PYOROALB

Geste de savoir

Petites fonctionnalités que j'aimerais voir dans plus de langages

mardi 15 octobre 2024

Table des matières

Introduction

> *i* Le billet que vous allez lire est la traduction française de ce billet rédigé par Hillelwayne le 5 janvier 2023.

On peut regrouper les fonctionnalités des langages de programmation en 3 catégories :

1. Les fonctionnalités autour desquelles le langage est conçu, celles qui ne peuvent pas être ajoutées après coup. On pense à l'évaluation paresseuse (*lazyness*) chez Haskell ou le *borrow checker* de Rust.
2. Les fonctionnalités qui influencent fortement l'utilisation du langage. Il est possible de les ajouter plus tard, cependant cela demande une phase de conception, d'ingénierie et de planification conséquente. Ici on pense par exemple au *pattern matching* ou encore les types algébriques de données.
3. Les fonctionnalités de confort, qui ne sont pas trop difficiles à ajouter après coup, et dont l'absence ne change pas fondamentalement le langage. Souvent ce sont les sucres syntaxiques, comme en Python avec les chaînes d'évaluations (`if 0 <= a < 100`).

La plupart des *PLT* et des travaux sur les design des langages se concentrent sur les deux premiers points car ce sont les plus importants, cependant j'ai un penchant pour les fonctionnalités de la dernière catégorie. *Parce que* ces dernières sont des fonctionnalités qui impliquent des ajouts mineurs, elles sont donc susceptibles d'être souvent présentes dans de multiples langages.
Comme je passe beaucoup de temps à explorer des langages de niche peu connus, j'ai découvert

de nombreuses fonctionnalités très sympathiques que peu de personnes connaissent. En voici quelques unes !

1. Représentation des nombres

Il y a plusieurs choses que nous pouvons faire pour travailler plus facilement avec les nombres. Dans un premier temps, nous pouvons ajouter un séparateur (pour les grands nombres) comme le font déjà de nombreux langages. Par exemple, à la place d'écrire `10000500` on préfèrera `10_000_500` ou `1_00_00_500` si vous êtes Indien. On peut aussi imaginer écrire `1e3` à la place de `1000`.

Pour d'autres exemples de facilité d'utilisation des nombres, on peut regarder du côté de J . Il arrive en science et en maths d'avoir des équations avec des exposants et des racines de . J a une manière standard d'exprimer cela avec le format `{x}p{y}` pour `xπ^y`. Par exemple, on écrira `5*sqrt(π)` comme `5p0.5`. Il y a aussi à disposition `x` pour les puissances de `e` et `r` pour les nombres rationnels exacts (`(2r3 + 1) = 5r3` le vinculum) est remplacé par `r`).

2. Chaines de caractères litérale

En Lua pour écrire des chaines de caractère brut sur plusieurs lignes on utilise les délimiteurs `[[` et `]]` :

```
[[
  Alice said "Bob said 'hi'".
]]
```

La plupart des langages proposent la fonctionnalité d'écrire des chaînes de caractères sur plusieurs lignes. Ce qui fait qu'en Lua l'usage est plus agréable est le fait que les délimiteurs de début (`[[`) et de fin (`]]`) soient différents. Cela résout le problème des quottes non échappables dans les chaînes de caractères et il n'y a donc pas besoin d'échapper tous les caractères spéciaux `\s`. NeoVim utilise la chaîne de caractère `[[\\]]` pour littéralement dire `\\`. Avec l'échappement des caractère cela serait quelque chose comme `"\\\\"`.

3. Syntaxe de mise à jour généralisée

Je suis tout de suite tombé amoureux de cette fonctionnalité de Noulith :

> Avez-vous déjà eu envie d'écrire `x max= y` lors de la récupération de la valeur maximale dans une boucle ? Et bien c'est possible ici. On peut le faire avec absolument toutes les fonctions.

En d'autres termes, écrire `x max= y` équivaut à `x = max(x, y)`. Il est donc logique de pouvoir écrire des expressions de la sorte : `text sub= (regex, replacement)`.

4. L'heure de gloire de Chapel

Chapel est un langage pour les hautes performances quand vous avez besoin d'exécuter du code très rapidement sur des centaines voir des milliers de CPU. Il possède beaucoup de fonctionnalités des groupes 1 et 2 que je n'avais jamais vues autre part, et je trouve le langage très intéressant. Bien que je ne le connaisse pas, il y a quelques fonctionnalités sur la qualité d'usage que j'ai trouvé dans la documentation que je souhaiterais retrouver dans tous les autres langages.

Premièrement, il y a le mot clé `config`. Si on écrit :

```
config var n = 1
```

le compilateur ajoutera pour nous un argument `--n` au binaire produit. Comme je suis quelqu'un qui 1. aime avoir des programmes configurables, et 2. déteste gérer les librairies CLI, une manière sale-et-rapide d'ajouter un argument au binaire me semble être un avantage évident.

