

PLD-Comp

Développement d'une chaîne complète de compilation

L'objectif de ce projet est la conception d'un compilateur pour un sous-ensemble bien défini 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 à terme est le support d'un sous-ensemble du langage C qui devra inclure les éléments suivants :

- Types de données de base : `char` et `int`. Les autres types (`int64_t`, `uint64_t`, `float`, `double`, ...) sont optionnels.
- Tableaux à une dimension.
- Définition de fonctions. Les procédures (qui n'ont pas de type de retour) devront utiliser le type retour `void`.
- Déclaration de variables seulement au début des fonctions. La possibilité de déclarer les variables n'importe où est optionnelle.
- Possibilité d'initialiser une variable lors de sa déclaration.
- Structure de blocs grâce à `{` et `}`.
- Structures de contrôle : `if`, `else`, `while`. Les autres structures (`for`, `do...while`, `switch...case`, etc.) sont optionnelles.
- Expressions : constantes entières et caractère (avec simple quote); variables; opérations arithmétiques de base; `+`, `-`, `*`; opérations logiques `|`, `&`, `^`; opérations de comparaisons `==`, `!=`, `<`, `>`; opérations unaires `!` et `-`. Tous les autres opérateurs sont optionnels.
- Affectation (qui, en C, retourne aussi une valeur).
- Possibilité de déclarer des variables globales.
- Utilisation des fonctions standard `putchar` et `getchar` pour les entrées-sorties.
- Les chaînes de caractères pourront être représentées par des tableaux de `char` (les constantes chaînes de caractères sont optionnelles).
- Un seul fichier source sans pré-processing (tout pré-processing est optionnel). Les directives du pré-processeur devront toutefois être autorisées par la grammaire, mais pourront être ignorées, ce afin de garantir que la compilation par un autre compilateur soit possible. (Exemple : inclusion de `stdint.h`).
- Commentaires

Exemple de programme :

```
#include <stdint.h>
#include <stdio.h>

int main() {
    char a;
    a='A';
    while (a<'Z'+1)
    {
        putchar(a);
        a=a+1;
    }
}
```

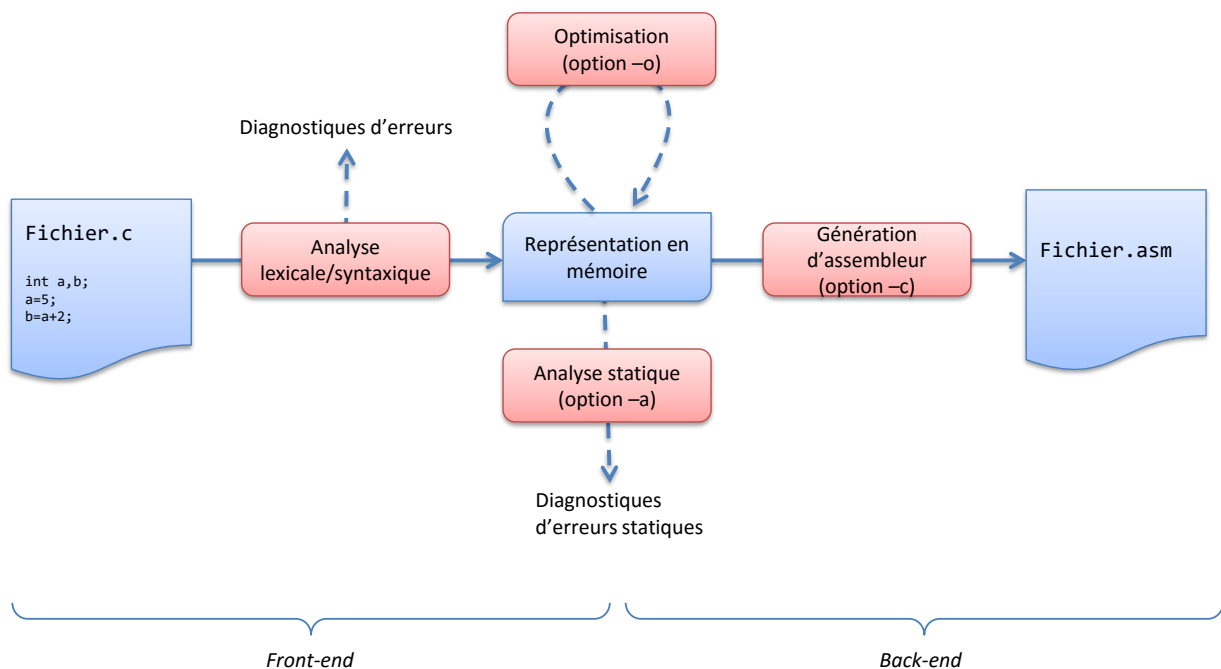
En général, implémenter les aspects optionnels rapportera des points, toutefois, cela demandera plus de travail sans vous faire comprendre de nouveaux concepts. L'implémentation correcte de tous les éléments requis est donc prioritaire.

