PLD-Comp
Développement d'une chaîne complète de compilation

Dans ce projet, nous allons concevoir un compilateur pour un sous-ensemble du langage C (voir sujet ci-dessous) grâce à flex/bison et l'utilisation du langage C++.

1 Le langage

Il s'agit d'un sous-ensemble du langage C avec les éléments suivants :

- Types de données : restriction aux types `char`, `int32_t` et `int64_t`
- Initialisation d'une variable possible lors de sa déclaration
- Tableaux à une dimension
- Fonctions (déclaration et définition), possibilité (et obligation) d'utiliser le type retour `void` pour construire des procédures. La déclaration de fonction est une partie optionnelle du projet. Si vous choisissez de ne pas la traiter, alors vos exemples devront tous avoir une définition des fonctions *avant* leur utilisation.
- Structures de contrôle : `if`, `else`, `while`, `for`
- Structure de blocs grâce à { et }
- Expressions : tous les opérateurs du langage C y compris l'affectation
- Au choix (les deux si vous en avez le temps) :
  - Déclaration de variables dans n'importe quel contexte (pas seulement au début d'une fonction)
  - Possibilité de déclarer des variables globales
- Utilisation des fonctions standard `putchar` et `getchar` pour les entrées-sorties
- Constante caractère (avec la simple quote)
- Les chaînes de caractères pourront être représentées par des tableaux de `char` (pas de constantes chaînes de caractères)
- Un seul fichier source sans pré-processing (pas d'inclusions externes ni macros, toutefois les directives du pré-processeur devront être autorisées mais ignorées afin de garantir que la compilation par un autre compilateur soit possible. Exemple : inclusion de `stdint.h`)

Exemple de programme :

```
#include <stdint.h>

void main(void) {
  int32_t a;
  int32_t b;
  a=0;
  b=6;
  if (a*5+3*b==18)
  {
    putchar('o');
    putchar('k');
  }
}
```

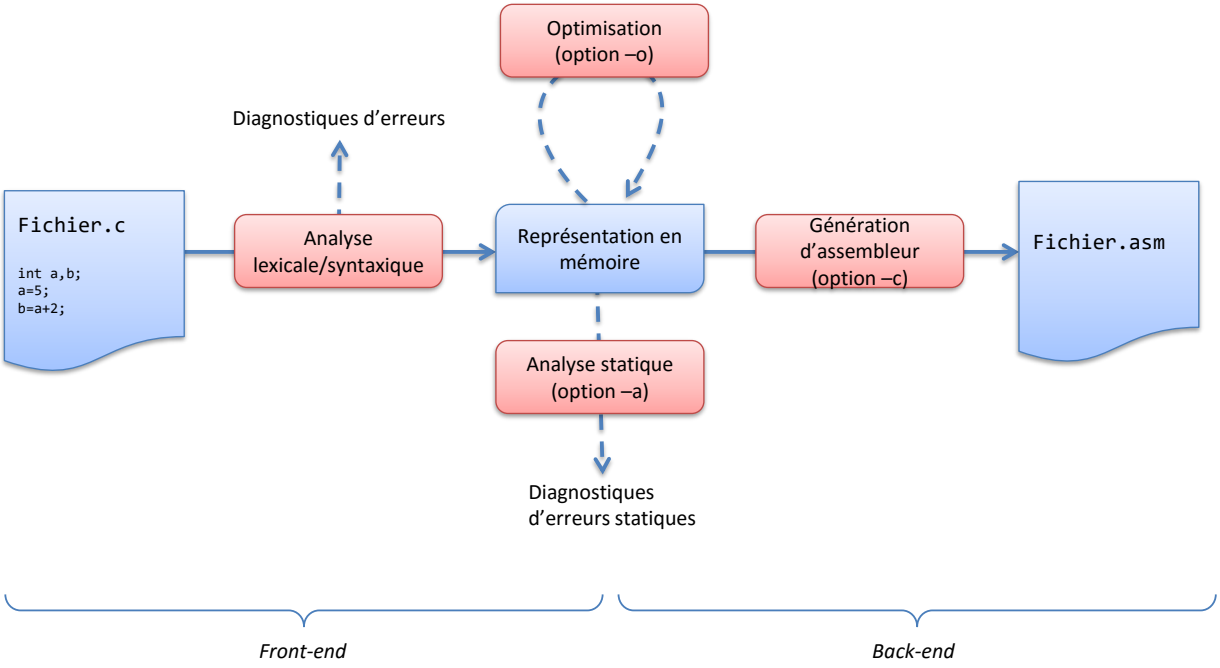
Plus particulièrement le langage qui devra être analysé **ne comprends pas les éléments suivants** :

