# Langages interprétés vs compilés

Prof d'info

Lycée Thiers

# Crédits

- Sur la différence entre langage interprété et compilé France IOI

# Crédits

- Sur la différence entre langage interprété et compilé France IOI
- sur les langages interprétés Wikipedia

# Langage interprété

- Dans un langage *interprété*, le code source (écrit par l'utilisateur) est interprété par un logiciel qu'on appele un *interpréteur* (voir figure 1).

# Langage interprété

- Dans un langage *interprété*, le code source (écrit par l'utilisateur) est interprété par un logiciel qu'on appele un *interpréteur* (voir figure 1).
- L'interprétation est une exécution dynamique du programme par un autre programme (l'interprète). Il n'y a pas de séparation entre les temps de conversion (traduction en langage machine) et d'exécution.

# Langage interprété

- Dans un langage *interprété*, le code source (écrit par l'utilisateur) est interprété par un logiciel qu'on appele un *interpréteur* (voir figure 1).
- L'interprétation est une exécution dynamique du programme par un autre programme (l'interprète). Il n'y a pas de séparation entre les temps de conversion (traduction en langage machine) et d'exécution.
- Un code source qui a vocation à être fourni à un interpréteur est appelé un *script* (bash, Python, PHP, HTML, réquêtes SQL).

# Cycle d'interprétation

Le *cycle* d'un interprète est le suivant :

- lire et analyser une instruction ou une expression,

# Cycle d'interprétation

Le *cycle* d'un interprète est le suivant :

- lire et analyser une instruction ou une expression,
- si l'instruction (resp. expression) est correcte syntaxiquement, l'exécuter (resp. évaluer)

# Cycle d'interprétation

Le *cycle* d'un interprète est le suivant :

- lire et analyser une instruction ou une expression,
- si l'instruction (resp. expression) est correcte syntaxiquement, l'exécuter (resp. évaluer)
- passer à l'instruction suivante

# Cycle d'interprétation

Le *cycle* d'un interprète est le suivant :

- lire et analyser une instruction ou une expression,
- si l'instruction (resp. expression) est correcte syntaxiquement, l'exécuter (resp. évaluer)
- passer à l'instruction suivante

Il y a donc exécution/évaluation au fur et à mesure de la lecture des instructions/expressions du code source.