D'autres traits du langage qui sont explicitement optionnels sont

- les pointeurs (toutefois, on pourra utiliser la notation `[]` comme argument d'une fonction pour passer en paramètre un tableau par son adresse)
- La possibilité de séparer dans des fichiers séparés les déclarations et les définitions
- ...

2 Architecture globale de l'outil

Voici le schéma de l'architecture globale de l'outil :



Le logiciel se présente sous la forme d'un outil en ligne de commande dont l'argument principal est le nom du fichier à analyser. Par défaut, l'outil va analyser le fichier, en faire une représentation en mémoire et afficher 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. En complément, il existe trois options en ligne de commande qui permettent d'enrichir ce fonctionnement de base :

1. Analyse statique du programme (option -a). Cette option fait une analyse du programme de manière statique afin d'en extraire des erreurs (sur la sortie d'erreur standard).
2. Transformation du programme afin de le simplifier (option -o). En propageant les constantes et en simplifiant certaines expressions (éléments neutres), la représentation interne du programme est modifiée.
3. Génération de code (option -c). Cette option permet de générer le code assembleur dans un fichier séparé.

3 Gestion de projet et livrables

Le projet se déroulera sous forme de sprints. Chaque sprint 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 du sprint. Ce document peut vous aider à mettre en place les premiers sprints, mais c'est uniquement à titre indicatif.

3.1 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.

3.2 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 zip¹ 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 :

- 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 copiables pour compiler votre projet sur les machines du département.
- vos jeux de tests

Enfin, une présentation orale (pendant les deux dernières heures de la dernière séance) montrera l'état de l'avancement du projet, et une exécution commentée des tests dans divers cas à l'issue du projet.

1. ce format d'archive permettra d'éviter d'encoder les droits d'accès sur les fichiers comme le ferait une archive tgz

4 Premiers sprints

Chaque sous-section peut correspondre à un sprint, mais il peut être plus productif de les regrouper. N'hésitez pas à construire votre gestion de projet personnelle.

4.1 De l'assembleur compliqué, un langage compliqué, et une infrastructure compliquée

La version 0.1 de votre compilateur devra reconnaître uniquement les programmes constitués d'un statement return suivi d'une constante, et les compiler en un fichier a.out qui, lorsqu'on l'exécute, renvoie la constante au shell.

Exemple

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

1. Recopiez-le dans un fichier `ret42.c`.
2. compilez-le avec gcc :
`gcc ret42.c` (regardez quel fichier a été produit)
3. exécutez-le par la commande shell :
`./a.out`
4. Observez la valeur de retour de ce programme par la commande shell :
`echo $?`

Voici un programme minimal en assembleur qui renvoie 42.

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

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

Exercice : changez le 42 pour autre chose et refaites tout cela. Les autres lignes du programme seront expliquées sous peu, mais vous pouvez poser des questions.

Donc si vous savez écrire le second listing à partir du premier, c'est tout bon.

Écrivez une grammaire adaptée à une analyse descendante pour le langage demandé. Dans cette grammaire, la seule partie variable sera l'analyse de la constante. Les grammaires ANTLR4 autorisent des répétitions de symboles ou de parties de règles, profitez de cette possibilité pour éviter les règles récursives lorsque cela est possible.

Ensuite (ou en même temps) il s'agit surtout de mettre en place toute l'infrastructure de votre projet. Il faudra au moins un fichier `.g4` (ANTLR²), quelques fichiers `.hpp` et `.cpp`, un `Makefile`, et quelques scripts qui font l'assemblage et le link, éventuellement le test, etc.

