



Interprétation et compilation

Chapitre 0 Introduction Présentation du cours



Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/ic

Introduction

- ▶ Il n'est pas déraisonnable de voir l'informatique comme une science des *langages*.



Noam Chomsky

Les langages sont des outils

- ▶ Les langages sont les outils de l'informaticien.
- ▶ Différents problèmes se règlent avec différents outils.
- ▶ Abstraction ultime : créer son propre langage.

- ▶ Un *interpréteur* est un programme qui prend en entrée un code source et exécute directement ses instructions.
- ▶ Les instructions sont interprétées dans le langage “hôte”.
- ▶ Le fonctionnement global d’un interpréteur est le cycle suivant :
 1. lire et analyser une expression ;
 2. si elle est valide (syntaxe, etc.), l’évaluer, sinon, rapporter une erreur ;
 3. passer à l’expression suivante.

Démonstration : mini interpréteur

- ▶ Démonstration d'un mini interpréteur "vite fait mal fait".
- ▶ Ajout d'une (ou deux) fonctionnalité(s).

- ▶ Un *compilateur* est un programme qui prend en entrée un code source et qui le traduit dans un autre langage, généralement de plus bas niveau.
- ▶ Le plus souvent le compilateur produit un fichier binaire exécutable, mais pas forcément.
- ▶ Le compilateur fait la traduction une fois pour toutes, contrairement à l'interpréteur.
- ▶ Un compilateur est *correct* si la *sémantique* du code produit est la même que celle du code source.

Démon : mini compilateur

- ▶ Démonstration d'un mini compilateur "vite fait mal fait".
- ▶ Ajout d'une (ou deux) fonctionnalité(s).

Structure globale d'un compilateur

- ▶ Idéalement, un compilateur est divisible en trois parties :
 - le *front end* (partie avant) est responsable de la lecture du code source et de sa transformation en *langage intermédiaire* ;
 - la *middle end* (partie centrale) est optionnelle, elle fait une ou plusieurs passes de traitement sur le langage intermédiaire (des optimisations par exemple) ;
 - le *back end* (partie finale) est responsable de produire le code final à partir du langage intermédiaire.

Structure globale d'un compilateur

- ▶ Idéalement, un compilateur est divisible en trois parties :
 - le *front end* (partie avant) est responsable de la lecture du code source et de sa transformation en *langage intermédiaire* ;
 - la *middle end* (partie centrale) est optionnelle, elle fait une ou plusieurs passes de traitement sur le langage intermédiaire (des optimisations par exemple) ;
 - le *back end* (partie finale) est responsable de produire le code final à partir du langage intermédiaire.

Quel est l'avantage de cette architecture selon vous ?

Structure globale d'un compilateur

- ▶ Idéalement, un compilateur est divisible en trois parties :
 - le *front end* (partie avant) est responsable de la lecture du code source et de sa transformation en *langage intermédiaire* ;
 - la *middle end* (partie centrale) est optionnelle, elle fait une ou plusieurs passes de traitement sur le langage intermédiaire (des optimisations par exemple) ;
 - le *back end* (partie finale) est responsable de produire le code final à partir du langage intermédiaire.

Quel est l'avantage de cette architecture selon vous ?

- ▶ Le code en langage intermédiaire n'a que rarement besoin d'être représenté textuellement, on manipule généralement des *arbres de syntaxe abstraite* (on parle d'AST, pour "abstract syntax tree").

Chaîne de compilation

- ▶ En fait, le compilateur à proprement parler n'est qu'une étape de la *chaîne de compilation*.
- ▶ Il y a parfois des outils qui passent avant le compilateur, par exemple :
 - un *préprocesseur* qui opère des transformations superficielles sur le code source,
 - un autre compilateur qui traduit d'un langage de haut niveau à un autre (par exemple vers le C).
- ▶ Selon le langage cible, il peut y avoir d'autres outils après le compilateur, par exemple :
 - un *assembleur* pour créer du code objet,
 - (puis) un *éditeur de lien* pour créer un exécutable,
 - une *machine virtuelle* dans le cas où le compilateur produit du "bytecode".

