But du TP

Vous trouverez sur la page Web du cours un fichier `tp11a.rkt` qui contient le source très incomplet d'un évaluateur du langage *MIPS* [MIPS Is Pseudo-Scheme] dont les constructions linguistiques et leur sémantique ont été détaillées en cours. L'interprète est presqu'immédiatement exécutable, même s'il ne donnera pas les résultats espérés avant la fin du TP : il ne sait évaluer que les constantes !

Votre mission consiste donc à programmer les fonctions manquantes ou incomplètes, et à rajouter quelques fioritures. Une fois ce TP assimilé, vous êtes censés être capable de programmer un tel interprète, en modèle réduit, un jour d'examen [voir le sujet de l'an passé]. Dès que vous pensez être en état de le faire, nous ne serions que trop vous conseiller de prendre 2 heures, de vous mettre devant une machine, et de reprogrammer complètement l'évaluateur sans faire appel à vos notes. Vous aurez alors fait un pas vers la Sagesse…

Dans cet interprète, et notamment dans les transparents du cours, nous avons fait le choix de ne pas utiliser de type abstrait pour représenter les expressions, et de les voir directement comme des listes [avec `car`, `cdr`, `cons`, etc]. D'une part faute de temps et d'autre part pour vous obliger à pénétrer au cœur des expressions LISP. Il s'agit bien entendu d'une faute logicielle, et un interprète en grandeur réelle ne le tolèrerait pas. L'interprète d'Abelson & Sussman, dont nous sommes proches, les utilise.

Section « GESTION DES ENVIRONNEMENTS »

Vous avez bien entendu vos notes de cours sous les yeux, à 3.14 cm du clavier. Les neurones ne sont plus à l'état de veille.

Exercice 11.1 Exécutez le fichier. Vous arrivez sur la boucle-toplevel de *MIPS*. Evaluez les constantes 2021 et `#t`, tout va bien ? Et pour les primitives ? Evaluez le symbole `$+` et réagissez en conséquence (définissez la fonction `%dict`).

Exercice 11.2 Complétez les fonctions `%new-dictionnary`, `%extend-env` et `%extend-dict!` (le *bang* fait référence à une mutation du dictionnaire).

Exercice 11.3 Comprenez bien le texte de la fonction (`$init-global-env`) qui installe les primitives arithmétiques dans l'environnement global. Il manque l'égalité numérique `$=`, rajoutez-la… Regardez bien la différence entre le traitement par la *quasiquote* ` du nom et de la valeur du symbole `$+` par exemple. Dans cette implémentation, les pointeurs `$global-env` et `$global-dict` sont fixes, ils pointeront toujours vers la même zone mémoire.

Section « L'EVALUATEUR *EVAL* »

Exercice 11.4 Complétez les fonctions de cette section sans en rajouter d'autres. La fonction (`$evlis Lexpr env`) renvoie la liste des résultats des évaluations des expressions éléments de la liste `Lexpr` dans l'environnement `env`.

A partir de maintenant, vous pouvez demander la valeur d'un symbole. Quelle est la valeur du symbole = au toplevel ?

Section « L'APPLICATEUR FONCTIONNEL *APPLY* »

Le second volet du couple `eval/apply`. La fonction (`$apply proc Largs`) applique la valeur procédurale `proc` [une primitive ou une fermeture] à la liste d'arguments `Largs` déjà évalués et retourne le résultat de cette application. Elle se moque de l'environnement courant, en vertu de la liaison lexicale. Dans le cas où `proc` est une *fermeture*, celle-ci contient [un pointeur vers] l'environnement lexical dans lequel seront recherchées les valeurs des variables libres de son corps.

Exercice 11.5 Complétez la fonction (`$apply proc Largs`). Vous pouvez maintenant évaluer des expressions arithmétiques au toplevel *Mips*, essayez…

Section « L'ABSTRACTION FONCTIONNELLE *LAMBDA* »

En λ-calcul, on parle d'*abstraction* pour une λ-expression et on la distingue d'une *application* qui est l'exécution de cette λ–expression sur des arguments. On rappelle que la valeur d'une lambda-expression est une **fermeture**, en anglais « *closure* ». Une fermeture sera ici représentée de manière interne par une liste:

```
(*closure* Lparams corps env)
```

où env pointe vers l'environnement lexical de la fermeture [environnement de création de la lambda].

