Help

Version 1.0 (Janvier 91)

Un dialecte Lisp paresseux inspiré de Scheme

e-mail: schiex@cert.fr, schiex@irit.fr

Table des Matières

Les consignes habituelles pour la diffusion d'un logiciel domaine public sont applicables à Help: ne diffusez pas l'archive si vous la modifiez (surtout si des fichiers disparaissent). Passez plutot par moi pour les ajouts ou modifications...

Thomas SCHIEX
Centre d'Etudes et de Recherche de Toulouse (ONERA)
2, Av Edouard Belin
BP 4025
31055 TOULOUSE CEDEX
e-mail: schiex@cert.fr

A titre indicatif, Help est essentiellement écrit en assembleur. Son portage en C ou autre langage portable n'est pas envisagé à court terme. Je dégage bien sur toute responsabilité pour les dommages que pourraient causer l'utilisation de Help. Vous pouvez évidemment me faire parvenir toute remarque, mais je ne garantit pas de la prendre en compte.

Enfin, les différents fichiers et dossiers de l'archive sont organisés comme suit:

Paresseux: l'interprete HELP lui-même ;
Help Files: comme son nom l'indique. On trouvera differents problemes classiques du lambda-calcul, un solveur de contraintes s'appuyant sur un "test and generate" bestial, un compilateur pour Help...
Docs: comme son nom l'indique.
Resources: Des ressources à coller dans Paresseux pour changer les menus en français ou anglais, des ressources pour changer la semantique, des resources à coller dans votre Resedit pour faciliter l'édition des ressources Help (cf .5.5).

Bonne chance !

5 Le manuel

Rappelons que Help est, avant tout, un dialecte Lisp. Toute personne connaissant déjà bien Lisp et plus particulièrement Scheme peut se permettre (dans un premier temps) de survoler rapidement la présentation de Help et de ses objets. Une utilisation de l'évaluateur sur MacII nécessite la lecture du §5.5.

5.1 Survol de Help

5.1.1 Sémantique

- Help est un langage statique[1] (portée des variables lexicales) tout comme Scheme, Algol… Chaque utilisation d'un identificateur est associée avec une liaison lexicalement visible de cet identificateur.

- Help est un langage non strict, employant l'évaluation paresseuse (call by need) pour **tous** ses passage d'arguments comme le font Lazy Miranda, Hope…

- Help est un langage faiblement typé. Les types sont associés aux valeurs plutôt qu'aux identificateurs de variables. C'est le cas de tous les dialectes Lisp mais aussi d'APL, Snobol…

- Les fermetures (ou procédures) Help sont des objets Help à part entière qui peuvent être créés dynamiquement, accumulés dans des structures de données de taille éventuellement infinie… Help partage cette qualité avec Scheme et la grande majorité des langages fonctionnels (Hope, Miranda, Daisy…)

- Les objets Help (fermetures , environnements inclus) ont une durée de vie illimitée. Le système de gestion de la mémoire ne se permettra de récupérer la place occuper par un objet que s'il peut démontrer qu'il ne peut plus être référencé (Garbage Collector). Tous les dialectes Lisp, APL et de nombreux langages fonctionnels reposent sur un système de gestion de la mémoire de ce type.

- Help ne possédant aucune structure de contrôle itérative (s'accommodant mal avec la paresse), les évaluateurs Help éliminent les récursions terminales simples. Ceci permet d'utiliser la récursivité en utilisant une place mémoire fixe si cela est possible.

5.1.2 Syntaxe

- Help emploie la notation polonaise avec parenthèses et crochets pour décrire la majorité des programmes et données. Cette syntaxe dont la pauvreté est une des grande richesse est employée (à quelques variantes prés) par tous les dialectes Lisp récents[2].

5.1.3 Notation et terminologie

La définition de Help est loin d'être figée et de nombreux retours de valeurs, domaine de définition … ne sont pas encore complètement spécifiés. Dans ces cas, on parlera de valeur…non spécifiée, ce qui est différent d'une valeur indéterminée qui s'exprime en Help par le biais du symbole d'erreur "`?`".

