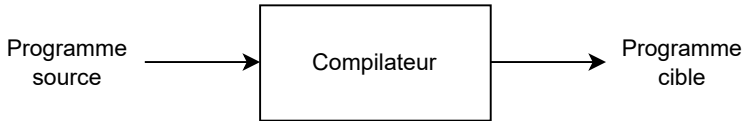
Notes de cours
Semaine 24

Cours Turing

1 Conception d'un compilateur

Pour ces dernières semaines de cours, nous allons nous plonger dans la conception d'un *compilateur*. Le but d'un compilateur est de transformer des programmes écrits dans un premier langage de programmation (le langage source) vers un second langage de programmation (le langage cible). Les constructions offertes par le langage cible sont généralement plus proches de celles offertes par les diverses machines sur lesquels les programmes doivent pouvoir s'exécuter. On appelle les langages dont les abstractions sont proches de celles offertes par un ordinateur un langage de bas niveau. À l'inverse, on appelle langage de haut niveau un langage qui offre des abstractions plus éloignées de celles offertes par les machines sur lesquels il s'exécute. Les langages de haut niveau ont pour vocation de faciliter la programmation par leurs utilisateurs.



Concevoir un compilateur est un moyen de donner vie à un langage de programmation ! Un compilateur permet de traduire des programmes exprimés dans un langage que l'on ne sait pas, a priori, exécuter, vers un autre langage dont on est capable d'exécuter les programmes. Les langages cibles sont souvent des *langages machine*, c'est-à-dire des langage directement exécutables par un ordinateur, mais pas obligatoirement.

Il existe de nos jours de nombreux compilateurs permettant de passer de nombreux langages sources à de nombreux langages cibles. Parmi les exemples les plus connu est `gcc`. Le programme `gcc` est un compilateur qui permet de traduire des programmes écrits en langage C (entre autres) vers divers langages machine proposés par différentes architectures d'ordinateurs.

Certains compilateurs génèrent du code qui n'est pas destiné à une machine physique mais à une machine *virtuelle* (un logiciel qui exécute du code). C'est par exemple le cas pour `java`, un compilateur qui permet de traiter des programmes écrits en Java et de les traduire en code directement exécutable sur la JVM (*Java Virtual Machine*).

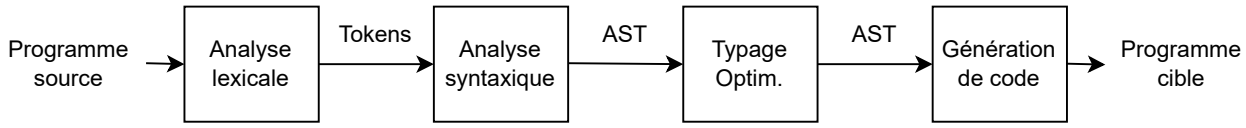
Notez que le langage cible d'un compilateur n'est pas forcément un langage machine : il existe des compilateurs qui permettent de traduire vers d'autres langages de plus haut niveau, comme vers JavaScript par exemple. Ainsi, il existe un compilateur pour traduire de TypeScript à JavaScript, un compilateur pour traduire de CoffeeScript à JavaScript, et ainsi de suite.

Comme tout programme, les compilateurs sont implémentés dans un langage de programmation. Dans notre cas, nous utiliserons comme langage d'implémentation Python. Notez que

ce choix ne restreint en rien ni le langage source ni le langage cible. De manière intéressante, de nos jours, un certain nombre de compilateurs sont implémentés dans *leur propre langage source*, selon un procédé appelé *bootstrapping*[1]. Ainsi, par exemple, le compilateur officiel du langage Scala, appelé `scalac`, est lui-même écrit en Scala.

1.1 Phases d'un compilateur

Le but d'un compilateur est de traduire un programme (sous la forme de texte) vers un autre programme (sous forme de texte ou de séquences d'octets), tout en préservant le sens du programme original.