Exercice 11.6 Complétez la fonction (`$eval-lambda expr env`) prenant une expression :

```
expr == (λ (x ...) e1 e2 ...)
```

et une environnement env, et retournant la fermeture correspondante sous la forme d'une liste à 4 éléments comme ci-dessus. Notez que le `begin` est optionnel dans le corps de la lambda. On le forcera donc à la création de la fermeture dans le cas où le corps de la lambda contiendrait plusieurs expressions. Vous testerez en évaluant par exemple au toplevel de *Mips* l'expression :

```
((λ (x y) ($* 2 ($+ x y))) 3 4)
```

Exercice 11.7 Evaluez au toplevel de Mips l'expression `(λ (x) ($+ x 2))` et contemplez le résultat. Le déluge qui apparaît n'est autre que la *représentation interne* d'une fermeture `(*closure* Lparams corps env)`. Cette fermeture étant anonyme, elle n'apparaît pas dans l'environnement.

Section « LA CONDITIONNELLE *IF* »

Exercice 11.8 Complétez la fonction (`$eval-if expr env`) dans laquelle `expr` est de la forme `($if p q r)`. On n'autorise pas la forme réduite `($if p q)` à moins que vous n'y teniez absolument [auquel cas le résultat si `p` est faux est *non spécifié* : il peut dépendre du compilateur - par exemple `(void)` - et on n'est pas censé l'utiliser sous peine de n'être plus portable]. Vous testerez en évaluant par exemple :

```
($+ ($* 2 3) ($if ($< 4 5) ($* 6 7) ($/ 8 9)))
```

Section « LES VARIABLES LOCALES : *LET* »

Exercice 11.9 **a)** Complétez la fonction (`$eval-let expr env`) dans laquelle `expr` est de la forme :

```
($let ((x v) ...) e1 e2 ...)
```

le `begin` étant ici encore forcé si besoin. Testez-le en évaluant l'expression en exemple dans l'exercice 6, puisque vous avez bien évidemment observé qu'il s'agissait d'un `let` déguisé…

b) Construisez au toplevel une fermeture dans un environnement privé et observez la valeur rendue.

Section « LA MUTUALITE RÉCURSIVE *LETREC* »

Exercice 11.10 **a)** Complétez la fonction (`$eval-letrec expr env`) dans laquelle `expr` est de la forme :

```
($letrec ((f func) …) e1 e2 …)
```

dont on rappelle la sémantique [extraite du R^{5}RS] :

Semantics: The variables f… are bound to fresh locations holding undefined values, the inits func… are evaluated in the resulting environment [in some unspecified order], each variable f is assigned to the result of the corresponding init, the body e1, e2,… is evaluated in the resulting environment, and the value of the last expression in the body is returned. Each binding of a variable f has the entire letrec expression as its region, making it possible to define mutually recursive procedures.

Vous vous inspirerez donc de `$let`, mais ce sera un peu plus compliqué, lisez bien la sémantique… En réalité, `$letrec` ne doit servir que pour des fonctions locales [mutuellement récursives], pas pour des variables locales numériques par exemple !

b) Testez en calculant 100! au toplevel [rép: 93326….], ou bien la mutualité classique even/odd qui exprime qu'un nombre est pair [resp. impair] si son prédécesseur est impair [resp. pair].

c) Evaluez au toplevel l'expression :

```
($letrec ((add ($λ (x y) ($if ($= x 0) y ($+ 1 (add ($- x 1) y))))))
   add)
```

qui est une fermeture, et observez la valeur affichée. Ce nouveau déluge qui s'abat sur vous n'est autre que la représentation interne d'une fermeture, qui contient un pointeur vers l'environnement qui… contient la fermeture, ce qui produit donc un

objet *cyclique* dont l'impression devrait boucler. Dans son infinie bonté, Racket maîtrise de tels objets et repère certains nœuds de ce graphe cyclique par ce qu'il nomme `#0`, `#1`, etc. Essayez de bien comprendre ce qui est affiché et d'y voir les pointeurs sous-jacents…

Section « LA DÉFINITION AU TOPLEVEL : *DEFINE* »

Enfin la possibilité de définir des variables, et de procéder à des tests plus élaborés ! Nous nous écartons un peu de la norme SCHEME en autorisant `define` comme introducteur de symbole lié dans l'environnement global uniquement. Donc pas de `define` interne, il faudra utiliser `letrec` ! Pour savoir si l'environnement courant est bien l'environnement global, on utilisera `eq?` qui compare des pointeurs. Dans le cas où le symbole est déjà lié, `define` se comportera ici comme une simple affectation.

