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 qui devra inclure environ les éléments suivants (dans le désordre) :

- Types de données de base : `char` et `int`. Les autres types (`float`, `double`, etc) sont optionnels.
- 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 bit-à-bit `|`, `&`,`^` ; opérations de comparaison `==`, `!=`, `<`, `>` ; opérations unaires `!` et `-`. Tous les autres opérateurs sont optionnels, par exemple les opérateurs logiques paresseux `||`, `&&`, et `!`
- Affectation (qui, en C, retourne aussi une valeur).
- Tableaux (à une dimension).
- Variables globales (optionnel).
- 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 `stdio.h`).
- Commentaires

Exemple de programme :

```
#include <stdio.h>

int main() {
  char a;
  a='A';
  while (a<'Z'+1)
  {
    putchar(a);
    a=a+1;
  }
  return 0;
}
```

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

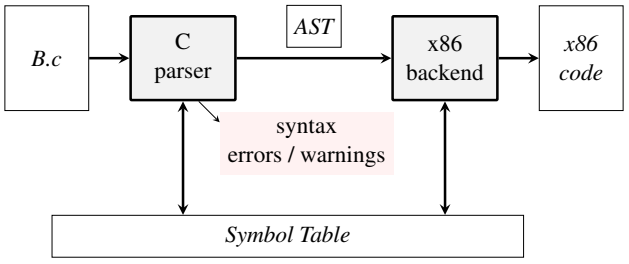
— les pointeurs (toutefois, on pourra utiliser la notation `[]` comme argument d'une fonction pour passer en paramètre un tableau par son adresse)
— La possibilité de séparer dans des fichiers distincts les déclarations et les définitions
— ...

2 Architecture globale du compilateur demandé

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.

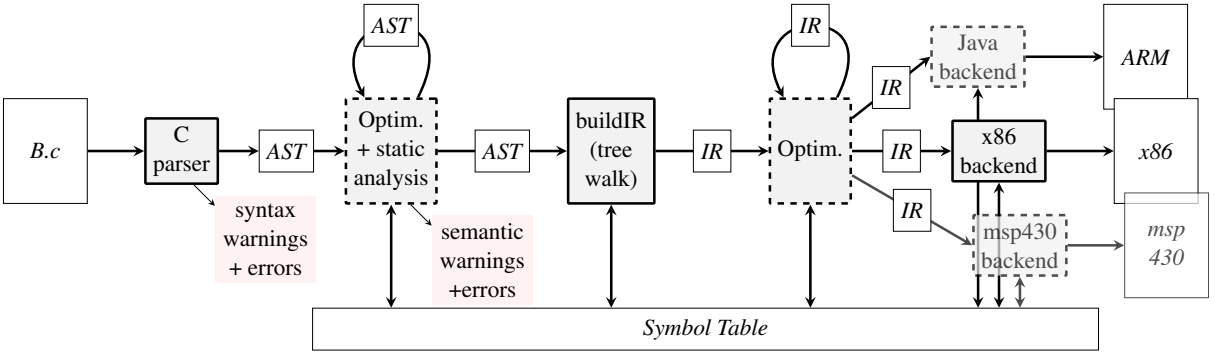
L'outil va analyser ce programme source, et afficher si nécessaire des diagnostics d'erreurs (lexicales, syntaxiques ou sémantiques simples). Chaque erreur devra faire apparaître le numéro de ligne et de manière optionnelle le numéro de colonne dans le fichier source. Pour les programmes syntaxiquement 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*) :



L'information transmise entre le parser et le générateur de code est un arbre de syntaxe abstraite (AST) ainsi qu'une table des symboles. Dans ce document, l'AST sera plusieurs fois évoqué. Il s'agit d'une abstraction de l'arbre de dérivation issu de l'analyse Antlr4. Vous pouvez en faire une structure de donnée explicite, ou vous pourrez si vous le souhaitez directement travailler sur cet arbre de dérivation par l'intermédiaire de visiteurs.

Au delà de ce minimum fonctionnel, nous attendons un compilateur qui se rapproche d'un vrai compilateur. Le schéma suivant donne l'architecture d'un compilateur optimiseur reciblable, dans lequel chacune des fonctionnalité optionnelle en pointillé a été implémentée par plusieurs groupes dans les années précédentes.



Ces options sont de trois types :

1. Analyse statique du programme.

2. Optimisations, soit sur l'AST, soit sur la représentation intermédiaire
3. Génération de code pour une autre cible.

Ces options seront contrôlées par des paramètres supplémentaires passés à la ligne de commande, par exemple -O pour optimisation.

3 Gestion de projet et livrables

Le projet se déroulera sous forme itérative. 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. Ce document peut vous aider à mettre en place les étapes, mais c'est uniquement à titre indicatif.

3.1 Gestion de configuration et intégration continue

Afin de simplifier le démarrage du projet, nous mettons à votre disposition un ensemble de fichiers qui serviront de point de départ. Vous pouvez bien entendu adapter cet environnement à vos contraintes. Les fichiers se trouvent sur Moodle. L'arborescence comprend deux sous-répertoires :

— `compiler/` contient les fichiers sources du compilateur
— `tests/` contient les fichiers permettant de faire les tests

Environnement de compilation Voir `compiler/README.md`.

