École Polytechnique

CSC_52064 – Compilation

Jean-Christophe Filliâtre

introduction

9 blocs, du 6 janvier au 10 mars

- **cours** 14h00–16h00
- **TD** dans la foulée, 16h15–18h15 en salles info 32–33
  - avec Wendlasida Ouedraogo et moi-même

- un **examen** écrit (17 mars, 14h–17h)

- un **projet** = un mini compilateur vers x86-64
  - réalisé en TD (à partir du TD 4) et un peu en dehors
  - seul ou en binôme
  - en Java ou en OCaml

$$note\ finale = \frac{examen + projet}{2}$$

page web du cours

https://www.enseignement.polytechnique.fr/informatique/CSC_52064/

- transparents
- TD

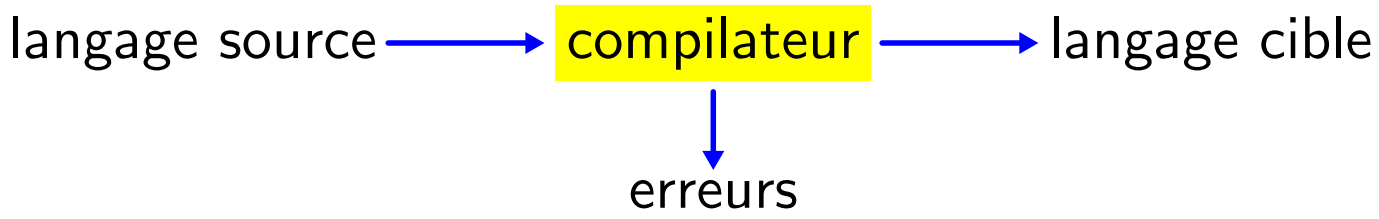
- archives examen
- ressources diverses : outils, lectures, sites web, etc.

objectif du cours

maîtriser les mécanismes de la **compilation**,
c'est-à-dire de la transformation d'un langage dans un autre

comprendre les différents aspects des **langages de programmation**
par le biais de la compilation

schématiquement, un compilateur est un programme qui traduit un ≪ programme ≫ d'un langage **source** vers un langage **cible**, en signalant d'éventuelles erreurs



compilation vers le langage machine

quand on parle de compilation, on pense typiquement à la traduction d'un langage de haut niveau (C, Java, OCaml...) vers le langage machine d'un processeur

```
% gcc -o sum sum.c
```

source `sum.c` → compilateur C (`gcc`) → exécutable `sum`

```
int main(int argc, char **argv) {
  int i, s = 0;
  for (i = 0; i <= 100; i++) s += i*i;
  printf("0*0+...+100*100 = %d\n", s);
}
```

