GNU Recursive Interpretation Processor

Gwenn Aubert Bastien Maureille

mai 2013

Résumé

GRIP (GNU Recursive Interpretation Processor) est un programme qui permet de définir et d'exécuter des fonctions récursives primitives. Ce document décrit les choix qui ont été faits pour le développement de ce programme ainsi que les différentes fonctionnalités qu'il implémente.

Table des matières

1 Langage utilisé

GRIP est écrit en LUA. Nous avons décidé de choisir ce langage de script pour sa légèreté mais surtout pour les fonctionnalités très intéressantes du langage offertes dans le cadre du développement d'un interpréteur en ligne de commande (comme les expressions régulières et le "pattern matching").

2 Structure du code et fonctionnement

2.1 Arborescence du projet

eval.lua Évalue la définition du prédicat saisi par l'utilisateur et retourne une fonction LUA correspondante si elle est valide.

functions.lua Déclaration des fonctions récursives primitives de base

GRIP Script shell (Bash) de lancement du programme

grip.lua Fichier principal d'exécution : contient la boucle principale

LICENSE.txt Licence GPL

loop.lua Corps de la boucle principale du programme

printf.lua Contient la déclaration de la fonction ANSI C printf en LUA

README.md Le fichier README du projet

terminal.lua Fonctions utiles au traitement textuel des inputs dans le terminal

useFullFunctions.grip Contient une librairie de fonctions récursives primitives utiles

2.2 Fonctionnalités

Les fonctionnalités que GRIP permet sont :
- la définition de prédicats grâce aux fonctions récursives primitives de base ainsi que les prédicats préalablement définis par l'utilisateur
- l'appel de ces prédicats pour obtenir leur résultat
- le chargement d'un fichier qui contient des instructions GRIP valides au démarrage, en particulier des définitions de prédicats

les différents opérateurs fournit par grip sont :
- z : fonction zero
- i : fonction identité
- s : fonction successeur
- $<$: arité gauche
- $>$: arité droite
- o : composition
- $(pred)$: utilisation du prédicat utilisateur *pred* dans une définition

3 Stratégie d'évaluation

3.1 Analyse syntaxique

L'une des raisons majeures pour lesquelles nous avons choisi le Lua pour l'implémentation de notre programme était liée aux facilités qu'il offrait pour le traitement des chaînes de caractères. Bien entendu c'est un avantage indéniable lorsqu'il faut concevoir un interpréteur en ligne de commande car il simplifie grandement l'analyste syntaxique des entrées de l'utilisateur.

La partie analyse syntaxique du problème est principalement implémenté dans deux classes : loop.lua et terminal.lua :

loop.lua : Ce fichier contient le corps de la boucle principale du programme. C'est dans ce fichier que l'instruction de l'utilisateur est traitée et c'est donc dans ce fichier que les erreurs de syntaxe potentielles de l'utilisateur sont vérifiées. Il va permettre de trouver la forme globale d'une instruction et déterminer si c'est une définition ou un appel.

terminal.lua : La chaîne obtenue est traitée grâce à des fonctions définie dans terminal.lua qui servent soit à informer l'utilisateur comme par exemple

des erreurs de syntaxe qu'il à commise ou qui servent tokeniser cette chaîne pour donner les tokens obtenus à l'analyseur sémantique.

3.2 Analyse sémantique

Avant de commencer la mise en place de l'analyse sémantique, nous avons dû faire plusieurs choix concernant la modélisation que nous souhaitions utiliser.

Pour la représentation des fonctions de base et des prédicats, le choix fut aisé. L'application garde deux tables associant des chaînes de caractères à des pointeurs sur fonction. Une de ces tables est connue dès le lancement du programme et recense les fonctions de base, l'autre recense les prédicats et est remplie lors de l'écriture de prédicats par l'utilisateur.

Un autre point clé de la modélisation était de choisir la représentation des arguments. La sensibilité de cet aspect vient de la nécessité d'avoir un système d'argument assez souple pour supporter le changement d'arité et permettre une manipulation aisée des arguments. Le système choisi représente tous les arguments dans une seule table. Cette table est également dotée d'un premier argument qui est l'indice du premier argument d'intérêt. Cet indice permet de faciliter les "décalages" d'arguments lors des changements d'arité.

