Rapport TP1 - Langage SLIP

Mathias La Rochelle & Michel Lamothe

Le samedi 29 septembre 2024

1 Introduction

1.1 Objectifs du TP

Afin d'avoir une session interactive sur GHCi avec aucuns avertissements et aucunes exceptions, nous allons devoir compléter les fonctions *s2l* et *eval*. Nous devons également écrire le code d'au moins 5 tests dans un fichier `tests.slip`. Tout doit bien s'exécuter évidemment.

À noter que le tout doit bien s'exécuter sur les machines du laboratoire du DIRO à travers une connexion SSH avec `ens.iro`.

1.2 Présentation du langage SLIP

Le langage SLIP est un langage qui se rapproche fortement de la famille de langages Lisp. Il s'agit d'un langage fonctionnel conçu pour être simple et expressif, tout en conservant certaines caractéristiques clés de Lisp. SLIP utilise une syntaxe basée sur les expressions S. Nous allons explorer en profondeur le fonctionnement interne de SLIP en implémentant ses composants essentiels.

1.3 Structure du rapport

Ce rapport est organisé de la manière suivante :

- La section 2 présente l'implémentation des fonctions principales, notamment *s2l* et *eval*.
- La section 3 détaille les exemples et les tests réalisés pour valider notre implémentation.
- La section 4 discute des difficultés rencontrées et des solutions apportées.
- La section 5 conclut le rapport en résumant nos réalisations et en proposant des pistes d'amélioration.

Tout au long de ce rapport, nous mettrons en évidence les concepts clés du langage SLIP et les choix d'implémentation effectués pour répondre aux exigences du TP.

2 Implémentation

Dans ce projet, l'objectif est d'implémenter une variante simplifiée de "Lisp" en utilisant Haskell. L'implémentation repose sur deux parties principales : la conversion des expressions symboliques "Sexp" vers une forme simplifiée "Lexp", et l'évaluation de ces expressions dans un environnement d'exécution. Nous avons dû construire deux fonctions manquantes, `s2l` et `eval`, pour assurer que le code fonctionne correctement en convertissant et évaluant les expressions de manière cohérente.

Après avoir analysé l'expression symbolique `Sexp`, nous devons la transformer en une forme de logique de programmation (`Lexp`), puis l'évaluée pour produire un résultat à l'exécution. L'implémentation met en jeu des concepts fondamentaux comme la définition de variables, l'évaluation d'expressions conditionnelles, les fonctions récursives et les appels de fonctions.

2.1 Structures de données utilisées

`Sexp` : Cette structure de données représente les expressions du langage que nous manipulons. Elle permet de représenter les entiers, les symboles et les listes imbriquées de manière récursive.

- `Snil` : Représente une liste vide.
- `Ssym String` : Représente un symbole sous forme de chaîne de caractères.
- `Snum Int` : Représente un entier.
- `Snode Sexp [Sexp]` : Représente une liste non vide d'expressions.

`Lexp` : C'est la forme intermédiaire d'une expression après analyse syntaxique. Elle simplifie les calculs et facilite l'évaluation.

- `Lnum Int` : Constante entière.
- `Lbool Bool` : Constante booléenne.
- `Lvar Var` : Référence à une variable.
- `Ltest Lexp Lexp Lexp` : Expression conditionnelle (équivalente à if).
- `Lfob [Var] Lexp` : Définition d'une fonction (objet fonctionnel).
- `Lsend Lexp [Lexp]` : Appel de fonction avec arguments.
- `Llet Var Lexp Lexp` : Déclaration locale non récursive.
- `Lfix [(Var, Lexp)] Lexp` : Déclaration de variables mutuellement récursives.

`VEnv :` Une liste de paires (Var, Value) où Var est le nom d'une variable et Value est la valeur correspondante. Utilisée dans l'évaluation pour suivre les variables définies et leur portée.

`Value :` Représente les différentes valeurs que peut manipuler le langage lors de l'exécution.

- `Vnum Int` : Entier.
- `Vbool Bool` : Booléen.
- `Vbuiltin ([Value] -> Value)` : Fonction primitive (comme + ou -).
- `Vfob VEnv [Var] Lexp` : Objet fonctionnel avec un environnement.

2.2 Fonctions principales

`s2l (Conversion de Sexp vers Lexp) :` La fonction s2l convertit une expression de type Sexp une expression logique Lexp. Cette conversion permet de simplifier la manipulation et l'évaluation des expressions. La façon dont s2l est implémenté :

- L'analyse commence par un pattern matching sur les différentes formes de Sexp.
- Si l'expression est un entier (Snum), elle est convertie en Lnum.
- Si l'expression est une fonction comme "let", elle est transformée en une structure de type "Llet".
- En cas de liste vide "Snil", la fonction "Lfob" est utilisée pour définir un objet fonctionnel avec une liste vide de paramètres.