→

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
...
```

langage cible

dans ce cours, nous allons effectivement nous intéresser à la compilation vers de **l'assembleur**, mais ce n'est qu'un aspect de la compilation

un certain nombre de techniques mises en œuvre dans la compilation ne sont pas liées à la production de code assembleur

certains langages sont d'ailleurs

- interprétés (Basic, COBOL, Ruby, etc.)
- compilés dans un langage intermédiaire qui est ensuite interprété (Java, Python, OCaml, Scala, etc.)
- compilés à la volée (Julia, etc.)
- compilés vers un autre langage de haut niveau

différence entre compilateur et interprète

un **compilateur** traduit un programme P en un programme Q tel que pour toute entrée x, la sortie de $Q(x)$ soit la même que celle de $P(x)$

$$\forall P \; \exists Q \; \forall x ...$$

un **interprète** est un programme qui, étant donné un programme P et une entrée x, calcule la sortie s de $P(x)$

$$\forall P \; \forall x \; \exists s ...$$

différence entre compilateur et interprète

dit autrement,

le compilateur fait un travail complexe **une seule fois**, pour produire un code fonctionnant pour n'importe quelle entrée

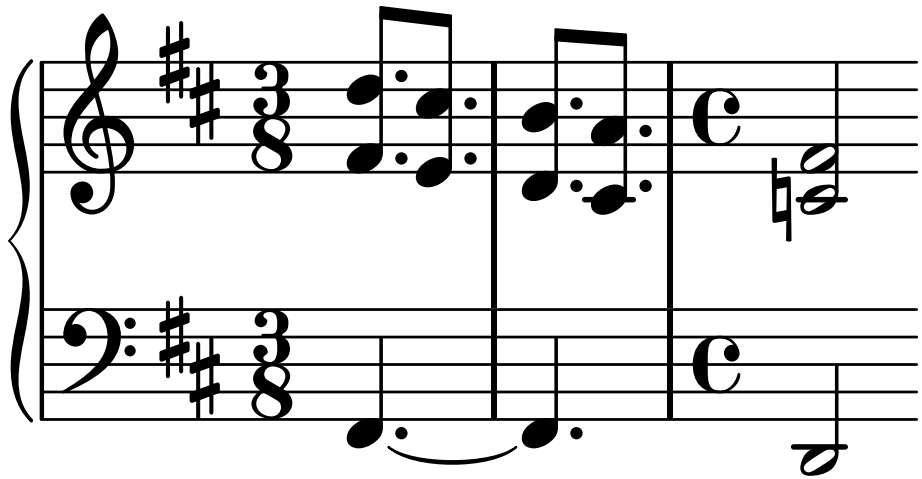
l'interprète effectue un travail plus simple, mais le refait sur chaque entrée

autre différence : le code compilé est généralement bien plus efficace que le code interprété

