ENS Paris-Saclay 2022-2023

Projet de programmation avancée
Un interpréteur pour le langage Lua

1 Le langage mini-Lua

Dans ce projet, on propose d'implémenter un interpréteur pour un sous-ensemble du langage Lua. L'objectif final est que notre interpréteur supporte les *coroutines*, une fonctionnalité avancée utile notamment pour implémenter des mécanismes d'itération ou de concurrence coopérative.

Lua est un langage interprété sans typage statique. Ceci le rend proche de langages comme Python, Javascript ou Lisp. Lua est bâti à partir d'un petit nombre de principes de base qui le rendent assez élégant.

Un interpréteur est un programme prenant en entrée un fichier contenant un programme sous forme textuelle et l'"exécute" en donnant du sens aux différentes constructions du langage dans lequel le programme est écrit. Le langage Lua est fourni avec un interpréteur de référence implémenté en C. Nous allons implémenter un interpréteur mini-Lua en OCaml. Nous testerons alors que notre interpréteur et l'interpréteur de référence renvoient bien le même résultat sur des programmes mini-Lua !

Au cours de ce projet, pour en savoir plus sur les fonctionnalités de Lua, on se réfèrera à la ressource suivante qui est une bonne introduction progressive à Lua à l'aide d'exemples :

Programming in Lua http://www.lua.org/pil/contents.html

En cas de doute sur des points précis, on pourra également se référer au manuel de référence de Lua (https://www.lua.org/manual/5.4/). Attention cependant, on ne cherche qu'à implémenter un *sous-ensemble* de Lua donc toutes les fonctionnalités décrites dans ces ressources ne nous concernent pas forcément.

La grammaire du langage mini-Lua que l'on considère est donnée dans la Figure 1. Elle correspond au sous-ensemble de Lua comportant les fonctionnalités suivantes :

— variables globales et locales
— assignation simple (variable = valeur)
— valeurs de base: entiers, flottants, booléens, chaînes de caractères, `nil`
— opérations arithmétiques de base, comparaisons et opérateurs logiques `and` et `or`
— conditions `if .. then .. else .. end`
— boucles `while`
— fonctions à plusieurs paramètres (mais une seule valeur de retour)
— fonctions de première classe et fermetures (*closures*)
— tables indicées par des entiers ou chaînes de caractères

Par ailleurs, on suppose que les programmes ont accès à une fonction primitive `print` (fonction d'affichage) et à une table `coroutine` contenant les fonctions primitives `coroutine.create`, `coroutine.mini_resume` et `coroutine.yield` (voir §6).

