PLD-Comp
Développement d'une chaîne complète de compilation

L'objectif de ce projet est la conception d'un compilateur pour un sous-ensemble du langage C. Le compilateur sera écrit en C++ et utilisera ANTLR4. Il faudra mettre en place une gestion de projet agile avec des sprints courts.

1 Le langage à compiler

L'objectif final est le support d'un sous-ensemble du langage C dont les fonctionnalités sont détaillées ci-dessous, ainsi que les fonctionnalités directement liées au compilateur.

	Fonctionnalité	Type	Priorité	Explications
	Un seul fichier source sans pré-processing. Les directives du pré-processeur sont autorisées par la grammaire, mais ignorées, ce afin de garantir que la compilation par un autre compilateur soit possible (exemple : inclusion de `stdio.h`)	L	DI	4.14.2 Vid Vid Vid
	Les commentaires sont ignorés	L	DI	
	Types de données de base `char` et `int` (ce dernier étant un type 32 bits)	L	O	4.8
Expressions	Variables	L	O	4.3, poly I
Expressions	Constantes entières et caractère (avec simple quote)	L	O	4.4, poly III.2, Vid
Expressions	Opérations arithmétiques de base : +,−, ⋆	L	O	
Expressions	Opérations logiques bit-à-bit : |, &, ^	L	O	
Expressions	Opérations de comparaison : ==, !=, <, >	L	O	
Expressions	Opérations unaires : ! et −	L	O	
	Déclaration de variables n'importe où	L	O	
	Affectation (qui, en C, retourne aussi une valeur)	L	O	poly III.3
	Possibilité d'initialiser une variable lors de sa déclaration	L	O	
	Utilisation des fonctions standard `putchar` et `getchar` pour les entrées-sorties	L	O	
	Définition de fonctions avec paramètres, et type de retour `int`, `char` ou `void`	L	O	4.9 4.17 A, poly II
	Vérification de la cohérence des appels de fonctions et leurs paramètres	C	O	4.10
	Structure de blocs grâce à { et }	L	O	
	Support des portées de variables et du *shadowing*	L	O	
	Les structures de contrôle `if`, `else`, `while`	L	O	4.11 4.13, poly III-5
	Support du `return` *expression* n'importe où	L	O	
	Vérification qu'une variable utilisée dans une expression a été déclarée	C	O	4.5
	Vérification qu'une variable n'est pas déclarée plusieurs fois	C	O	4.5

	Vérification qu'une variable déclarée est utilisée au moins une fois	C	O	4.5
	Division et modulo	L	F	
	Opérateurs d'affectation `+=`, `-=` *etc.*, d'incrémentation `++` et décrémentation `--`	L	F	
	Tableaux (à une dimension)	L	F	4.16
	Pointeurs	L	F	
	`break` et `continue`	L	F	
	Les chaînes de caractères représentées par des tableaux de `char`	L	F	
	`switch...case`	L	F	
	Les opérateurs logiques paresseux `||`, `&&`	L	F	
	Reciblage vers plusieurs architectures : x86, MSP430, ARM	C	F	4.7 poly I-III
	Propagation de constantes simple	C	F	4.6
	Propagation de variables constantes (avec analyse du *data-flow*)	C	F	4.14 poly IV
	Les autres types de base comme les types de `inttypes.h`, les `float`, `double`	L	NP	
	Le support dans les moindres détails de tous les autres opérateurs arithmétiques et logiques : <=, >=, << et >> *etc.*	L	NP	
	Les autres structures de contrôle : `for`, `do...while`	L	NP	
	Les variables globales	L	NP	
	La possibilité de séparer dans des fichiers distincts les déclarations et les définitions	L	D	
	Le support du préprocesseur (`#define`, `#include`, `#if`, *etc.*)	L	D	
	Les structures et unions	L	D	

Le type L correspond à des éléments de support du langage, tandis que le type C correspond à une fonctionnalité du compilateur.

Les priorités sont les suivantes :

DI (Déjà Implémenté) Ces fonctionnalités sont déjà implémentées dans le code de base distribué, vous n'avez donc pas à les faire, mais il est indispensable de bien les comprendre

O (Obligatoire) Ce sont des objectifs prioritaires du projet, ne pas les implémenter serait dommage

F (Facultatif) Éléments optionnels mais faisables et pleins d'enseignements, ils rapportent des points

NP (Non Prioritaire) Éléments optionnels qui n'apprennent pas grand chose de plus, ne vous privez pas de les implémenter mais ils n'apporteront pas beaucoup de points

D (Déconseillé) Éléments trop durs, ou longs et sans intérêt à implémenter : on n'encourage pas à essayer

Exemples de programmes que votre compilateur final devrait pouvoir traiter :

```c
#include <stdio.h>

int alphabet(int n)
{
  char a;
  a='A';
  while (a<'A'+n)
  {
    putchar(a);
    a=a+1;
  }
  return a;
}

int main()
{
  int c;
  c = alphabet(15);
  return c;
}
```

```c
int fibo(int n)
{
  if ( n <= 0 )
  {
    return 0;
  }
  else if (n == 1)
  {
    return 1;
  }

  return fibo(n-1)
         + fibo(n-2);
}

int main()
{
  return 2*fibo(8);
}
```

```c
void print_int(int x)
{
  if(x<0)
  {
    putchar('-');
    x = -x;
  }
  if(x/10 != 0)
    print_int(x/10);

  putchar(x%10 + '0');
}

int main() {
  print_int(-273);
  putchar(10);// newline
}
```

2 Architecture globale du compilateur

Votre compilateur se présentera sous la forme d'un outil en ligne de commande dont l'argument principal est le nom d'un fichier contenant un programme source.