Exercice 11.11 Complétez la fonction (`$eval-define expr env`) dans laquelle `expr` est de la forme (`$define var e`) puis testez-la.
b) Demandez à voir la valeur d'une fermeture définie au toplevel ou dans un environnement privé.

Section « LE SÉQUENCEMENT : *BEGIN* »

Pour la programmation impérative, l'évaluation séquentielle d'expressions.

Exercice 11.12 **a)** Complétez la fonction (`$eval-begin Lexpr env`) dans laquelle `Lexpr` est de la forme (`e1 e2 …`) donc non vide. On évalue les expressions `e1,e2,...` dans cet ordre et on retourne le résultat de la dernière expression. Testez sur la fonction : (`$define foo ($lambda (x) ($+ x 2) ($* x 3))`). En l'absence d'affectation et d'entrées-sorties, `begin` n'est pas encore très utile…
b) *[optionnel]* La norme Scheme impose le caractère implicite du `begin` dans le corps d'une lambda. Comment en Scheme usuel pourriez-vous alors définir `begin` s'il n'existait pas ?…

Section « L'AFFECTATION : *SET !* »

Pour la programmation impérative, l'instruction d'affectation qui ne retourne aucun résultat [on dit plutôt que le résultat est *non spécifié*]. Cette instruction cherche la première liaison d'un symbole dans l'environnement courant et modifie physiquement [écrase] la valeur associée sans création de nouvelle liaison. Contrairement au `setq` de Emacs-Lisp, c'est une erreur en Scheme d'affecter un symbole non encore lié dans l'environnement courant !

Exercice 11.13 Complétez la fonction (`$eval-set! expr env`) dans laquelle `expr` est de la forme (`$set! var e`).
Le résultat de `$set!` est non spécifié. Testez sur l'inévitable factorielle impérative.

Section « EXTENSION SOFT DE L'ENVIRONNEMENT GLOBAL»

L'environnement global ne contient jusqu'à présent que des fonctions très primitives comme les fonctions arithmétiques. Vous allez lui ajouter du logiciel chargé automatiquement à l'allumage, c'est-à-dire des fonctions prédéfinies écrites en *Mips*.

Exercice 11.14 Ecrivez la fonction (`$install-software`) chargée d'étendre le dictionnaire global en lui rajoutant du "soft", par exemple une constante `pi = 3.14…` ainsi que la fonction factorielle (`fac n`) et la fonction (`zero? n`).
Utilisez `$extend-dictionary!` *et prenez exemple sur* `$init-global-env`.

Exercice 11.15 *[terriblement facile]* **a)** Ajoutez à la fonction `$eval` le traitement de la forme spéciale `quote` :

```
> (define foo 'tralala)    ; rappel : en Lisp, 'expr est une abréviation de (quote expr)
--> #<void>
> foo
--> tralala
```

b) *[terriblement astucieux]* Installez en « soft » les fonctions `cons`, `car` et `cdr`. Il s'agit bien d'une installation « soft », sans utiliser les vraies fonctions Scheme `cons`, `car` et `cdr`, ni les listes. Si vous séchez, faites une installation « hard »…

Section « LE TOPLEVEL »

Le toplevel est déjà écrit à la fin du fichier source, ne serait-ce que pour pouvoir tester tout ce qui précède. Les fonctions `$print` [le *scribe*] et `$read` [le *lecteur*] sont spartiates et directement issues de leurs analogues Scheme. En les reprogrammant, on pourrait se tailler sur mesure les lectures et affichages de la boucle REP.

Exercice 11.16 Modifiez le scribe et peut-être autre chose pour que l'on ait au toplevel *MIPS* presque le même style d'affichage de la représentation interne d'une primitive ou d'une fermeture qu'au toplevel Racket :

```
? ($lambda (x) ($+ x 2))
#<closure>
? ($define foo ($lambda (x) ($* x 4)))   ; au passage, supprimez l'affichage du résultat de define…
? foo
#<closure:foo>
? $+
#<primitive:+>
```