`Exemple` : s2l (Snode (Ssym "let") [Ssym "x", Snum 2, Snode (Ssym "+") [Lvar "x", Lnum 3]])
`Résultat` : Llet "x" (Lnum 2) (Lsend (Lvar "+") [Lvar "x", Lnum 3])

2.3 Difficultés rencontrées et solutions

1. `Problème :` Extraction des arguments et liaisons récursives
   - `Description :` Lors de la conversion avec Lfix, il a été difficile d'extraire correctement les variables et leur corps.
   - `Solution :` L'utilisation de fonctions auxiliaires (bindExtraction et argsExtraction) a permis de gérer efficacement les expressions imbriquées et de structurer les arguments de manière cohérente.
2. `Problème :` Gestion de l'environnement d'exécution
   - `Description :` Il était complexe de maintenir un suivi précis de l'environnement avec des déclarations imbriquées. Par exemple, les variables définies dans un "let" ne doivent pas affecter l'environnement global.
   - `Solution :` L'approche consistant à ajouter temporairement des variables dans l'environnement a permis de résoudre ce problème sans modifier les environnements globaux. L'utilisation de listes immuables pour représenter l'environnement a également facilité la gestion.
3. `Problème :` Gestion de l'environnement d'exécution
   - `Description :` La gestion des fonctions et objets fonctionnels définis avec Lfob posait plusieurs défis. Contrairement à un simple appel de fonction, Lfob permet de définir une fonction en capturant un environnement local (appelé closure). Cela signifie que la fonction peut être appelée plus tard avec des arguments, en se basant sur les variables définies lors de sa création. Cependant, plusieurs problèmes sont apparus :
     i. `Correspondance des paramètres et arguments :` Il fallait s'assurer que le nombre de paramètres fournis lors de l'appel corresponde exactement au nombre d'arguments attendus par la fonction.
     ii. `Gestion de l'environnement capturé :` L'environnement d'exécution lors de la définition de la fonction devait être stocké et réutilisé lorsque la fonction était appelée.
     iii. `Appels imbriqués :` Dans certains cas, une fonction pouvait retourner une autre fonction, ce qui nécessitait une gestion des closures imbriquées.
     iv. `Traitement des erreurs :` Une mauvaise correspondance entre les paramètres et les arguments devait être signalée immédiatement avec un message d'erreur explicite.
   - `Solution :`
     i. `Validation du nombre d'arguments :` Lors de l'évaluation d'une fonction avec Lsend, une vérification explicite a été ajoutée pour s'assurer que le nombre d'arguments correspond au nombre de paramètres. En cas de déséquilibre, une erreur descriptive est levée:

        **if length params == length args
        then ...
        else error ("Manque " ++ show (abs (length params - length args))
        ++ " arguments.")**
     ii. `Gestion de l'environnement capturé :` L'environnement d'exécution lors de la définition de la fonction devait être stocké et réutilisé lorsque la fonction était appelée.
     iii. `Appels imbriqués :` Dans certains cas, une fonction pouvait retourner une autre fonction, ce qui nécessitait une gestion des closures imbriquées.
     iv. `Traitement des erreurs :` Une mauvaise correspondance entre les paramètres et les arguments devait être signalée immédiatement avec un message d'erreur explicite.
4. La section 5 conclut le rapport en résumant nos réalisations et en proposant des pistes d'amélioration.

Je ne sais pas :(

3 Exemples et tests

3.1 Cas de test simples

Test 1 : Fonction sans paramètre

Expression :

```
(fob () 42))
```

Explications :

- Teste la création et l'appel d'une fonction sans paramètre.
- Vérifie que la fonction retourne bien la valeur 42.

Test 2 : Fonction avec paramètre simple

Expression :

```
((fob (x) (if (= x 5) true false)) 5)
```

Explications :

- Teste une fonction qui prend un paramètre.
- Vérifie si le paramètre vaut 5 pour retourner `true`, sinon `false`.

Test 3 : Fonction avec let et fermeture

Expression :

```
(let x 5 ((fob (x) (+ x 3)) x))
```

Explications :

- Déclare une variable `x` locale avec la valeur 5.
- Appelle une fonction qui ajoute 3 à `x` et retourne le résultat.

3.2 Cas de test complexes

Test 4 : Fonction récursive (factorielle)

Expression :

```
(fix ((fact (fob (n)
                    (if (= n 0) 1
                    (* n (fact (- n 1)))))))
     (fact 5))
```

Explications :

- Teste la récursion pour calculer la factorielle de 5.
- Vérifie que la base de récursion (`n = 0`) retourne 1.

Test 5 : Fonction imbriquée avec trois paramètres

Expression :

```
(let volume ((((fob (x) (fob (y) (fob (z) (* x (* y z))))) 3) 2) 4)
            (if (= volume 16) true false))
```

Explications :

- Teste une fonction imbriquée prenant trois paramètres (`x`, `y`, `z`).
- Calcule `volume` et vérifie si la valeur obtenue est 16.

Extension du langage

Nouvelles fonctionnalités implémentées

Justification des choix

Conclusion

Récapitulatif des réalisations

Perspectives d'amélioration

Annexes

Code source complet

Exemples supplémentaires