L'outil va analyser ce programme source, et afficher si nécessaire des diagnostics d'erreurs (lexicales, syntaxiques ou sémantiques simples). Chaque erreur devra faire apparaître le numéro de ligne et de manière optionnelle le numéro de colonne dans le fichier source. Pour les programmes syntaxiquement corrects, l'outil devra générer le code assembleur x86 dans un fichier séparé, et l'assembler pour produire un exécutable.

Un compilateur minimal se compose d'un anlayseur syntaxique *(parser)* et d'un générateur de code (ou *back-end*) (Figure 1)

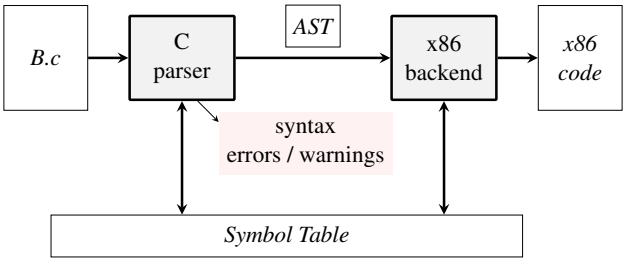


FIGURE 1 – Le compilateur attendu à mi-parcours

L'analyseur syntaxique (ou *parser*) est chargé de parcourir l'arbre syntaxique du programme. Il en abstrait certaines constructions syntaxiques, comme les espaces ou même les parenthèses, pour ne garder que l'essence du programme : un arbre de syntaxe abstrait ou AST. Il faut noter qu'ANTLR ne construit pas un AST, mais construit un visiteur qui le parcourt, ce qui est équivalent.

Lors de ce parcours, une table des symboles est également construite : elle contient essentiellement la déclaration de chaque variable avec son type, et éventuellement le numéro de ligne de cette déclaration (utile à

des messages d'erreurs futurs), etc.

Puis un générateur de code parcourt cet AST (encore une fois ce parcours est implicite dans ANTLR) pour produire du code machine, ici x86.

Au delà de ce minimum fonctionnel, nous attendons un compilateur qui se rapproche d'un vrai compilateur optimisant et reciblable.

La Figure 2 donne l'architecture d'un compilateur optimiseur reciblable, dans lequel presque chacune des fonctionnalités optionnelles en pointillé a été implémentée par plusieurs groupes dans les années précédentes.

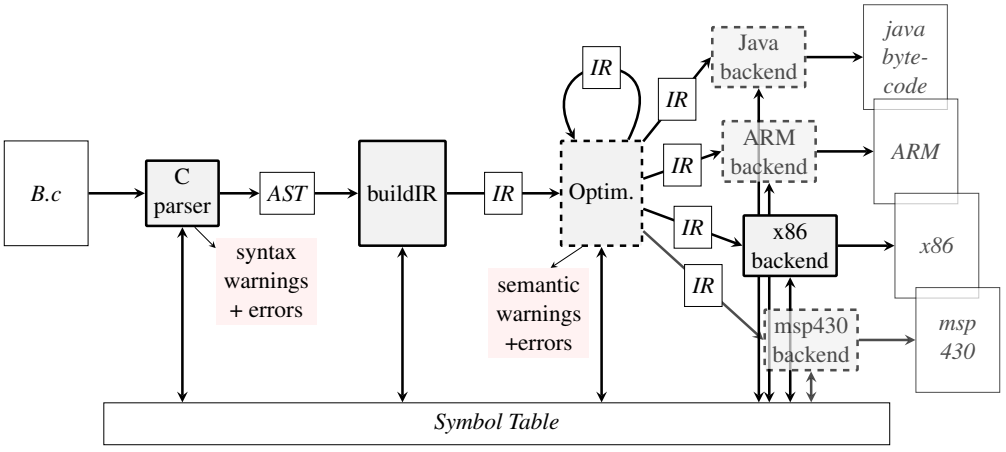


FIGURE 2 – Le compilateur final (dans nos rêves)

Les termes utilisés sur cette figure seront introduits en temps utile.

3 Gestion et déroulé du projet

Afin de simplifier le démarrage du projet, nous mettons à votre disposition sur Moodle un ensemble de fichiers qui serviront de point de départ. Vous pouvez bien entendu adapter et améliorer cet environnement.

3.1 Gestion de configuration

Vous ferez en sorte que votre compilateur compile et tourne sur les machines du département – c'est le cas du squelette fourni, dont le Makefile est configuré pour ces machines. Les README inclus devraient vous permettre d'adapter ce squelette à vos machines préférées, n'hésitez pas à demander de l'aide. Il faut avoir installé antlr4, ainsi que son runtime C++, et il faut (en général) indiquer au Makefile où ils se trouvent.

3.2 Cycles agiles

Le projet se déroulera sous forme itérative, avec une séquence d'étapes dont un exemple est détaillé dans la section 4. Chaque étape produira un compilateur fonctionnel de bout en bout (c'est-à-dire de l'analyseur syntaxique à la génération de code), mais sur un langage restreint bien défini au début de l'étape.

La section 4 a pour but de vous aider à mettre en place ces étapes, mais elle reste indicative.

3.3 Développement dirigé par les tests – TDD

Nous parlons ici uniquement de tests fonctionnels (par opposition aux tests de performance, sans objets malheureusement dans ce projet qui fera très peu d'optimisation).

Un environnement de tests vous est fourni dans le sous-répertoire `tests` pour automatiser les tests fonctionnels : un cas de test, ici, consiste en un petit programme C sur lequel on compare les sorties de gcc et de votre compilateur. Cet environnement devra être pris en main dès le début du projet, et on attend de vous que vous l'enrichissiez au fur et à mesure du développement. À la fin du projet, vous aurez normalement plus d'une centaine de cas de tests, qui vous serviront à éviter les régressions.