$$
\begin{array}{lcl}
\langle \textit{fichier} \rangle & ::= & \langle \textit{block} \rangle \ \texttt{EOF} \\
\langle \textit{block} \rangle & ::= & \langle \textit{locals} \rangle ? \\
 & & \langle \textit{statement} \rangle \\
 & & \langle \textit{ret} \rangle ? \\
\langle \textit{locals} \rangle & ::= & \texttt{local} \ \langle \textit{name} \rangle^{+}_{\texttt{,}} \\
\langle \textit{ret} \rangle & ::= & \texttt{return} \ \langle \textit{expr} \rangle \\
\langle \textit{statement} \rangle & ::= & \varepsilon \mid \langle \textit{statement} \rangle \ \langle \textit{statement} \rangle \\
 & \mid & \langle \textit{var} \rangle \ \texttt{=} \ \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle \texttt{(} \ \langle \textit{expr} \rangle^{\star}_{\texttt{,}} \ \texttt{)} \\
 & \mid & \texttt{while} \ \langle \textit{expr} \rangle \ \texttt{do} \ \langle \textit{statement} \rangle \ \texttt{end} \\
 & \mid & \texttt{if} \ \langle \textit{expr} \rangle \ \texttt{then} \ \langle \textit{statement} \rangle \ \texttt{else} \ \langle \textit{statement} \rangle \ \texttt{end} \\
\langle \textit{expr} \rangle & ::= & \texttt{nil} \mid \texttt{false} \mid \texttt{true} \mid \langle \textit{int} \rangle \mid \langle \textit{float} \rangle \mid \langle \textit{string} \rangle \\
 & \mid & \langle \textit{var} \rangle \mid \langle \textit{expr} \rangle \texttt{(} \ \langle \textit{expr} \rangle^{\star}_{\texttt{,}} \ \texttt{)} \\
 & \mid & \texttt{function} \ \texttt{(} \ \langle \textit{name} \rangle^{\star}_{\texttt{,}} \ \texttt{)} \ \langle \textit{block} \rangle \\
 & \mid & \langle \textit{expr} \rangle \ \langle \textit{op} \rangle \ \langle \textit{expr} \rangle \mid \langle \textit{unop} \rangle \ \langle \textit{expr} \rangle \\
 & \mid & \texttt{\{} \ \langle \textit{tableelt} \rangle^{\star}_{\texttt{;}} \ \texttt{\}} \\
\langle \textit{tableelt} \rangle & ::= & \langle \textit{name} \rangle \ \texttt{=} \ \langle \textit{expr} \rangle \mid \texttt{[} \ \langle \textit{expr} \rangle \ \texttt{]} \ \texttt{=} \ \langle \textit{expr} \rangle \\
\langle \textit{binop} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{==} \mid \texttt{\~{}=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{and} \mid \texttt{or} \\
\langle \textit{unop} \rangle & ::= & \texttt{-} \mid \texttt{not} \\
\langle \textit{var} \rangle & ::= & \langle \textit{name} \rangle \mid \langle \textit{expr} \rangle \texttt{[} \langle \textit{expr} \rangle \texttt{]} \\
\langle \textit{name} \rangle & ::= & \textit{(chaîne alphanumérique)} \\
\langle \textit{int} \rangle & ::= & \textit{(nombre entier)} \\
\langle \textit{float} \rangle & ::= & \textit{(nombre flottant)} \\
\langle \textit{string} \rangle & ::= & \texttt{"} \ \textit{(chaîne de caractères)} \ \texttt{"}
\end{array}
$$

FIGURE 1 – Grammaire des fichiers mini-Lua

2 Structure du projet

On détaille ci-dessous l'organisation des différents fichiers du projet. Les fichiers qu'il vous faut lire et/ou compléter sont indiqués explicitement. Pour les autres, ils sont nécessaires au fonctionnement du projet mais vous n'avez pas besoin de les lire.

Général :

— `README.md` **(à lire)**: instructions pour compiler et exécuter le code du projet

Interpréteur OCaml (§4) :

— `ocaml/parser/ast.ml` **(à lire)**: AST de mini-Lua obtenu après analyse syntaxique (*parsing*);

— `ocaml/interp/value.{ml,mli}` **(à lire)**: définition des valeurs et des environnements de l'interpréteur mini-Lua;

— `ocaml/interp/interp.ml` **(à compléter)**: code de l'interpréteur, partant d'un AST mini-Lua et l'exécutant.

Interpréteur Rust (§5) :

— `rust/src/parser/ast.rs` **(à lire)**: AST de mini-Lua;

— `rust/src/interp/value.rs` **(à lire)**: définitions des valeurs mini-Lua;

— `rust/src/interp/env.rs` **(à lire)**: définitions des environnements mini-Lua;

— `rust/src/interp/mod.rs` **(à compléter)**: code de l'interpréteur.

Interpréteur OCaml avec support des coroutines (§6) :

— `ocaml/interp-cps/value.{ml,mli}` **(à lire)**: définitions des valeurs et environnement mini-Lua, avec coroutines;

— `ocaml/interp-cps/interp.ml` **(à compléter)**: code de l'interpréteur.

Programmes de test :

— `tests/*` **(à lire)**: programmes de test mini-Lua, *numérotés par complexité croissante.*

Voir le `README.md` pour savoir comment lancer l'interpréteur du projet ou l'interpréteur de référence Lua sur un fichier d'entrée, ou exécuter les tests.

Par ailleurs, il vous est demandé d'implémenter une ou plusieurs extensions à l'un de ces interpréteurs (§7), et d'écrire un compte-rendu de votre travail expliquant les difficultés rencontrées et les choix réalisés.

3 Comment démarrer ?

Ce projet est conçu pour être implémenté **incrémentalement**.

Le but est d'ajouter petit à petit le support pour chaque fonctionnalité de mini-Lua, en *compilant et testant dès que possible*, entre l'ajout de chaque fonctionnalité.