Le processus de compilation se décompose généralement en plusieurs phases distinctes :

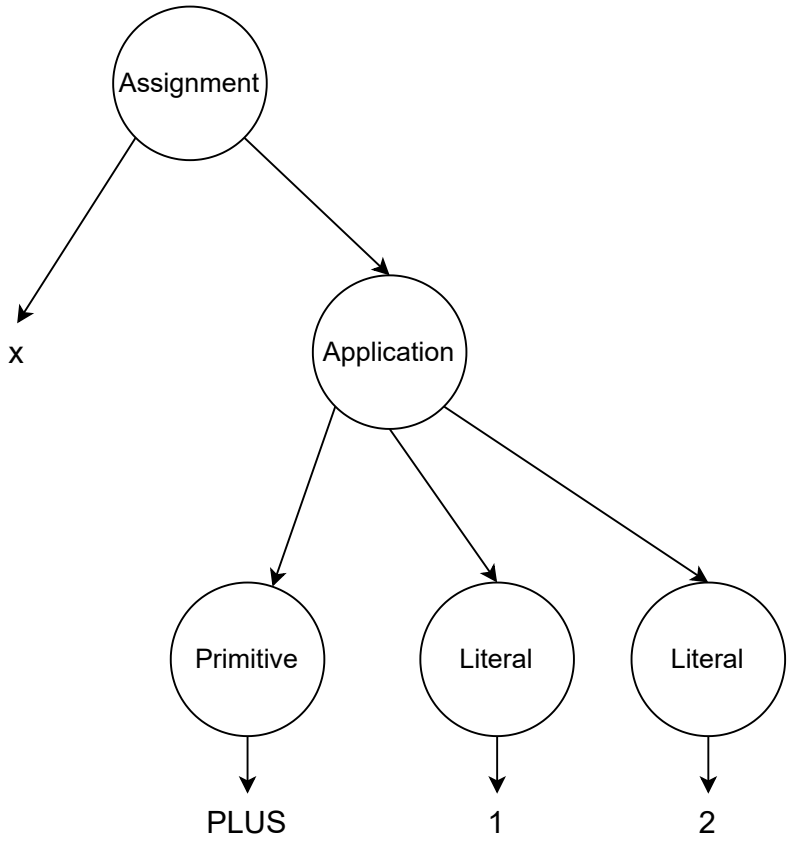
1. En premier lieu, lors de la phase d'analyse lexicale, le texte du programme d'entrée (une suite de caractères) est découpé en ce qu'on appelle des *tokens*. Les tokens représentent les différents mots, les différents composants lexicaux du programmes : mots-clés, identifiants, ponctuations, valeurs littérales, etc.
2. Dans un second temps, lors de la phase d'analyse syntaxique, la séquence de tokens produite par la phase précédente est analysée et une représentation structurée du programme est construite : l'*arbre de syntaxe abstraite* (*abstract syntax tree*, ou simplement AST). On appelle ce processus d'analyse syntaxique le *parsing*.
3. Ensuite, diverses phases d'analyse sont faites sur l'AST du programme. Dans les langages avec un système de typage statique, c'est lors de cette phase que l'on vérifie qu'il n'y a pas d'erreurs de types dans le programme.
4. Optionnellement, on poursuit ensuite par une phase d'optimisations, où l'on cherche à améliorer les performances du programme à générer (rapidité d'exécution, taille du code, etc.). Parmi les optimisations courantes, on peut citer l'élimination de code mort (*dead code elimination*, la suppression de code qui n'est pas atteignable dans un programme) ou encore la recherche de sous-expressions communes (*common subexpression elimination*, une technique permettant d'éviter de calculer à de multiples reprises la même expression).
5. Finalement, on termine le processus de compilation par la génération de code (*code generation*). C'est lors de cette phase que l'on retranscrit le code dans le langage cible.

Les phases ci-dessus présente une vision idéale, volontairement simple, d'un processus qui est parfois plus compliqué en pratique. Dans le cadre de ces derniers cours et du projet associé, nous allons essayer d'adhérer à cette organisation en phases.

1. Voir `https://fr.wikipedia.org/wiki/Bootstrap_(compilateur)` à ce sujet si vous êtes intéressé !

1.2 Arbre de syntaxe abstraite

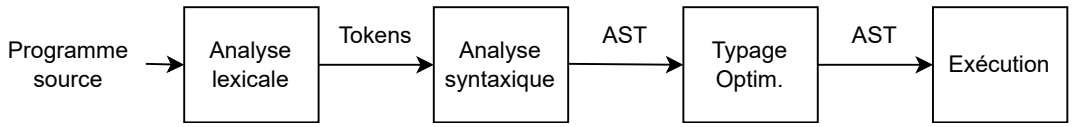
Comme vu plus haut, il est une structure de données qui joue un rôle central dans un compilateur : l'arbre de syntaxe abstraite, ou AST (pour *abstract syntax tree*). Un arbre de syntaxe abstraite est une représentation structurée d'un programme sous la forme d'un arbre. Par exemple, ci-dessous est représenté l'AST d'une expression qui correspondrait à `x = 1 + 1` selon la syntaxe Python. En rond dans le diagramme sont indiqués les noeuds qui correspondent à une expression du langage. Les flèches indiquent les parties constituantes de l'expression. Notez que certaines expressions sont formées à partir d'autres expressions, formant ainsi une structure en arbre.