Dans la suite, les exemples codés en Help seront systématiquement imprimés dans la fonte `exemples`. Le symbole "➠" utilisé dans ces exemples doit se lire "dont la valeur s'imprime". Du fait de la paresse, il y a une différence importante entre forme interne (contenant des formes suspendues) et forme externe (forme affichée), cette distinction est donc primordiale. Les valeurs de taille infinie sont, en général, limitée par l'imprimeur. Des points de suspension (…) indiqueront que la structure n'a pas été épuisée.Par exemple:

[1]Du moins, c'est la cas dans le cadre d'une utilisation normale. Il est possible d'élargir la portée d'une variable (Cf § 3.5.1.1).

[2]Les premiers évaluateurs Lisp utilisaient une syntaxe différente, distinguant programmes et données (M Expressions opposées aux S Expressions actuelles).

```
(letrec [(x (cons 1 x))] x) ➠ (1 1 1 1 1 1 1 …)
```

Les § 5.3 et 5.4 sont organisés par “fiches”. Chaque fiche présente une fermeture ou une forme syntaxique du langage et commence par un en-tête comportant un modèle d’utilisation de la forme précisant son type (forme syntaxique, fermeture (séparées en `ProcN` d’arité fixe et `NProc` d’arité variable) ainsi que son arité (arité minimale pour les `NProcs`). Le domaine de définition de la forme est indiqué dans le modèle d’appel par le nom des arguments en fonte *`arguments`*. Si il n’y a pas de restriction sur le type des arguments, le mot *`quelconque`* sera utilisé. Les types référencés sont:

`nombre` ---------- entier taille fixe, “bignum” ou flottant
`entier` ---------- entier taille fixe, “bignum”
`fix` ---------- entier taille fixe
`posfix` ---------- entier taille fixe positif ou nul
`bignum` ---------- “bignum”
`flottant` ---------- flottant
`smallnul` ---------- entier de taille fixe ou flottant
`liste` ---------- doublet ou ()
`doublet` ---------- doublet
`cellule` ---------- cellule
`environ`---------- environnement
`applicable`---------- une fermeture ou un *`fixpos`*
`symbole` ---------- symbole, erreur, constante,mot-clé…
`ident`---------- identificateur de variable ou contante
`identv`---------- identificateur de variable
`mot-clé` ---------- mot clé syntaxique
`erreur` ---------- erreur
`bit-array` ---------- tableau de bits
`io-unit`---------- unité d’entrées/sorties
`quelconque` ---------- tous type d’argument

Exemple:

(**`cell=?`** *`cellule posfix`*)	Fermeture:`ProcN`	2

indique que **`cell=?`** est une fermeture d’arité fixe et égale à 2, prenant en argument une cellule et un nombre entier positif de taille fixe.

5.2 Objets lisibles et non lisibles

Un grand nombre de types d’objets différents cohabitent dans la mémoire de Help à un moment donné. La majorité des objets dont la représentation externe a une importance pour l’utilisateur sont aussi “lisibles” (i.e. il est possible de décrire un de ces objets de façon à ce que le lecteur Help créé en mémoire une représentation interne de cet objet). Pour des raisons pragmatiques ou de faisabilité, certains objets possèdent une représentation interne, éventuellement une représentation externe mais ne sont pas lisibles.

Le lecteur Help ne fait aucune distinction entre majuscules et minuscules (si ce n’est à l’intérieur des chaînes de caractères et des symboles internés via le caractère “backslash”). Ainsi les symboles `Foo`, `fOO` représentent un seul symbole. Les caractères d’espace (ASCII 32) et de retour chariot (ASCII 13) sont des délimiteurs.