Pour savoir dans quel ordre implémenter les fonctionnalités de mini-Lua, on se laissera diriger par les fichiers de test : ceux-ci sont ordonnés par difficulté croissante.

L'approche conseillée est donc la suivante :

— Lancer les tests; identifier le premier test qui ne passe pas;

— Aller lire le fichier `.lua` correspondant au test

— Comprendre quel est le comportement attendu sur ce fichier; pour cela, on pourra : lancer l'interpréteur de référence sur le fichier (cf `README.md`), ou aller lire les ressources en ligne sur Lua.

— Inspecter la structure de l'arbre de syntaxe correspondant à ce fichier (cf `README.md`);
— Étendre l'interpréteur pour supporter la nouvelle fonctionnalité de façon à ce que le test passe.

NB : Ne pas hésiter à insérer des `failwith "todo"` dans le code de l'interpréteur pour les cas que l'on ne gère pas encore (ou `unimplemented!()` en Rust). Il est important d'avoir un programme qui compile et que l'on peut tester, quitte à laisser des trous à remplir plus tard.

NB : les Sections 5 et 6 sont indépendantes. Elles peuvent être implémentées dans l'ordre de votre choix.

4 Interpréteur OCaml pour mini-Lua sans coroutines

Quelques indications complémentaires concernant l'implémentation des fonctionnalités de base de mini-Lua, avant l'introduction des coroutines :

— Lorsque plusieurs arguments sont passés à la fonction `print`, ceux-ci sont affichés espacés par un caractère *tabulation*, qui s'écrit `\t` dans une chaîne OCaml.
— La gestion de l'environnement de l'interpréteur est importante et un peu subtile.
  — En plus des variables assignées globalement, les arguments d'une fonction et les déclarations `local` introduisent des variables *locales*;
  — L'environnement garde une liste de blocs de déclarations locales (les plus récentes en premier);
  — Les fonctions (de première classe) s'évaluent en des fermetures (*closures*); une fermeture a toujours accès aux variables qui étaient disponibles à sa création. Autrement dit, une fermeture capture son environnement et le réinstaure à chaque fois qu'elle est appelée.

5 Interpréteur Rust pour mini-Lua sans coroutines

En plus de l'utilisation d'un autre langage de programmation, de sa syntaxe, etc. . ., la principale difficulté de cette partie du projet consiste en une gestion différente de la mémoire utilisée par l'interpréteur, en adéquation avec le système de types de Rust.

Afin de comprendre la technique de gestion de la mémoire choisie, veuillez lire les commentaires des fichiers `rust/src/interp/value.rs` et `rust/src/interp/env.rs`.

6 Interpréteur OCaml avec support des coroutines

À ce stade, vous devez avoir un interpréteur de mini-Lua implémenté en OCaml passant tous les tests n'utilisant pas de coroutines. C'est aussi le bon moment pour faire un peu de nettoyage et de refactoring dans le code de votre interpréteur; essayez de faire en sorte qu'il soit le plus simple et uniforme possible.

6.1 Les coroutines c'est quoi ?

Une coroutine représente un *calcul pouvant s'interrompre au cours de son exécution*. Si l'on dispose de plusieurs calculs sous la forme de coroutines, on peut alors effectuer ces calculs de façon « entrelacée », en exécutant une première coroutine jusqu'à ce qu'elle s'interrompe, puis en en exécutant une seconde, puis en continuant le calcul de la première, *et cætera*. On dit que les coroutines fournissent une forme de « concurrence coopérative ».

À titre d'exemple, considérons la fonction Lua `myloop` définie ci-dessous:

```lua
function myloop ()
  local i; i = 1
  while i <= 10 do
    print(i)
    i = i + 1
  end
end
```

Il s'agit d'une fonction ordinaire affichant les entiers de 1 à 10 à l'aide d'une boucle. Lorsque l'on exécute `myloop()`, celle-ci s'exécute entièrement en affichant 1, puis 2, etc, jusqu'à 10.

