Notes de cours
Semaine 28

Cours Turing

1 Génération de code

La dernière phase d'un compilateur est la génération de code. Cette phase consiste à traduire l'AST, qui est une représentation abstraite du programme, en code concret exprimé dans le langage cible. Suivant le langage cible, la génération de code peut être plus ou moins complexe.

Plus les abstractions du langage cible sont éloignées de celles du langage source, plus la génération de code est compliquée. Pour les langages cibles de bas niveau, la génération de code à partir d'un langage de haut niveau est relativement complexe car il faut traduire les abstractions du langage source en instructions simples du langage cible. Par exemple, pour générer du code assembleur à partir d'un langage de haut niveau comme C++, il faut prendre en compte la gestion de la mémoire, les registres, les appels de fonctions, les objets, etc.

Dans le cas de notre projet, le langage cible est Python, qui est très proche de notre langage source en termes d'abstractions. La génération de code est donc relativement simple. Il suffit de parcourir l'AST du programme et de générer du code Python en fonction des nœuds rencontrés. Il faudra au passage procéder à quelques transformations pour traiter les différences entre les deux langages. Par exemple, dans notre langage, l'assignation est une expression qui a une valeur, alors que ce n'est pas le cas en Python.

Pour traiter de manière spécifique chaque type de noeud de l'AST, nous avons vu jusqu'à présent une unique technique basée sur la programmation orientée objet : définir une méthode dans la classe parente et en offrir une implémentation différente pour chaque type de noeud. C'est ainsi que nous avions procédé pour l'évaluation de l'AST ou encore pour la vérification des types. Cette technique est connue sous le nom de *polymorphisme* dans le jargon de la programmation orientée objet.

Aujourd'hui, nous allons voir une autre technique pour réaliser le même genre de tâches : le *pattern matching*.

Remarque

Le *pattern matching* est une fonctionnalité empruntée aux langages fonctionnels qui est apparue en Python à partir de la version 3.10. Pour l'utiliser, il faut donc s'assurer que la version de Python utilisée est bien 3.10 ou une version supérieure.

2 Le *pattern matching* en Python

Le pattern matching est une technique qui permet de définir des règles pour exécuter du code en fonction de la forme (du *pattern*) d'une valeur. En Python, il est introduit à l'aide du mot clé `match`.

2.1 Instruction `match`

En Python, une instruction `match` permet de définir une série de cas, chacun introduit par le mot-clé `case`. Chaque `case` définit un *pattern* et un bloc de code correspondant. Voici la syntaxe générale de l'instruction `match` :

```
match value:
    case pattern1:
        # code à exécuter si la valeur correspond à pattern1
    case pattern2:
        # code à exécuter si la valeur correspond à pattern2
        # mais pas aux patterns précédents
    ...
    case _:
        # code à exécuter si la valeur ne correspond
        # à aucun des patterns précédents
```

Lorsque Python exécute une instruction `match`, il va essayer de faire correspondre la valeur donnée à chacun des *patterns* dans l'ordre. Le premier *pattern* qui correspond à la valeur est sélectionné et le code associé est exécuté. Si aucun *pattern* ne correspond à la valeur, le *pattern* `_` est utilisé comme cas par défaut. Notez qu'un unique `case` est sélectionné, même si plusieurs *patterns* correspondent à la valeur. Cela signifie que l'ordre des *patterns* est important.

2.2 Les *patterns*

Au coeur du *pattern matching* se trouve le concept de *pattern*. Un *pattern* est une sorte d'expression qui permet de décrire une forme que doit prendre une valeur. Il existe différents types de *patterns* en Python. Nous allons passer en revue les principaux dans la suite de ce document.

Les *patterns* littéraux

Un *pattern* littéral est un *pattern* qui correspond à une valeur littérale. Par exemple, un *pattern* littéral peut correspondre à un entier, une chaîne de caractères, un booléen, etc. Dans l'exemple suivant, on utilise différents *patterns* littéraux pour vérifier à quelle valeur correspond la variable `value`.

```
match value:
    case 0:
        print("La valeur est 0")
    case "hello":
```

```
        print("La valeur est 'hello'")
    case [1, 2, 3]:
        print("La valeur est [1, 2, 3]")
    case _:
        print("La valeur correspond à autre chose")
```

Un *pattern* littéral n'accepte qu'une seule valeur. Si la valeur donnée ne correspond pas à la valeur littérale du *pattern*, le *pattern* est falsifié et le programme passe au cas suivant.

Les *patterns* de variable

Un autre type de *pattern* est le *pattern* de variable. Un *pattern* de variable est un *pattern* qui permet de **capturer** une valeur et de l'associer à une variable.

```
match value:
    case x:
        print(f"La valeur est {x}")
```

Dans cet exemple, le *pattern* `x` est un *pattern* de variable qui va capturer la valeur de `value` et l'associer à la variable `x`. À noter qu'un *pattern* de variable ne peut jamais être falsifié. Cela signifie que si un *pattern* de variable est utilisé, il correspondra toujours à la valeur donnée.

