. Présentation	2
2. Prérequis	2
3. Utilisation de base	2
4. Architecture cible	3
5. Compilation manuelle	3
6. Fonctionnalités supportées	4
7. Erreurs détectées	5
8. Exemples de programmes	5
9. Questions fréquentes	6
Feuille de route	6

1. Présentation

IFCC est un compilateur qui traduit un sous-ensemble du langage C en code assembleur x86-64 ou ARM.

2. Prérequis

Systemes d'exploitation supportés

- Linux
- macOS
- Windows (avec limitations WSL)

Dépendances requises

- ANTLR4 (inclus dans le projet)
- Compilateur C++ (g++ ou clang++)
- Make
- Java (pour exécuter ANTLR)

Installation du compilateur

```
# Cloner le dépôt
git clone https://github.com/saadelg12/PLD_COMP.git
cd PLD_COMP

# Compiler le compilateur
cd src/compiler
make
```

3. Utilisation de base

Le compilateur IFCC produit du code assembleur en sortie standard (stdout) :

```
# Syntaxe de base
./ifcc chemin/vers/fichier.c

# Rediriger la sortie vers un fichier
./ifcc chemin/vers/fichier.c > fichier.s

# Compiler et exécuter
./ifcc fichier.c > fichier.s
gcc -o executable fichier.s
```

```
./executable  
echo $? # Affiche la valeur de retour
```

Guide pour lancer les tests :

- Lancer tous les tests en une fois : depuis le dossier /compiler, lancer la commande `make test` ou `make test_arm`

Attention : certains tests attendent une entrée de l'utilisateur !

- Lancer les tests sur un dossier entier : Vous pouvez exécuter tous les tests présents dans un dossier en spécifiant simplement le chemin du dossier.

Exemple : `python3 ifcc-test.py testfiles_assignment/`

- Lancer les tests sur un fichier spécifique : Si vous souhaitez exécuter les tests uniquement sur un fichier précis, fournissez le chemin du fichier.

Exemple : `python3 ifcc-test.py testfiles_assignment/return_bizzare.c`

- Option `--verbose` : Ajoutez l'option `--verbose` pour obtenir des informations détaillées sur les erreurs rencontrées lors de l'exécution des tests.

Exemple avec un dossier :

```
python3 ifcc-test.py --verbose testfiles_assignment/
```

Exemple avec un fichier :

```
python3 ifcc-test.py --verbose testfiles_assignment/return_bizzare.c
```

4. Architecture cible

Par défaut, IFCC génère du code assembleur pour l'architecture x86-64. Pour cibler ARM :

```
./ifcc --arm chemin/vers/fichier.c > fichier_arm.s
```

5. Compilation manuelle

Une fois le code assembleur généré, vous pouvez le compiler avec GCC :

```
# Pour x86-64  
gcc -o programme fichier.s  
./programme
```

```
# Pour ARM (sur une machine ARM)
```

```
gcc -o programme fichier_arm.s  
./programme
```

6. Fonctionnalités supportées

Types de données

- ☒ int
- ☒ double (entiers 32 bits) **(non supporté en ARM)**
- ☒ char (Support partiel de ASCII (int a = 'A'))
- ☒ float, etc.

Opérateurs

- ☒ Arithmétiques : +, -, *, /, %
- ☒ Comparaison : ==, !=, <, >, <=, >=
- ☒ Bit à bit : |, &, ^
- ☒ Unaires : - (négation), ! (non logique)
- ☒ Logiques : &&, ||
- ☒ Incrémentation : ++, --
- ☒ Composés : +=, -=, etc.

Structures de contrôle

- ☒ Blocs de code avec {}
- ☒ Instructions return
- ☒ Structures conditionnelles if/else
- ☒ Boucles while, for
- ☒ switch/case

Fonctions **(non supportées en ARM)**

- ☒ Fonction main() (sans paramètres)
- ☒ Définition de fonctions personnalisées
- ☒ Appels de fonctions avec paramètres

E/S

- ☒ getchar **(non supporté en ARM)**
- ☒ putchar






Autres

- ☒ Déclarations de variables
- ☒ Initialisation à la déclaration
- ☒ Tableaux
- ☒ Pointeurs

-  Structures et unions

7. Erreurs détectées

Le compilateur IFCC effectue plusieurs vérifications pour détecter les erreurs :

-  Variables utilisées non déclarées
-  Double déclaration de variables
-  Absence de return dans la fonction main
-  Variables déclarées mais non utilisées (avertissement)
-  Vérification des appels de fonctions et leurs paramètres

Messages d'erreur courants :

- error: syntax error during parsing : Erreur de syntaxe
- error: undeclared variable X : Variable X non déclarée
- error: variable X already declared : Variable X déjà déclarée
- warning: main function does not contain a return statement : Pas de return dans main

8. Exemples de programmes

Exemple 1 : Calcul de factorielle

```
int main() {
    int n = 5;
    int result = 1;

    while (n > 0) {
        result = result * n;
        n = n - 1;
    }

    return result; // Retourne 120
}
```

Exemple 2 : PGCD (Plus Grand Commun Diviseur)

```
int main() {
    int a = 56;
    int b = 98;
    int temp;

    while (b != 0) {
        temp = b;
        b = a % b;
    }
}
```

```
        a = temp;
    }

    return a; // Retourne 14
}
```

Exemple 3 : Expression avec opérateurs bit à bit

```
int main() {
    int a = 12;
    int b = 10;
    int c = a & b;
    int d = a | b;
    int e = a ^ b;

    return c + d + e; // Retourne 28
}
```

9. Questions fréquentes

Pourquoi `3 = x;` ne fonctionne-t-il pas ?

En C, l'opérateur d'affectation = attend une variable à gauche (lvalue) et une expression à droite. `3 = x;` n'est pas valide car 3 est une constante et ne peut pas recevoir de valeur.

Pourquoi `return;` sans valeur plante ?

La fonction main doit retourner une valeur entière selon la norme C. `return;` sans valeur n'est pas valide pour main. Utilisez toujours `return 0;` ou une autre valeur entière.

Comment compiler pour ARM sur une machine x86 ?

Le compilateur peut générer du code ARM avec l'option `--arm`, mais pour l'exécuter sur une machine x86, vous aurez besoin d'un émulateur comme QEMU ou d'une chaîne de compilation croisée.

Comment compiler plusieurs fichiers ?

Le compilateur IFCC ne prend actuellement en charge que la compilation d'un seul fichier à la fois. Le support pour plusieurs fichiers n'est pas encore implémenté.

Feuille de route

Le compilateur IFCC a été développé en méthode AGILE avec les échéances suivantes :

- Sprint 1 (26 mars 2025) : Support des structures conditionnelles, boucles et fonctions.
- Sprint 2 (2 avril 2025) : Support des tableaux, pointeurs, et types supplémentaires.

- Sprint 3 (9 avril 2025) : Support des structures avancées et optimisations.

Vous pouvez consulter la présentation du compilateur ([README/presentation.pdf](#)) pour plus de détails concernant la gestion de projet.

Pour toute question ou problème, merci de contacter l'équipe de développement PLD_COMP de l'hexanome H4233 à l'INSA Lyon (4IF).