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

Lorsque tout tombe en marche, vous avez 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".

4.2 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`
3. ouvrez `ret42.s` dans votre éditeur préféré.
4. Retrouvez-y le code généré par le premier sprint.

(ceci crée `ret42.s`)

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')  
.global 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. Sur les machines Linux du département, ANTLR4 est installé dans `/shares/public/tp/antlr4-cpp` et ce répertoire contient aussi un exemple avec un `makefile` qui vous aidera à bootstraper le vôtre.

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 de sprint 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.

Cela dit, si vous êtes vraiment à l'aise, vous pouvez attaquer directement avec le sprint suivant.

4.3 Sprint des expressions arithmétiques

Côté front-end, il faudra gérer les expressions avec leurs priorités, les parenthèses, etc. Les feuilles seront, à ce stade, des variables ou des constantes.

Côté back-end, il faudra implémenter l'algorithme qui parcourt l'AST et génère le code assembleur correspondant.

C'est aussi le moment de mettre en place une infrastructure de tests un peu systématique : il faudra tester votre compilateur sur des tas de programmes C exploitant les priorités des opérateurs, par exemple. D'autres programmes pourront tester vos analyses statiques.

Au terme de ce sprint vous avez votre livrable de mi-parcours.

Vous pouvez aussi rendre le résultat du sprint suivant.

4.4 On reprend tout en passant à l'IR

Ce sprint va ajouter une fonctionnalité : la génération de code ARM ou MSP430 (au choix).

Il utilisera l'IR fourni dans `IR.h` et s'appuiera sur le poly. Les tâches principales sont

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.

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 avec grâce (ce pourra être réparé par un appel de fonction, mais c'est le sprint suivant).

Par rapport à `IR.h`, 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 du premier sprint. 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, ce n'est pas grave.

4.5 Mise en place l'enregistrement d'activation et l'ABI

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, connaît 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 ce sprint est de mettre en place les appels de fonction d'une part, les déclarations de fonctions d'autre part.

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 la section 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 au sprint suivant.

4.6 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.

À nouveau tout est dans le poly, mais cela reste une marche assez haute.

Note sur le retour de valeur

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

Une option est de définir une variable temporaire spéciale `retvalue`.

`return expr;` évalue `expr`, copie le résultat dans `retvalue`, puis saute au BB de sortie.

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

4.7 Compiler les boucles `while`

À la surprise générale, si 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 un microprint.

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

4.8 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).

4.9 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))].
```

4.10 Compiler les boucles `for`

Rien de bien méchant si vous connaissez leur sémantique.

4.11 Les autres types de base

Ajoutez au front-end les types `char` et `int64_t`. Essayez de ne consommer dans votre AR qu'un octet pour les chars, et 8 pour les `int64`. Vous avez le droit de regrouper tous les chars d'un programme ensemble et tous les `int32` d'un programme ensemble.

Attention, cela ne marchera que si votre génération d'instruction gère également les types : il faut utiliser des `movb`, `movl`, `movq` à bon escient et avec les bons noms de registres (par exemple `%eax` pour 32 bits et `%al` pour 8 bits). En effet une lecture de la mémoire vers `%rax` n'est légale que si l'adresse est un multiple de 8.

Pour valider cette étape, vérifiez que vos prologue et épilogue sont corrects sur des variations du programme suivant :

Remarque si vous êtes en avance et que vous voulez produire un compilateur recible : 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 être fournies par la classe processeur qui encapsule toute la transformation d'IR en assembleur.

4.12 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 quelque chose 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 l'ABI : il faut 1/ gérer les types proprement, et 2/ empiler les paramètres comme spécifié par l'ABI.

A Annexe : éléments d'ABI des PC linux

A.1 Passage de paramètres

En mode 64-bits, on utilise les registres pour passer les paramètres³.

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) order⁴.

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`⁵ dont les 32 bits de poids faible s'appellent `eax`⁶, les 16 bits de poids faible s'appellent `ax`⁷, 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.

3. Grosse différence avec l'ABI 32 bits, dans laquelle tous les arguments étaient passés sur la pile...

4. 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.

5. “r” comme *really extended*

6. “e” comme *extended*

7. “x” comme *extended*