Attention, le principe même du TDD, c'est d'écrire des cas de tests pour qu'ils échouent ! Dans votre rendu final, vous êtes donc priés de laisser les tests qui échouent, ne les glissez surtout pas sous le tapis. Deux bonnes raisons pour ça : d'une part nous avons, depuis le temps, accumulé assez de cas de tests pour trouver tous vos bugs. D'autre part, les cas de tests qui échouent montrent que vous avez correctement adopté la méthodologie TDD préconisée dans ce projet.

Par contre, il serait inutile d'écrire les tests trop en amont du développement : c'est pour cela qu'on ne vous les fournit pas.

3.4 Evaluation et livrables

Un premier livrable est demandé à mi-parcours : vous devez fournir un compilateur fonctionnel et bien emballé supportant un sous-ensemble très limité du C (détaillé en annexe B). Ce livrable est évalué et sert à corriger le tir si nécessaire, mais son évaluation ne participe pas à la note finale.

La dernière séance de chaque groupe est consacrée à des soutenances : chaque hexanôme présente son compilateur et en fait une démonstration, en environ une demi-heure, plus questions.

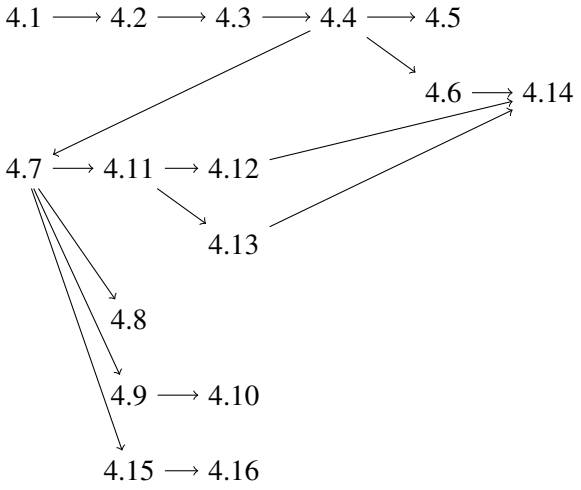
Le second livrable, final (un zip de votre compilateur) doit être fourni le dimanche suivant cette soutenance, ce qui vous permet de corriger des remarques faites en soutenance. Ce livrable (détaillé en annexe B) est évalué et noté selon le barême indicatif donné en annexe D. La note tient compte de la qualité de votre compilateur (couverture du C, stabilité, qualité du code), sur votre gestion de projet (méthodologie et outils utilisés, partage du travail, tests), et sur votre soutenance.

4 Décomposition en tâches – Backlog

Cette section est un exemple de séquencement des étapes qui vous emmèneront à un compilateur fonctionnel. Chaque sous-section correspond à une fonctionnalité et donc une tâche. Il peut être plus productif de les regrouper. N'hésitez pas à construire votre gestion de projet personnelle.

Certaines de ces tâches (la première ligne du graphe ci-dessous) sont très guidées et peuvent être vues comme des tutoriels. Il est important que **tous** les membres de l'équipe puissent bien comprendre ces premières étapes.

Le graphe de dépendances des différentes sous-tâches est le suivant (en utilisant les numéros des sections) :



4.1 Prise en main du code fourni

Votre tâche consiste ici à prendre en main la base de code qui vous est distribuée sur Moodle. Commencez par télécharger et décompresser le fichier `squelette-pld-comp.tgz`. Attention, vous devez vous placer dans un répertoire qui ne contient pas d'espace jusqu'à la racine.

- Le répertoire `compiler` est destiné à contenir le code de votre compilateur. Pour faciliter le démarrage du projet, nous vous en donnons une implémentation minimale avec la bonne structure.
- Le répertoire `tests` contient l'infrastructure de test, que vous prendrez en main à l'étape 4.2.

Installation de ANTLR4 Voici différents cas de figure pour l'installation d'Antlr4 (qui n'est en principe à faire qu'une seule fois sur chaque machine) :

- Machines linux du département (salle 208) : utilisez le script `./install-antlr.sh` pour installer ANTLR en local. Ensuite, pour compiler votre projet utilisez directement `make` dans `./compiler/`. Attention, vous devez être sur un sous-répertoire de votre home par défaut, sinon les liens symboliques ne marchent pas et l'installation échouera.
- ubuntu $\geq$ 20 : ANTLR est précompilé, il faut installer trois paquets, la commande `apt` correspondante est en commentaire dans `./compiler/runmake_ubuntu.sh`. Ensuite ce script doit être utilisé en place de `make` (il appelle `make`).
- ubuntu <20 : upgrader à un ubuntu $\geq$ 20, la sécurité de ces versions n'est plus assurée.
- Windows : utiliser WSL, version 2 minimum, et mise à jour (`wsl -update` dans un powershell). Ensuite ouvrir un terminal linux et procéder comme pour ubuntu ci-dessus.
- Linux-2D : vous n'avez rien à faire, tout est déjà installé.
  Pour compiler votre projet utilisez le script `./compiler/runmake_ubuntu.sh`
- Autre distribution Linux : regardez si la distribution offre des paquets pour Antlr4 et Antlr4-runtime, si c'est le cas vous éviterez de tout recompiler. Par exemple sous Fedora, il s'agit des paquets `antlr4`,

`antlr4-cpp-runtime` et `antlr4-cpp-runtime-devel`. Sinon lancez le script `./install-antlr.sh`.