On va maintenant transformer cette fonction en une coroutine exécutant cette fonction pas-à-pas, en s'arrêtant après chaque tour de boucle. La première étape est d'indiquer dans le code de la fonction le moment où l'on veut que le calcule s'interrompe. On le fait en appelant la fonction `coroutine.yield()` dans la boucle `while`.

```lua
function myloop ()
  local i; i = 1
  while i <= 10 do
    print(i)
    i = i + 1
    coroutine.yield()
  end
end
```

Enfin, on construit une coroutine à partir de la fonction `myloop` grâce à la fonction `coroutine.create`:

```lua
co = coroutine.create(myloop)
```

À ce stade, `co` contient une coroutine dite *suspendue* initialisée avec la fonction `myloop`; on n'a pas commencé à exécuter le code contenu dans `myloop`. Pour exécuter le code d'une coroutine *jusqu'au prochain appel à* `coroutine.yield()`, on appelle `coroutine.mini_resume`:

```lua
coroutine.mini_resume(co)  -- affiche 1
```

On a ici commencé l'exécution de `myloop()` (affichant l'entier 1 et incrémentant la variable locale `i`) jusqu'à recontrer un appel à `coroutine.yield()`. L'exécution s'arrête alors, mais la coroutine `co` *mémorise l'endroit du calcul où elle s'est arrêtée* ! On peut reprendre l'exécution de la coroutine (jusqu'à un nouvel appel à `coroutine.yield()`) en appelant à nouveau `coroutine.mini_resume`:

```lua
coroutine.mini_resume(co)  -- affiche 2
coroutine.mini_resume(co)  -- affiche 3
...
coroutine.mini_resume(co)  -- affiche 10
coroutine.mini_resume(co)  -- n'affiche rien et termine immédiatement:
                           -- la coroutine a terminé son calcul
```

Quelques remarques supplémentaires :

1. Il n'y a pas de restriction sur les endroits où peuvent apparaitre les appels à `coroutine.yield()`; tant que le code s'exécute bien à l'intérieur d'une coroutine créée par `coroutine.create`. Autrement dit, `coroutine.yield()` peut être appelé via des fonctions auxiliaires, au milieu de fonctions récursives, de boucles, etc; et une coroutine peut contenir autant d'appels à `coroutine.yield()` que l'on veut;

2. **On peut passer un argument supplémentaire à `coroutine.yield` et `coroutine.mini_resume`.** Ceci permet d'échanger des valeurs entre la coroutine et le code appelant la coroutine. Un argument passé à `coroutine.yield` devient la *valeur de retour* du `coroutine.mini_resume` ayant exécuté la continuation. Et de manière symmetrique, un argument passé à `coroutine.mini_resume` (en plus de la continuation) devient la *valeur de retour* du `coroutine.yield` d'où reprend l'exécution.
3. La fonction `coroutine.mini_resume` est spécifique au langage mini-Lua de ce projet. Le langage Lua fournit plutôt une fonction `coroutine.resume`. Contrairement à `coroutine.mini_resume` qui renvoie une seule valeur (la valeur passée en argument à `coroutine.yield`, ou `nil` s'il n'y en a pas), `coroutine.resume` renvoie à la fois un booléen et la (ou les) valeurs passées à `coroutine.yield`. Comme notre langage mini-Lua ne supporte pas les fonctions renvoyant plusieurs valeurs, on se contentera dans l'interpréteur d'implémenter la fonction `coroutine.mini_resume`.
4. On dispose d'une fonction supplémentaire `coroutine.status` permettant d'obtenir l'état d'une coroutine. Elle renvoie l'une des chaînes de caractères: `"dead"`, `"suspended"` ou `"running"`.

Pour compléter votre compréhension des coroutines, lire au moins les sections 9.1 et 9.2 du *Programming in Lua* : http://www.lua.org/pil/9.html#CoroutineSec.

6.2 Interpréteur en style "passage de continuations"

Pour que notre interpréteur mini-Lua supporte les coroutines, celui-ci doit être capable de représenter un calcul partiellement exécuté. Plus précisément, lorsque l'interpréteur rencontre un `coroutine.yield`, il doit alors stocker dans la coroutine correspondante « *ce qu'il reste à exécuter après* », code qui sera relancé par un appel futur à `coroutine.mini_resume`.

Ceci n'est pas facilement faisable avec l'implémentation actuelle de l'interpréteur. Avant de se consacrer à l'implémentation des coroutines, on va donc d'abord transformer le code de l'interpréteur en un interpréteur *équivalent* mais écrit dans le style « passage de continuations », dit CPS (*Continuation Passing Style* en anglais).