Dans la suite, nous distinguerons séquence de caractères (suite de caractères dans le flot d'entrée délimitée par des caractères … délimiteurs (Cf § 5.2.7)) et chaîne de caractères qui est un type d'objet Help.

5.2.1 Les nombres (fixnums, bignums et flottants)

Pour l'instant, rationnels et complexes ne sont implémentés en Help. De plus, il existe encore de grosses restrictions sur les nombres entiers en précision illimitée. Ceci est principalement dû à l'age du langage Help.

5.2.1.1 Les entiers

Un nombre entier est toujours lu dans la base d'entrée courante du lecteur. Cette base est fixée via la fermeture `ibase` décrite par ailleurs. La définition des caractères formant ou non un nombre entier dépend de cette base. Si la base est inférieure ou égale à 10 (en décimal), alors les caractères compris entre "`0`" et la base moins un forment le jeu de caractère dédiés aux nombres entiers. Si cette base dépasse 10 (elle est limitée à 36 de façon interne), le jeu de caractère est étendu avec "`a`"…"`z`". Ainsi, en base hexadécimal, la chaîne de caractères "`ff`" décrit elle un nombre entier (et non un symbole) de valeur 255 (décimal).

Soit `CarNum`, l'ensemble des caractères couramment dédiés à la lecture des nombres.

```
nombre::=               nombre_non_signé        |
                        +nombre_non_signé       |
                        -nombre_non_signé
nombre_non_signé::=     CarNum*
```

Exemples: `123 +123 -123 +235987459862345` (en base 10 au moins)
`12af` (base 16) `-foobar` (base 29)

Dans l'avenir, la lecture se fera par défaut en décimal, et c'est le flot de lecture lui-même qui pourra éventuellement décider de changer localement de base via l'utilisation de caractères spéciaux.

5.2.1.1.1 Entiers de taille fixe

Quand il existe une représentation d'un nombre en entier de taille fixe, le lecteur générera automatiquement cette représentation interne. La valeur d'un entier de taille fixe est comprise entre -2 147 483 648 et 2 147 483 647.

5.2.1.1.2 Entiers en précision illimitée

Tout nombre entier dépassant la capacité d'un entier fixe sera automatiquement lu sous forme de "bignum". Il faut tout de même être sensible au fait que l'utilisation des "bignums" repose sur un usage intensif des piles et dimensionner la taille de celles-ci en conséquence. De même, l'impression d'un gros "bignum" (…) peut nécessiter un travail important.

L'allocation de "bignums" temporaires sur la pile lors de calculs sur ceux-ci implique que la taille d'un "bignum" est limitée: 1- par la place restante dans le Tas;
2- par la place restante dans la pile !

Il est donc possible de générer une erreur du type "Piles pleines" avec une écriture **récursive terminale** de Fibonacci pour des valeurs importantes (essayez `(fib 50000)`) du fait seul de la place occupée sur la pile par les "bignums" manipulés 5NB/ il existe à l'heure actuelle un bug dans la multiplication de bignums)..

5.2.1.2 Les flottants

Toute séquence de caractères (qui n'est pas à l'intérieur d'une chaîne de caractères ou d'un symbole interné via le "backslash") contenant le caractère "." sera considéré par le lecteur comme l'expression d'un nombre flottant.

Le standard Motorola IEEE "extended-precision floating point" est actuellement utilisé. Il permet de coder des flottants d'approximativement $1.9*10^{-4951}$ à $1.1*10^{4932}$. Ils possèdent entre 19 et 20 chiffres de précision. Il permet aussi de représenter le résultat d'opérations illicites (division par zéro...) en générant des "NaNs[3]" ou des "infinités". Ces derniers ne sont pas lisibles, mais sont imprimables et manipulables comme comme n'importe quel flottant.

La lecture peut se faire en notation scientifique (emploi du caractère "`e`" ou "`f`" pour séparer mantisse et exposant) ou en notation classique (mantisse seule). Le nombre flottant est toujours lu en décimal. La mantisse, comme l'exposant peuvent être précédées d'un signe "`+`" ou "`-`".