exemple de compilation et d'interprétation

source → `lilypond` → fichier PDF → `evince` → image

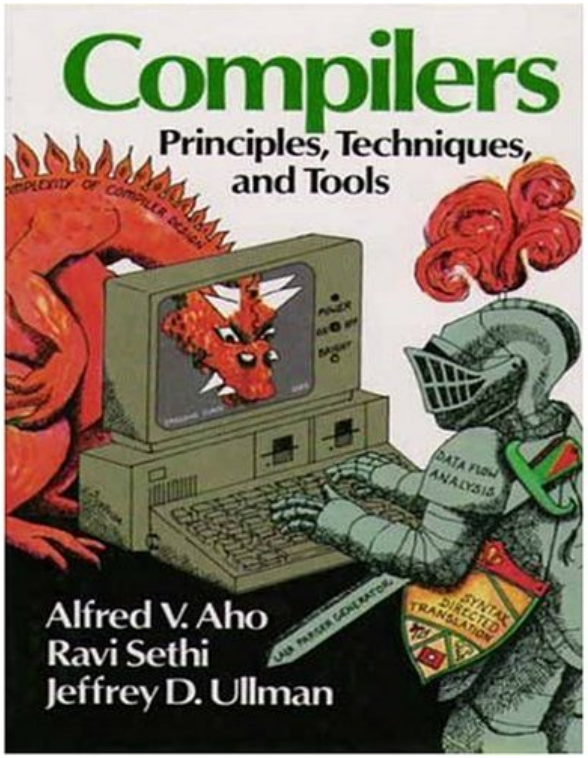
```
\new PianoStaff <<
  \new Staff { \clef "treble" \key d \major \time 3/8
               <<d8. fis,8.>> <<cis'8. e,8.>> | ... }
  \new Staff { \clef "bass" \key d \major
               fis,,4. ~ | fis4. | \time 4/4 d2 }
>>
```



qualité d'un compilateur

à quoi juge-t-on la qualité d'un compilateur ?

- à sa correction
- à l'efficacité du code qu'il produit
- à sa propre efficacité



"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free ! Thus, the most important objective in writing a compiler is that it is correct."

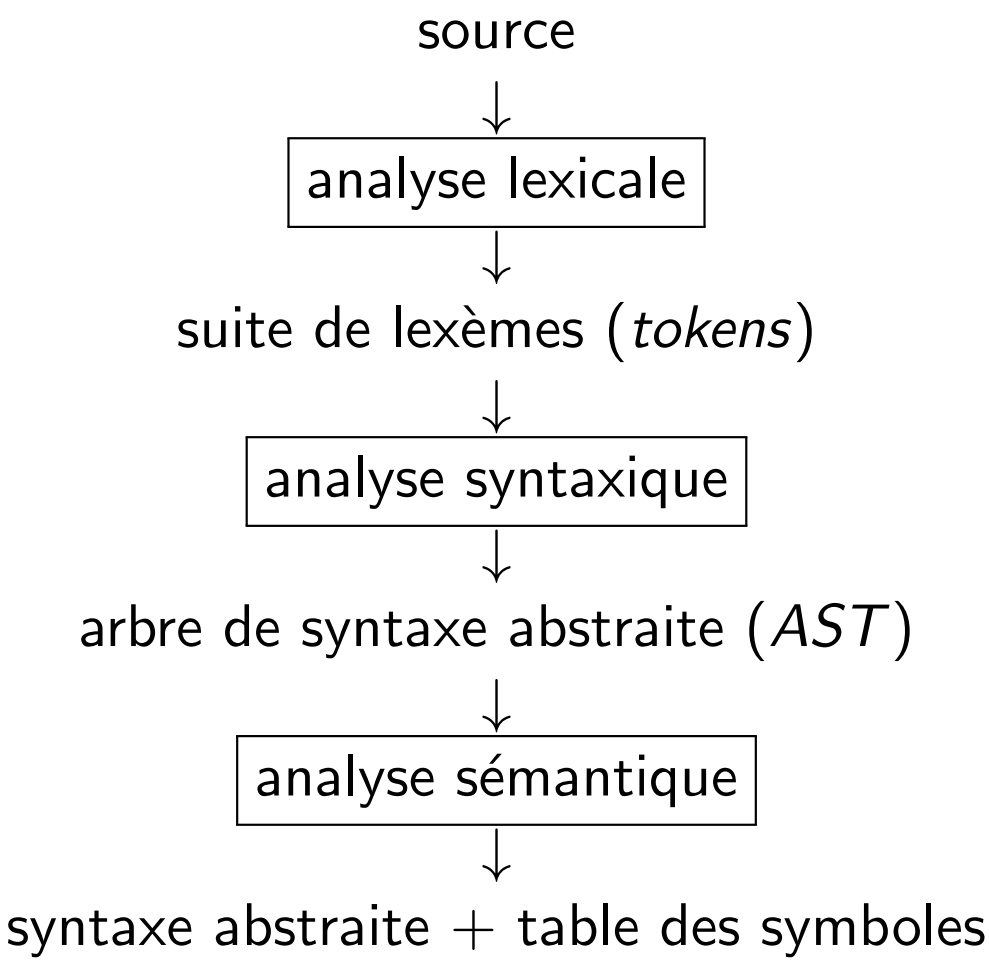
(Dragon Book, 2006)

phases d'un compilateur

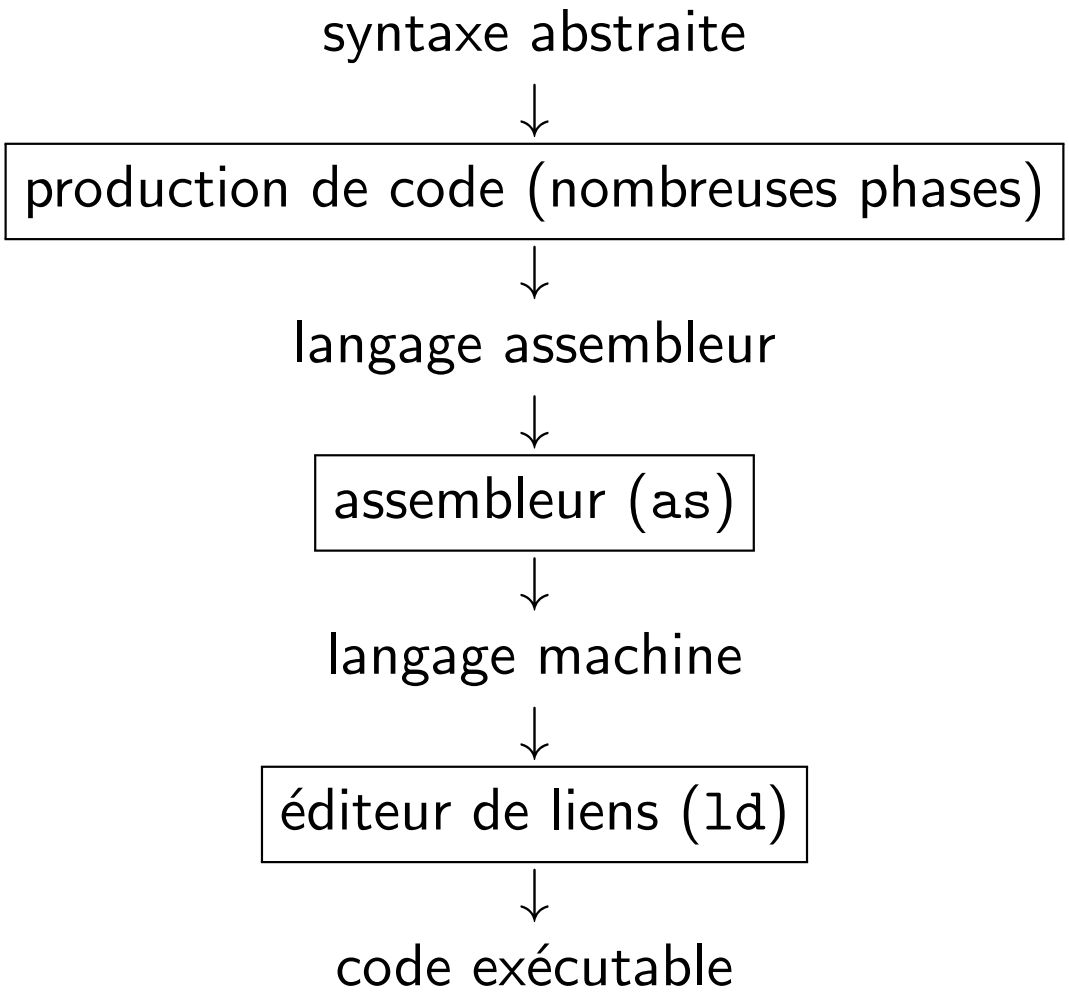
typiquement, le travail d'un compilateur se compose

- d'une phase d'**analyse** (*frontend*)
  - reconnaît le programme à traduire et sa signification
  - signale les erreurs et peut donc échouer
    (erreurs de syntaxe, de portée, de typage, etc.)

- puis d'une phase de **synthèse** (*backend*)
  - production du langage cible
  - utilise de nombreux langages intermédiaires
  - n'échoue pas


source
analyse lexicale
suite de lexèmes (*tokens*)
analyse syntaxique
arbre de syntaxe abstraite (*AST*)
analyse sémantique
syntaxe abstraite + table des symboles

phase de synthèse



plan de ce cours