Compilation

Guillaume Bonfante

Département d'Informatique - École des Mines de Nancy

Contents

1 Introduction

Dans ce cours, nous allons implémenter un petit compilateur en partant d'un sous langage de C à destination du processeur x86_64. Chacune des sections correspond peu ou prou à une séance de cours. Pour toutes les séances sauf la première, il est attendu que vous disposiez de

- une machine Linux, éventuellement virtuelle, sous x86_64, la machine étant dotée de :
- un compilateur `gcc`,
- le compilateur `nasm`,
- le langage `python`,
- avec la bibliothèque `Lark`.

Le contenu de ce cours pourra être retrouvé en grande partie dans le livre de Neil Jones, *Computability and complexity from a programming perspective*, [Jon97], pour toute la partie concernant la sémantique. Pour la partie concernant la compilation, LE livre est le dragoon book [ALSU06] d'Alfred Aho, Monica Lam, Ravi Sethi et Jeffrey Ullman. Pour information, un millier de pages en perspective.

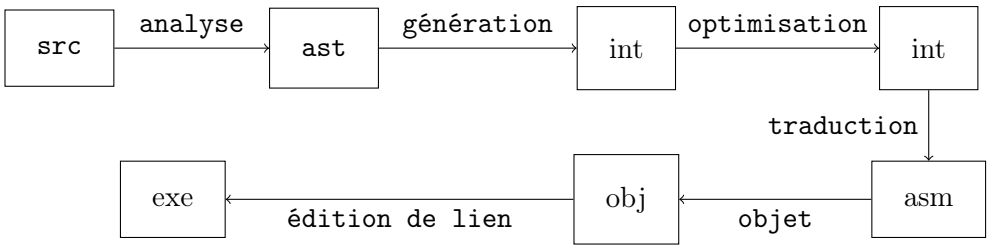
Autre source d'inspiration de ce cours, celui de Xavier Leroy au collège de France, et tout particulièrement la saison 1, `https://www.college-de-france.fr/chaire/xavier-leroy-sciences-du-logiciel-chaire-statutaire/events`.

Ici, le contenu est adapté à un cours de cinq séances de trois heures. De ce fait, le style est assez direct. Les deux livres mentionnés ci-dessus sont une bonne base pour aller plus loin.

La compilation consiste à transformer les programmes d'un langage dans un autre. De ce fait, on va avoir besoin de décrire deux langages (de programmation). Le premier sera le nano-C décrit dans la section à venir. Le second, ce sera notre cible, il s'agit du langage x86_64, le langage machine de la plupart des machines de bureau.

1.1 Une chaine de compilation

Globalement, la compilation va prendre la forme d'une chaine de transformations plus élémentaires. Ici, nous décrivons une chaîne simple en 6 étapes. Les compilateurs modernes sont beaucoup plus sophistiqués.



Analyse Le code source va être transformé en un objet qu'on nomme *arbre de syntaxe abstraite*, objet dont le rôle est de ne conserver que la structure fondamentale du code source, celle qui *fait sens*. En d'autres termes, il s'agit d'oublier tous les détails inutiles (les signes de ponctuation, les espaces, etc). C'est l'étape d'analyse syntaxique. C'est à ce moment qu'on va pouvoir recevoir un message d'erreur comme « il manque une parenthèse ouvrante à la ligne 18 ». Nous implémenterons l'analyse syntaxique à l'aide de la bibliothèque `Lark` en `python`.

Génération La deuxième étape transforme l'arbre de syntaxe abstraite en un code dit intermédiaire. Il s'agit d'instructions de bas niveau correspondant à une machine abstraite. Par bas niveau, on entend le fait que la machine abstraite a un fonctionnement proche des processeurs actuels, ou plutôt que la traduction des instructions de ce langage vers le langage de la machine est raisonnablement simple. Cette abstraction est fort utile dans un monde où il existe des processeurs différents (par exemple, x86_64 et ARM). Dans la vie courante, on trouvera par exemple `LLVM` qui est assez commun. Pour nous simplifier la vie dans ce cours, nous ne distinguerons pas les deux langages : `nasm` $\simeq$ `int`.

Nous utiliserons le langage `python` pour réaliser cette étape.

Optimisation Le code produit à l'étape précédente peut être optimisé. Les optimisations sont obtenues pas des raffinements successifs. La chaine des compilateurs modernes est assez compliquée. On retrouvera les couches d'optimisation de `gcc` à l'aide de l'option `-fverbose-asm`. Le cours d'Architecture revient plus en détail sur cette partie.

Traduction Le code intermédiaire est transformé en assembleur. Comme nous avons choisi le même langage, cette étape sera triviale dans notre cas.

Pour retrouver l'assembleur x86_64 produit par `gcc`, on emploie l'option "-S".

Objet L'étape consiste à construire un code objet. C'est à dire un fichier binaire qui contient les instructions machines du programme à venir ainsi que les données (qu'on retrouve dans les variables globales, les constantes, les chaines de caractères, etc). Cette étape sera réalisée avec l'outil `nasm`.

Édition de lien Enfin la dernière étape. Celle-ci construit le code binaire final. Elle va intégrer les liens (cf cours sur le C). Nous utiliserons `gcc` pour cette étape.

1.2 Plan général du compilateur

En suivant ce qui précède, le code que nous allons construire va ressembler schématiquement au code suivant. Nous nous arrêterons à la boite `asm` du schéma précédent. La suite sera réalisée à l'aide des outils externes : `nasm` et `gcc`.

```
def get_source(filename : str) -> str:
  """
  get the source code contained in filename
  """

def get_ast(file_content : str) -> ast:
  """
  get the ast from the source code
  """

def compile(ast) -> asm:
  """
  compile to assembly code
  """

def save(asm, filename : str):
  """
  save assembly code to some filename
  """

if __name__ == "__main__":
  ast = get_ast(get_source(sys.argv[1]))
  asm = compile(ast)
  save(asm, sys.argv[2])
```

Les deux fonctions `get_source` et `save` sont des fonctions usuelles, elles ne seront pas considérée outre mesure. La Section 3 donne les moyens d'implanter la fonction `get_ast`. Les deux sections 5 et 6 traitent de la fonction `compile`.

1.3 Notations

Nous utilisons par la suite un certain nombre de notations. Essayons de toutes les grouper ici.

Étant donné deux entiers $a \leq b$, on note $[a, \dots, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$. Et $[a, \dots, b[= \{x \in \mathbb{Z} \mid a \leq x < b\}$.

Étant donné deux ensembles X, Y et $y \in Y$, on note $\underline{y} : X \to Y$ la fonction $x \in X \mapsto y$. Étant donné une fonction $f : X \to Y$, $x \in X$ et $y \in Y$, on note $f[x \mapsto y] : X \to Y$ la fonction définie par

$$\begin{cases} x' \mapsto f(x') & \text{pour tout } x' \in X \text{ tel que } x' \neq x \\ x \mapsto y & \text{sinon.} \end{cases}$$

Étant donné une fonction $f : X \to Y$, des éléments $x_1, \dots, x_k \in X$ et $y_1, \dots, y_k \in Y$, on peut composer la notation $f[x_1 \mapsto y_1, x_2 \mapsto y_2, \dots, x_k \mapsto y_k] =$

$$((f[x_1 \mapsto y_1])[x_2 \mapsto y_2] \cdots)\,[x_k \mapsto y_k].$$

Quand les éléments $x_1, \dots, x_k$ sont deux à deux distincts, l'ordre importe peu. Par exemple :

$$f[x_1 \mapsto y_1, x_2 \mapsto y_2] = f[x_2 \mapsto y_2, x_1 \mapsto y_1].$$

Nous notons pour la suite `BYTE` l'ensemble des octets ("byte" en anglais), c'est-à-dire les entiers entre 0 et $2^8 - 1$, `WORD` l'ensemble des mots ("word"), les entiers entre 0 et $2^{16} - 1$, `DWORD` l'ensemble des mots doubles, les entiers entre 0 et $2^{32} - 1$ et `QWORD` l'ensemble des mots quadruples, les entiers entre 0 et $2^{64} - 1$.

Etant donné une fonction mem : `QWORD` $\to$ `BYTE`, et deux entiers $a \leq b$, on utilise alternativement les notations $mem[a] = mem(a)$ selon qu'on a un penchant d'informaticien ou de mathématicien. Étant donnés deux entiers $a \leq b$, $mem[a : b] = (mem[a], \dots, mem[b - 1])$ est la suite des éléments de mem entre a et b, la suite étant vide si $a = b$.

Questions/observations
Par la suite, vous verrez des blocs en jaune. Ils correspondent à des questions/observations qu'on m'a posées/faites auparavant.

Formalisme
Les blocs qui sont en rose abordent le problème sous l'angle mathématique/abstrait. Ils sont rédigés à la manière de ce qu'on peut trouver dans un article scientifique. La section notation aurait pu être entièrement en rose. C'eut été un peu lourd.

2 Le nano-C

Dans cette section, on présente le nano-C dans sa version (partiellement) abstraite. C'est la syntaxe qu'on utilise pour raisonner à propos du langage. Par la suite, nous serons amenés à compiler le langage, et pour cela, nous aurons besoin d'une syntaxe concrête présentée plus loin. La syntaxe que nous introduisons ici est la version mathématique qui correspond aux arbres de syntaxe abstraite dans l'implémentation du compilateur.

Mais avant de commencer, nous supposons donné l'ensemble $\mathbb{Z}$ des entiers relatifs et $\mathcal{V} = \{\mathtt{X}, \mathtt{Y}, \ldots\}$ un ensemble de variables. Enfin, on suppose donné un ensemble d'opérations binaires $OP = \{+, \geq, \ldots\}$ sur $\mathbb{Z}$.

Le nano-C est composé de trois catégories d'objets. D'abord, les expressions qui désignent les objets qu'on peut calculer directement : ce qui est à droite du signe "=" des affectations.

$$\mathcal{E} ::= \mathbb{Z} \mid \mathcal{V} \mid \mathcal{E}\ OP\ \mathcal{E}$$

Comment lire ce qui précède ?
Une expression est soit un entier, soit une variable, soit la somme, le produit (ou tout autre opération binaire) de deux sous-expressions. Et... il n'y a pas d'autres expressions.

Mathématiquement, les expressions sont le plus petit ensemble qui peut être construit selon les trois clauses. Cela nous donne un principe d'induction (de récursion, en langage courant).