- 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
— La structure de contrôle `switch..case`
— La structure de contrôle `do..while`

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. (facultatif) 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. (facultatif) 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 Étapes de travail

Les étapes sont ici données à titre indicatif, organisez vous comme vous le souhaitez. Certaines étapes sont parallélisables, à vous de les détecter. On peut par exemple très bien construire à la main une représentation d'un programme en faisant appel aux constructeurs (donc sans analyse syntaxique ni lexicale). Cela permet de tester certaines fonctionnalités avant l'intégration.

Écriture de la grammaire. Reprenez les éléments du langage exprimés dans la première section et déduisez-en une grammaire. Gardez en vue que vous devrez implémenter cette grammaire dans bison donc qu'il faut le plus possible être adapté à une analyse ascendante (privilégier les récursivités gauches par exemple).

Identification des expressions régulières. Pour chacun des éléments du langage (symboles terminaux), identifiez les expressions régulières associées. Si cela est nécessaire (expressions régulière ayant des éléments communs), mettez des priorités à chacune.

Conception des structures de données. Il s'agit ici de définir les structures qui contiennent le programme lui-même (arbre syntaxique abstrait) et les différents dictionnaires nécessaires (table des symboles, etc).

On demande un diagramme de classes UML.

Vous êtes invités à faire valider cette modélisation par l'équipe enseignante dans le but de ne pas perdre de temps.

Implémentation de l'analyseur avec flex/bison. À partir de la grammaire et des expressions régulières identifiées précédemment, produisez les fichiers flex et bison qui permettent d'analyser le code source afin d'en valider la construction grammaticale. Cette phase n'est qu'une transcription de votre grammaire. Si la grammaire est bien conçue, ce sera une formalité. Malheureusement, vous aurez sûrement des surprises et bison vous indiquera des conflits. Vous devez ou bien résoudre le conflit, ou bien expliquer pourquoi le choix fait par bison lors du conflit est le bon. Pour vous aider à comprendre les conflits, activez l'option `-v` de bison et allez voir le fichier `.output` ainsi généré. N'oubliez pas de paramétrer dans bison les priorités d'opérateurs qui feront disparaître énormément de conflits (directives `%left` et `%right`).

Construction de la représentation en mémoire. Complétez les règles bison par des actions afin de construire une représentation en mémoire du programme. À ce moment du projet, vous pourrez faire la vérification de la conformité de la signature des déclarations et définitions de fonctions (seulement si vous avez choisi d'autoriser les déclarations de fonctions).

Résolution de portée de variables. Dans une expression, lorsqu'une variable est utilisée, on ne connait que son nom. Le but de cette partie est de faire le lien entre les variables utilisées et leurs déclarations. Attention, il y a plusieurs types de variables dont l'emplacement mémoire sera différent au moment de la génération de code assembleur. Par exemple une variable de fonction n'est pas au même endroit qu'une variable globale ou qu'un paramètre de fonction. À l'issue de cette partie, toutes les ambigüités sont levées (par exemple, une variable de fonction qui masque un paramètre de fonction, ou une variable globale). Vous serez à même de détecter dans cette partie l'utilisation de variables non déclarées.

Vérification statique du programme (optionnel). Réfléchissez à l'algorithme qui va permettre de vérifier la validité du programme de manière statique. Voici les éléments qui sont vérifiables dans cette partie :

- Une variable est utilisée (en partie droite d'une affectation ou dans une opération d'écriture) sans avoir jamais été affectée.
- Une variable a été déclarée et jamais affectée ou utilisée.