— Machines MacOS : utilisez le script `./install-antlr.sh`. Pour compiler votre projet utilisez directement `make` dans `./compiler/`

Si vous devez exécutez `./install-antlr.sh` : après de longs écrans de messages divers et variés (avec même de la couleur pour les chanceux), si vous voyez finalement une ligne disant `PLD COMP: ANTLR OK` alors c'est que votre installation s'est passée correctement. Dans le cas contraire, demandez de l'aide à vos enseignants.

Le pourquoi des scripts runmake*

Vous allez collaborer par git à un fichier Makefile partagé entre des gens qui ont des systèmes différents, donc les chemins qu'il y a au début de ce Makefile ne peuvent pas être bons pour tout le monde.

La solution, c'est par exemmple `./compiler/runmake_ubuntu.sh` : ouvrez le dans un éditeur de texte, tout ce qu'il fait, c'est appeler `make` en écrasant les chemins par défaut du Makefile.

Dans votre Git, vous pouvez très bien avoir un runmake par membre de l'hexanome : `runmake-riri.sh`, `runmake-fifi.sh`, `runmake-fifi.sh`, etc.

Construction du compilateur

Dans le répertoire `compiler`, tapez la commande `make` ou `./runmake_xxxx.sh` suivant votre installation (cf. ci-dessus). Vous devriez observer successivement : un appel à l'outil ANTLR, traduisant votre grammaire `ifcc.g4` en plusieurs classes C++ (dans le répertoire `generated`) ; plusieurs appels à GCC pour compiler tout ce code ; et enfin un dernier appel à GCC pour lier le tout et produire l'exécutable `ifcc`.

Compilateur ifcc, version 0 Le compilateur fourni reconnaît uniquement les programmes constitués d'une instruction `return` suivie d'une constante entière. Exemple :

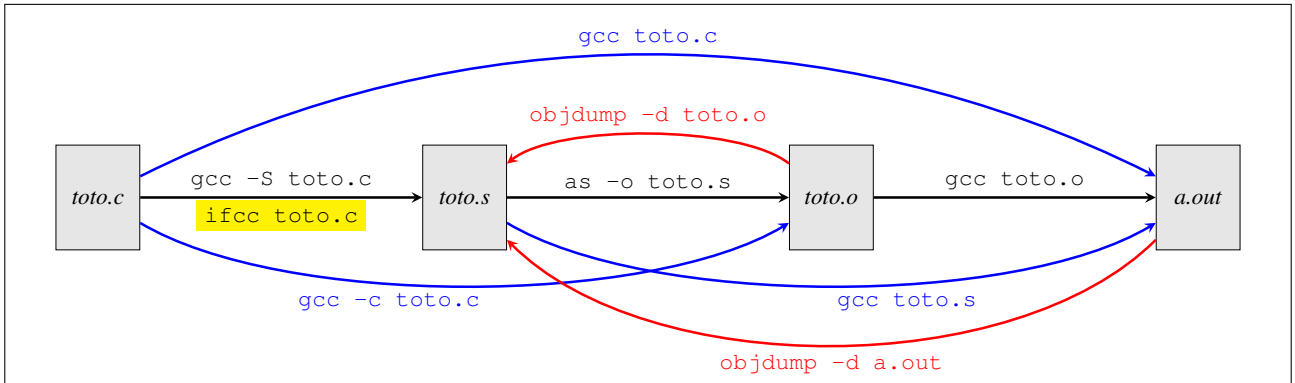
```
int main() {
  return 42;
}
```

GCC : source vers exécutable Pour s'échauffer, on va tout d'abord compiler ce programme avec GCC. Exercice :

1. Recopiez ce programme dans un fichier `ret42.c`.
2. compilez-le avec gcc :
   `gcc ret42.c`
   Note : on n'a pas spécifié de nom pour notre exécutable, donc GCC utilise par défaut : `a.out`
3. exécutez-le :
   `./a.out`
4. observez la valeur de retour du programme depuis le shell :
   `echo $?`

Tout au long du PLD, vous utiliserez cette même méthodologie pour valider le bon fonctionnement de vos programmes : compilation / exécution / observation de la valeur de retour. Attention, ce «statut de sortie du processus» est traité par le noyau comme un nombre non signé de 8 bits : si votre programme fait `return -1`, alors le `echo $?` affichera 255. C'est normal et ce n'est pas un bug. En pratique, il y a largement assez de valeurs entre 0 et 255 pour écrire des exemples intéressants.

GCC : ASM vers exécutable Pour l'instant, vous avez utilisé GCC pour réaliser plusieurs opérations d'un seul coup : compilation, assemblage, et édition des liens. La figure 3 montre les différentes étapas de ce processus, et les fichiers intermédiaires en jeu. Dans ce PLD, vous allez implémenter vous-même la traduction de



Ce n'est pas tout... voici les outils installés par le packet `gcc-msp430` :

msp430-addr2line	msp430-as	msp430-c++filt	msp430-elfedit	msp430-gcc
msp430-gcov	msp430-ld	msp430mcu-config	msp430-objcopy	msp430-ranlib
msp430-size	msp430-strip	msp430-ar	msp430-c++	msp430-cpp
msp430-g++	msp430-gcc-4.6.3	msp430-gprof	msp430-ld.bfd	msp430-nm
msp430-objdump	msp430-size	msp430-strip	msp430-readelf	msp430-strings

FIGURE 3 – GCC veut dire : GNU Compiler Collection

C vers l'assembleur (en jaune sur la figure 3) mais vous continuerez d'utiliser GCC pour traduire cet assembleur en un exécutable.

