Auteur : R. Namyst
Enseignants : L. Bougé A. Bonhomme
Email : `luc.bouge,alice.bonhomme@ens-lyon.fr`

le 18/12/00

TD11 : Interprète LISP en CAML

L'objectif de ce TP est de continuer l'étude du fonctionnement de l'interprète du langage LISP par le biais d'une implantation simple en CAML.

1 Prise en main

- Récupérez les sources du mini-interprète développé par Luc Bougé à l'adresse

  `http://www.ens-lyon.fr/~bouge/Teaching/Programmation/Lisp_Caml/`

  (fichiers `lisp_dynamic_pure.ml`, `lisp_dynamic_pure_defs.ml`, `eval.l`)
- Testez-le à l'aide d'exemples simples. Référez-vous au source pour découvrir les fonctions supportées.

2 Visualisation de l'environnement

- Écrivez une fonction `print_env` qui affiche la totalité de l'environnement courant.
- Forcez son exécution à chaque étape de l'évaluation interne, en attendant une entrée au clavier entre chaque étape. Ceci permet de faire une évaluation pas à pas.

3 Ordre d'évaluation des paramètres

- Utilisez la fonction `print` pour observer l'ordre dans lequel l'interprète évalue les éléments d'une liste.
- Modifiez l'interprète de manière à inverser cet ordre.

4 Rattrapage d'erreurs

- Modifiez l'interprète pour qu'il rattrape les erreurs et affiche un message d'erreur en fonction du type d'erreur (syntaxe, sémantique, etc.), ceci sans sortir du `toplevel`.

5 Méta-interprétation

- Vous disposez maintenant de tout ce qu'il vous faut (ou presque) pour évaluer sous cet interprète CAML l'interprète LISP que vous aviez développé la semaine dernière... Testez-le !
- Améliorez votre interprète de manière à pouvoir définir des liaisons au niveau du `toplevel` et évaluer des lambda-expressions.
- Pourriez-vous faire exécuter votre interprète LISP en LISP par lui-même ? Chiche !
- Regarder le méta-interprète `Scheme` (situé dans le même répertoire que les sources, `*.scm`). Que faudrait-il changer pour qu'il fonctionne avec l'interprète actuel ?

6 Ajout de `subroutines` dans l'environnement

Dans l'interprète LISP en Caml actuel, la fonction `apply` teste successivement la fonction à exécuter contre chacune des fonctions prédéfinies. Ceci n'est pas *extensible* : pour ajouter une fonction prédéfinie, il faut modifier la fonction `apply`. Une autre solution est de placer les noms des fonctions prédéfinies dans l'environnement initial. La valeur associée est simplement la fonction Caml : par exemple, le symbole `+` est associé à la fonction `do_plus`. Ainsi, il suffit d'ajouter un cas à la fonction `apply` : si le symbole à appliquer est une fonction Caml `f`, alors lancer cette fonction sur la liste des arguments `(f lval)`.

Pour mettre en place cette idée, il suffit de pouvoir stocker les fonctions Caml comme valeurs dans l'environnement. Dans la tradition LISP, ces fonctions sont appelées des *subroutines*, `SUBR`. Le plus simple est d'étendre le type des objets LISP :

```
type lisp_object =
  | Lisp_Atom of string
  | Lisp_Num of int
  | Lisp_List of (lisp_object list)
  | Lisp_Subr of (lisp_object -> lisp_object)
  ;;
```

- Étendez le type `lisp_object` avec une nouvelle composante `Lisp_Subr` et modifiez l'ensemble des fonctions de manipulation en conséquence.
- Définissez une fonction `subr_to_obj` qui retourne un objet LISP correspondant à la fonction passée par paramètre. Définissez la réciproque `obj_to_subr`. Définissez le prédicat `subrp`.
- Modifiez la fonction `apply` pour prendre en compte ces nouveaux objets.
- Définissez une fonction `extend_subr_env name function env` qui ajoute une *subroutine* `subr` de nom `name` à l'environnement `env` et retourne l'environnement ainsi obtenu.
- Construisez l'environnement initial sur lequel lancer la fonction `toplevel`.
- Testez ces modifications sur des exemples. (Notez que l'on peut maintenant *redéfinir* la fonction `+` !)
- Ne pourrait-on pas étendre ces idées au cas de `quote`, `if` et `progn` ? Traditionnellement, ces fonctions sont appelées `FSUBR` (*functional subroutine*).