# Langage interprété

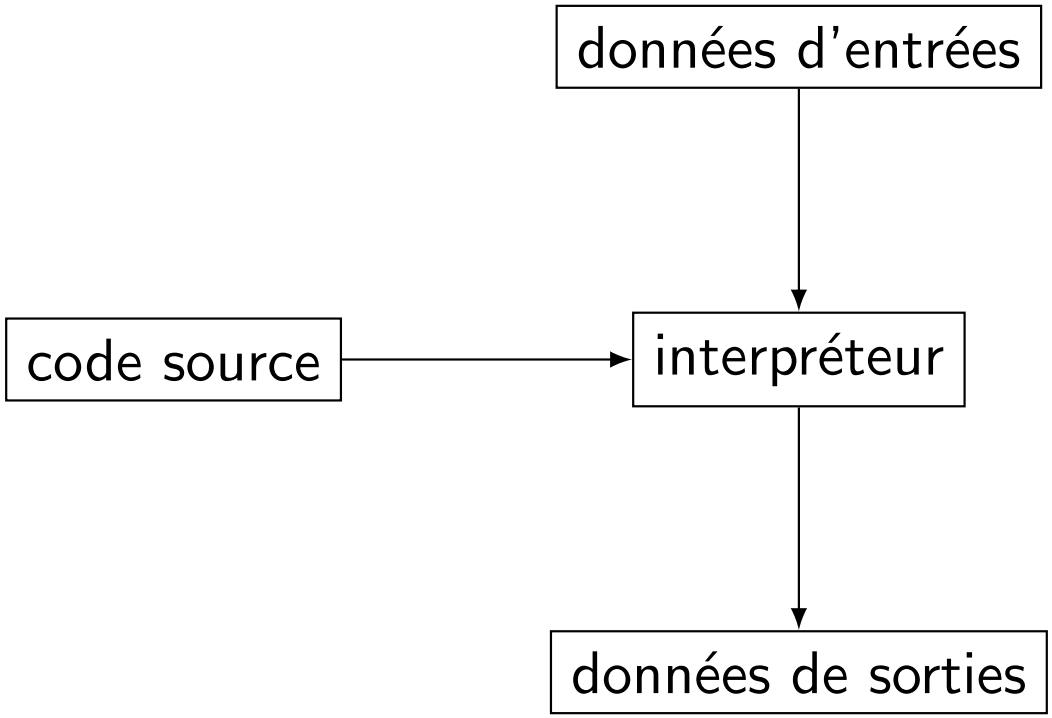


FIGURE 1 – Langage interprété

# Langage compilé

- Au contraire, la *compilation* repose sur la traduction du code source en un autre langage : le langage machine (voir figure 2)...

# Langage compilé

- Au contraire, la *compilation* repose sur la traduction du code source en un autre langage : le langage machine (voir figure 2)...
- L'exécution se fait une fois la compilation effectuée, directement à partir de la traduction en langage machine.

# Langage compilé

- Au contraire, la *compilation* repose sur la traduction du code source en un autre langage : le langage machine (voir figure 2)...
- L'exécution se fait une fois la compilation effectuée, directement à partir de la traduction en langage machine.
- Il y a dans ce cas une séparation des temps de conversion et d'exécution.