Voici donc à présent un programme en assembleur qui renvoie 42.

```
.text           # declaration of 'text' section (which means 'program')
.globl main     # entry point to the ELF linker or loader.
main:
  movl $42, %eax
  ret
```

Exercice :

1. Recopiez ce programme dans un fichier `main.s`.
2. Assemblez-le :
   `as -o main.o main.s` (regardez quel fichier a été produit)
3. Linkez-le pour obtenir un exécutable :
   `gcc main.o` (regardez quel fichier a été produit)
4. exécutez-le :
   `./a.out`
5. observez la valeur de retour :
   `echo $?`

Changez le `$42` pour une autre valeur et refaites tout cela. Les autres lignes du programme seront expliquées sous peu, mais vous pouvez poser des questions.

IFCC : source vers ASM Le squelette de compilateur fourni se contente de produire le second listing à partir du premier ! Exécutez-le en tapant `./ifcc ret42.c`, puis lisez tous les fichiers distribués dans le répertoire `compiler` et posez des questions sur ce que vous ne comprenez pas.

Bonus : visualisation de l'AST Pour l'instant, la grammaire `ifcc.g4` est très simple, mais vous allez la faire évoluer tout au long de votre PLD. Afin de vous aider dans la mise au point de la grammaire, vous pouvez utiliser l'outil `testrig` de ANTLR, qui affiche l'arbre de dérivation dans une fenêtre graphique. Pour cela, tapez par exemple `make gui FILE=ret42.c` (remplacer `make` par `runmake_bidule.sh` qui va bien) puis relisez le Makefile pour en savoir plus.

Note : cette visualisation nécessite le compilateur `javac` donc assurez-vous d'avoir installé les paquets nécessaires, par exemple en tapant `sudo apt install default-jdk`.

4.2 Prise en main de l'environnement de test

Reste à prendre en main l'infrastructure de test dans le répertoire `tests`.

Le principe est assez simple, un code source C est compilé avec `gcc` ainsi qu'avec votre compilateur. Les binaires sont exécutés et les résultats sont comparés au niveau du code de sortie (le `return` du `main`) ainsi que la sortie standard. Si les deux se comportent de la même manière, le test est validé, sinon il échoue.

Un test peut échouer pour des tas de raisons différentes, par exemple

— votre compilateur ne compile pas ;
— votre compilateur compile mais plante à l'exécution ;
— votre compilateur compile et tourne mais produit de l'assembleur invalide ;
— votre compilateur ... et produit de l'assembleur valide mais qui ne calcule pas le bon résultat.

(et on en oublie).

De plus, un test est un succès si votre compilateur échoue à compiler un programme que `gcc` échoue également à compiler.

... Allez lire tout ce qu'il y a dans le répertoire `tests`. Il est important de comprendre les trois tests fournis, mais aussi de lire les scripts pour comprendre ce qu'ils font.

1. Lancez le script de test fourni avec l'option `--help`
   ```
   python3 ifcc-test.py --help
   ```
2. Comprenez comment il est utilisé par `ifcc-wrapper.sh`
3. Lancez les tests
4. Ajoutez un programme incorrect dans le répertoire `tests/testfiles`

Validez auprès d'un enseignant votre compilateur sur au moins deux programmes corrects et deux programmes incorrects.

Lorsque vous avez compris tout cela, vous avez une infrastructure du projet pour un compilateur fonctionnel, mais pour un langage (très) restreint. Dans votre gestion de projet, chaque objectif de sprint devra être un "compilateur fonctionnel mais pour un langage restreint". Vous êtes encouragés à commencer chaque sprint par l'écriture de quelques programmes de tests qui illustrent les fonctionnalités ajoutées par le sprint (développement dirigé par les tests).

4.3 Les variables en mémoire

Reprenons un programme C qui renvoie 42.

```
int main() {
  return 42;
}
```

1. Recopiez-le dans un fichier `ret42.c`.
2. compilez-le vers de l'assembleur par la commande shell :
   ```
   gcc -S ret42.c
   ```
   (ceci crée `ret42.s`)
3. ouvrez `ret42.s` dans votre éditeur préféré.

4. Retrouvez-y le code généré par la première étape.

Ce code est emballé dans du code magique que nous comprendrons en temps utile. Si on enlève tout ce que je ne comprends pas, on arrive au code minimal assembleur suivant :

```
.text           # declaration of 'text' section (which means 'program')
.globl main    # entry point to the ELF linker or loader.
main:
    # prologue
    pushq %rbp          # save %rbp on the stack
    movq %rsp, %rbp      # define %rbp for the current function

    # body
    movl $42, %eax

    # epilogue
    popq %rbp            # restore %rbp from the stack
    ret                  # return to the caller (here the shell)
```

Les deux lignes du prologue servent à fournir à votre code un Base Pointer (BP) vers un coin de mémoire où il pourra placer ses variables et variables temporaires. On comprendra plus tard dans les détails.

En attendant, ajoutez à votre générateur de code les deux lignes de prologue et les deux lignes d'épilogue.

À présent, voici un autre programme en C qui renvoie 42.

```
#include <inttypes.h>
int main() {
  int a=42;
  return a;
}
```

1. Recopiez-le dans un fichier `ret42aff.c`.
2. compilez-le vers de l'assembleur par la commande shell :
   `gcc -S ret42aff.c` (ceci crée `ret42aff.s`)
3. ouvrez `ret42aff.s` dans votre éditeur préféré.
4. Retrouvez-y vos petits. Qu'est-ce qui a changé ? Où est rangée la variable `a` ?

En assembleur x86, la notation `-12(%rbp)` dénote le contenu de la case mémoire d'adresse BP-12. En syntaxe C, cela s'écrirait `*(rbp-12)`.

Recommencez avec un programme C qui déclare deux variables, puis trois, qui les recopie les unes dans les autres, etc. Comprenez où sont rangées les variables.