Pour prouver qu'une propriété est vraie pour toutes les expressions, il suffit de montrer :

- qu'elle est vraie pour les entiers,
- qu'elle est vraie pour les variables,

- et que pour toutes sous-expressions E_1 et E_2 pour lesquelles on suppose qu'elle est vraie, pour tout opérateur $\otimes \in OP$, la propriété est vraie pour $E_1 \otimes E_2$.

On va retrouver le caractère inductif de la construction des expressions pendant l'implémentation du générateur. Les fonctions qui s'appliquent aux expressions auront une furieuse tendance à être récursives.

Ensuite, nous avons des commandes (ou "statement" selon les auteurs) :

$$\begin{array}{rll} & \mathcal{V} = \mathcal{E} & | \text{ affectation} \\ & \texttt{while } (\mathcal{E})\{\mathcal{C}\ \} & | \text{ boucle while} \\ \mathcal{C} ::= & \texttt{if } (\mathcal{E})\{\mathcal{C}\} & | \text{ test} \\ & \mathcal{C};\mathcal{C} & | \text{ sequence d'instructions} \\ & \texttt{printf } (\mathcal{E}) & \text{ un print} \end{array}$$

Enfin, on considère le programme lui-même.

$$\mathcal{P} ::= \texttt{main}(\mathcal{V}^*)\{\mathcal{C}; \texttt{return}(\mathcal{E})\}$$

où $\mathcal{V}^*$ désigne une suite finie de variables.

Tout est récursif
Comme les expressions, les commandes et les programmes (dans une version un peu dégénérée) sont des objets inductifs, et donc susceptibles de raisonnement par induction.

2.1 Discussions autour du langage

Le langage est très réduit, son rôle est de présenter l'essentiel des concepts. Et c'est une bonne base pour la suite. Il manque de nombreuses constructions, constructions qui seront introduites en projet.

Le langage n'est pas typé de manière explicite. Toutefois, on utilise des entiers dans les expressions. On peut donc voir le langage comme typé, mais avec un type unique, les entiers. On reviendra sur la question dans la section sur la sémantique.

A propos des entiers, ici, on parle d'entiers naturels, pas d'entiers machines. Mais ce qui suit pourrait être adapté sans problème à un cadre où les entiers seraient signés, sur 32 bits, etc.

Le langage est décrit de manière semi-abstraite pour raisons de lisibilité. Dans une version encore plus abstraite, nous ne verrions pas des parenthèses, des accolades, ni des mots comme `return`.

Le test `if` ne contient pas de clause `else`. Est ce un oubli ? Ou juste parce qu'on peut simuler une telle commande $\texttt{if}(E)\{C_1\}\ \texttt{else}\ \{C_2\}$ de la manière suivante :

$$\texttt{Z} = E; \texttt{if}(\texttt{Z})\{C_1\}; \texttt{if}(\max(0, 1 - \texttt{Z} * \texttt{Z}))\{C_2\}$$

où `Z` est une variable fraîche (c'est à dire qui n'apparaît pas dans le reste du programme).

Les boucles `for` se calculent facilement avec les boucles `while`. Notons que les boucles `for` ont tendance à se généraliser dans les langages modernes au point qu'elles peuvent remplacer la construction `while`, par exemple en GO.

Par la suite, pour présenter un programme, nous rajouterons des retours à la ligne pour raison de lisibilité et quelques fois, nous rajouterons même des ';' et des parenthèses. Enfin, nous rajouterons des indications de type, même si cela ne sert à rien. De fait, ce sera bien utile pour ceux qui auront à implémenter la vérification de type par la suite.

2.2 Sémantique

L'objectif de la sémantique est de décrire formellement le *sens* du langage, à préciser ce que "calculer" veut dire. L'intérêt de la sémantique est d'expliciter le calcul. Par exemple, quand on dit que 0 signifie 'faux' tandis que toute autre valeur indique 'vrai', c'est une convention et cette convention sera spécifiée (autrement dit explicite) dans notre sémantique.

Si on revient à notre objectif de compiler un langage vers un autre. Il est attendu que la traduction soit fidèle. C'est à dire que la sémantique du programme source soit la même que la sémantique du programme cible. Plus prosaïquement, il faut comprendre ce que le programme source est sensé faire pour le traduire. C'est l'objet de la sémantique.

Il existe plusieurs catégories de sémantiques, citons les sémantiques dénotationnelles, axiomatiques, opérationnelles. La première vise à attribuer à chaque programme un objet mathématique. La seconde vise à exprimer les axiomes sous-jacents de chaque constructions, voir les contributions de Tony Hoare [Hoa69] et Floyd [Flo67]. Enfin, la troisième rentre plus concrètement sur les procédés de calcul.

Ici, vu la simplicité du nano-C, nous allons pouvoir nous permettre de décrire le langage à l'aide d'une sémantique dénotationnelle. L'étendre à des

langages plus sophistiqués peut vite devenir une gageure.

Comme nous avons vu au dessus, le nano-C s'appuie sur des constructions préalables : les entiers, les variables et les opérations binaires. Pour ces dernières, on va écrire "+" pour le symbole et "$\underline{+}$" pour la fonction mathématique. Cela peut paraître couper les cheveux en quatre, mais il y a bien une distinction entre le mot et la chose.

Pour décrire la sémantique, nous allons nous appuyer sur la notion de valuation. Une valuation attribue à chaque variable sa valeur.

Valuation
Une *valuation* est une fonction $\mathcal{V} \to \mathbb{Z}$. On note $\mathfrak{S}$ l'ensemble de toutes les valuations.

Le $\mathfrak{S}$ de la valuation vient de 'store', une autre dénomination possible du même concept.

Sémantique des expressions
Etant donné une expression $E \in \mathcal{E}$, sa sémantique est une fonction dans $\mathfrak{S} \to \mathbb{Z}$, dénotée $[\![E]\!]_{\mathcal{E}}$. La définition procède par induction sur la construction de l'expression. Dans les équations qui suivent, $\sigma \in \mathfrak{S}$.

$$\begin{cases} n \in \mathbb{Z} & [\![n]\!]_{\mathcal{E}}(\sigma) = n \\ \mathtt{X} \in \mathcal{V} & [\![\mathtt{X}]\!]_{\mathcal{E}}(\sigma) = \sigma(\mathtt{X}) \\ E_1 \in \mathcal{E}, E_2 \in \mathcal{E} & [\![E_1 \otimes E_2]\!]_{\mathcal{E}}(\sigma) = [\![E_1]\!]_{\mathcal{E}}(\sigma) \mathbin{\underline{\otimes}} [\![E_2]\!]_{\mathcal{E}}(\sigma) \end{cases}$$

Exceptions
Le cas des divisions par 0 n'est pas géré ici. Pour cause de place, on va "oublier" ce problème. La bonne méthode consiste à envisager des fonctions partielles. Et par la suite, la sémantiques du programme est non définie si l'on rencontre une telle évaluation.

Sémantique des commandes
Etant donnée une commande $C \in \mathcal{C}$, sa sémantique est une fonction $\mathfrak{S} \to \mathfrak{S}$, dénotée $[\![C]\!]_{\mathcal{C}}$. Par induction sur les commandes, σ désignant toujours une valuation dans les équations,

- Étant donnés $\mathtt{X} \in \mathcal{V}, E \in \mathcal{E}$, on définit :

$$[\![\mathtt{X} = E]\!]_{\mathcal{C}}(\sigma) = \sigma[\mathtt{X} \mapsto [\![E]\!]_{\mathcal{E}}(\sigma)]$$

- Étant donnés $E \in \mathcal{E}, C \in \mathcal{C}$, posons $W = \texttt{while}\ (E)\ \{C\}$, on définit :

$$[\![W]\!]_{\mathcal{C}}(\sigma) = \begin{cases} \sigma & \text{si } [\![E]\!]_{\mathcal{E}}(\sigma) = 0 \\ [\![W]\!]_{\mathcal{C}}([\![C]\!]_{\mathcal{C}}(\sigma)) & \text{sinon} \end{cases}$$

- Étant donnés $E \in \mathcal{E}, C \in \mathcal{C}$, posons $I = \texttt{if}\ (E)\ \{C\}$, on définit :

$$[\![I]\!]_{\mathcal{C}}(\sigma) = \begin{cases} \sigma & \text{si } [\![E]\!]_{\mathcal{E}}(\sigma) = 0 \\ [\![C]\!]_{\mathcal{C}}(\sigma) & \text{sinon} \end{cases}$$

- Étant donnés $C_1, C_2 \in \mathcal{C}$,

$$[\![C_1; C_2]\!]_{\mathcal{C}}(\sigma) = [\![C_2]\!]_{\mathcal{C}}([\![C_1]\!]_{\mathcal{C}}(\sigma))$$

- Étant donnée $E \in \mathcal{E}$,

$$[\![\texttt{printf}(E)]\!](\sigma) = \sigma$$

Retour sur $[\![W]\!]_{\mathcal{C}}$

La définition de la fonction $[\![W]\!]_{\mathcal{C}}$ est récursive. Est elle correcte ? Cette question a été l'objet de nombreuses recherches dans les années 70. Les sémantiques visaient à donner du sens à une telle définition. Citons ici le travail de Dana Scott et Christopher Strachey [SS71].

Il reste qu'il y a un moment où la définition peut être problématique. C'est précisément quand la boucle est infinie.

Que fait `printf`?
La commande `printf` ne fait pas grand chose dans la définition. Cela vient de notre modèle qui est trop faible. Celui-ci ne rend compte que

des valeurs des variables. Nous n'avons pas modélisé "stdout". Comment rendre compte du comportement de `printf`? La solution habituelle s'appuie sur la notion mathématique de monade. Mais cela va bien trop loin pour ce cours.

Sémantique des programmes
Etant donné un programme $P = \texttt{main}(\texttt{X}_1, \ldots, \texttt{X}_k)\{C; \texttt{return}(E)\}$, sa sémantique est la fonction $\mathbb{Z}^k \to \mathbb{Z}$, dénotée $[\![P]\!]_{\mathcal{P}}$, définie comme ceci :

$$(n_1, \ldots, n_k) \mapsto [\![E]\!]_{\mathcal{E}}([\![C]\!]_{\mathcal{C}}\,(\underline{0}[\texttt{X}_1 \mapsto n_1, \ldots, \texttt{X}_k \mapsto n_k]))$$

Valeurs initiales
On pourra noter que dans la définition qui précède, la valeur par défaut des variables est 0 qu'on retrouve dans $\underline{0}$. C'est un choix conventionnel (voir les discussions sur les initialisations de variables en C++). D'autres auraient été possibles. En exercice (difficile) : gérer les valeurs prises "au hasard" au démarrage du programme, ce qui ressemble plus aux conventions en C.