Sur les machines linux du département antlr4 et le runtime C++ sont installés. Vous pouvez utiliser le Makefile tel quel.

Si vous voulez travailler sur votre machine, les scripts script `compile_*.sh` vous montrent comment utiliser une autre installation de antlr4 et de son runtime C++.

Pour plus de simplicité, vous pouvez utiliser le script `compile_docker.sh` qui lancera antlr4, ainsi que la compilation du projet à travers docker. Ce script devrait marcher sur vos machines personnelles quel que le soit le système hôte à condition que docker soit correctement installé avec les droits adaptés. Inutile de l'utiliser sur les machines du département.

Attention, un binaire créé sous docker devra être exécuté dans docker. Il en va de même pour les binaires créés localement sur les machines linux du département.

Environnement de tests Nous parlons ici uniquement de tests fonctionnels. Nous fournissons un environnement de tests qui se trouve dans le sous-répertoire `tests` et qui permet d'automatiser les tests fonctionnels. 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. Le script `python` permettant de faire cela s'appelle `pld-test.py` et deux scripts bash permettent de l'invoquer : `test.sh` pour utiliser docker, et `test_if.sh` pour les machines du département.

Intégration continue Si vous souhaitez vous lancer dans l'intégration continue, un environnement de départ est fourni, qui utilise Gitlab et la même image docker que précédemment. Le fichier `.gitlab-ci.yml` qui contient les informations sur les différentes tâches est fourni. Il permet de découper en deux tâches distinctes : celle de *build* qui exécute antlr4, et compile votre projet, et celle de *test* qui lancera les scripts de tests décrits précédemment. Les deux tâches sont reliées par un *artefact* qui contient le binaire exécutable du compilateur. La tâche de tests produit aussi un artefact qui contient les résultats détaillés des tests. L'INSA ne propose pas encore d'hébergement de type Gitlab, vous aurez donc recours à un outil extérieur comme le site `https://about.gitlab.com/` qui permet de faire 2000 minutes d'intégration continue par mois. Attention, pour l'instant, les

pipelines d'intégration continue tels qu'ils sont présents dans le fichier s'exécutent à chaque nouvelle version poussée sur le dépôt. Cela peut vite engendrer beaucoup de compilations coûteuses et dépenser le budget en peu de temps.

3.2 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.

3.3 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 [1] 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 copicollables pour compiler votre projet sur les machines du département ;
- vos jeux de tests ;
- une documentation utilisateur et une documentation développeur.

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 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. À vous de trouver quelles sont les tâches parallélisables.

Certaines de ces tâches (les premières notamment) 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.

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

Cette année, nous vous fournissons cette étape.

La version 0.1 du compilateur doit reconnaitre 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 de programme :

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

Observation du comportement attendu :

1. Recopiez ce programme 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 à présent 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
```

Observation du comportement attendu :

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 le squelette de compilateur fourni se contente de produire le second listing à partir du premier. Ouvrez tous les fichiers et posez éventuellement des questions.

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

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

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).

4.2 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
   ```
   (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')
.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. 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.

4.3 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.

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

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

Vous pouvez aussi rendre le résultat de la tâche suivante.

4.4 Vérifications statiques

Cette tâche peut faire les traitements directement sur l'arbre de dérivation grâce à l'implémentation d'un visiteur, ou alors sur l'AST si vous avez décidé d'en implémenter un. 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 deux fois
— Une variable déclarée est utilisée au moins une fois

4.5 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). C'est donc très encouragé pour le hexanômes qui sont à l'aise.

Si vous passez à l'IR, nous vous encourageons à utiliser l'IR fournie dans `IR.h`, et à vous appuyer sur le poly. 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 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 la tâche suivante).

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 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.

La suite de ce document est rédigé en supposant une IR. Les choses peuvent être légèrement plus simple lorsqu'on fait le choix de ne pas en avoir, mais il faudra tout de même comprendre certaines notions telles que *basic block* et *control flow graph* pour les traduire dans votre compilateur.

4.6 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.

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.

4.7 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 ou 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 contenue de `retvalue` dans `%rax`, puis émet l'épilogue.

4.8 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.

4.9 Optimiser les expressions arithmétiques dans le front-end

Que ce soit au niveau de l'AST si vous en avez un ou directement au moment où vous visitez vos expressions afin d'en construire l'IR, 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)

— Propager des valeurs de variables connues. Une ligne affecte une variable, la suivante l'utilise, on connait donc sa valeur et on peut mettre une constante à la place (attention, plus compliqué à mettre en œuvre). Note : on peut aussi faire ce genre d'optimisation directement sur l'IR.

4.10 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).

4.11 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))]`.

4.12 Compiler les boucles `for`

Rien de bien méchant si vous connaissez leur sémantique.

4.13 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 so l'adresse est un multiple de 8.

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.

4.14 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.

A Annexe : éléments d'ABI des PC linux

A.1 Passage de paramètres

En mode 64-bits, on utilise les registres pour passer les paramètres [2].
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 [3].

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

2. Grosse différence avec l'ABI 32 bits, dans laquelle tous les arguments étaient passés sur la pile...
3. 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.
4. "r" comme *really extended*
5. "e" comme *extended*
6. "x" comme *extended*