Un bon objectif à ce stade sera un compilateur de bout en bout pour un langage qui permet les affectations variable=constante, et aussi variable=variable. Le front-end devra associer à chaque variable un index, qui sera stocké dans la table des symboles, et utilisé par le back-end pour générer le code qui accède à la variable. Tous vos index seront des multiples de 4, car toutes vos variables seront des entiers 32 bits.

C'est aussi le moment de mettre en place les premières analyses statiques : vérifier si une variable déclarée est utilisée par exemple, et produire un warning sinon.

4.4 Expressions arithmétiques

Coté front-end, il faudra gérer les expressions avec leurs priorités, les parenthèses, etc. Les feuilles seront, à ce stade, des variables ou des constantes. Mettez en place des tests qui vérifient que votre grammaire gère bien le − unaire, les priorités et l'associativité à gauche par défaut.

Côté back-end, il faudra implémenter l'algorithme qui parcourt l'AST et génère le code assembleur correspondant. Il y a une vidéo et un jeu de slides qui expliquent cet algorithme.

Au terme de cette tâche vous avez votre délivrable de mi-parcours. Il est recommandé de bétonner les tests.

Vous pouvez aussi intégrer dans votre livrable la tâche suivante.

4.5 Vérifications statiques sur les expressions et variables

Il s'agit de vérifier la cohérence des déclarations de variables :

— Une variable utilisée dans une expression a été déclarée ;

— Une variable n'est pas déclarée plusieurs fois ;

— Une variable déclarée est utilisée au moins une fois.

Ces analyses peuvent se faire dans votre visiteur, dans un autre visiteur spécifiquement dédié à cette tâche, ou simplement être intégrées dans les méthodes d'accès à la table des symboles.

4.6 Propagation de constantes dans les expressions

Cette tâche est une optimisation donc elle est optionnelle, vous pouvez la garder pour plus tard. Vous pouvez faire des optimisations qui vont simplifier les expressions là où c'est possible :

— Éliminer les éléments neutres des opérateurs (le 1 pour la multiplication, le 0 pour l'addition, *etc.*)

— Transformer des expressions constantes en leur valeur (exemple : 1+4*2 sera transformé en 9)

Ce qui précède est une optimisation sur les expressions, qui peut donc se faire lors de leur visite. Vous pouvez aussi propager les constantes à travers les variables.

4.7 On reprend tout en passant à l'IR

Passer à une représentation intermédiaire pour le code cible n'est pas indispensable pour un compilateur fonctionnel C vers x86. Mais cela ouvre la possibilité de nombreuses optimisations, et permet une nouvelle fonctionnalité : la génération de code ARM ou MSP430 (au choix), comme illustré sur la figure 2

Nous vous encourageons à utiliser l'IR fournie dans `IR.h`, et à vous appuyer sur la video qui la décrit. Toutefois, vous êtes libres de l'améliorer.

Cette tâche se décompose en sous-tâches dont certaines sont parallélisables :

1. mettre en place la structure de donnée pour l'IR ;
2. faire construire l'IR par le front-end ;
3. transformer l'IR en assembleur x86 dans le back-end.
4. transformer l'IR en assembleur ARM ou MSP430 ou ce que vous voulez dans le back-end.

Pour le MSP430, vous pourrez choisir que le type `int` fait 16 bits, ou 32 bits. Remarquez que cela a un impact sur les index. Comme il n'y a pas d'instruction de multiplication, votre compilateur MSP430 est autorisé à refuser les programmes avec multiplications, tant que c'est fait avec grâce (ce pourra être réparé par un appel de fonction, mais c'est la tâche suivante).

Dans cette tâche, on restera dans un seul bloc de base (BB), ce qui correspond à du code linéaire, sans test ni branchement : c'est le cas du langage-cible des tâches précédentes. Le graphe de contrôle de flot (le graphe de BB) sera mis en place plus tard, et donc si vous n'avez pas compris cette phrase c'est normal.

4.8 Ajout du type `char` et de l'inférence de type

Cette tâche peut arriver plus ou moins tôt. Une subtilité est qu'il faut à présent attacher un type à tous les noeuds intermédiaires (implicites) de l'AST. Par exemple dans `char a, b, c; int n,m; n=a+b+m+c;` on a des additions 8 bits et des additions 32 bits (dessinez l'arbre et dites ou).

Votre génération d'instruction doit gérer également les types : il faut utiliser des movb, movl, movq (etc) à bon escient et avec les bons noms de registres (par exemple %eax pour 32 bits et %al pour 8 bits). Par exemple une lecture de la mémoire vers %rax n'est légale que si l'adresse est un multiple de 8.

4.9 Mise en place de l'enregistrement d'activation, de l'ABI et des appels de fonction

L'ABI (application binary interface) est spécifique à chaque processeur. Il est indispensable de faire cette section pour x86. C'est intéressant de réfléchir au support de deux cibles, mais si vous vous rendez compte alors qu'il faudrait tout refactorer, laissez tomber la cible secondaire (MSP430 ou ARM) pour la suite.

Compilez le programme suivant avec le vrai `gcc -S -O0` :

```
void toto() {
  int x=1;
  int y=2;
  int z=3;
  putchar('a');
}
```

Retrouvez dans le `.s` les trois constantes, le putchar, et enfin identifiez (en gros) son prologue et son épilogue, c'est-à-dire ce qu'il y a au début et à la fin.

La nouveauté c'est une opération sur le SP (stack pointer) : c'est elle qui réserve la mémoire pour la fonction. Normalement votre compilateur, à ce stade, connait la taille de la zone mémoire à réserver : il sera facile de modifier votre générateur de code pour qu'il réserve ce qu'il faut.

Le but de cette tâche est de mettre en place les appels de fonction d'une part, les déclarations de fonctions d'autre part. C'est une grosse tâche, n'hésitez pas à la décomposer.