Par la suite, on va se permettre de simplifier $[\![E]\!]_{\mathcal{E}}$ en $[\![E]\!]$ et de même pour les autres structures. Normalement, par typage, on sait de quelle fonction on parle.

2.3 Exercice

Développons un exemple complet. Nous allons faire la preuve que le programme `A` qui suit calcule l'addition sur les entiers naturels.

```
main(X, Y){
    while (X){
        X = X - 1;
        Y = Y + 1
        }
    return Y;
    }
```

La preuve qui suit est *extrêmement détaillée*, horrible à lire humainement parlant. En revanche, elle montre l'utilisation de toutes les équations vues à la première section, à l'exception de la définition de $[\![I]\!]_{\mathcal{C}}$.

On veut donc montrer que le programme calcule l'addition sur les entiers naturels. En d'autres termes, on prouve que $[\![\mathtt{A}]\!](n, m) = n + m$ pour tout $n, m \in \mathbb{N}$. On va procéder par induction sur les entiers. Mais dans un premier temps, nommons B le bloc :

```
X = X - 1;
Y = Y + 1
```

et W la boucle :

```
while (X){
    X = X - 1;
    Y = Y + 1
    }
```

D'abord, on montre le lemme suivant :

Lemme 1. *Pour toute valuation σ telle que $\sigma(\mathtt{X}) = n > 0$ et $\sigma(\mathtt{Y}) = m$, la valuation $\sigma' = [\![B]\!](\sigma)$ vérifie $\sigma'(\mathtt{X}) = n - 1$ et $\sigma'(\mathtt{Y}) = m + 1$.*

Proof. En suivant les définitions,

$$\begin{aligned}
[\![B]\!](\sigma) &= [\![\mathtt{Y} = \mathtt{Y} + 1]\!]([\![\mathtt{X} = \mathtt{X} - 1]\!](\sigma)) \\
&= [\![\mathtt{Y} = \mathtt{Y} + 1]\!](\sigma[\mathtt{X} \mapsto [\![\mathtt{X} - 1]\!](\sigma)]) \\
&= [\![\mathtt{Y} = \mathtt{Y} + 1]\!](\sigma[\mathtt{X} \mapsto [\![\mathtt{X}]\!](\sigma) - [\![1]\!](\sigma))] \\
&= [\![\mathtt{Y} = \mathtt{Y} + 1]\!](\sigma[\mathtt{X} \mapsto n - 1]) \\
&= \sigma[\mathtt{X} \mapsto n - 1][\mathtt{Y} \mapsto [\![\mathtt{Y} + 1]\!](\sigma[\mathtt{X} \mapsto n - 1]) \\
&= \sigma[\mathtt{X} \mapsto n - 1][\mathtt{Y} \mapsto [\![\mathtt{Y}]\!](\sigma[\mathtt{X} \mapsto n - 1]) + [\![1]\!](\sigma[\mathtt{X} \mapsto n - 1]) \\
&= \sigma[\mathtt{X} \mapsto n - 1][\mathtt{Y} \mapsto m + 1]
\end{aligned}$$

Cela montre $\sigma'(\mathtt{X}) = n - 1$ et $\sigma'(\mathtt{Y}) = m + 1$. □

Maintenant, on prouve que pour toute valuation σ tel que $\sigma(\mathtt{X}) = n$ et $\sigma(\mathtt{Y}) = m$, la valuation $\sigma' = [\![W]\!](\sigma)$ vérifie $\sigma'(\mathtt{X}) = 0$ et $\sigma'(\mathtt{Y}) = n + m$.

Par induction sur n,

- Le cas de base. Supposons $n = 0$, alors, par définition $[\![\mathtt{X}]\!](\sigma) = \sigma(\mathtt{X}) = 0$. De ce fait, $[\![W]\!](\sigma) = \sigma = \sigma'$. Par conséquent, $\sigma'(\mathtt{X}) = \sigma(\mathtt{X}) = 0$ et $\sigma'(\mathtt{Y}) = \sigma(\mathtt{Y}) = m = m + 0 = n + m$.

- Sinon, supposons $n = n'+1$. Par définition, $[\![\texttt{X}]\!](\sigma) = \sigma(\texttt{X}) = n'+1 \neq 0$. De ce fait, $\sigma' = [\![W]\!](\sigma) = [\![W]\!]([\![B]\!](\sigma))$. Comme $\sigma(\texttt{X}) > 0$, on a $\sigma' = [\![W]\!](\sigma[\texttt{X} \mapsto n-1][\texttt{Y} \mapsto m+1]) = [\![W]\!](\sigma[\texttt{X} \mapsto n'][\texttt{Y} \mapsto m+1])$. On peut appliquer l'induction sur le magasin $\sigma[\texttt{X} \mapsto n'][\texttt{Y} \mapsto m+1]$. Ce qui donne $\sigma'(\texttt{X}) = 0$ et $\sigma'(\texttt{Y}) = n' + (m+1) = n + m$.

Et sur $\mathbb{Z}$?
Le programme boucle si $\texttt{X} < 0$, variable qui nous a servi pour la récursion.

3 Analyse syntaxique

L'objectif de l'analyse syntaxique est de produire des arbres de syntaxe abstraite à partir du code source donné sous la forme d'une chaîne de caractères. En résumé, l'objet de la section est d'implémenter la fonction `get_ast` du plan général.

En termes d'implémentation, nous allons nous appuyer sur la bibliothèque `Lark`. Il y a bien sûr bien d'autres possibilités, parmi lesquelles citons `lex, yacc, antlr, menhir`.

Pour toute cette section, on pourra se référer à la documentation en ligne de `Lark`, `https://lark-parser.readthedocs.io`. On y trouvera bien plus de détails que ce qui est proposé ici.

L'analyse syntaxique s'appuie sur une notion de grammaire « qui distingue les codes sources correctement écrits de ceux qui ne le sont pas ». La grammaire du langage est décrite par une grammaire algébrique (cf cours de fondements de l'informatique). Les terminaux de la grammaire seront décrits par des expressions régulières. Les terminaux sont nommés tokens ou lexèmes dans ce cadre. Les non-terminaux sont décrits au travers de règles (cf Section 3.2).

Pour fixer les idées, revenons à notre objectif sur un exemple concret. Reprenons le programme de l'addition.

```
main(X, Y){
  while (X){
    X = X - 1;
    Y = Y + 1
  }
  return Y
}
```

L'arbre de syntaxe abstraite que l'on veut obtenir est présenté à la figure 1.

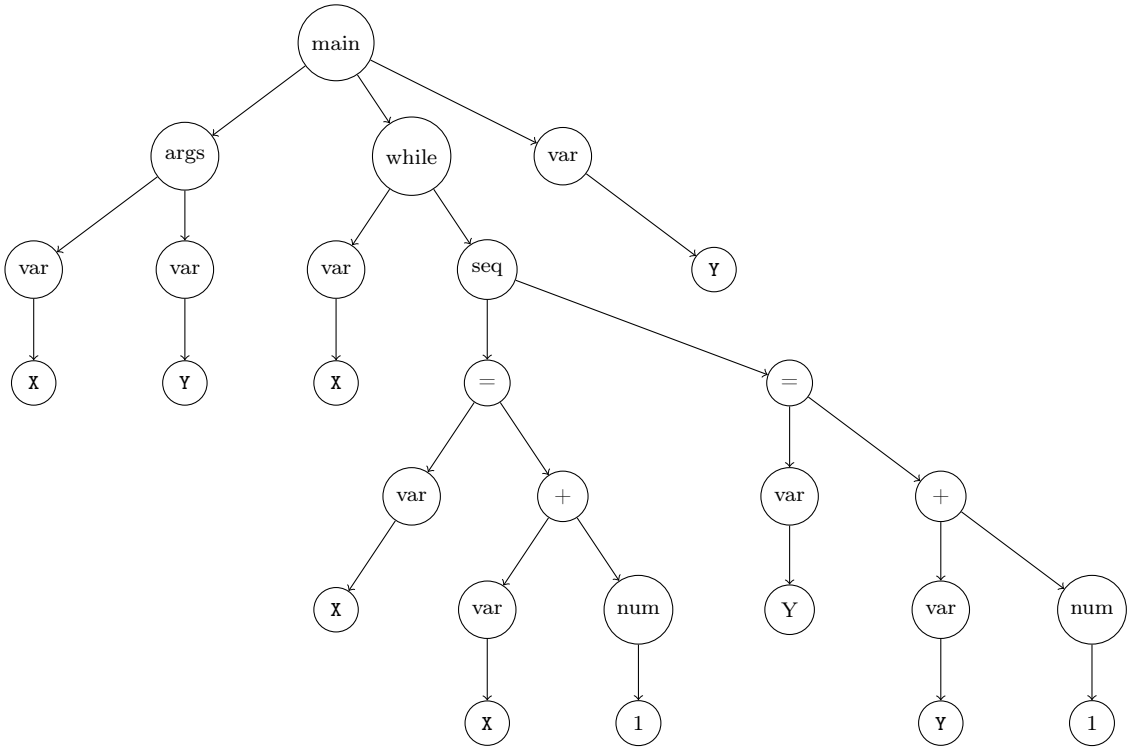
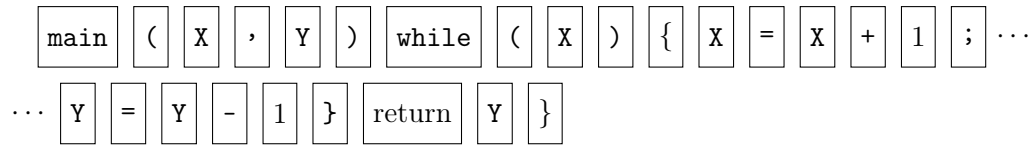


Figure 1: AST du programme de l'addition.

Cela peut paraître volumineux par rapport à ce qui précède. Toutefois, on dispose là d'une structure récursive qui rend compte de la syntaxe abstraite définie à la Section 2. Nous allons bénéficier de cette structure pour décomposer la génération en sous-parties simples.