Typage des noeuds intermédiaires. Il s'agit dans cette partie de donner à chaque nœud intermédiaire d'expression un type en fonction du type de ses opérandes. Cela dépend de l'opérateur aussi. Exemple, une addition ne sera pas traitée pareil qu'une affectation.

À la moitié du projet (4 séances), vous devriez avoir terminé toutes les tâches précédentes, et donc avoir le *front-end* opérationnel. Vous pouvez maintenant passer au *back-end*.

Définition des structures de données pour une représentation intermédiaire linéaire. Il s'agit de définir une structure de donnée qui reste abstraite mais se rapproche de l'assembleur. Ce sera un un graphe orienté d'instructions élémentaires à trois adresses (deux variables opérandes, une variable destination). Dans ce graphe, la plupart des instructions élémentaires n'ont qu'un successeur, seul le saut conditionnel a deux successeurs possibles. Les dictionnaires seront partagés avec l'AST.

Transformation de l'AST vers la représentation linéaire. Le travail ici consiste à "désucrer" le code initial :

— démonter les expressions en séquences de code trois adresse
— démonter les `for`, `while` et `if` en sauts conditionnels
— mettre toutes les variables à plat dans une seule portée

Ceci se fait par un parcours de l'AST. Dans ce processus, on créera autant de variables intermédiaires que nécessaire.

Génération de code assembleur à partir de la représentation linéaire. On produira dans cette partie de l'assembleur, a priori pour un processeur de PC (x86 64-bits), mais l'une des cibles suivantes sera la bienvenue :

— assembleur ARM (par exemple pour Raspberry Pi)
— assembleur MSP430
— java bytecode (sans utiliser le modèle objet)

On demande de l'assembleur sous forme textuelle qui sera assemblé en code machine par `as`. Vous avez donc droit aux *labels* et autres facilités offertes par GNU `as`. On ne demande pas d'allocation de registre ici.

Gestion des appels de procédures. Pour les programmes qui ont plusieurs procédures, il faudra gérer la pile des appels, sur laquelle seront placées toutes les variables locales de chaque procédure.

Outil en ligne de commande - intégration. Il s'agit là de concevoir et implémenter l'outil qui va permettre d'interfacer l'analyseur.

Optimisations (facultatif mais recommandé). Certaines optimisations peuvent se faire sur l'AST :

— propagation de constantes : une expression dont toutes les opérandes sont des constantes sera remplacée par la valeur résultante.
— éléments neutres : pour chaque opérateur qui possède un élément neutre (l'addition le 0, la multiplication le 1, *etc.*), les opérations correspondantes peuvent être supprimées lorsqu'elles surviennent.

Certaines optimisations se font sur la représentation linéaire :

— destruction des instructions redondantes (optimisation à fenêtre)
— calcul de vivacité des variables et allocation des registres

Et voici une extension/optimisation qui touche à la fois *front-end* et *back-end* : le support du `switch/case` du langage C, implémenté par une table de sauts. C'est une optimisation car, sur des gros `switch`, la table des sauts va bien plus vite que des `if` imbriqués.

4 Livrables (zéro papier)

Le projet se déroulera en deux temps avec un document de conception à mi-parcours à déposer sur moodle qui reprendra les éléments suivants :

— la grammaire du langage que vous avez utilisée
— la description des structures de données sous forme d'un diagramme de classes
— la liste des fonctionnalités implémentées

À l'issue du projet, un livrable de réalisation sera déposé sur moodle. Ce dossier de réalisation (application testable sur les machines LINUX du département) sera déposé sur moodle sous la forme d'une archive zip [1]. Merci de nettoyer l'archive pour ne pas qu'elle contienne des binaires ni des répertoires cachés. Le zip contiendra :

— tous les sources de votre application
— le `makefile` avec une cible par défaut qui crée l'exécutable et une cible « test » qui lance les jeux de tests
— vos jeux de tests

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

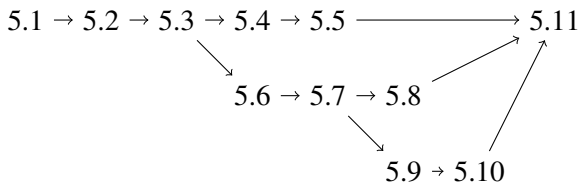
5 Phasage recommandé pour le back-end

On peut distinguer en gros trois tâches parallèles :
— Tâche 1 : Définir l'IR et transformer l'AST en IR
— Tâche 2 : Transformer l'IR en assembleur x86
— Tâche 3 : Optimiser l'IR

On va ignorer la tâche 3 tant que le compilateur n'est pas complètement opérationnel.