Pour les appels, commencez par relire le chapitre *Compiling the body* du poly : tout y est. Vous pourrez tester ces développements avec la fonction de bibliothèque standard `putchar`. L'intérêt est qu'il suffit d'avoir modifié le parseur pour qu'il reconnaisse les appels de fonctions, pas leur déclaration.

- `CallExpr::genIR` doit évaluer la valeur de chaque expression passée en paramètre et se souvenir des variables dans lesquelles il les a rangées. Puis il doit émettre une seule instruction IR `call` qui prend en paramètre cette liste de variables.
- `genAsm()` émet l'assembleur qui recopie chaque variable dans un registre suivant l'annexe A. Puis il émet une instruction x86 `call`.

On commencera par se limiter à compiler les appels de fonction ayant jusqu'à 6 arguments (quand il y en a plus il faut passer des arguments sur la pile). Les registres à utiliser sont décrits dans l'ABI *(aplication binary interface)* en annexe.

Pendant ce temps, côté front-end, on peut implémenter les déclarations de fonctions.

Testez tout ceci sur un programme structuré en plusieurs fonctions.

On aimerait essayer une fonction récursive, mais pour cela il faut qu'elle puisse terminer, ce qui nous amène à la tâche suivante.

4.10 Vérifications statiques sur les fonctions

À ce stade on espère que vous trouverez tout seul (enfin, tout votre hexanome) ce qui est attendu dans cette tâche. N'hésitez pas à en discuter avec un enseignant.

4.11 Compiler le `if ... else`

A partir de là il faut créer des blocs de base, donc gérer les successeurs d'un BB. Il est conseillé d'avoir deux BB spéciaux : le point d'entrée de la fonction (il devra générer le prologue) et un point de sortie unique qui générera l'épilogue.

Coté front-end, mettez en place des tests pour le cas de deux `if` fermés par un seul `else`. Côté back-end, tout est dans le poly, mais cela reste une marche assez haute.

4.12 Gestion du return n'importe où

Il faut réfléchir au cas ou il y a plusieurs `return expr;` dans un programme.

C'est une bonne idée d'avoir un seul épilogue par fonction : pour cela, `return expr;` évalue expr, copie le résultat dans une variable spéciale `!retvalue`, puis saute au BB de sortie.

La génération d'assembleur pour ce BB de sortie copie le contenue de `!retvalue` dans `%rax`, puis émet l'épilogue.

4.13 Compiler les boucles `while`

À la surprise générale, une fois que le if-then-else marche, le while vous prendra quelques minutes à faire tomber en marche, car il ne demande rien en plus. C'est une micro-tâche.

Et vous voilà capables d'implémenter la factorielle de deux manières différentes : récursive, et avec un while.

4.14 Propagation des variables constantes (avec analyse du *data-flow*)

Cette tâche est une optimisation donc elle est optionnelle. Elle s'appuie sur la propagation des constantes de la tâche 4.6.

Plus intéressant, mais aussi beaucoup plus compliqué, on peut essayer de propager des valeurs de variables connues (ou localement constantes) : Une ligne affecte à une variable une valeur constante. Alors, dans toutes les expressions qui suivent jusqu'à la prochaine affectation à cette variable, on peut remplacer la variable par la constante. Sur du vrai code la détermination de ce qu'est "toutes les instructions qui suivent..." nécessite une analyse *data-flow*.

4.15 Compiler l'affectation à une lvalue quelconque

Ici il faut mettre en place l'affectation à une lvalue quelconque. Implémentez l'usine à gaz décrite dans le poly, section *Generic Lvalue-based assignment code*. Testez que cela marche sur des programmes qui ne font que des affectations dans des variables (et marchaient jusque là).

4.16 Compiler des tableaux

Rien de bien méchant si vous avez compris 1/ la distinction entre `lvalue` et `rvalue` et 2/ que `a[i]` c'est du sucre syntaxique pour
`Mem[a+i*sizeof(type(a))]`.

Remarque si vous êtes en avance et que vous voulez produire un compilateur reciblable : Ces contraintes d'alignement dépendent du processeur cible. Par exemple sur MSP430 les variables 32 bits doivent juste être alignées sur des adresses paires. Les contraintes d'alignement doivent donc êtres fournies par la classe processeur qui encapsule toute la transformation d'IR en assembleur.

4.17 Compiler les appels de fonction ayant plus de 6 arguments

Les arguments supplémentaires doivent être empilés par l'appelant, et utilisés par l'appelé. Là il y a quelquechose d'intéressant et nouveau, c'est qu'il faut utiliser des offsets positifs pour aller piocher les paramètres empilés.

Exercice : trouver dans l'ABI qui fait le dépilement des paramètres ainsi passés, l'appelé ou l'appelant ?

Attention cependant, grosse difficulté pratique si vous voulez respecter la vraie ABI Linux : il faut 1/ gérer les types proprement, et 2/ empiler les paramètres comme spécifié par l'ABI.

Annexes

A Éléments d'ABI des PC linux

A.1 Passage de paramètres

En mode 64-bits, on utilise les registres pour passer les paramètres [1].
Ces registres sont spécialisés par l'ABI comme suit :
(...) the registers get assigned (in left-to-right order) for passing as follows :
— (... for integer parameters) the next available register of the sequence %rdi, %rsi, %rdx, %rcx, %r8 and %r9 is used.
— Once all registers are assigned, the arguments are passed in memory. They are pushed on the stack in reversed (right-to-left) order [2].

Dans un premier temps on pourra se limiter à des fonctions qui passent tous leurs arguments par les registres. Cela nous limite à des fonctions à 6 paramètres ou moins. Dans ce cas, produisez un message d'excuse si une fonction a plus de 6 paramètres.