Exemples:
```
1.123456789123456789
1.1234f-2001
-233.2e+63
```

5.2.2 Les listes (doublets)

La notation des listes est parenthèsée et repose sur la notion de doublets[4] (comme en Lisp):

- Les caractères "`(`", "`)`" et "`|`" sont réservés à la représentation des listes et sont des délimiteurs;
- Une liste vide sera représentée par un symbole spécial noté "`()`";
- Un doublet \x(`car` | `cdr`) est représenté par: `(` <car> `|` <cdr> `)` ou <car> est la représentation de "`car`"...;
- Une notation simplifiée permet d'éviter d'écrire (dans le cas où un `cdr` est un doublet ou la liste vide) les couples "`| (`" et "`)`" correspondante. Ainsi, on écrira indifféremment: `(1 | ())` ou `(1)`.

[3]NaN= Not a Number. Retourné lors de la lecture d'une séquence de caractères ne formant pas un nombre flottant.

[4]structure de données comportant deux champs non typés appelés (pour des raisons historiques) `CAR` et `CDR` (prononcé "coudair).

Exemples:
```
(1 | 2)
(1 2 3 4)
(1 | (2 | ()))
(1 (2 3 4) (5 (6)) 1)
```

5.2.3 Les cellules (vecteurs)

Une cellule (ou vecteur) sera représentée au moyen des caractères spéciaux délimiteurs "`[`" et "`]`". Ainsi, une cellule de trois éléments `e1`, `e2`, `e3` sera représentée par `[<e1> <e2> <e3>]`

Exemples:
```
[]
[1 2 3]
[[1 2][2 3]]
```

5.2.4 Les tableaux de chiffres binaires

Le caractère délimiteur "`%`" permet de représenter les tableaux de chiffres binaires (bit-arrays). La séquence de caractère qui le suit doit être formée uniquement de caractères "`0`" ou "`1`".

Exemples:
```
%
%11011000110110001101100011011000
```

5.2.5 Les chaînes de caractères

Le caractère "`"`" sert de délimiteur aux chaînes de caractères. Tous les caractères suivant un "`"`" jusqu'au prochain "`"`" sont considérés comme faisant partie de la chaîne de caractères formée (y compris les caractères de contrôle comme retour chariot, linefeed...)

5.2.6 Les symboles atomiques (symboles, constantes, erreurs...)

Toute séquence de caractères n'entrant pas dans les domaines syntaxiques définis précédemment et ne contenant pas de caractère délimiteur "`:`"sera interprété comme un symbole atomique simple.

Il est possible de "forcer" le lecteur à reconnaître un symbole dans une unité syntaxique quelconque en utilisant le caractère spécial "`\`". Ainsi, la lecture du flot de caractères "`\12\`" retournera un symbole dont le nom sera "12".

De plus, les symboles atomiques forment une arborescence (identique à l'arborescence des packages en Le_Lisp) à la disposition de l'utilisateur. Un symbole atomique simple a pour parent "`()`". Il est possible de désigner un des fils d'un symbole en faisant suivre ce symbole du caractère délimiteur "`:`" et du symbole fils. Ainsi:

`a` ---------- symbole "`a`" de père "`()`"
`a:b` ---------- symbole "`b`" de père "`a`", lui même de père "`()`"
`a:b:c:d:e`----------symbole "`e`" de père "`d`"...

Les symboles atomiques ont de nombreuses utilisations en Help. Ils permettent de représenter:

- symboles---------- `'a`
- identificateurs de variable---------- `a`
- mots clés---------- `(lambda (x) (1+ x))`
- macro-expressions---------- `(defmacro useless (lambda (t) '()))`
- erreurs---------- `?:var-undef`
- constantes---------- `(1+ 1)`

Il n'y a pas véritablement de syntaxe particulière pour chacune de ces représentation. Des moyens sont (en général) fournis à l'utilisateur pour associer à un symbole donné un type donné.