Et on va essayer de développer les tâches 1 et 2 en parallèle pour que votre compilateur soit toujours opérationnel de bout en bout, mais sur des programmes de plus en plus compliqués.

Le graphe de dépendances des différentes sous-tâches obligatoires est en gros le suivant :



5.1 Générer un squelette vide

Vous trouverez dans le répertoire 1-Minimal-Skeleton un fichier assembleur `main.s` et le Makefile qui l'assemble et le linke avec la `glibc`.

Votre compilateur doit produire `main.s`, puis l'assembler comme le montre ce Makefile.
— Lancez ce makefile.
— Modifiez `main.s` et observez le résultat sont prises en compte.
— Mettez en place un squelette de compilateur qui produit (en gros) ce squelette.

Le programme qui doit valider cette étape :

```
void main() {
}
```

5.2 Comprendre l'enregistrement d'activation et l'ABI

Allez lire la section B en détail et faites les expériences décrites.

Du boulot pour la tâche 2 :
— écrivez une fonction `gen_prologue` qui prend un nom de fonction et une taille d'AR à réserver.
— écrivez une fonction `gen_epilogue`.

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

Il y aura du boulot pour T1 aussi : il devra calculer quelle est la taille de l'AR à réserver. Dans un premier temps, on va y metttre toutes les variables locales, d'entrée et de retour :

— associez à chaque variable, dans votre table des symboles, un index dans l'AR. Remarque : les paramètres sont fâcheux puisqu'ils sont dans des registres. La solution la plus simple est que le prologue les copie vers des variables locales, après quoi ils auront un index comme tout le monde.
— une fois ceci fait vous connaissez la taille totale (à passer à `gen_prologue`)

Dans un premier temps vous êtes très fortement encouragés à ne manipuler que des index multiples de 8 octets quitte à perdre de la place dans votre AR. Autrement dit, donnez 8 octets à chaque char et à chaque int32. Dans un second temps, vous pourrez optimiser l'allocation de vos différentes variables locales dans l'AR. L'ABI n'impose rien à ce sujet, mais l'ISA impose que les variables 64bits aient des adresses multiples de 8 et les variables 32 bits des adresses multiples de 4. **En particulier rsp et rbp doivent toujours être des multiples de 8**.

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

```
#include<inttypes.h>
void main() {
  char b;
  int32_t a; /* doit avoir une adresse multiple de 4 */
  char c;
  int64_t d; /* doit avoir une adresse multiple de 8 */
  a=5;
  b=6;
  c=7;
  d=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.

5.3 Compiler un programme qui fait un putchar()

Le but de cette étape est de compiler

```
#include<inttypes.h>
void main() {
putchar('O');
putchar('K');
putchar(' ');
}
```

Il faut avouer que c'est une marche assez haute, mais une fois ceci fait on aura les appels de fonctions.

La génération de code pour une constante est un cas particulier de génération de code pour une expression : mettez en place les squelettes de méthodes `genIR(CFG* cfg)` sur votre AST.

Pour compiler la constante, il faut créer une nouvelle variable intermédiaire dans la table des symboles : on y accède par le CFG passé en paramètre.

Ensuite, le code IR est juste constitué d'une seule instruction, et le `genAsm()` pour ce code produit aussi une seule instruction.

L'appel de fonction peut générer pas mal de code (mais pas dans notre exemple).

— `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 un seul `call` qui prend en paramètre cette liste de variables.
— `genAsm()` émet l'assembleur qui recopie chaque variable dans un registre suivant la section B. Puis il émet une instruction `call`.

Quand vous avez réussi à passer cette étape, le reste devrait aller plus vite et être plus intéressant.

5.4 Compiler les appels de fonction en général

Un jour ou l'autre il faudra vérifier que vos appels de fonctions marchent pour 2, 3 arguments.

5.5 Compiler le retour de valeur

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

5.6 Affectation, lvalues et rvalues

L'étape suivante est d'arriver à compiler le programme suivant :

