Comment fonctionne l'interpréteur d'un langage de programmation ?

Jean Dubois

12 mai 2024

1 Introduction

Les ordinateurs sont des machines traitant des données et pouvant être programmées. On estime que le premier vrai algorithme a été écrit pour la machine analytique imaginée en 1834 par Charles Babbage, par la première informaticienne de l'humanité : Ada Lovelace. Cette machine ne fut cependant jamais construite. Les programmes écrits par Ada Lovelace étaient rédigés en langage mathématique, et non dans un langage de programmation tel qu'on en connaît aujourd'hui. Cependant, elle fut la première à comprendre la différence entre matériel et logiciel. [1, p. 454]

Les premiers ordinateurs électroniques, tel que l'ENIAC (Electronic Numerical Integrator and Computer), se programmait en connectant physiquement les différents modules entre eux. [2] Après cela, les ordinateurs ont progressivement adoptés un langage machine, où chaque instruction est représentée par un nombre, puis un langage assembleur, qui fait correspondre chaque instruction à un mot-clé mnémotechnique. [3]

Le premier vrai langage de programmation à proprement parler est Fortran, pour mathematical FORmula TRANslating system, créé en 1954 pour le calcul scientifique [1, p. 455]. Fortran était un langage compilé et haut-niveau, qui essayait au mieux de reproduire la notation mathématique. Jusqu'alors, un programme était une suite d'instructions pas-à-pas pour le processeur. En Fortran, l'idée était plutôt de décrire le problème que l'on veut résoudre [4], et cette idée va être à l'origine de tous les autres langages de programmations jusqu'à nos jours.

Fortran est ensuite suivi par LISP (LISt Processor), qui est le premier langage interprété.

Nous allons nous intéresser dans cet exposé au fonctionnement des langages interprétés, bien que nous parlerons également des langages compilés. Nous expliciterons d'abord les points communs et les différences entre ces deux types de langage. Nous verrons ensuite dans le détail comment un interpréteur fonctionne. Enfin, nous nous intéresserons à comment implémenter un interpréteur en langage Python, et notamment en nous intéressant à l'interpréteur du langage Nougaro, écrit par mes soins.

2 Langages interprétés et langages compilés

Tout d'abord, qu'est-ce qu'un langage de programmation ? Un langage de programmation permet de communiquer des instructions à un ordinateur. Comme on l'a déjà discuté, les ordinateurs ne parlent qu'en langage machine, dans lequel on parle en cases mémoires et instructions basiques. Dans un langage de programmation, les instructions sont plus abstraites, plus proches de comment pense un être humain. [1, p. 55]

Cependant, il faut traduire ce langage de programmation en langage machine. C'est le rôle des compilateurs. Un compilateur prend en entrée un programme écrit dans un certain langage et le traduit dans un autre langage. [5] À l'origine, cet autre langage était le langage machine, mais de plus en plus de compilateurs traduisent maintenant vers le C, qui a le mérite de posséder des dizaines de compilateurs stables vers le langage machine, et d'être un des langages les plus utilisés sur Terre.

On peut remarquer que la plupart des compilateurs sont écrits dans le langage qu'ils sont censés interpréter (*GCC* est écrit en C, *RustCompiler* est écrit en Rust, etc.) Cela peut paraître paradoxal, mais a en fait une explication : il suffit de programmer le compilateur avec un certain langage, et d'écrire une version très simple de notre langage. Puis, on utilise notre compilateur pour le faire lui-même se compiler. On peut ensuite utiliser notre langage pour écrire notre vrai compilateur. On compile ce programme grâce

à notre premier compilateur puis on le recompile avec le résultat de la compilation. Ce dernier compilateur ne contient plus aucune trace du langage originel ni des étapes intermédiaires. Ce principe se nomme le *bootstraping*. [1, p. 448]

Un interpréteur, quant à lui, est un programme qui exécute pas-à-pas les instructions indiquées dans le programme. Un interpréteur ne traduit pas vraiment vers un autre langage, mais interprète ce que l'utilisateur a rentré. [5] Nous ne nous intéresserons désormais qu'aux interpréteurs.

3 Fonctionnement détaillé d'un interpréteur

Il y a trois grandes façons d'interpréter un programme : analyser directement le code et l'exécuter dans la foulée, transformer le code vers une représentation intermédiaire et exécuter cette représentation, et compiler vers un langage intermédiaire plus simple et interpréter ce résultat. [6] Je m'intéresserai à la seconde méthode.

Un interpréteur est constitué de trois parties distinctes, chacune s'exécutant l'une après l'autre. Ces trois parties sont l'analyseur lexical (lexer), l'analyseur syntaxique (parser), et l'interpréteur à proprement parler.

3.1 Analyse lexicale

L'analyse lexicale a pour but de découper le code source en petits bouts, des « tokens » en anglais, que l'on pourrait traduire en « unités » .

Une unité est composée d'un type, d'une position, et éventuellement d'une valeur. Ces unités peuvent être de plusieurs types : entier, flottant, chaîne de caractères, signe `+`, signe `=`, mot-clé, symbole (identifiant), etc.

Par exemple, le code suivant :

```
var a = 0
for i = 0 to 10 then
    var a += i
    print(a)
end
```

sera traduit par l'analyseur lexical vers la liste d'unités suivante :

```
keyword:var, identifier:a, eq, int:0, newline,
keyword:for, identifier:i, eq, int:0, keyword:to, int:10, keyword:then,
    newline
keyword:var, identifier:a, pluseq, identifier:i, newline,
identifier:print, lparen, identifier:a, rparen, newline,
keyword:end, end-of-file
```

L'analyseur lexical repère les identifiants (`identifier`), et les distingue des des mot-clés (`keyword`) grâce à une liste. Les symboles `=` et `+=` sont traduits par les unités `eq` et `pluseq`.