- ▶ Comment a-t-on écrit le premier compilateur ?

- ▶ Comment a-t-on écrit le premier compilateur ?
- ▶ Historiquement, le premier compilateur est A-0, il a été écrit par Grace Hopper.
Il s'agit plutôt d'un genre d'éditeur de lien, selon les standards actuels.



Grace Hopper

- ▶ Comment a-t-on écrit le premier compilateur ?
- ▶ Historiquement, le premier compilateur est A-0, il a été écrit par Grace Hopper.
Il s'agit plutôt d'un genre d'éditeur de lien, selon les standards actuels.
- ▶ Aujourd'hui, le problème ne se pose plus vraiment, sauf pour le *bootstrapping* d'un compilateur, c'est à dire écrire le compilateur d'un langage dans ce langage.

Bootstrapping

- ▶ Quels sont selon vous les avantages du bootstrapping ?
- ▶ Pouvez-vous imaginer quelques méthodes de bootstrapping ?

Reflections on Trusting Trust

- ▶ En 1984, Ken Thompson a reçu le prix Turing.
- ▶ Pendant son discours, il a présenté une idée intéressante à propos de la confiance qu'on peut avoir en un compilateur...
- ▶ <https://pablo.rauzy.name/dev/trustingtrust.pdf>

- ▶ Les liens entre interprétation et compilation sont assez étroits.

Langages interprétés et langages compilés

- ▶ On entend souvent parler de langages interprétés ou compilés, par exemple “python est un langage interprété, C est un langage compilé”.
- ▶ Il s’agit en fait d’un abus de langage.
- ▶ Il est théoriquement possible d’écrire des interpréteurs et des compilateurs pour tout type de langage.
- ▶ En pratique, il existe tout un spectre de techniques intermédiaires entre l’interprétation pure et la compilation pure, par exemple :
 - bytecode et machine virtuelle (VM),
 - compilation “just-in-time” (JIT),
 - et parfois même les deux en même temps !

Des parties communes dans le “front end”

- ▶ L'*analyse lexicale* découpe le texte du code source en une suite de *lexèmes*.
- ▶ L'*analyse syntaxique* structure le code en fonction d'une *grammaire formelle*.
- ▶ L'*analyse sémantique* vérifie la *validité* du code.

Ce qui diffère

- ▶ Comme on l'a déjà vu :
 - l'interpréteur évalue/exécute le code directement et *au fur et à mesure*,
 - le compilateur analyse d'abord l'entièreté du code source puis génère du code cible.

Présentation du cours

- ▶ En cours :
 - Détailler et approfondir tout ce qu'on a vu superficiellement aujourd'hui.
 - Écrire des interpréteurs et des compilateurs.
 - Utiliser les bons outils pour ça.
- ▶ En TP :
 - Mettre en pratique les notions du cours.
 - Apprendre à se servir des outils.
- ▶ Projet :
 - Écrire un compilateur d'un sous-ensemble du langage Python vers l'assembleur MIPS.
 - En utilisant un langage fonctionnel (Racket ou OCaml) et le simulateur `spim`.

- ▶ Le cours va être composé de modules d'une ou deux semaines.
 1. Analyse lexicale.
 2. Analyse syntaxique.
 3. Analyse sémantique.
 4. Langage assembleur.
 5. Génération de code.
 6. Transformations de code.

- ▶ Chaque module fera l'objet d'au moins un TP.

- ▶ Votre évaluation pour ce cours prendra en compte (dans l'ordre d'importance) :
 - le projet,
 - les TPs (potentiellement tous les TPs seront notés !)
 - un exam écrit sur la partie théorique.

- ▶ La propreté du code (nommage, indentation, organisation) est importante est sera prise en compte autant pour les TP que pour le projet.

▶ Pour Racket :

- Plus d'information :
<https://racket-lang.org/>
- Tutoriel d'initiation :
<https://docs.racket-lang.org/guide/to-scheme.html>.

▶ Pour OCaml :

- Plus d'information :
<https://ocaml.org/>
- Tutoriel d'initiation (en français) :
<https://www.lri.fr/~filliatr/publis/ipf.pdf>