```
void main() {
  char a, b, c;
  a='O';
  b=a;
  c='K';
  a='\n';
  putchar(b);
  putchar(c);
  putchar(a);
}
```

Voir le poly pour vous rafraîchir sur le code à générer pour les parties droites et gauche du =.

Le travail commun à T1 et T2 est de mettre en place la structure de donnée pour l'IR. Dans un premier temps on pourra se contenter de développer le bloc de base (BB), qui correspond à du code linéaire, sans test ni branchement. Le graphe de contrôle de flot (qui est un graphe de BB) devra être mis en place à partir de la section 5.9.

5.7 Compiler des expressions

Sur le fond c'est un parcours d'arbre, voir le poly. La nouveauté est que ce parcours peut créer des variables temporaires pour les calcul intermédiaires. Ajoutez les dans votre table des symboles à la suite de vos variables locales.

5.8 Compiler des tableaux

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

5.9 Compiler le contrôle de flot

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.

5.10 Compiler les boucles `for`

Rien de bien méchant si vous connaissez la sémantique.

5.11 Test sur des programmes complexes

A ce stade votre compilateur doit être capable de compiler une factorielle récursive, la suite de Syracuse et la fonction d'Ackermann.

A Spécification de la représentation intermédiaire

L'IR est à mi-chemin entre le C et un assembleur.

Ce qui ressemble à de l'assembleur :

- Elle ne connait que les types de base (int64, int32, int8). En particulier les adresse sont des entiers (int64 sur notre cible 64-bits), et les accès au tableaux ont été transformés en calculs sur les adresses.
- Il y a un registre particulier `!bp` qui contient le pointeur de base (%rbp sur notre cible). Les adresses des variables locales et paramètres sont définies par un offset relatif à ce `!bp`.
- Elle ne connait que des instructions à 1 ou 2 opérandes (sauf le `call`)
- Elle ne connait que le saut et le saut conditionnel.

Ce qui ressemble à du C :

- Elle travaille avec un nombre arbitraire de registres : le registre spécial `!bp`, et les registres normaux `!r1` jusqu'à `!r`∞. On peut demander la création d'un nouveau registre lors de la construction de l'IR.
- Dans un premier temps, tous ces registres seront mappés vers autant de nouveaux symboles dans la table des symboles (et donc autant de cases mémoires dans l'enregistrement d'activation)[2]. C'est pour les différencier des variables C (qui vivent aussi dans la table des symboles) que les noms des registres commence par un `!`. Si j'avais mis un `%` vous auriez été capables de confondre avec les registres du Pentium.
- L'instruction `call` a un nombre arbitraire de paramètres, mais ces paramètres sont tous des registres.

Ce qui ne ressemble à rien du tout : le graphe de flot de contrôle.

- Les instructions d'une fonction sont groupées en séquences linéaires appelées *basic block* ou BB. Dans un BB, toutes les instructions s'exécutent en séquence.
- Le CFG d'une fonction se compose d'un BB d'entrée (il générera le prologue), d'un BB de sortie (il générera l'épilogue), et d'un nombre quelconque de BBs intermédiaires.
- Un BB commence par une étiquette (une destination de saut) et termine
  - par un saut à un autre BB,
  - ou par deux sauts choisis en fonction du résultat d'un test,
  - ou par le retour de la fonction si c'est le BB de sortie.

A.1 Jeu d'instructions

Voici le jeu d'instruction minimal à utiliser.

Mnémonique	Description
reg <- const	met une constante dans un registre
reg <- reg1+reg2	operation binaire (aussi +, -, *, ainsi que les comparaisons <, <=, >, >=, =)
call label (reg1, reg2, reg3...)	appel de sous-routine
reg1 <- (reg2)	lecture mémoire : le contenu de la case d'adresse reg2 est placé dans reg1
(reg1) <- reg2	écriture mémoire.
jump label	uniquement comme dernière instruction d'un BB
if reg then label1 else label2	branchement, uniquement comme dernière instruction d'un BB

2. En fait, avec nos restrictions au C, on peut même considérer aussi toutes les variables locales et paramètres du programme initial comme des registres de l'IR. Cela simplifie la génération de code, mais attention, il faudra être soigneux pour l'affectation : la variable à gauche est une lvalue, il faudra prendre son adresse.

Vous avez remarqué que les résultats des comparaisons sont dans des registres, pas dans de drapeaux comme sur un vrai processeur.

Dans le Cooper et Torczon, l'IR n'a pas de call, c'est glissé sous le tapis. Par contre ils ont aussi des opérations binaires avec une constante en place de reg2, et un adressage indexé par une constante... Restons sur une IR minimale : c'est une optimisation à implémenter ensuite de fusionner plusieurs instructions IR en une vraie instruction assembleur complexe.

A.2 La classe IRInstr

... a comme attributs et méthodes
— un mnémonique du tableau ci-dessus,
— ses paramètres (les reg, const et label du tableau),
— un pointeur vers le CFG qui la contient (qui fournira entre autres la table des symboles),
— une méthode de génération d'assembleur.

A.3 La classe BasicBlock

... a comme attributs et méthodes
— une séquence de IRInstr,
— un pointeur vers le CFG qui la contient,
— une méthode de génération d'assembleur.

A.4 La classe CFG

... a comme attributs et méthodes
— une table des symboles, et les différentes méthodes qui y accèdent,
— une liste de BasicBlocks,
— un pointeur vers l'AST qu'elle implémente,
— une méthode de génération d'assembleur.

B L'ABI des PC linux de la salle 219

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

