

## Les types rékursifs

1. On considère la définition du type `form` des formules propositionnelles suivant :

```
type form = Bool | Not | And | Or

@dataclass
class Bool: value: bool

@dataclass
class Not: operand: form

@dataclass
class And: left: form; right: form

@dataclass
class Or: left: form; right: form
```

2. Ecrire une fonction `string_of_form(f: form) -> str` qui affiche une formule donnée dans la syntaxe de Python.
3. Ecrire une fonction `eval_form(f: form) -> bool` qui calcule la valeur de vérité d'une formule donnée.
4. Que fait la fonction `mystery` définie ainsi ?

```
def mystery(f1: form, f2: form) -> form:
    return Not(And(Not(f1), Not(f2)))
```

Définir l'implication `impl` de la même manière.

5. On souhaite simplifier une formule en supprimant les occurrences de `Not(Not(...))`. Ecrire la fonction `simpl(f: form) -> form` qui implante cette simplification.
6. De manière à prendre en compte des variables propositionnelles, on complète la définition précédente de `form` ainsi :

```
type form = Prop | ...

@dataclass
class Prop: id: str
```

Modifier la définition de `eval` pour pendre aussi en argument un paramètre fonctionnel qui associe une valeur de vérité à chaque formule atomique. Son prototype est maintenant le suivant :

```
eval_form(f: form, v: Callable[[str], bool]) -> bool
```

7. Définir une variante de `eval` qui prend en paramètre un dictionnaire pour coder les valeurs de vérité des formules atomiques. Son prototype est alors le suivant :

```
eval_form2(f: form, m: dict[str, bool]) -> bool
```

8. On définit le type `expr` des expressions arithmétiques suivant :

```
type expr = Int | Minus | Plus | Times

@dataclass
class Int: value: int

@dataclass
class Minus: operand: expr

@dataclass
class Plus: left: expr; right: expr

@dataclass
class Times: left: expr; right: expr
```

9. Ecrire une fonction `string_of_expr(e: expr) -> str` qui affiche une expression donnée dans la syntaxe de Python.
10. Ecrire une fonction `eval_expr(e: expr) -> int` qui évalue une expression donnée.
11. De manière à prendre en compte des variables entières dans les expressions arithmétiques, on complète la définition précédente de `expr` avec le constructeur :

```
type expr = Var | ...

@dataclass
class Var: id: str
```

Modifier la définition de `eval_expr` pour prendre aussi en argument un paramètre `m` de type `dict[str, int]`. Son prototype est maintenant le suivant :

```
eval_expr2(f: expr, m: dict[str, int]) -> int
```

12. Compléter la fonction `string_of_expr : expr -> string` qui affiche une expression donnée dans la syntaxe de Python.
13. Ecrire une fonction `translate(f: form) -> expr` qui traduit une formule en expression, en utilisant comme toujours les constantes 1 et 0 pour représenter `True` et `False`.
14. En déduire une nouvelle fonction d'évaluation des formules propositionnelles utilisant l'évaluation des expressions.