3.1 Analyse lexicale

La première étape d'analyse est la segmentation du texte en lexèmes (token). On la retrouve aussi bien avec `Lark` qu'avec d'autres outils (comme l'ancêtre : `lex`). Le texte du programme de l'addition ci-dessus est transformé en la liste :



En Lark, la description des lexèmes est réalisée à l'aide d'expressions régulières. Cet exemple, minimal, reconstitue un identifiant :

```
1 import lark
2 gidentifier = lark.Lark(r"""start:IDENTIFIER
3 IDENTIFIER : /[a-zA-Z][a-zA-Z0-9]*/""", start="start")
4 ast = gidentifier.parse("hello")
5 print(ast.pretty())
```

La variable `gidentifier` est l'objet qui représente la grammaire. Le mot `IDENTIFIER` est le terminal qui décrit les variables. L'expression régulière est donnée dans la syntaxe `python`. A la ligne 4, le texte `"hello"` est analysé selon la grammaire `gidentifier`. Le résultat est imprimé ligne 5.

Les lexèmes doivent être identifiés par des lettres capitales.

Voici un deuxième cas intéressant.

Opérateurs binaires

```
OPBIN : /[+\-*/<>]|>>|<=/
```

Dans le cas des opérateurs binaires, la chaîne `">>"` doit être interprétée par un lexème, pas deux. Par défaut, `Lark` favorise les lexèmes les plus longs. Voir la section du manuel sur les ambiguïtés sur les lexèmes au besoin.

Le signe `"-"` sert pour décire les intervalles de lettres. Pour décrire le caractère '-', on utilise la notation `"\-"`.

Espaces et caractères à oublier Pour éviter la gestion des espaces et des retours à la ligne, il est suggéré de les oublier (à mettre dans la déclaration de la grammaire):

```
1 %import common.WS
2 %ignore WS
```

La première ligne charge la définition `WS` correspondant aux espaces et retours à la ligne. La seconde indique que ces lexèmes seront ignorés.

3.2 Analyse sémantique

La seconde étape va transformer la liste des tokens en un arbre de syntaxe abstraite. Les chaînes de lexèmes suivent des règles de grammaire

Un mélange des deux
Les deux analyses, lexicale et sémantique sont en fait imbriquées : en cherchant un non terminal d'un certain type, `Lark` va déclencher l'analyse lexicale pour les terminaux correspondants à ce type là.

Expressions Les expressions sont construites en suivant la grammaire suivante :

```
1 gr_expressions = lark.Lark(r"""
2 exp : SIGNED_NUMBER              -> exp_nombre
3 | IDENTIFIER                     -> exp_var
4 | exp OPBIN exp                  -> exp_opbin
5 | "(" exp ")"                    -> exp_par
6 %import common.SIGNED_NUMBER
7 ...
8 """, start = "exp")
```

Dans cette grammaire, le non-terminal qui sert de symbole de départ est "exp" (cf. `start="exp"`). L'analyse sémantique va construire un arbre à partir de cette grammaire. Par exemple, sur `"(3+x) - xy"`, l'affichage de l'ast donne :

```
exp_opbin
  exp_par
    exp_opbin
      exp_nombre   3
      +
      exp_var   x
  -
  exp_var       xy
```

Nom de règles
Les indications comme `"-> exp_nombre"` vont donner un nom à chaque

clause (règle). Ce nom se retrouve pour chaque noeud interne de l'arbre. Conseil : le nom est optionnel, mais cela simplifie largement la suite.

Autant pour les lexèmes, il faut éviter les blancs et les retours à la ligne dans les définitions, autant pour les règles, on peut être plus libéral. Attention toutefois à ne pas laisser une ligne vide qui peut être (mal) interprétée.

Syntaxe EBNF

La syntaxe des règles est la syntaxe EBNF (Extended Backus Normal Form). Elle autorise les listes dans les règles. Par exemple, pour définir la liste (éventuellement vide) des variables d'une fonction, on pourra définir :

```
1 var_list :                        -> liste_vide
2 | IDENTIFIER ("," IDENTIFIER)*  -> aumoinsune
```

3.3 Les AST

L'analyseur de `Lark` produit des objets de type `Tree`. Il y a deux sortes de nœuds, les noeuds internes (qui correspondent aux non-terminaux) et les noeuds feuilles qui correspondent aux terminaux.

Un noeud interne `n` est de type `Tree`, il a deux attributs :

- `n.data` qui donne la règle appliquée ("exp_opbin", "exp_var", etc) et
- `n.children` qui donne la liste de ses enfants.

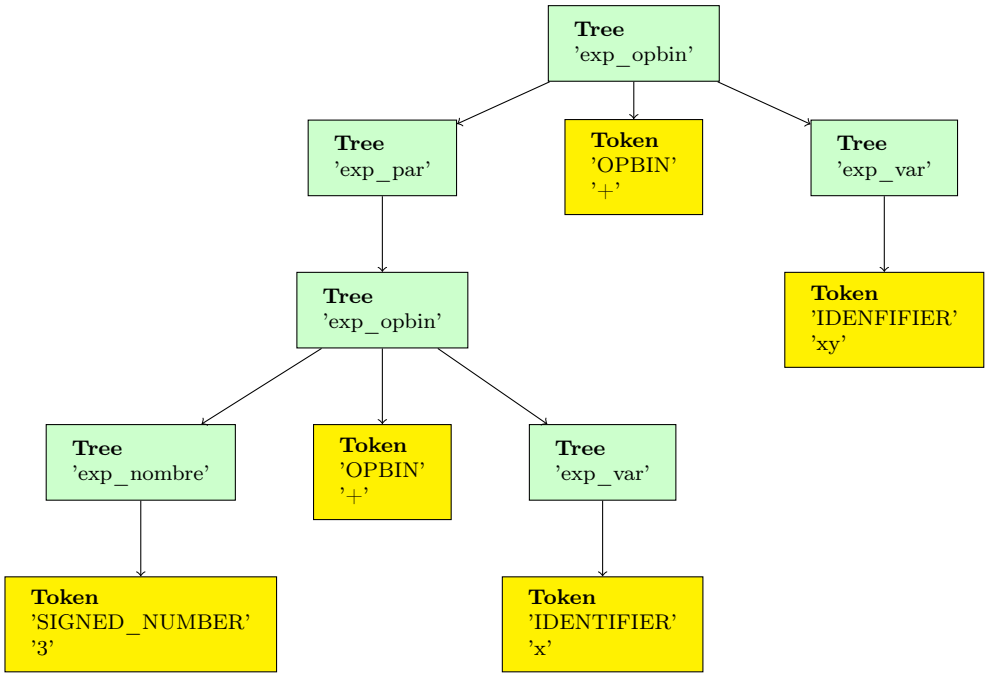
Un noeud feuille `t` est de type `Token`, il a également deux attributs :

- `t.type` qui donne le type de terminal ("IDENTIFIER", "OPBIN", etc)
- `t.value` qui donne le contenu du lexème ("44", "x", etc).

De ce fait, l'ast au dessus peut se voir :

```
Tree('exp_opbin', [
    Tree('exp_par', [
        Tree('exp_opbin', [
            Tree('exp_nombre', [
                Token('SIGNED_NUMBER', '3')
            ]),
            Token('OPBIN', '+'),
            Tree('exp_var', [
                Token('IDENTIFIER', 'x')
            ])
        ])
    ]),
    Token('OPBIN', '-'),
    Tree('exp_var', [
        Token('IDENTIFIER', 'xy')
    ])
])
```

ou de manière graphique :



3.4 Quelques conseils pour utiliser `Lark`

Premier point. Dans l'expression

```
(3 + x),
```

les deux parenthèses n'apparaissent pas dans l'AST. Cela vient de la règle :

```
exp: "(" exp ")" -> exp_par
```

pour laquelle les deux `Token`s correspondant aux parenthèses ne sont pas nommés. A contrario, le `Token` '+' est bien présent. Cela vient de la règle

```
exp: exp OPBIN exp -> exp_opbin
```

qui identifie le token `+` par un terminal nommé, ici `OPBIN`. En résumé, quand un terminal est nommé, il apparait dans l'AST, il est omis dans le cas contraire. Ce procédé est fort utile pour évacuer de très nombreux éléments de syntaxe, les parenthèses, les accolades, et plus généralement les signes de ponctuation. Et même les mots clés.

Deuxième cas important. Par défaut, pour une règle :

```
var_list : IDENTIFIER ("," IDENTIFIER)* -> aumoinsune
```

`Lark` va produire un nœud de type 'aumoinsune' qui a autant d'enfants que de variables dans la liste.

Règle ou Lexème ?
Avec les expressions régulières, on peut exprimer de nombreux lexèmes. Il ne serait pas difficile de définir par exemple la liste des variables avec un lexème :

```
/[a-z]+(,[a-z]+)*|/
```

Alors, faut-il une règle ou un lexème ? Si on veut accéder à chacune des variables, il est préférable d'utiliser une règle comme nous l'avons vu auparavant. En effet, dans le cas d'un token, celui-ci contiendra la chaîne "X,Y,Z,T". Il faudrait retransformer le token en une liste ["X","Y","Z","T"] pour disposer de chaque variable individuellement.

D'où la règle de bon sens : si la structure que l'on considère est composée de sous éléments, il vaut mieux utiliser une règle. Dans le cas contraire, les lexèmes font l'affaire.

Pour aller plus loin
Ici, nous utilisons directement l'arbre depuis `Lark`. Dans le cadre d'un projet plus conséquent, transformer l'arbre ainsi extrait dans sa propre structure est souvent une bonne idée.

4 `nasm`, un assembleur x86_64

Venons maintenant à notre langage cible, le langage `nasm`, un des assembleurs des processeurs x86_64.

Les assembleurs sont une syntaxe humaine pour les instructions du processeur. Chaque instruction assembleur correspond à une instruction du processeur, enfin presque, voir [BT21]. On pourra retrouver le lien entre code machine et assembleur sur un site comme `https://defuse.ca/online-x86-assembler.htm`.

Nous allons voir un sous-sous ensemble du langage. Il y a de nombreuses sources pour aller plus loin. Citons :

- La documentation du constructeur INTEL : `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`;
- Le page de Félix Cloutier, `https://www.felixcloutier.com/x86/`;
- Le forum d'Agner Fog, `https://www.agner.org/forum`.

Avant de parler de la syntaxe de l'assembleur, nous avons besoin de quelques éléments de fonctionnement d'une machine.

4.1 Un modèle de machine

Vu d'un programme, le processeur est muni de registres et d'une mémoire virtuelle. Chaque instruction machine modifie les (des) registres et la mémoire.