Parfois, les compilateurs feront usage de plusieurs ASTs différents, utilisés successivement au cours des différentes phases du compileur. Les ASTs en début de processus seront plus proches de la syntaxe du langage source, tandis que les ASTs en fin de processus sont généralement plus proches dans l'esprit du langage cible. Dans le cadre de notre projet, nous allons nous restreindre à l'utilisation d'un seul type d'AST.

1.3 Interpréteur

Les compilateurs ne sont pas les seuls outils capables de donner vie à un langage de programmation. Une autre technique est celle de l'*interprétation*. Un interpréteur est un logiciel qui permet d'exécuter directement des programmes écrits dans un certain langage source. Lorsqu'un interpréteur a un langage source de haut niveau, le processus d'interprétation suit dans les grandes lignes les mêmes phases que pour la compilation, à l'exception de la dernière phase.



Dans un premier temps, le texte du programme source est transformé en AST par la succession des phases d'analyse lexicale et d'analyse syntaxique. Ensuite, l'AST est analysé (par ex. pour les erreurs de types) et éventuellement optimisé. Ensuite, à la place de générer du code, l'AST est directement interprété par le logiciel. C'est-à-dire que l'interpréteur simule l'exécution du programme qu'il traite.

À noter que les machines virtuelles, que nous évoquions plus tôt, sont un type d'interpréteur. Elles adoptent cependant généralement un langage source de bas niveau.

2 Programmation Orientée Objets (POO)

Concevoir un compilateur, ou même un interpréteur, n'est pas une tâche simple. Un compilateur doit manipuler des programmes, qui sont des données avec une structure non triviale, et leur appliquer diverses opérations, elles aussi non-triviales. Pour nous aider dans cette tâche, nous allons faire recours à une technique de programmation appelée *Programmation Orientée Objets*, ou POO pour faire court.

2.1 Objets

La notion centrale en POO est celle d'*objet*. Un objet regroupe et encapsule des données. De manière simple, un objet peut être vu comme un moyen de rassembler des données et de les traiter comme une unique entité. De plus, un objet offre généralement des *méthodes* afin d'accéder à ces données de manière plus ou moins directe, et éventuellement des moyens de modifier ces données, là aussi de manière plus ou moins directe. L'objet est un outil de conception très intéressant car il permet de regrouper des données et de contrôler la manière dont on peut interagir avec ces données.

Vous avez tout au long de ce cours utilisé des objets à de nombreuses reprises. Par exemple, quand nous avons fait de la manipulation d'images avec la librairie Pillow, les images sur lesquelles nous travaillions étaient représentées par des objets. Ces objets contenaient les données de l'image et offraient des méthodes (telles de `putpixel`, `getpixel`) afin d'accéder à ces données.

De manière générale, beaucoup de valeurs en Python sont représentées par des objets. Ainsi, les listes sont des objets, tout comme les dictionnaires, pour ne citer que quelques exemples.

2.2 Classes et méthodes

Comme nous l'avons vu, beaucoup de valeurs en Python sont représentées par des objets. Nous allons maintenant nous intéresser à la façon de concevoir nos propres types d'objets. En Python, et dans la plupart des langages dits *orientés objets*, la conception de différents types d'objets se fait par le biais de *classes*. Une classe est une sorte de patron de conception d'objets, une description de l'objet et de ses méthodes. Par exemple, observez la définition suivante en Python :

```
class Chat:
    def __init__(self, nom_naissance):
        self.nom = nom_naissance

    def caresser(self):
        print(self.nom, "ronronne de plaisir !")

    def renommer(self, nouveau_nom):
        self.nom = nouveau_nom
```

La classe ci-dessus définit comment créer un objet de type `Chat` et comment interagir avec cet objet. La classe contient ici trois définitions de méthodes. Ces définitions de méthodes suivent les mêmes règles syntaxiques que les définitions de fonctions. Chaque méthode doit avoir pour premier paramètre `self` : un paramètre qui fait référence à l'objet sur lequel la méthode est appelée. Examinons de plus proche ces méthodes !