Souvent, les patterns de variables sont utilisés pour capturer des parties d'une valeur plus complexe. Par exemple, pour capturer le premier élément d'une liste, il est possible d'écrire :

```
match value:
    case [x, *rest]:
        print(f"Le premier élément est {x}")
    case _:
        print("La valeur n'est pas une liste")
        print("ou la liste est vide")
```

Dans l'exemple ci-dessous, le *pattern* `[x, *rest]` correspond à une liste dont le premier élément est capturé par la variable `x` et le reste de la liste est capturé par la variable `rest`. Le pattern accepte donc toute liste non vide.

Parfois, on ne sera pas intéressé à capturer la valeur d'une variable. Dans ce cas, on peut utiliser le *pattern* `_` qui correspond à n'importe quelle valeur mais sans la capturer. On parle du *wildcard pattern*. Nous avons fait usage de ce pattern dans les exemples précédents pour définir un cas par défaut.

Les *patterns* d'objets

Il est également possible de définir des *patterns* qui décrivent la forme d'un objet à partir de la classe de cet objet. Par exemple, pour vérifier si une valeur est une instance de la classe `Rectangle` ou de la classe `Circle`, on peut écrire :

```
match shape:
    case Rectangle():
        print("La valeur est un Rectangle")
    case Circle():
```

```
        print("La valeur est un Circle")
    case _:
        print("La valeur n'est ni un Rectangle ni un Circle")
```

De manière intéressante, il est possible de spécifier des conditions supplémentaires pour un *pattern* d'objet, en utilisant des sous-patterns pour les attributs de l'objet. Par exemple, pour vérifier si un rectangle a une largeur et une hauteur de 10, il est possible d'écrire :

```
match shape:
    case Rectangle(width=10, height=10):
        print("C'est un carré de côté 10")
    case _:
        print("Ce n'est pas un carré de côté 10")
```

Les *patterns* imbriqués

Comme nous l'avons vu à l'instant, il est possible d'imbriquer des *patterns* pour décrire des valeurs plus complexes. Cela peut être très utile pour décrire des structures de données imbriquées. Par exemple, dans un compilateur, il est possible d'utiliser le *pattern* suivant pour détecter si un noeud correspond à une addition de deux entiers littéraux :

```
match expr:
    case Application(fun=Primitive(name="PLUS"),
                     args=[Literal(value=x), Literal(value=y)]):
        print(f"L'expression est {x} + {y}")
    case _:
        print("Il s'agit d'une autre expression")
```

Les *patterns* alternatifs

Il est possible de combiner plusieurs *patterns* en utilisant l'opérateur `|` pour définir des *patterns* alternatifs. Par exemple, pour vérifier si une valeur est soit "q", soit "quit", soit "exit", on peut écrire :

```
match command:
    case "q" | "quit" | "exit":
        print("Fin du programme")
    case _:
        print("Commande inconnue")
```

2.3 Ajout de conditions

Dans une construction de pattern matching, il est aussi possible d'ajouter une condition en supplément du *pattern* à chaque `case` en utilisant l'opérateur `if`. Par exemple, pour vérifier si un entier est pair, on peut écrire :

```
match value:
    case x if x % 2 == 0:
        print(f"{x} est pair")
    case x:
        print(f"{x} n'est pas pair")
```

2.4 *Pattern matching* versus *polymorphisme*

Comme nous l'avons mentionné auparavant, le pattern matching permet de réaliser une tâche similaire à celle que nous avions réalisée avec la programmation orientée objet via ce qu'on appelle le *polymorphisme*, à savoir effectuer une action spécifique en fonction du type d'un objet.

En règle générale, on préférera utiliser le polymorphisme lorsque les actions à réaliser sont spécifiques à chaque sous-classe et y correspondent naturellement. Au contraire, on préférera utiliser le pattern matching lorsque les actions à réaliser dépendent de la forme plus profonde de la valeur et non uniquement de sa classe. En pratique, les deux méthodes sont complémentaires et peuvent être utilisées conjointement.

Les deux techniques ont aussi des avantages et des inconvénients en lien avec l'*extensibilité du code*, c'est-à-dire la facilité à modifier le code pour y ajouter de nouveaux comportements. On distingue deux cas de figure :

1. l'ajout d'une nouvelle fonction,
2. l'ajout d'un nouveau type de données.

Dans le cas où l'on souhaite ajouter une nouvelle fonction, le pattern matching est souvent plus simple car la fonction est définie en un seul endroit. En revanche, avec le polymorphisme, il faut ajouter une nouvelle méthode dans chaque sous-classe. Le changement est donc plus localisé avec le pattern matching.

Au contraire, dans le cas où l'on souhaite ajouter un nouveau type de données, le polymorphisme est souvent plus simple car il suffit d'ajouter une nouvelle sous-classe. Avec le *pattern matching*, il faut ajouter un nouveau `case` pour chaque nouvelle sous-classe, et ce dans chaque `match` où la nouvelle sous-classe peut apparaître, ce qui peut être fastidieux.