CPS. L'idée centrale de l'écriture de code en CPS est de paramétrer chaque fonction de notre programme par un *paramètre supplémentaire*, un fonction appelée **continuation** qui correspond au code à exécuter « *après que l'on ait fini notre calcul* ». Alors:

— Si notre calcul est fini et que l'on a une valeur à renvoyer, *on appelle la continuation en lui passant cette valeur* au lieu de renvoyer la valeur directement;

— Si on a besoin d'appeler une autre fonction elle-même écrite en CPS, on l'appelle directement *en lui donnant une continuation contenant le reste du calcul que l'on veut faire après.*

CPS, sur un exemple. À titre d'exemple, considérons la fonction `size` calculant récursivement le nombre de nœuds d'un arbre de type `tree` défini ci-dessous.

```
type tree = Node of tree * tree | Leaf of int

let rec size1 (t: tree): int =
  match t with
  | Node (t1, t2) -> size1 t1 + size1 t2
  | Leaf _ -> 0
```

Avant la transformation CPS proprement dite, une première étape pour se simplifier la vie est d'introduire des noms explicites pour les calculs intermédiaires correspondant à des appels récursifs, avec des `let ... in`:

```
let rec size2 (t: tree): int =
  match t with
  | Node (t1, t2) ->
    let n1 = size2 t1 in
    let n2 = size2 t2 in
    n1 + n2
  | Leaf _ -> 0
```

On réécrit alors la fonction en style « passage de continuation », en lui passant un argument supplémentaire `k` (la *continuation*). À chaque appel à `size_cps`, on lui passe en deuxième argument une fonction indiquant *la suite du calcul*. Cette fonction prend un argument : la valeur calculée par `size_cps`. Lorsque l'on a fini un calcul et que l'on veut « renvoyer » une valeur, *on ne la renvoie jamais directement*: à la place, on appelle la continuation avec cette valeur.

```
let rec size_cps (t: tree) (k: int -> 'a): 'a =
  match t with
  | Node (t1, t2) ->
    size_cps t1 (fun n1 ->
    size_cps t2 (fun n2 ->
    k (n1 + n2)))
  | Leaf _ -> k 0
```

Finalement, pour effectivement calculer la taille d'un arbre, on appelle `size_cps` avec une continuation qui renvoie directement la valeur qui lui est passée : la fonction identité. La fonction `size3` obtenue est *équivalente* aux fonctions `size1` et `size2`.

```
let size3 (t: tree): int = size_cps t (fun x -> x)
```

Qu'à t-on gagné à réécrire notre code en CPS ? Dans le cas de la fonction `size`, pas grand chose. Pour notre interpréteur, chaque fonction prend maintenant en argument une *fonction OCaml* (la continuation) qui représente explicitement la « suite du calcul ». On pourra alors l'utiliser pour représenter l'état d'une coroutine ! Un appel à `coroutine.yield` pourra être supporté par l'interpréteur en stockant la continuation courante dans la coroutine environnante.

À faire Réécrire le code de votre interpréteur mini-Lua en OCaml en style CPS. **Écrire la version CPS de l'interpréteur dans le dossier `interp-cps/` du projet**, afin de préserver une version fonctionnelle de l'interpréteur de la Section 4 dans le dossier `interp/`. La définition des valeurs (`Value.t`) et fonctions (`Value.func`) contient des nouveaux constructeurs liés aux coroutines; on ne les implémentera pas pour le moment.

Vérifier empiriquement que l'interpréteur en CPS est bien *équivalent* à l'interpréteur avant réécriture. En particulier, tous les tests qui passaient avant doivent toujours passer.

Attention : que faire si on utilisait une fonction d'ordre supérieur comme `List.iter` ou `List.map` dans laquelle on appelle une fonction que l'on veut transformer en CPS ? Peut-on toujours utiliser `List.iter`/`List.map` ?