Au démarrage, un certain nombre de symboles sont déjà ainsi définis.

5.2.6.1 Mots clés:

Les symboles suivants sont réservés comme mots clés du langage:

```
lambda cond define =! begin step nomemo
defmacro quote let letrec bindings warn
```

5.2.6.2 Erreurs:

Le tableau suivant permet de considérer l'ensemble des erreurs prédéfinies du système Help. En général (ce n'est pas obligatoire) une erreur est un symbole fils du symbole" `?`".

Symbole	Cause de l'erreur
`?`	Erreur indéfinie
`?:too-args`	Trop d'arguments
`?:few-args`	Trop peu d'arguments
`?:bad-type`	Mauvais type d'arguments
`?:bad-expr`	Expression mal formée
`?:syn-keyw`	Mot clé en mauvaise position
`?:syntx-er`	Erreur de syntaxe
`?:mem-full`	Plus de mémoire
`?:no-apply`	Objet non applicable
`?:indx-out`	Index hors limite
`?:strange!`	Erreur indescriptible
`?:overflow`	Dépassement de capacité
`?:cb-break`	Break utilisateur
`?:lispstck`	Pile Lisp inconsistante
`?:contstck`	Pile de contrôle inconsistante
`?:varundef`	Variable non définie
`?:stckfull`	Piles pleines (collision)
`?:maxlengt`	Longueur max atteinte
`?:io-error`	I/O error

`?:eof-error`	Fin de fichier
`?:dead-cont`	Continuation chronologique morte

5.2.6.3 Constantes:

De très nombreuses constantes sont prédéfinies. La grande majorité de celles-ci dénotent des fermetures et seront décrites dans §5.4.

Le symbole "()" est une constante égale à lui-même dénotant la liste vide;
Le symbole "†" est une constante égale à lui-même dénotant le booléen VRAI;
Le symbole "ƒ" est une constante égale à lui-même dénotant le booléen FAUX;

A la différence de la grande majorité des dialectes Lisp, le symbole "()" est interprété comme VRAI en HELP. Seul le booléen "ƒ" dénote FAUX.

5.2.7 Objets non lisibles

Parmi tous les objets apparaissant dans le mémoire, un certain nombre ne sont pas descriptibles directement par l'utilisateur. Il s'agit des:

- Environnements (imprimables);
- Fermetures (imprimables);
- Code (imprimable partiellement);
- Unités d'I/O (non imprimables);
- Formes suspendues (non imprimables);

5.2.8 Caractères spéciaux et délimiteurs

Le tableau suivant présente la liste des caractères spéciaux, s'ils sont délimiteurs, ainsi que (si ils existent) les délimiteurs correspondants et leur rôle. Lorsqu'un nombre est indiqué, il indique le code Ascii du caractère en question. Ces caractères sont modifiables par l'utilisateur (Cf §4.2.6).

C	Délim	Corresp	Rôle
(	oui	) |	délimite le début d'une liste, d'un doublet
)	oui	(|	délimite la fin d'une liste, d'un doublet
|	oui	( )	sépare CAR et CDR d'un doublet
[	oui	]	délimite le début d'une cellule
]	oui	[	délimite la fin d'une cellule
%	oui	delim.	marque le début d'un tableau de chiffres binaires
"	oui	"	fin et début d'une chaîne de caractère
;	oui	13	début de remarque sur une fin de ligne
{	oui	}	début de remarque
}	oui	{	fin de remarque
32	oui		espace: séparateur
13	oui		Retour chariot: séparateur
\	oui	\	force l'interneur

5.3 Les expressions primitives Help et leurs dérivées

Tout programme Help est formé d'une séquence de définitions réalisée via la forme `define`.

(**define** <ident> <quelconque>)	Forme syntaxique	x

Permet de définir `<ident>` en tant que valeur de `<quelconque>`.