```
#include<inttypes.h>
void toto() {
  int64_t x=1;
  int64_t y=2;
  int64_t 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.

Cette section vous explique tout cela, essayez de tout comprendre en détail et appelez dès que vous coincez.

Nous vous encourageons aussi à aller consulter la source, qui se trouve sur le Ternet si on cherche :
System V Application Binary Interface, AMD64 Architecture Processor Supplement

B.1 L'enregistrement d'activation

L'enregistrement d'activation, c'est la zone de mémoire dans laquelle sont allouées les variables locales à l'entrée dans une procédure/fonction.

On accède aux variables locales relativement à `%rbp`, qui est le *base pointer*. En assembleur x86, `-8(%rbp)` signifie `*(%rbp-8)` en syntaxe C. Voici à quoi ressemble la pile lors de l'exécution d'une fonction `toto` qui a trois variables locales de 64 bits et qui a été appelé par `tata` :

		AR
20(%rbp)	éventuel paramètre sur la pile	de
16(%rbp)	éventuel paramètre sur la pile	tata
8(%rbp)	adresse de retour de la procédure	
0(%rbp)	valeur précédente de %rbp	AR
-8(%rbp)	variable 1	de
-16(%rbp)	variable 2	toto
0(%rsp) = -20(%rbp)	variable 3	

Exercice : retrouvez sur ce dessin l'oeuvre des instructions de prologue et d'épilogue. On rappelle que l'adresse de retour de la procédure a été empilée par le `call`.

Exercice : retrouvez dans le .s les emplacements des variables x, y et z.

Une remarque : dans toute la suite du projet, les variables locales ne seront pas seulement les variables de votre programme, il y aura aussi des variables implicites créées pour recevoir les calculs intermédiaires...

B.2 Passage de paramètres

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

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.

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

— Once all registers are assigned, the arguments are passed in memory. They are pushed on the stack in reversed (right-to-left) order [4].

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.

La valeur de retour est passée dans `%rax`.

Remarque : en mode 32 bits les registres s'appellent `esp`, `ebp`, etc. Le "e" voulait déjà dire "extended" (to 32 bits). En mode 64 bits ils s'appellent `rsp`, `rbp`, etc. Dans les deux cas `sp` c'est *Stack Pointer*, et `bp` c'est *Base Pointer*.

B.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.

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.