# Langage compilé

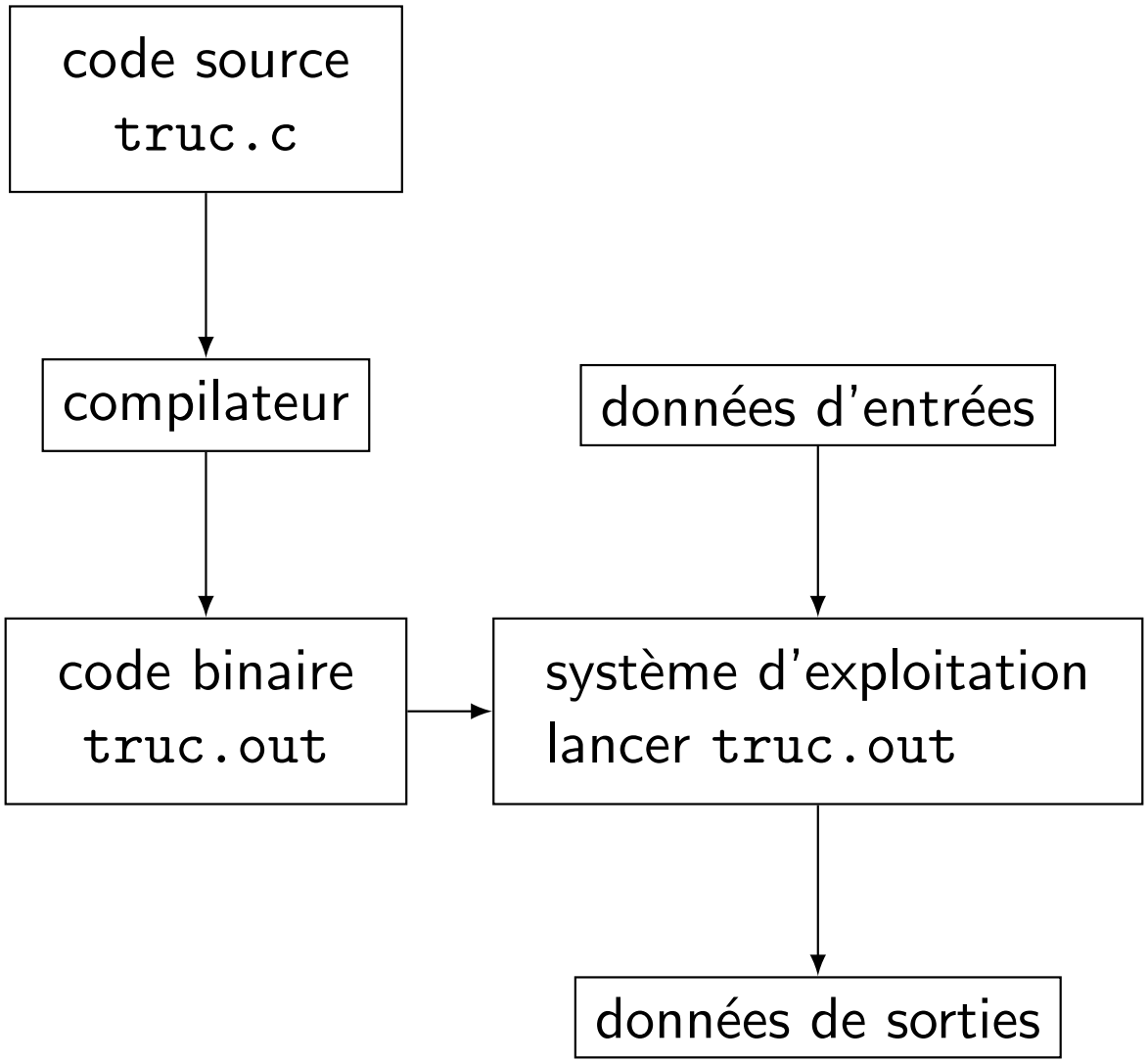


FIGURE 2 – Langage compilé

# Notion de shell

Le *shell* est un programme qui reçoit des commandes informatiques données par un utilisateur à partir de son clavier pour les envoyer au système d'exploitation qui se chargera de les exécuter.

# Rappel

Une fois écrit le code source, deux possibilités s'offrent pour l'exécution selon la nature du langage :

- Un programme *script* est un programme exécuté à partir du fichier source via un interprète de script (ex : shell bash, ipython).

# Rappel

Une fois écrit le code source, deux possibilités s'offrent pour l'exécution selon la nature du langage :

- Un programme *script* est un programme exécuté à partir du fichier source via un interprète de script (ex : shell bash, ipython).
- Un programme *compilé* est exécuté à partir d'un bloc en langage machine issu de la traduction du fichier source (ex :OCAML, C, Java...).

# Avantages/inconvénients

- Dans un langage interprété, le même code source pourra marcher directement sur tout ordinateur : plus grande portabilité.
  Avec un langage compilé, il faudra (en général) tout recompiler à chaque fois ce qui peut être long.

# Avantages/inconvénients

- Dans un langage interprété, le même code source pourra marcher directement sur tout ordinateur : plus grande portabilité.
  Avec un langage compilé, il faudra (en général) tout recompiler à chaque fois ce qui peut être long.
- Dans un langage compilé, le programme est directement exécuté sur l'ordinateur, donc il sera en général plus rapide que le même programme dans un langage interprété.

# Avantages/inconvénients

- Dans un langage interprété, le même code source pourra marcher directement sur tout ordinateur : plus grande portabilité.
  Avec un langage compilé, il faudra (en général) tout recompiler à chaque fois ce qui peut être long.
- Dans un langage compilé, le programme est directement exécuté sur l'ordinateur, donc il sera en général plus rapide que le même programme dans un langage interprété.
- Langage interprété : la recherche de bug est facilitée (pas besoin de tout recompiler pour trouver l'erreur).

# Hybridation

- Il existe des méthodes intermédiaires entre les deux techniques de comiplation et d'interprétation.

# Hybridation

- Il existe des méthodes intermédiaires entre les deux techniques de comiplation et d'interprétation.
- Certains langages sont mis en œuvre à partir d'une *marchine virtuelle applicative*. Il s'agit d'une technique à mi-chemin entre les interprètes et les compilateurs.