(**define** <closdef> <corps>)	Forme syntaxique	x

Permet de définir une fermeture. `<corps>` est une séquence d'expressions quelconques et constitue le corps de la fermeture. `<closdef>` est une liste dont le CAR doit être un identificateur (nom de la fermeture ainsi définie) et le CDR une "liste" de paramètres formels (Cf. `lambda`)

5.3.1 Références à un littéral (constantes, symboles "quotés"...)

(**quote** <quelconque>)	Forme syntaxique	x
(' <quelconque>)	Forme syntaxique	x

Retourne l'objet dont la représentation externe est `<quelconque>`. Cette forme est utilisée pour référencer des littéraux dans du code Help. `(quote <x>)` peut être abrégé en `'<x>`.

```
(quote (+ 1 2))      ➠ (+ 1 2)
```

<constant>	Forme syntaxique	x

Retourne l'objet dont la représentation externe est *`constant`* dans le cas où cette représentation dénote un objet "constant" (invariant par évaluation). Cette forme est utilisée pour référencer des littéraux dans du code Help.

```
†       ➠ †
1       ➠ 1
```

5.3.2 Références aux variables

<ident>	Forme syntaxique	x

La valeur retournée est celle stockée à l'emplacement auquel l'identificateur est lié dans l'environnement courant. La référence à une variable non liée dans l'environnement courant retournera l'erreur `?:varundef`.

5.3.3 Application de fermeture

(<operateur> <operande1>…)	Forme syntaxique	x

L'appel à une fermeture est écrit simplement et systématiquement en mettant entre parenthèses les expressions dénotant la fermeture à appeler et les arguments qui doivent lui être passés.

L'<opérateur> est évalué, mais les <operande1>… ne seront évalués que si la fermeture “accede” à ces arguments.

```
(+ 1 2)                          ➠ 3
((0 (list - +)) 32 10)     ➠ 22
```

De nombreuses procédures sont prédéfinies dans l'environnement initial. De nouvelles procédures peuvent être définies en utilisant la forme syntaxique `(lambda…)`.

5.3.4 Création de fermeture

`(`**`lambda`** `<formels> <corps>)`	Forme syntaxique	`x`

Retourne une fermeture contenant (entre autres) l'environnement existant lors de l'évaluation de la forme (environnement capturé). L'application ultérieure de la fermeture ainsi créée à d'éventuels arguments entraînera l'évaluation du `<corps>` de la forme dans l'environnement capturé étendu avec les liaisons des paramètres `<formels>` à de nouveaux emplacements contenant les arguments. La valeur retournée sera la valeur retournée par la dernière évaluation. Si le `<corps>` est vide, l'erreur `?` sera retournée.

`<formels>` doit avoir une des formes suivantes:

- `(<ident1> … <identn>)`: La fermeture retournée sera alors une procédure à n arguments. Lors de l'application, chaque paramètre formel sera "lié" à chaque argument.
- `<ident>`: La fermeture retournée sera alors une procédure d'arité quelconque. Lors de l'application, le paramètre formel sera "lié" à la liste (fraîchement allouée) des arguments.
- `(<ident1> … | <identn>)`: La fermeture retournée sera alors une procédure à (n - 1) arguments au moins. Lors de l'application, le dernier paramètre sera "lié" à la liste des arguments en excès.

`<corps>` est une séquence éventuellement vide d'expressions.

```
(lambda(x) x)                    ➟ {Closure (x) x) in ()}
((lambda x x) 1 2 3)                    ➟ (1 2 3)
```

5.3.5 Conditionnelle

Notez que la forme syntaxique COND n'est conservée que pour des raisons d'efficacité (vu sa grande fréquence d'utilisation). Une version entièrement compilée de Help ne comprendra plus cette forme.

`(`**`cond`** `<clause1>…)`	Forme syntaxique	`x`

Chaque `<clause>` est une séquence de deux expressions quelconques, la dernière `<clause>` pouvant éventuellement se limiter à une séquence de une expression.