La modélisation fixée, nous avons pu commencer la conception de l'analyse sémantique.

En nous basant sur nos cours de compilation du premier semestre, nous avons choisi une stratégie mettant en place un arbre d'évaluation. L'analyseur part du "haut" de l'arbre (le début de l'expression à analyser) et descend de manière récursive dans les nœuds. Après cela, en partant du bas, la récursion fait remonter des pointeurs sur fonction jusqu'à obtenir la fonction finale, celle qui sera sauvegardée en tant que prédicat.
Chaque nœud de l'arbre correspond à la lecture d'un ou plusieurs symboles de l'expression. Le symbole est lu, l'action correspondante est appelée et, si besoin, l'évaluation est rappelé sur la suite de l'expression.

Tout au long de ce processus, si une erreur est rencontrée, le message correspondant est affiché et l'évaluation s'arrête.

Suite à l'écriture de l'algorithme d'évaluation, nous avons dû faire face à un problème autrement plus complexe, comment créer des fonctions à partir d'autres fonctions de manière dynamique.
En effet, la fonction retournée par l'évaluation de l'expression : $ri >< s$ n'est pas la même et n'a pas le même résultat que celle retournée pas l'évaluation de l'expression : $r > z <> i$, bien que le symbole lu (r) soit le même. Aucun des paradoxes habituels de la programmation objet ne permet de réaliser ce type d'opération.
Heureusement, le Lua est un langage plein de ressources et en nous plongeant dans sa documentation, nous avons trouvé la solution à ce problème.
En effet, le Lua possède un mécanisme nommé "function closure" qui permet de faire du lambda calcul.
Grâce à ce mécanisme, nous pouvons écrire des fonctions génératrices de fonctions, ce qui correspond exactement à ce dont nous avons besoin pour la récursion, la composition et les changements d'arité.

L'exemple ci-dessous permet d'expliquer mieux ce fonctionnement.

```
function left_arity(func)

    if func == nil then
       print("ERROR : expecting function after arity change call")
    end

    local originalFunc = func[1]

    local modifiedFunc = function (args)

        args2 = TableCopy(args)
        args2[1] = args2[1] + 1
        return originalFunc(args2)
    end

    return {modifiedFunc, func[2]+1}
end
```

La fonction $left\_arity$ est l'exemple de fonction génératrice le plus simple que nous ayons.
Lorsque $left\_arity$ est appelée, la fonction originale passée en argument est sauvegardée par le système, elle fait partie de l'environnement de la fonction

que nous retournerons.
Une fonction $modifiedFunc$ est créée, elle est triviale, elle se contente de changer l'indice d'intérêt des arguments et de rappeler la fonction originale avec ces arguments modifiés. De cette manière, chaque appel à $left\_arity$ avec un paramètre différent générera une fonction unique.
Nous avons également utilisé ce mécanisme pour écrire des fonctions génératrices de récursion et de composition.

La simplicité et l'efficacité de cette solution est ce qui nous a particulièrement attiré. En effet, vu que l'on ne fait que manipuler des fonctions, chaque fonctions peut être passer en argument d'une autre, et ce jusqu'à obtenir la fonction finale. Cela rend le système d'évaluation particulièrement robuste, maintenable et facile à étendre.

4 Problèmes rencontrés

Nous n'avons pas rencontrés d'autres problèmes majeurs que ceux que nous avons résolu qui sont décrits dans la section **3.2 Analyse sémantique**.

5 Conclusion

Nous avons donc su mener à bien ce projet sans rencontrer de problèmes majeurs.
Le fait d'avoir pris le temps de chercher une solution à la fois adéquate et élégante nous a également permis d'avoir un résultat plus solide et maintenable.
De plus ce projet nous aura permis de découvrir le Lua et tous les avantages qu'il présente pour le traitement des données et le lambda calcul.