# Hybridation

- Il existe des méthodes intermédiaires entre les deux techniques de comiplation et d'interprétation.
- Certains langages sont mis en œuvre à partir d'une *marchine virtuelle applicative*. Il s'agit d'une technique à mi-chemin entre les interprètes et les compilateurs.
- Le langage `Python` (via les fichiers `.pyc`, fichiers binaires précompilés), en est un exemple mais des portages via une machine virtuelle sont aussi réalisés par Java, OCAML, Perl....

# Hybridation

- Il existe des méthodes intermédiaires entre les deux techniques de comiplation et d'interprétation.
- Certains langages sont mis en œuvre à partir d'une *marchine virtuelle applicative*. Il s'agit d'une technique à mi-chemin entre les interprètes et les compilateurs.
- Le langage `Python` (via les fichiers `.pyc`, fichiers binaires précompilés), en est un exemple mais des portages via une machine virtuelle sont aussi réalisés par Java, OCAML, Perl....
- Nous utiliserons souvent la *boucle interractive* de OCaml pour écrire nos programmes et les debugger. Il s'agit en gros d'un interpréteur OCaml.

# Production d'un fichier `.pyc`

- Les fichiers `.pyc` sont des bytecodes compilés qui sont générés par l'interpréteur Python (et ensuite exécutés par une machine virtuelle). Leur code peut être exécuté directement par l'interpréteur sans refaire tout le processus d'analyse de code source et de traduction en instructions machines.

# Production d'un fichier `.pyc`

- Les fichiers `.pyc` sont des bytecodes compilés qui sont générés par l'interpréteur Python (et ensuite exécutés par une machine virtuelle). Leur code peut être exécuté directement par l'interpréteur sans refaire tout le processus d'analyse de code source et de traduction en instructions machines.
- Ces fichiers `.pyc` sont par exemple générés automatiquement par l'interpréteur Python quand un script importe un module.

# Production d'un fichier `.pyc`

- Les fichiers `.pyc` sont des bytecodes compilés qui sont générés par l'interpréteur Python (et ensuite exécutés par une machine virtuelle). Leur code peut être exécuté directement par l'interpréteur sans refaire tout le processus d'analyse de code source et de traduction en instructions machines.
- Ces fichiers `.pyc` sont par exemple générés automatiquement par l'interpréteur Python quand un script importe un module.
- Ils sont stockés dans le même répertoire que celui d'où le script a été appelé. Ils ont le même nom que le fichier `.py` mais leur extension est `.pyc` (passage de `truc.py` à `truc.pyc`).

# Production d'un fichier `.pyc`

- Les fichiers `.pyc` sont des bytecodes compilés qui sont générés par l'interpréteur Python (et ensuite exécutés par une machine virtuelle). Leur code peut être exécuté directement par l'interpréteur sans refaire tout le processus d'analyse de code source et de traduction en instructions machines.
- Ces fichiers `.pyc` sont par exemple générés automatiquement par l'interpréteur Python quand un script importe un module.
- Ils sont stockés dans le même répertoire que celui d'où le script a été appelé. Ils ont le même nom que le fichier `.py` mais leur extension est `.pyc` (passage de `truc.py` à `truc.pyc`).
- Ils sont dépendants de la version de Python utilisés et ne sont donc pas vraiment portables.

# Exemple

```python
# dans un fichier my_module.py
def my_function():
    print("Hello, world!")

my_function()
```

```python
# dans un fichier main.py
import my_module

# Quand main.py est exécuté la 1ere fois
# Python génère un fichier my_module.cpython-311.pyc
```

# Exécution

```
$ python main.py
Hello, world!
```

```
$ls *py*
main.py   my_module.py

__pycache__:
my_module.cpython-311.pyc
```

- Un fichier `.pyc` a bien été créé et compilé.

# Exécution

```
$ python main.py
Hello, world!
```

```
$ls *py*
main.py   my_module.py

__pycache__:
my_module.cpython-311.pyc
```

- Un fichier `.pyc` a bien été créé et compilé.
- Si le fichier `.pyc` existe et est à jour, Python l'utilise au lieu de recompiler le code source de `my_module.py`.