Registres Premier point. La machine utilise des registres, ce sont des petites zones de mémoire. Étant inscrites directement sur le processeur, elles forment une zone de stockage rapide à modifier et à lire.

Sur les processeurs x86_64, lister les registres étant une sinécure (voir le billet amusant d'Agner Fog à ce sujet), on va restreindre notre attention à ceux qu'on va réellement utiliser.

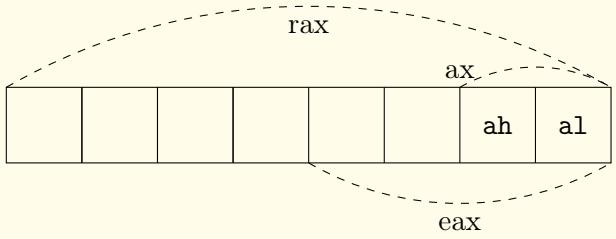
Les registres courants sont : `rax, rbx, rcx, rdx, rsi, rdi, rsp, rbp, r8, ..., r15, rip, rflags`. Ils font tous 8 octets (64 bits). Ils sont prévus pour contenir des adresses ou des entiers.

Pour les calculs avec les flottants, il y a des registres spécialisés. Ce sont les registres `xmm0, ..., xmm7`. Ces registres font 128 bits (et même plus si on prend leur forme `ymmi` (256 bits) ou `zmmi` (512 bits)).

Les registres ont une histoire
Les quatre premiers registres datent des premiers processeurs INTEL, le 8080, dans les années 1970. Ils faisaient au début 8 bits. Et il fallait les combiner pour obtenir des nombres au dessus de 255.

Pour des raisons de retro-compatibilité, ces registres sont toujours accessibles aujourd'hui. Par exemple, le registre `ebx` qui date de l'architecture `Intel-386` (32 bits) correspond aux 32 bits de poids faible du registre `rbx`. Le registre `ax` qu'on retrouve sur `Intel-286` correspond aux 16 bits de poids faible de `rax`. Et enfin, `ah` et `al`, tous deux faisant 8 bits correspondent respectivement aux bits de 8 à 16 et aux bits de 0 à 8 du registre `rax`.

En résumé, les 8 octets de `rax` se déclinent :



Parmi les registres, il y en a trois qui se distinguent.

- Le registre `rip`, instruction pointer, est le registre qui contient l'adresse de l'instruction à exécuter.
- Le registre `rsp` désigne le "haut de la pile".
- Le registre `rflags` est composé de drapeaux. L'un d'entre eux, le drapeau `ZF`, est mis à 1 quand un calcul arithmétique aboutit à 0. Parmi les autres flags : SF, drapeau du signe (1 quand le nombre est positif, 0 s'il est négatif), CF (carry, 1 si l'opération se termine avec

une retenue), OF, (overflow, en cas de dépassement de capacité du registre), PF, (parity).

Modèle des registres
Par la suite, on note `REG` l'ensemble des registres. Au même titre qu'on a défini la valuation des variables, on peut le faire pour les registres.

Définition 1. *Une valuation de registres est une fonction dans* `REG` $\to$ `QWORD`. *On note* $\mathfrak{R}$ *l'ensemble des valuations de registres.*

Mémoire virtuelle Le second élément qui constitue la machine, c'est la mémoire virtuelle. Elle prend la forme d'un tableau de 2^{64} octets sur une machine 64 bits. Chaque octet de la mémoire peut être désigné en assembleur :

```
byte [0x1234]
```

désigne l'octet à la position 0x1234 en mémoire. Il est également possible d'extraire des blocs plus longs:

`word [0x1234]`	2 octets
`dword [0x1234]`	4 octets
`qword [0x1234]`	8 octets

Modèle de la Mémoire

Définition 2. *Une mémoire virtuelle est une fonction* `QWORD` $\to$ `BYTE`. *On note* $\mathfrak{M}$ *l'ensemble de ces fonctions.*

Si $a \in$ `QWORD` et $mem \in \mathfrak{M}$, on désigne par $mem_2(a)$, $mem_4(a)$ et $mem_8(a)$ respectivement les entiers codés sur 2, 4 et 8 octets à partir de a.

Les octets retournés
Le mot "`word [0x1234]`" stocké à l'addresse 0x1234 s'interprète comme l'entier `byte [0x1234]` $+ 2^8 \times$ `byte [0x1235]`.

En d'autres termes, l'octet de poids faible apparait en mémoire *avant* l'octet de poids fort. Cela va à l'encontre de notre notation habituelle. La remarque vaut pour les mots doubles et quadruples. L'encodage est dit little-endian.

Configuration
A la section 2, nous avons introduit la notion de valuation. Chaque instruction modifie la valuation courante. Quelle est la notion de valuation pour x86_64?

Un état de la machine est caractérisé par l'état de ses registres et l'état de sa mémoire.

Définition 3 (Configuration, état de la machine)**.** *Un état de la machine(alternativement une configuration) est une paire* $\gamma = \langle reg, mem \rangle \in \mathfrak{R} \times \mathfrak{M}$.

L'ensemble des configurations est noté $\mathfrak{C}$. C'est le pendant de la valuation du nano-C.

Étant donné une configuration $\gamma = \langle r, m \rangle \in \mathfrak{C}$, à la manière des informaticiens, on note $\gamma.regs$ la première composante de la paire (i.e. r) et $\gamma.mem$ sa deuxième composante, ici m. De ce fait, $\gamma = \langle \gamma.regs, \gamma.mem \rangle$.

A la manière de ce qu'on à fait pour les valuations, étant donné γ, étant donné $r \in \texttt{REG}$, et $a \in \texttt{QWORD}$, on note $\gamma[r \mapsto a] = \langle \gamma.regs[r \mapsto a], \gamma.mem \rangle$. Si $a \in \texttt{QWORD}$ et $n \in \texttt{BYTE}$, $\gamma[@(a) \mapsto n] = \langle \gamma.regs, \gamma.mem[a \mapsto n] \rangle$.

Pour la suite, étant donné un entier m sur plus d'un octet (2, 4 ou 8), on note $\gamma[@(a) \mapsto m]$ le fait de mettre à jour la mémoire en stockant l'entier m à l'adresse a. On se permet également d'écrire $\gamma[\texttt{rax}]$ pour $\gamma.regs[\texttt{rax}]$.

Notes sur le modèle
La machine se résume donc à une paire de fonctions $\gamma = \langle regs, mem \rangle$. Bien sûr, ce modèle est insuffisant pour décrire le fonctionnement d'une machine en toute généralité. Le fonctionnement de la machine va dépendre de nombreux autres facteurs comme l'heure, les entrées sorties, l'état des caches, la température du processeur, etc. Vous en verrez certains en cours d'Architecture.

Mais il vaut mieux avoir un modèle qui permette de travailler que pas de modèle du tout. En outre, le code produit par le compilateur est sensé ne pas dépendre de ce type de paramètres.

Autre faiblesse du modèle, en fait, la mémoire est scindée en pages,

chacune correspondant à un bloc 0x1000 octets successifs de mémoire. À chaque page est associée des droits, en lecture, en écriture et en exécution : il n'est pas possible d'écrire n'importe quoi n'importe où. Pour la compilation, cela n'aura pas (vraiment) d'incidence.

Exécution Troisième point à propos du fonctionnement d'une machine. Après la présentation des données, la manière de calculer. A chaque top de l'horloge, le processeur lit l'instruction stockée à l'adresse donnée par le registre `rip` en mémoire. Il met à jour la configuration de la machine en fonction de cette instruction. En particulier, il met à jour le registre `rip` lui-même. La section suivante présente un petit nombre d'instructions sous leur forme assembleur.

Exceptions, interruptions

Nous donnons ici une présentation largement simplifiée du fonctionnement du processeur. Le top de l'horloge est une version très simplifiée de ce qui se passe réellement.

En outre, le fonctionnement du processeur peut être "perturbé" par des exceptions ou des interruptions. Elles apparaissent à plusieurs occasions, s'il y a un évènement matériel—une clé USB branchée—par exemple. Ce type de détail n'a pas d'incidence pour l'implémentation de notre compilateur.

A l'instar de la sémantique des expressions en nano-C, étant donné une configuration γ, on note :

$$\begin{cases} r \in \texttt{REG} & [\![r]\!](\gamma) = \gamma.regs(r) \\ a \in \texttt{QWORD} & [\![a]\!](\gamma) = a \\ a \in \texttt{QWORD} & [\![\texttt{byte [a]}]\!](\gamma) = \gamma.mem(a) \\ a \in \texttt{QWORD} & [\![\texttt{qword [a]}]\!](\gamma) = \gamma.mem_8(a) \\ r \in \texttt{REG} & [\![\texttt{byte [r]}]\!](\gamma) = \gamma.mem([\![r]\!](\gamma)) \\ \dots \end{cases}$$

où $m_8(x)$ désigne le `QWORD` à l'adresse x de la mémoire m.

4.2 Syntaxe `nasm`

Un énoncé (une ligne de code) en `nasm` est de la forme suivante :

```
etiquette : element
```

L'étiquette désigne la position en mémoire de l'élément. On peut mettre plusieurs étiquettes sur le même élément :

```
label:
etiquette: element
```

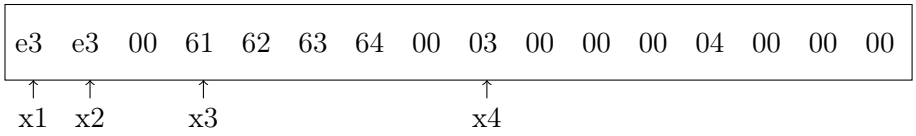
ou aucune :

```
element
```

Les éléments sont soit une déclaration directe de contenu, soit une instruction. Les déclarations directes de contenu ont la forme suivante :

```
1 x1 : db 0xe3
2 x2 : dw 0xe3
3 x3 : db "abc", "e", 0
4 x4 : dd 3, 4
```

A la ligne 1, i.e. à l'adresse `x1`, on met en mémoire un octet qui vaut 0xe3. A la ligne 2, on met en mémoire le même entier mais sur 2 octets : [0xe3, 0]. L'adresse x3 qui suit vérifie dans ce cas `x3 == x2 + 2`. La syntaxe nous permet de mettre plusieurs caractères : la mémoire en `x3` sera ['a', 'b', 'c', 'e', 0]. A la ligne 4, c'est pareil, mais attention, cela pose 2×4 octets en mémoire. En résumé, après la ligne 4, la mémoire virtuelle a la forme suivante :



Nous allons lister maitenant une vingtaine d'instructions en donnant en même temps leur sémantique. Dans ce qui suit, γ désigne une configuration de la machine.