A.2 Valeur de retour

La valeur de retour est passée dans `%rax`. Argh ! Tous nos programmes jusqu'ici utilisaient `%eax` ! ? ! En fait, c'est le même registre 64-bit `rax` [3]dont les 32 bits de poids faible s'appellent `eax` [4], les 16 bits de poids faible s'appellent `ax` [5], et les 8 bits de poids faible s'appellent `al`. De la poésie ? Non, de l'histoire.

Idem pour les autres registres : en mode 32 bits on avait `esp`, `ebp`, etc. En mode 64 bits ils s'appellent `rsp`, `rbp`, etc. Dans tous les cas `sp` c'est *Stack Pointer*, et `bp` c'est *Base Pointer*.

A.3 Qui est propriétaire de quels registres ?

Voici la convention qui dit quels registres une fonction peut écraser, et quels registres elle est priée de laisser dans l'état où elle les a trouvés :

Registers %rbp, %rbx and %r12 through %r15 "belong" to the calling function and the called function is required to preserve their values. In other words, a called function must preserve these registers' values for its caller. Remaining registers "belong" to the called function. If a calling function wants to preserve such a register value across a function call, it must save the value in its local stack frame.

Le respect de cette convention est facile : il suffit de vivre uniquement avec deux registres bien choisis.

1. Grosse différence avec l'ABI 32 bits, dans laquelle tous les arguments étaient passés sur la pile...
2. Right-to-left order on the stack makes the handling of functions that take a variable number of arguments (such as `printf`) simpler. The location of the first argument can always be computed statically from the stack pointer, based on the type of that argument. It would be difficult to compute the address of the first argument if the arguments were pushed in left-to-right order.
3. "r" comme *really extended*
4. "e" comme *extended*
5. "x" comme *extended*

B Livrables

B.1 Livrable à mi-parcours

L'objectif à mi-parcours est un compilateur qui fonctionne de bout en bout sur un sous-ensemble de C permettant ce type de programme :

```
int main()
{
  int a,b,c;
  a=17;
  b=42;
  c = a*a + b*b +1;
  return c;
}
```

- il n'y a qu'une fonction main, sans arguments, et il n'y a qu'une paire d'accolades ;
- il n'y a qu'une seule ligne pour déclarer toutes les variables, et le seul type supporté est `int` ;
- un programme est une séquence d'affectations, sans boucle, ni test ;
- le programme compilé n'affiche rien, mais retourne une valeur au shell. Vous pourrez tester cette valeur par exemple par
  `echo $?`
- Pour le reste, la syntaxe des expressions est celle de C.

À mi-parcours, vous déposerez sur moodle un dossier incluant votre code, ainsi qu'un document qui 1/ décrit les fonctionnalités implémentées, 2/ permet de naviguer dans votre code et 3/ décrit succinctement votre gestion de projet passée et à venir.

En principe, le sprint correspondant sera terminé avant ce rendu. Si vous anticipez que ce ne sera pas le cas, n'hésitez pas à demander de l'assistance à l'équipe enseignante.

B.2 Livrable final

À l'issue du projet, un livrable de réalisation sera déposé sur moodle.

Ce livrable inclura une application testable sur les machines LINUX du département.

Il sera déposé sur moodle sous la forme d'une archive zip nommée <numéro de l'hexanôme>.zip. Merci de nettoyer l'archive pour ne pas qu'elle contienne ni binaires ni répertoires cachés (tels que fichiers .git). Le zip contiendra :

- une présentation de l'hexanôme (sa composition et les rôles de chacun si cela est pertinent) ainsi que les éventuels supports utilisés lors de la soutenance.
- tous les sources de votre application ;
- un `makefile` avec une cible par défaut qui crée l'exécutable et une cible « test » qui lance les jeux de tests ; Si vous utilisez CMake ou autre, un README devra donner des instructions copicollables pour compiler votre projet sur les machines du département ;
- vos jeux de tests ;
- une documentation utilisateur et une documentation développeur.

C Soutenances

La dernière séance sera dédiée à des présentations orales (une heure par hexanôme). Votre exposé devra nous montrer l'état de l'avancement de votre compilateur, et une exécution commentée des tests dans divers cas à l'issue du projet.

Les attendus sont, en gros :
— un tour d'horizon des fonctionnalités supportées et non supportées, avec une discussion sur vos choix de conception.
— une présentation des points originaux (de votre compilateur, par rapport à celui des autres) par exemple les structures de données que vous avez utilisées, avec des dessins.
— des démonstrations interactives : c'est bien si on peut aussi vous demander «compilez-nous donc tel ou tel programme»
— des éléments de gestion de projet : organisation du temps, prise de décision collective, répartition des tâches...

Vous avez le droit de faire un powerpoint mais ce n'est pas du tout obligatoire. Par contre on veut entendre tous les membres de chaque henanôme dans la présentation, donc préparez à l'avance votre répartition des temps de parole. Prévoyez un exposé+démo d'environ 20 ou 25 minutes, ce qui nous laissera environ autant de temps pour les questions-réponses.

D Barême indicatif

Voici le barême appliqué :
— 4 points pour la gestion de projet et la qualité (incluant la gestion des tests et la qualité du code : lisibilité, modularité, commentaires, *etc.*)
— 4 points pour la soutenance : préparation, démo, équilibre de la parole
— 8 points pour le support du langage C, dont
  — 4 points pour expressions + affectations (le rendu de mi-parcours)
  — 4 points pour les autres *features* du C : if/while, tableaux, appels+déclaration de fonctions, etc – pas de barême précis ici
— 4 points pour les extras de votre compilateur : analyses statiques poussées, optimisations, multi-cible, ...