Deuxièmement, on peut écrire les séquences `1, 2, ..., n-1` comme `1..<n`. C'est une extension élégante de l'opérateur `..` comme il y a en Ruby ou en TLA+.

Troisièmement (et sûrement la plus contestable) la "promotion", une sorte d'élévation automatique du type. Si nous avons une fonction du type `a -> a`, Chapel nous laisse l'appeler avec un tableau retournant le tableau *mappé*.
Imaginons `f(x) = x*2`, alors `f([1, 2, 3]) = [2, 4, 6]` (le tout était fortement typé). On peut avoir un résultat similaire en utilisant les fonctions ou méthodes `lift` et `map` dans d'autres langages, mais la promotion est une fonctionnalité native intéressante à avoir.

(Chapel est encore plus fort dans ce cas : il parallélise automatiquement les calculs.)

C'était seulement quelques fonctionnalités cool présentes en Chapel, et nous n'avons pas encore évoqué le coeur de la chose. Je recommande chaudement à toutes les personnes intéressées en théorie de langages ou en calculs scientifiques de l'essayer.

5. Date litérale

Frink possède une syntaxe spéciale pour les dates. On peut écrire `# 2001-08-12 #` qui équivaut à la *date* du 12 août 2001, à la place d'écrire quelque chose comme `Date(2001, 8, 12)` et avoir des bugs parce que les mois (contrairement aux jours) sont indexés à partir de 0 .

6. Blocks de paramètres

Les fonctions PowerShell peuvent avoir un bloc de `params` dans lequel on ajoute des attributs aux paramètres de la fonction comme, les valeurs par défaut, de la documentation, de la validation, … C'est à dire qu'à la place d'écrire `fun f(str path="/", int x)` on écrira quelque chose comme :

```
fun f {
  params (
    [default = "/"]
    [help = "The path to your file"]
    [mandatory]
    [str]path
  , [optional][int] x
  )
}
```

Cela fait très sens pour un langage utilisé pour un shell car nous pouvons avoir beaucoup d'options, et d'alias. Cependant cela peut aussi être très utilisé pour les langages de programmation compilés.

7. kebab-case

Comme dans la plupart des dialectes Lisp. À l'inverse de nommer les éléments `two_things` ou `TwoThings` on les nomes `two-things`. C'est plus facile à écrire et à lire. Bien-sûr, la raison principale pour laquelle la majorité des autres langages ne font pas ça est car l'opérateur `-` est infixé. Cela rend ambiguë `x-y`, à savoir si c'est l'expression de `x` moins `y` ou l'invocation de la fonction qui a pour nom `x-y`.
Cela ne semble pas être le compromis adéquat. À quelle fréquence utilisez-vous `-`, et à quelle fréquence écrivez-vous les fonctions contenant plusieurs mots ? On pourrait définir que `x-y` est toujours le nom d'une fonction, et, si on utilise l'opérateur moins mettre des espaces autour.

(Ce n'est pas quelque chose qui peut-être ajouté à un langage sans devoir faire de gros changements à ce dernier, mais c'est peut-être quelque chose à penser si on en crées un nouveau ?)

8. Symbols

Ruby possède un type spécial appelé Symbol qui s'écrit de cette forme `:symbol`. Un *symbol* est égal à lui-même quand on le compare et n'a pas d'autre utilité, c'est un identifieur. Il permet de remplacer les chaînes de caractères d'un mot. Par exemple, à la place d'écrire `dict["employee_id"]` on écrira `dict[:employee_id]`.

L'avantage des *symbol* est qu'ils permettent de travailler plus facilement avec les chaînes de caractère. Dans la plupart des langages, les chaînes de caractères sont utilisées pour représenter de multiples éléments comme des *tokens*, du texte, des données structurées, … Si on voit `"book"`, nous ne sommes pas certains, sans contexte, si nous manipulons la clé d'un dictionnaire, un champ texte, une valeur d'un CSV ou encore autre chose. Avec les *symbol* on peut commencer pas exclure le premier cas cité car dans le cas contraire nous aurions eu `:book`.

9. Syntax dédié pour les tests

On peut observer cela au sein des blocs de test dans les fonctions en D ou avec les moniteurs en P . Bien qu'il fasse sens d'avoir les tests en tant que librairie, les tests sont universels sur les gros projets, qu'il est préférable d'avoir un support syntaxique pour ces derniers.

Conclusion

Dans tous les cas je ne pense pas qu'ajouter toutes ces fonctionnalités arbitrairement aux langages de programmation soit facile ou n'aura pas d'impact par la suite, mais elles semblent cependant arriver de manière perpendiculaire aux fonctionnalités du second groupe. C'est pourquoi les fonctionnalités du troisième groupe sont souvent ajoutées après coup (ayant un impact positif certain).
Ces fonctionnalités se propagent donc plus facilement d'un langage à l'autre.
Sauf pour le cas du *kebak-case*. Nous vivons dans un monde déchu.

Liste des abréviations

PLT Programming Language Theory - Théorie des langages de programmation. 1