Mouvements de la mémoire vers les registres et inversement Nous allons traiter en détail deux instructions machine.

```
mov rax, 13
mov [0x1234], rax
```

La première instruction, `mov rax, 13` se lirait en C:

```
rax = 13;
```

Elle attribue au registre `rax` la valeur 13. En termes de sémantique, cela donne :

$$[\![\texttt{mov rax, 13}]\!](\gamma) = \gamma[\texttt{rax} \mapsto 13, \texttt{rip} \mapsto [\![\texttt{rip}]\!](\gamma) + 7].$$

Remarque
On pourra noter que deux registres sont modifiés. Le registre `rax` bien sûr mais aussi le registre `rip`. L'instruction faisant 7 octets en mémoire, pour passer à l'instruction suivante, le registre `rip` est décalé d'autant.

Tous les registres dans la liste :

$$Lmov = \{\texttt{rax}, \texttt{rbx}, \texttt{rcx}, \texttt{rdx}, \texttt{rsi}, \texttt{rdi}, \texttt{rsp}, \texttt{rbp}, \texttt{r8}, ..., \texttt{r15}\}$$

sont gérés de manière analogue. Les registres `rip` et `rflag` ne peuvent pas être modifiés à l'aide de l'instruction `mov`.

La seconde instruction, `mov [0x1234], rax` se lirait en C:

```
((uint64_t *) 0x1234)[0] = rax;
```

Sa sémantique est :

$$[\![\texttt{mov [0x1234], rax}]\!](\gamma) = \gamma[@(0x1234) \mapsto [\![\texttt{rax}]\!](\gamma), \texttt{rip} \mapsto [\![\texttt{rip}]\!](\gamma) + 8].$$

Pour vérification
En fonction de la taille de l'adresse indiquée, l'instruction machine pourra avoir une taille différente ; le décalage de `rip` est fonction de cette taille.

Pour les autres instructions, on donne une sémantique un peu plus informelle. Les instructions sont présentées avec des registres/valeurs concrètes.

nasm	"C-like"	taille
`mov rax, rbx`	`rax = rbx`	3
`mov rax, [0x12]`	`rax = ((uint64_t *)0x12)[0]`	7
`mov rax, [rbx]`	`rax = ((uint64_t *)rbx)[0]`	3
`mov [0x12], 0x34`	`((uint64_t *)0x12)[0]` = 0x34	12
`mov [rax], rbx`	`((uint64_t *)rax)[0]` = rbx	3
`mov ah, [0x12]`	`ah = ((uint8_t *) 0x12)[0]`	5

Exercice 1. *Donner la sémantique opérationnelle des instructions ci-dessus.*

Voici des variantes de l'instruction `mov`,

```
mov eax, dword ptr[0x12]
mov rax, [rbx]
mov rax, [rcx + 4*rdx]
mov dword ptr[0x12], 0x34
mov byte ptr[0x12], bl
mov qword ptr [rbx], rax
```

L'instruction suivante est interdite :

```
mov qword ptr[0x12], [0x34]
```

En principe, on a pas le droit de transférer directement une cellule mémoire vers une autre. Il faut passer par un registre.

Opérations arithmétique

nasm	"C-like"	taille
`add rax, rbx`	`rax += rbx`	3
`mul rax, [0x12]`	`rax *= 0x12`	7
`imul rax, [rbx]`	`rax *= *rbx`	3
`xor rax, rax`	`rax ^= rax`	12
`cmp rax, rbx`	***	1

Notes sur les drapeaux
Pour la dernière instruction, `cmp rax, rbx`, le processeur fait le calcul `rax - rbx` et met à jour le registre des flags. Le flag `ZF` prend la valeur 1 si le résultat qui précède aboutit à 0 (i.e. `rax == rbx`), 0 sinon. Le registre de signe `SF` suit la même logique. Pareil pour les autres drapeaux.

Pour toutes les instructions arithmétiques, le registre `rflags` est mis à jour. Par exemple, le calcul `xor rax,rax` donne toujours 0. Le flag `ZF` (ainsi que le flag de parité `PF`) est mis à 1 par conséquent.

Signé et non signé
La multiplication `imul` est la multiplication signée tandis que la multiplication `mul` est la multiplication non signée.

Instructions de pile

nasm	"C-like"	taille
`push rax`	`rsp = rsp - 8;` `*((uint64_t *) rsp) = rax`	1
`push 0x12345678`	`rsp = rsp - 4;` `*((uint32_t *) rsp) = 0x12345678`	5
`push [0x1234]`	`rsp = rsp - 8;` `*((uint64_t *) rsp) = ((uint64_t *) 0x1234)[0]`	7
`pop rax`	`rax = *((uint64_t *) rsp);` `rsp = rsp + 8;`	1

Remarques
La taille de l'instruction `push 0x12345678` dépend de la taille de l'immédiat. Toutefois, sur la pile, les ajouts se font par paquet de 4 octets même si l'immédiat n'en fait que 1. Pour empiler un entier sur 8 octets, il faut utiliser un registre.

L'instruction `push [0x12]` est une instruction qui viole le principe selon lequel il n'y a pas de mouvements directs de mémoire à mémoire. Cela vient de la gestion de la pile, spécifique depuis les processeurs 8086.

Deux instructions supplémentaire :

```
pushad ; sauvegarde de tous les regitres sur la pile
popad ; mise à jour des registres après une sauvegarde
```

Instructions de saut (conditionnel)

Considérons le petit programme en `nasm`:

```
hello : xor rax, rax
        cmp rax, rbx
        je fin
        jmp hello
fin:    mov rax, 12
```

L'étiquette `hello` désigne l'adresse (du premier octet) de l'instruction `xor rax, rax`. L'adresse `fin` est l'adresse de l'instruction `mov rax, 12`.

nasm	"C-like"	taille
`jmp hello`	`goto hello; rip = hello`	5
`je fin`	`if (ZF) goto fin;`	5

Le registre `rflags`
On rappelle que le registre `rflags` (dont `ZF` fait partie) est mis à jour par les instructions arithmétiques. Comme dans l'exemple qui précède, on va trouver très souvent une telle instruction juste avant un test. Typiquement une instruction de comparaison `cmp` ou `test`.

Quelques exemples d'instructions de saut conditionnel et leur flag correspondant :

Saut	Flags	Sur cmp
je	$ZF = 1$	rax == rbx
jne	$ZF = 0$	rax != rbx
jg	$OF = 0 \wedge ZF = 0$	rax $>$ rbx
jge	$OF = SF \vee ZF = 1$	rax $\geq$ rbx

La taille des instructions dans le tableau précédent est donnée à titre indicatif. Elle dépend de la position relative de l'instruction et de l'étiquette cible.

L'instruction `je` a un alias équivalent `jz`. De même avec `jne` et `jnz`. Toutes les quatre utilisent le flag `ZF`.

Appels de fonctions Considérons le petit programme suivant :

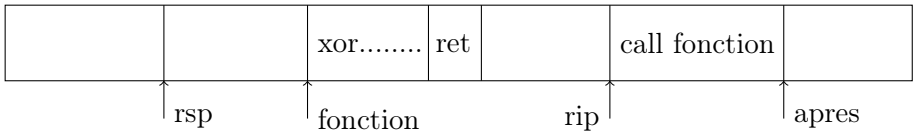
```
fonction : xor rax, rax
           mov rbx, 21
           add rax, rbx
           ret
           ...
           call fonction
apres :    mov rcx, rax
           ...
```

Globalement, la fonction `fonction` est appelée par l'instruction `call fonction`. La fonction se termine lors de l'exécution de l'instruction `ret` et ensuite, le calcul se poursuit par l'instruction `mov rcx, rax` à l'adresse `apres`.

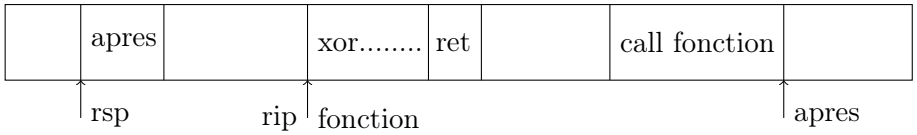
Difficile de donner une description "C-like" de l'instruction `call`. En termes de sémantique, cela donne :

$$[\![\texttt{call fonction}]\!](\gamma) = \\ \gamma[\texttt{rsp} \mapsto [\![\texttt{rsp}]\!](\gamma) - 8][@([\![\texttt{rsp}]\!](\gamma)) \mapsto \texttt{apres}][\texttt{rip} \mapsto \texttt{fonction}]$$

En d'autres termes, la valeur `apres` est empilée. Et ensuite, le pointeur d'instruction est positionné en `fonction`. Graphiquement, juste avant l'exécution de l'instruction `call`, la configuration de la machine est de la forme :



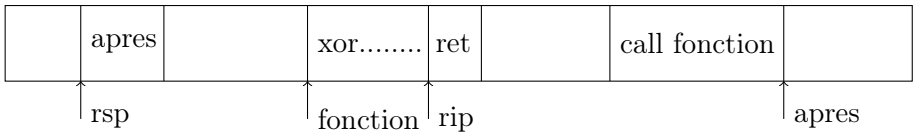
Juste après, au début de l'exécution de la fonction, on est dans l'état suivant.



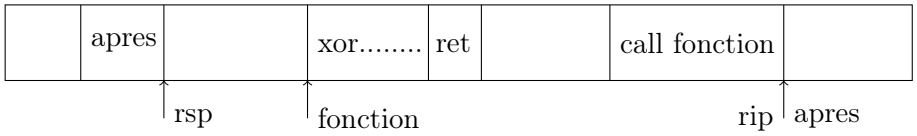
Pour l'instruction `ret`, la sémantique est

$$[\![\texttt{ret}]\!](\gamma) = \gamma[\texttt{rip} \mapsto [\![[\texttt{rsp}]]\!](\gamma)][\texttt{rsp} \mapsto [\![\texttt{rsp}]\!](\gamma) + 8]$$

Le pointeur d'instruction prend la valeur en haut de la pile qui est dépilée. Juste avant le `ret`, la configuration (attendue) est résumée comme suit :



Juste après,



En résumé, comme on a empilé l'adresse `apres` lors de l'appel de la fonction, *si cette adresse sur la pile n'a pas été modifiée et que le haut de la pile est le même qu'à l'entrée de la fonction au moment où l'on arrive à l'instruction* ***`ret`***, le haut de la pile contient l'adresse `apres` de l'instruction qui suit l'appel de la fonction.