Il est important de noter que pour l'analyseur lexical, la notion de syntaxe n'existe pas encore. Par exemple, lui donner `while for True in print 78.5 += "chaîne"` ne sortira pas d'erreur, et on aura la liste d'unités correspondante en sortie. Il n'y a pas non plus encore de notion de contexte et de variables définies ou non, il ne donnera donc pas d'erreur si un identifiant n'est pas défini.

La liste d'unités que l'analyseur lexical donnera en sortie sera donnée en entrée à l'étape suivante : l'analyseur syntaxique (ou parser).

3.2 Analyse syntaxique

L'analyse syntaxique a pour but de construire un arbre de syntaxe (en anglais Abstract Syntax Tree, AST), constitué de nœuds, à partir des unités de l'analyseur lexical et d'une grammaire définie.

Pour cela, il parcourt les unités une à une et repère les structures. Le but est, pour chaque unité, de parcourir la grammaire qu'on s'est définie.

Par exemple, avec notre programme de tout à l'heure, on trouve d'abord un mot-clé `var`. On sait qu'on est dans une déclaration de variable, on sait alors que l'unité suivante doit impérativement être un identifiant. Si ce n'est pas le cas, on renvoie une erreur. Ici, on trouve un identifiant, de valeur `a`. On

doit ensuite trouver un signe `=`, puis une valeur. La valeur peut être n'importe quelle expression, ici il s'agit simplement d'un entier : 0. On s'attend ensuite à trouver un retour-ligne, car cette instruction est terminée. Une fois qu'on a fait ça, on ajoute à notre liste de nœuds un nœud de déclaration de variable. Ce nœud possède une position, un nom de variable, une valeur.

On continue à parcourir les unités, en on trouve le mot-clé `for`. On sait qu'on doit trouver ensuite un identifiant, puis il y a deux syntaxes (`for i in` et `for i = to`). Ici, on trouve un signe `=`, on sait qu'on est dans la deuxième syntaxe. On continue donc jusqu'au `then`, puis au retour-ligne. À partir de là, on ajoute chaque instruction présente dans la boucle à une liste de nœuds, qui sera elle-même stockée dans le nœud de la boucle `for`. On continue chaque instruction jusqu'au `end`.

Une chose intéressante est l'appel de fonction : on ne sait pas si c'est un simple accès à une variable ou un appel avant d'avoir trouvé l'unité `lparen` correspondant à la parenthèse ouvrante.

C'est donc l'analyseur syntaxique qui s'occupe de savoir si la grammaire est respectée. Dans notre exemple `while for True in print 78.5 += "chaîne"`, l'analyseur syntaxique ne s'attendrait pas à trouver le `for` après le `while`, et renverrait donc une erreur. En revanche, l'analyseur syntaxique ne s'occupe toujours pas de si une variable est définie ou non. Ainsi, si l'on n'avait pas définit la variable `a` au début de notre code, l'analyseur syntaxique n'aurait pas renvoyé d'erreur.

À la fin de l'analyse syntaxique, on se retrouve avec une liste de nœuds, que l'on peut envoyer vers l'interpréteur, qui va s'occuper d'exécuter réellement le code.

3.3 Interpréteur

L'interpréteur, en anglais *interpreter* ou plus spécifiquement *runtime*, est la dernière étape, celle qui exécute l'arbre de syntaxe directement. L'interpréteur stocke un contexte, qui contient plusieurs informations utiles : les variables dans un dictionnaire identifiant/valeurs, ou encore s'il faut arrêter ou continuer une boucle au cas où on aurait trouvé un mot-clé `continue` ou `break`.

L'interpréteur parcourt l'arbre de syntaxe, et exécute chaque nœud. Par exemple, s'il trouve un nœud `CallNode` d'identifiant `print` et d'argument `a`, il va résoudre le nom `a` dans sa table de symboles, puis va résoudre le nom `print`, qui va afficher la valeur de `a` à l'écran.

4 Une implémentation en Python

(détailler le jour de l'oral si le temps)
(parler des limites : Python n'est pas fait pour de la production et surtout c'est très lent)

Références

[1] Gérard Berry. *L'Hyperpuissance de l'informatique. Algorithmes, données, machines, réseaux*. Odile Jacob, 2017. ISBN : 9782738139535.

[2] Paul A. Frieberger et Michael R. Swaine. *ENIAC*. 2008. URL : `https://www.britannica.com/technology/ENIAC`. (sur *Britannica*, accédé le 2024/04/30).

[3] William L. Hosch. *Assembly Language*. 2009. URL : `https://www.britannica.com/technology/assembly-language`. (sur *Britannica*, accédé le 2024/04/30).

[4] *John Backus*. URL : `https://www.ibm.com/history/john-backus`. (sur *IMB*, accédé le 2024/04/30).

[5] *What is a programming language ?* URL : `https://resources.github.com/software-development/what-is-a-programming-language/`. (sur *GitHub Resources*, accédé le 2024/05/01).

[6] *Interpreter (computing)*. URL : `https://en.wikipedia.org/wiki/Interpreter_(computing)`. (sur *Wikipedia*, accédé le 2024/05/01).

Licence