Le code de votre interpréteur doit avoir la forme ci-dessous (aussi fournie dans le code de démarrage). En particulier, toutes les fonctions de l'interpréteur (`interp_*`) sont écrites en forme CPS. Seule la fonction `run` s'appliquant à un programme entier n'est pas en forme CPS et appelle l'interpréteur en lui passant une continuation triviale (qui ne fait rien).

```
let rec interp_block (env : env) (blk : block) (k : value -> unit) : unit =
  ...
and interp_stat (env : env) (stat : stat) (k : unit -> unit) : unit  =
  ...
and interp_exp (env : env) (e : exp) (k : value -> unit) : unit =
 ...

let run ast =
   ...
   interp_block env ast (fun _ -> ())
```

6.3 Interprétation des coroutines

Le type des valeurs mini-Lua de notre interpréteur (`Value.t`) a un nouveau constructeur correspondant aux coroutines. Les « valeurs coroutines » sont les valeurs créées par `coroutine.create`, par exemple `co` dans la Section 6.1.

Inspectons leur définition (dans `value.mli`):

```
type t =
  | ...
  | Coroutine of coroutine

and coroutine = {
  mutable stat : coroutine_status
}

and coroutine_status =
  | Dead
  | Suspended of (t -> unit)
  | Running of (t -> unit)
```

Une coroutine est représentée par un enregistrement `coroutine` stockant son *état actuel* `stat`, qui peut changer au cours de l'exécution (d'où le fait que le champ soit mutable). Une coroutine peut être dans trois états différents :

- `Dead` : la coroutine a terminé son calcul. Un appel à `coroutine.mini_resume` sur la coroutine renvoie une erreur.
- `Suspended k` : la coroutine est *suspendue* (n'est pas en train de s'exécuter). La continuation `k` correspond à la fonction à appeler pour continuer l'exécution de la coroutine lors d'un appel à `coroutine.mini_resume`.
- `Running k` : la coroutine est *en cours d'exécution*. La continuation `k` correspond à la continuation *de l'appel à `coroutine.mini_resume` qui a déclenché son exécution.*

Quand l'interprète s'exécute, on peut considérer qu'il exécute du code appartenant à une coroutine donnée (si l'on est en dehors d'une coroutine, on considère que l'on est dans une coroutine « spéciale » d'état `Running` avec une continuation qui renvoie une erreur si on essaye de l'appeler). On peut donc paramétrer chaque fonction de l'interpréteur par la « coroutine courante ».

Lors d'un appel à `coroutine.create`, on crée une nouvelle continuation initialement dans l'état `Suspended`. Lors d'un appel à `coroutine.mini_resume` sur une coroutine `Suspended`, on passe cette coroutine dans l'état `Running` en y enregistrant la continuation actuelle, et on exécute la coroutine qui devient la « coroutine courante ». Finalement, lors d'un appel à `coroutine.yield`, on récupère la continuation stockée dans la coroutine courante, et on celle-ci de l'état `Running` à l'état `Suspended`. La fonction `coroutine.status` permet de consulter l'état d'une coroutine, elle renvoie la chaîne de caractères `"dead"`, `"suspended"` ou `"running"`.

À faire

- **Ajouter un paramètre** de type `continuation` à chaque fonction `interp_*` de l'interpréteur implémenté en Section 6.2 : la « coroutine courante ».
- Implémenter le support pour les fonctions primitives `CoroutCreate`, `CoroutResume`, `CoroutYield` et `CoroutStatus` (correspondant respectivement à `coroutine.create`, `coroutine.mini_resume`, `coroutine.yield` et `coroutine.status`).

Tester votre implémentation sur les tests dédiés aux coroutines (sans casser les tests concernant les autres fonctionnalités du langages !).

7 Extensions

Lors de la notation de votre projet, quelques points seront réservés à l'évaluation d'une ou plusieurs extensions aux interpréteurs que vous aurez programmés jusqu'ici.

Vous pouvez choisir une extension dans la liste suivante, ou en inventer une vous-même:

- Se documenter sur les *effect handlers* de OCaml 5 et les utiliser pour implémenter un interpréteur avec support des coroutines sans utiliser la CPS;
- Même chose en utilisant les threads en Rust;
- Utiliser une autre technique pour implémenter le support des coroutines en Rust (avec une pile explicite);
- Closure conversion pour un lookup plus rapide dans les environnements des closures;
- Utiliser les coroutines pour écrire des programmes non-triviaux ? (générateurs, scheduler, ...)
- Écrire des programmes qui font du calcul intensif (sur les entiers ou flottants) et améliorer la performance de l'interprète sur ces programmes;
- Une autre extension qui vous semble intéressante...