4.3 Structure globale d'un code assembleur

Commentons le programme "Hello world".

```
1  extern printf                ; a C external function,
2  section .data                ; Data section, initialized variables
3    msg:  db "Hello world", 0  ; C string needs 0
4    fmt:  db "%s", 10, 0       ; The printf format, "\n",'0'
5
6  section .text                ; Code section.
7  global main                  ; the standard gcc entry point
8    main:                      ; the program label for the entry point
9    push rbp                   ; set up stack frame, must be alligned
10   mov rdi, fmt
11   mov rsi,msg
12   mov rax,0                  ; usually  xor  rax,rax
13   call printf                ; call to a C function
14   pop rbp                    ; restore stack
15   mov rax, 0                 ; normal, no error, return value
16   ret                        ; return
```

Un programme en assembleur se structure sous la forme de sections. A une section va correspondre une zone de mémoire. Ici, on a deux sections, la section `.data` et la section `.text`. Ce sont les noms dévolus respectivement à la zone de mémoire des variables globales et à celle des instructions.

- La ligne 1 qui précède indique l'import d'une fonction externe, la fonction `printf`. Elle sera liée au programme au moment de l'édition de liens.
- La ligne 7 donne le point d'entrée du programme, ici `main`. Pour l'édition des liens, le compilateur `gcc` demande (par défaut) que le point d'entrée soit `main`.

Conventions d'appel
L'instruction d'appel des fonctions, `call`, n'indique pas *où* sont les arguments. Ils peuvent être donnés via des registres ou en mémoire. La manière dont les arguments sont transmis est une convention. Pour les bibliothèques comme la `libc`, la convention d'appel est la convention ABI-V.

Le texte de la convention fait une centaine de pages. Le voici résumé en quelques phrases. Premier point, les arguments entiers (ou pointeurs) sont donnés en premier lieu via les registres `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` dans cet ordre. Ensuite, les arguments sont passés par la pile.

Les arguments flottants sont passés via les registres `xmm0, xmm1, ..., xmm7`.

Pour les fonctions variadiques comme `printf`, la valeur de `rax` (en fait `al`) indique le nombre d'arguments flottants.

Les fonctions dont le retour est un entier ou un pointeur donnent leur résultat dans le registre `rax`.

- Les lignes 10, 11, 12 correspondent à la mise en place des arguments de `printf`.
- La ligne 15 indique que le résultat du code est `0` (pas d'erreur donc).

4.4 Vademecum en assembleur

Voici un petit nombre d'exemples pour inspiration éventuelle :

```
extern printf
section .data
fmt: db ">> %s <<",0
section .text
global main
main:
mov rbx, rsi   ; read argv
mov rsi, [rbx+8] ; argv[1]
mov rdi, fmt
xor rax, rax
call printf
ret
;
```

Read argv

```
extern printf
section .data
fmt: db "%d", 10, 0
section .text
global main
main:
mov rbx, rdi ; read argc
mov rdi, fmt
mov rsi, rbx
xor rax, rax
call printf
ret
;
```

Read argc

```
extern printf, atoi
section .data
nombre: db "32", 0
fmt:    db "%d",10,0
section .text
global main
main:
mov rdi, nombre
call atoi
inc rax
mov rdi, fmt
mov rsi, rax
xor rax, rax
call printf
ret
```

String to int

```
extern        printf
section .data
fmt:    db ">> %lf",0
pi: dq 4614253070214989087
section .text
global main
main:
push rbp
movsd xmm0, qword [pi]; xmm0=pi
mov rax, 32 ;
cvtsi2sd xmm2, rax; (double) rax
addsd   xmm0, xmm2
mov rax, 1
mov rdi, fmt
call printf
pop rbp
ret
```

Print 3.14 + 32 as float

On pourra également s'inspirer de code produit avec un compilateur. Une bonne adresse : `https://godbolt.org`. Essayez les optimisations `-O0` et `-O1`.

4.5 Cinq algorithmes pour s'entraîner en assembleur

Exercice 2. *Écrire le code assembleur pour faire la somme des entiers d'un tableau* `t` *de taille* `size`*. On fera l'hypothèse que* `t` *est stocké dans* `RDI` *et que* `size` *l'est en* `RSI`*.*

Exercice 3. *Écrirre le code assembleur qui fait la somme d'un vecteur* `v` *(alias* `RDI`*) et d'un vecteur* `t` *(alias* `RSI`*) de taille* `size` *(alias* `RDX`*), le résultat étant stocké dans le premier vecteur.*

Exercice 4. *Écrire en assembleur l'algorithme du tri à bulles. On fera l'hypothèse que le tableau est stocké en* `rsi` *et qu'il est de taille* `size` *stocké en* `rdi`*.*

Exercice 5. *Écrire en assembleur la fonction qui permet de savoir si un entier est premier. L'entier est supposé être donné dans* `rsi`*.*

Exercice 6. *Écrire en assembleur une fonction qui range tous les éléments pairs avant les éléments impairs.*

Solutions à la fin du polycopié.

5 Plan du générateur de code

Dans cette section, nous définissons de manière formelle le générateur de code. Le générateur est défini comme une fonction qui prend une expression, une commande ou un programme et dont le résultat est une liste d'énoncés (au sens de `nasm`).

Quelques principes de notation avant de commencer.

- À la manière des formatages de chaînes en `python`, si $n = 12$, par définition, `mov rax, {n}` doit se lire `mov rax, 12`. On utilise les substitutions de manière un peu informelle. Mais le contexte devrait être clair.

On suppose qu'à chaque variable $\texttt{X} \in \mathcal{V}$ correspond une étiquette (une adresse) $\Gamma(\texttt{X})$ dans le code. Le plus simple est de choisir $\Gamma(\texttt{X}) = \texttt{X}$ et c'est ce que nous ferons dans un premier temps.

On note $\Gamma_{\mathcal{E}} : \mathcal{E} \to \mathcal{LI}^*$ le compilateur d'expression. L'intention du générateur est qu'après l'évaluation de $\Gamma_{\mathcal{E}}(E)$, le registre `rax` a la valeur de l'expression. Par induction sur chacun des cas.

- Si $E = n \in \mathbb{Z}$. On définit :

$$\Gamma_{\mathcal{E}}(n) = \boxed{\texttt{mov rax, \{n\}}}.$$

- Si $E = \texttt{X} \in \mathcal{V}$. On définit :

$$\Gamma_{\mathcal{E}}(\texttt{X}) = \boxed{\texttt{mov rax, [\{}\Gamma(\texttt{X})\texttt{\}]}}.$$

- Supposons $E = E_1 \otimes E_2$, $E_1, E_2 \in \mathcal{E}$, $\otimes \in OPBIN$. On fait l'hypothèse que si $\gamma(\texttt{rax}) = n$ et $\gamma(\texttt{rbx}) = m$, alors, étant donné $\gamma' = [\![I_\otimes]\!](\gamma)$, on suppose que $\gamma'(\texttt{rax}) = n \underline{\otimes} m$, la mémoire restant inchangée. En résumé, $I_\otimes$ calcule $\otimes$. On définit alors :

$$\Gamma_{\mathcal{E}}(E_1 \otimes E_2) = \boxed{\begin{array}{l}\{\Gamma_{\mathcal{E}}(E_2)\}\\ \texttt{push rax}\\ \{\Gamma_{\mathcal{E}}(E_1)\}\\ \texttt{pop rbx}\\ \{I_\otimes\}\end{array}}.$$

Note
Pour le calcul de $\Gamma_{\mathcal{E}}(E_1 \otimes E_2)$, nous évaluons E_2 avant E_1. C'est un choix qui n'est pas toujours pertinent. Par exemple, dans la documentation du langage C, l'évaluation de l'opérateur binaire logique `&&` est spécifiée aller de gauche à droite. À vous de faire mieux.

La génération du code des commandes est $\Gamma_{\mathcal{C}} : \mathcal{C} \to \mathcal{LI}^*$. Elle se définit également par induction :

- Si $C = \texttt{X} = E$ où $\texttt{X} \in \mathcal{V}$ et $E \in \mathcal{E}$, on définit :

$$\Gamma_{\mathcal{C}}(C) = \boxed{\begin{array}{l}\{\Gamma_{\mathcal{E}}(E)\}\\ \texttt{mov [X], rax}\end{array}}$$

- Si $C = C_1; C_2$ avec $C_1, C_2 \in \mathcal{C}$, on définit :

$$\Gamma_{\mathcal{C}}(C) = \boxed{\begin{array}{l}\{\Gamma_{\mathcal{C}}(C_1)\}\\ \{\Gamma_{\mathcal{C}}(C_2)\}\end{array}}$$

- Si $C = \texttt{if}(E)\{C_0\}$ où $E \in \mathcal{E}, C_0 \in \mathcal{C}$, on définit :

$$\Gamma_{\mathcal{C}}(C) = \boxed{\begin{array}{ll} & \{\Gamma_{\mathcal{E}}(E)\} \\ & \texttt{cmp rax, 0} \\ & \texttt{jz fin} \\ & \{\Gamma_{\mathcal{C}}(C_0)\} \\ \texttt{fin:} & \texttt{nop} \end{array}}$$

  où l'on suppose que l'étiquette `fin` est "fraîche", c'est-à-dire n'apparaît pas dans le reste du code.

- Si $C = \texttt{while}(E)\{C_0\}$ où $E \in \mathcal{E}, C_0 \in \mathcal{C}$, on définit :

$$\Gamma_{\mathcal{C}}(C) = \boxed{\begin{array}{ll} \texttt{debut:} & \{\Gamma_{\mathcal{E}}(E)\} \\ & \texttt{cmp rax, 0} \\ & \texttt{jz fin} \\ & \{\Gamma_{\mathcal{C}}(C_0)\} \\ & \texttt{jmp debut} \\ \texttt{fin:} & \texttt{nop} \end{array}}$$

  avec le même commentaire que ci-dessus, `debut` et `fin` sont supposées être des étiquettes fraîches.