- La méthode `__init__` est appelée le *constructeur* de la classe. Cette méthode est appelée automatiquement à la création de l'objet et permet de paramétrer sa création. Ainsi, dans le cas présent, lorsque l'on crée un chat, on devra lui fournir un nom. Le nom est stocké dans l'objet à l'aide de l'instruction `self.nom = nom_naissance`. L'instruction assigne au *champ* `nom` de l'objet la valeur donnée, ici celle du paramètre `nom_naissance`.
- La méthode `caresser` ici a pour effet d'afficher un message à la console. Au début de ce message est affiché la valeur stockée dans le champ `nom` de l'objet.
- Finalement, la méthode `renommer` offre la possibilité de donner un nouveau nom, une nouvelle valeur au champ `nom`. Notez à nouveau l'usage de `self.nom` afin de faire référence au champ et, dans ce cas précis, d'en modifier la valeur.

2.3 Création et utilisation d'objets

La classe indique l'interface de l'objet, la façon dont on crée et peut interagir avec l'objet. Étant donné la définition de la classe `Chat` plus haut, voici comment créer un objet de type `Chat` et comment accéder à ses méthodes.

```
minet = Chat("Sylvestre")

minet.caresser()  # Affiche "Sylvestre ronronne de plaisir !"

minet.renommer("Sissi")

minet.caresser()  # Affiche "Sissi ronronne de plaisir !"
```

La première instruction plus haut permet de créer un objet de type `Chat` et de le stocker dans une variable appelée `minet`. Notez que l'on utilise le nom de la classe comme une fonction, et que l'on fournit ici un unique argument (ici `"Sylvestre"`). Les arguments doivent correspondre aux paramètres du constructeur de la classe, en faisant abstraction du paramètre `self` qui est automatiquement renseigné par Python.

Les diverses méthodes de l'objet sont appelées en utilisant la syntaxe :

```
objet.methode(arguments)
```

Notez que l'on ne donne pas d'argument pour le paramètre `self` entre les parenthèses. La valeur à gauche du point est utilisée comme valeur pour `self`.

De manière intéressante, plusieurs objets peuvent être créés en utilisant la même classe, comme dans l'exemple plus bas. Chaque objet a ses propres données.

```
minet_1 = Chat("Sylvestre")
minet_2 = Chat("Moquette")

minet_1.caresser()  # Affiche "Sylvestre ronronne de plaisir !"
minet_2.caresser()  # Affiche "Moquette ronronne de plaisir !"

minet_1.renommer("Sissi")

minet_1.caresser()  # Affiche "Sissi ronronne de plaisir !"
minet_2.caresser()  # Affiche "Moquette ronronne de plaisir !"
```

2.4 Héritage

En Python, et dans de nombreux langages orientés objets, il est possible de définir des classes qui spécialisent une autre classe (appelée la classe *parente*) selon une technique appelée l'*héritage*.

```
class Animal:
    def dormir(self):
        print("ZzZzz")

class Chat(Animal):
    def __init__(self, nom):
        self.nom = nom

    def caresser(self):
        print(self.nom, "ronronne de plaisir !")

    def renommer(self, nom):
        self.nom = nom

class Chien(Animal):
    def __init__(self, nom, bruyant):
        self.nom = nom
        self.bruyant = bruyant

    def dormir(self):
        if self.bruyant:
            print("ZZZZZZZZzzzZZZZZZ")
        else:
            super().dormir()
```

```
minet = Chat("Sylvestre")
minet.dormir()  # Affiche "ZzZzz"

chien_1 = Chien("Lola", False)
chien_1.dormir()  # Affiche "ZzZzz"
chien_2 = Chien("Sam", True)
chien_2.dormir()  # Affiche "ZZZZZZZZzzzZZZZZZ"
```

Dans l'exemple ci-dessus, on montre comment une classe, `Animal`, peut se spécialiser, ici en deux classes distinctes, `Chien` et `Chat`. Notez le nom de la classe parente entre parenthèse après le nom de la classe. La classe `Animal` fournit une unique méthode `dormir`. La classe `Chat` ne définit pas de méthode de même nom, la méthode `dormir` reste accessible tout comme si `Chat` l'avait définie. Si la méthode est redéfinie par la classe enfant, comme c'est le cas pour `Chien`, c'est la nouvelle méthode qui prime. Cependant, il est toujours possible de faire référence à la méthode de la classe parente via l'utilisation de la fonction `super`, comme exemplifié dans la méthode `dormir` de la classe `Chien`.

L'héritage est une notion relativement avancée de la programmation orientée objets et nous n'avons ici qu'effleuré les possibilités qui sont offertes par ce mécanisme. Ce bref aperçu devrait cependant nous être suffisant pour nos besoins dans l'immédiat !