- Si $C = \texttt{printf}(E)$ avec $E \in \mathcal{E}$, on définit :

$$\Gamma_{\mathcal{C}}(C) = \boxed{\begin{array}{l} \{\Gamma_{\mathcal{E}}(E)\} \\ \texttt{mov rdi, rax} \\ \texttt{mov rsi, long\_format} \\ \texttt{xor rax, rax} \\ \texttt{call printf} \end{array}}$$

La compilation du programme lui-même est traitée dans la section suivante.

5.1 La fidélité du compilateur

Nous allons montrer pour une instruction simple en quoi notre compilateur est fidèle. C'est-à-dire que le programme cible calcule bien ce qui est décrit dans le programme source. Comment montrer cela ? Nous allons modéliser l'adéquation entre les valuations et les configurations. Ensuite, étant donné une valuation σ et une configuration γ en adéquation, on montre qu'après

exécution d'une commande, disons C, le valuation obtenue est en adéquation avec la configuration résultant de l'exécution du code $\Gamma(C)$ sur γ. Ouf !

Une valuation σ et une configuration γ sont en adéquation si elles s'accordent sur la valeur des variables :

$$\text{Adequation}(\gamma, \sigma) \Leftrightarrow \forall \texttt{X} \in \mathcal{V} : \sigma(\texttt{X}) = \gamma.mem(\Gamma[\texttt{X}])$$

Quelques formules de logique pour modéliser proprement le problème. Étant donné une expression E ou une commande C (par la suite, un énoncé), on note $Size(E)$ (resp. $Size(C)$) la taille en mémoire du code de E (resp. C). La formule :

$$\text{PointsTo}(\gamma, T, a) \Leftrightarrow \gamma[@(a : a + Size(T))] = \Gamma(T)$$

où γ est une configuration, T un énoncé et $a \in QWORD$ indique que le contenu en mémoire entre a et $a+Size(T)$ est précisément le code de l'énoncé T. En d'autres termes, à l'adresse a, γ "pointe" sur le code de T.

La formule suivante :

$$\text{Run}(\gamma, T) \Leftrightarrow \text{PointsTo}(\gamma, T, \gamma[\texttt{rip}])$$

indique que le pointeur d'instruction est au début du code de T.

La formule :

$$\text{After}(\gamma, T) = \text{PointsTo}(\gamma, T, \gamma[\texttt{rip}] - Size(T))$$

indique que la configuration γ pointe sur le code apparaissant en mémoire juste après le code de T.

La fidélité du code d'une commande C revient à dire que

$$\text{Run}(\gamma, \texttt{C}) \wedge \text{Adequation}(\gamma, \sigma) \Rightarrow \text{Adequation}(\gamma', \sigma') \wedge \text{After}(\gamma', \texttt{C})$$

où $\gamma' = [\![\texttt{C}]\!](\gamma)$ et $\sigma' = [\![\texttt{C}]\!]_{\mathcal{C}}(\sigma)$.

Montrons que le calcul `X = 12` est fidèle. Si on suit la compilation, on obtient :

$\Gamma_{\mathcal{C}}(\texttt{X = 12}) =$

```
mov rax, 12
mov [X], rax
```

La fidélité du code revient à dire que :

$$\text{Run}(\gamma, \texttt{X = 12}) \wedge \text{Adequation}(\gamma, \sigma) \Rightarrow \text{Adequation}(\gamma', \sigma') \wedge \text{After}(\gamma', \texttt{X = 12})$$

où $\gamma' = [\![\texttt{X = 12}]\!](\gamma)$ et $\sigma' = [\![\texttt{X = 12}]\!]_{\mathcal{C}}(\sigma)$. En d'autres termes, si le pointeur d'instruction est au début du code de `X = 12`, que le magasin et la configuration s'accordent sur les variables, alors après le calcul :

- le nouveau magasin et la nouvelle configuration s'accordent sur les variables et
- la pointeur d'instruction sur la nouvelle configuration pointe sur l'instruction après le code `X = 12`.

Etant donné les définitions, $Size(\texttt{X = 12}) = 15 = 7 + 8$. Calculons

$$\begin{aligned} \gamma' &= [\![\texttt{mov [X], rax}]\!]([\![\texttt{mov rax, 12}]\!](\gamma)) \\ &= [\![\texttt{mov [X], rax}]\!](\gamma'') \end{aligned}$$

avec

$$\gamma'' = [\![\texttt{mov rax, 12}]\!](\gamma) = \gamma[\texttt{rax} \mapsto 12, \texttt{rip} \mapsto \gamma[\texttt{rip}] + 7] \tag{1}$$

Poursuivons l'équation :

$$\begin{aligned} \gamma' &= [\![\texttt{mov [X], rax}]\!](\gamma'') && (2)\\ &= \gamma''[@(\Gamma(X)) \mapsto \gamma''[\texttt{rax}], \texttt{rip} \mapsto \gamma''[\texttt{rip}] + 8] && (3)\\ &= \gamma''[@(\Gamma(X)) \mapsto 12, \texttt{rip} \mapsto \gamma[\texttt{rip}] + 7 + 8] && (4)\\ &= \gamma[@(\Gamma(X)) \mapsto 12, \texttt{rax} \mapsto 12, \texttt{rip} \mapsto \gamma[\texttt{rip}] + Size(\texttt{X = 12})] && (5) \end{aligned}$$

L'équation (4) est justifiée par l'équation (1), de même pour l'équation (5).

De l'autre côté, un calcul immédiat donne

$$\sigma' = [\![X = 12]\!](\sigma) \quad = \quad \sigma[X \mapsto 12] \tag{6}$$

De ce fait, il vient Adequation(γ', σ'). En effet, d'après les équations (6) et (5), pour tout $\texttt{Y} \neq \texttt{X}$, $\sigma'(\texttt{Y}) = \sigma(\texttt{Y}) = \gamma(\texttt{Y}) = \gamma'(\texttt{Y})$. Et $\sigma'(\texttt{X}) = 12 = \gamma'(\texttt{X})$.

Ensuite, l'équation (5) implique After$(\gamma', \texttt{X = 12})$.

Oups !
On fait dans ce qui précède l'hypothèse que $\texttt{X} \neq \texttt{Y} \Rightarrow [\Gamma(\texttt{X}), \ldots, \Gamma(\texttt{X}) + 8[\cap [\Gamma(\texttt{Y}), \ldots, \Gamma(\texttt{Y}) + 8[= \emptyset$. En d'autres termes, les variables sont stockées séparément en mémoire !

6 Implémentation

Nous arrivons maintenant à la dernière étape de notre générateur. Nous savons compiler les expressions et les commandes. Maintenant, il s'agit de mettre tout ensemble. Le compilateur va avoir globalement la structure suivante :

```
extern prinf, atoi   ;déclaration des fonctions externes
global main           ; declaration main
section .data        ; section des données
long_format: db "%lld",10, 0 ; format pour les uint64_t
argc : dd 0 ; copie de argc
argv : dd 0 ; copie de argv
; DECLARATION_DES_VARIABLES

section .text         ; instructions
main:
; OPENING
; INITIALISATION_DES_VARIABLES_MAIN
; BODY
; RETURN
; CLOSING
```

Voici une description courte du rôle des parties à implémenter.

- La partie `DECLARATION_DES_VARIABLES` réserve de la place pour les variables du programme. Cela nous permet d'implémenter la fonction $\Gamma(\mathtt{X})$ pour $\mathtt{X} \in \mathcal{V}$.
- La partie `OPENING` est en charge de préparer la pile et de sauvegarder `argc` et `argv`.
- La partie `INITIALISATION_DES_VARIABLES_MAIN` sera en charge d'initialiser les variables de `main` à partir du tableau `argv`.
- La partie `BODY` est le résultat de la génération du corps de `main`.
- La partie `RETURN` est en charge de calculer la valeur de retour et de l'afficher sur la sortie standard.
- La partie `CLOSING` remet la pile en bon état et termine le programme.

DECLARATION_DES_VARIABLES. Chaque variable est déclarée comme un "quad" :

```
X: dq 0
```

OPENING. Sauvegarde des paramètres `argc` et `argv`.

```
push rbp    ; Set up the stack. Save rbp
mov [argc], rdi
mov [argv], rsi
```

INITIALISATION_DES_VARIABLES_MAIN. Chaque argument de la fonction `main` est relié à sa valeur donnée via `argv` comme suit. Etant donné `X`, le i-ème argument de la fonction, on utilise :

```
mov rbx, [argv]
mov rdi, [rbx + { 8*(i+1)}]
xor rax, rax
call atoi
mov [{X}], rax
```

BODY. Obtenu directement avec la section qui précède.

RETURN. Étant donné l'expression E de retour,

$\{\Gamma_{\mathcal{E}}(E)\}$
```
mov rax, rdi
mov rsi, long_format
xor rax, rax
call printf
```

CLOSING. Remise en place de la pile, et `return` de la fonction.

```
pop rbp
xor rax, rax
ret
```

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[BT21] Guillaume Bonfante and Alexandre Talon. At the bottom of binary analysis: Instructions. In Esma Aïmeur, Maryline Laurent, Reda Yaich, Benoît Dupont, and Joaquín García-Alfaro, editors, *Foundations and Practice of Security - 14th International Symposium, FPS 2021, Paris, France, December 7-10, 2021, Revised Selected Papers*, volume 13291 of *Lecture Notes in Computer Science*, pages 311–320. Springer, 2021.

[Flo67] Robert W. Floyd. Assigning meaning to programs. 1967.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 1969.

[Jon97] Neil D. Jones. *Computability and Complexity, from a Programming Perspective*. MIT press, 1997.

[SS71] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. *Proceedings of the Symposium on Computers and Automata*, 21, 01 1971.

A Solution de certains exercices

La somme des `size` premiers entiers du tableau T. On suppose size stocké dans `RSI` et T dans

```
xor     EAX, EAX ; s = 0
test    RSI, RSI ; size == 0
je     fin
body_for:
add     RAX, QWORD PTR [RDI] ; s += T[size]
add     RDI, 8 ; T++
dec     RSI ; SIZE -- ; SIZE == 0
jne     body_for  ; if (SIZE != 0) goto body_for
fin:

test    RDX, RDX ; RDX == 0
je     fin ; if (RDX == 0) goto fin
debut:
mov     RAX, QWORD PTR [RSI] ; RAX = t[size]
add     QWORD PTR [RDI], RAX ; v[size] += RAX
add     RSI, 8 ; t++
add     RDI, 8 ; v++
dec     RDX ; size -- ; size == 0
jne     debut ; if ()size != 0) goto debut
fn:
```