La première expression de la première `<clause>` est évaluée. Si elle délivre une valeur fausse (égale à `ƒ`), alors on passe à la `<clause>` suivante si il y en a, sinon on retourne `ƒ`. Si elle délivre une erreur, alors on retourne cette erreur, sinon on retourne la valeur de la seconde expression de la `<clause>` si il y an une, sinon on retourne la valeur de la première expression.

```
(cond (=? 2 (1+ 1)) "oui" "ca va pas")  ➠ oui
(cond (>? -3 0) -3
      (<? -3 -5) (- 0 -3)
      (+ 2 2))                          ➠ 4
```

5.3.6 Affectation

(**=!** <ident> <quelconque>)	Forme syntaxique	x

L'emplacement auquel <ident> est lié reçoit la valeur de la forme <quelconque> dans l'environnement courant. Cette forme syntaxique est une des caractéristiques "non fonctionnelle" de Help.

```
(=! x 2)            ➠ 2
(=! z (cons 1 z))   ➠ (1 1 1 1 1 1 1 …)
```

5.3.7 Suspension non mémoïzante

(**nomemo** <booléen> <quelconque>…)	Forme syntaxique	x

Le <booléen> est d'abord évalué. Sa veleur indique si la suppression de la mémoïzation doit être activée (vrai) ou désactivée (faux). Les formes <quelconque> vont ensuite être évaluées en séquence. Si le <booléen> n'a pas pour valeur ƒ (faux), toute suspension qui pourrait être créée par ces évaluations sera une suspension définitive… (sans mémoïzation, et si la mémoïzation n'est pas forcée localement par l'utilisation d'un (nomemo ƒ …)) Il faut noter la fragilité du caractère "définitif" de ces suspensions. En effet, si une suspension s1 créée par nomemo est suspendue classiquement (formant s2) , tout accès à la suspension s2 entraînera l'évaluation de s1 et la mémoïzation de la valeur dans s2 empêchant toute nouvelle évaluation de la forme de s1 via s2. Si le <booléen>a pour valeur ƒ, la mémoïzation est localement réactivée.

```
(define z
  (nomemo † (cons (print "CAR")
                  (print "CDR"))))
z                                   ➠ (? | ?)
                                        et imprime "CAR" et "CDR"
z                                   ➠ (? | ?)
                                        et imprime "CAR" et "CDR"
(define zz (list z z))
zz                                  ➠ ((? | ?) (? | ?))
                                        et imprime "CAR", "CDR" (2 fois)
zz                                  ➠ ((? | ?) (? | ?))
                                        mais n'imprime rien…
```

5.3.9 Capture de l'environnement

(**bindings**)	Forme syntaxique	x

Retourne l'environnement courant. Si celui ci est l'environnement global, le symbole () est retourné. Ne peut être une fermeture puisque le corps des fermetures sont évalués dans l'environnement capturé et non dans l'environnement courant auquel elles n'ont plus accès.

```
(bindings)                          ➠ ()
((lambda(x) (bindings)) 1)          ➠ {Env: «x=1» Then ()}
```

5.3.10 Définition de macros

(**defmacro** <quelconque>)	Forme syntaxique	x

Permet de définir une macro. La syntaxe est la même que celle de `define`. Sous l'interprète, l'appel à une macro est remplacé physiquement par la valeur retournée lors de la première exécution (macros déplaçantes). Si on utilise le compilateur, les macros sont expansées lors de la compilation. Il est donc indispensable de définir les macros avant que le code qui les utilisent ne soit compilé.

```
(defmacro (mapause m)
  `(begin (prin "Pause à ")
          (print ',m)
          (pause)))
(define (test n)(mapause test)(1+ n))
```

Après une exécution:

```
test                ➟ {Closure:((n) (begin (prin "Pause à ")
                                            (print 'test)
                                           (pause)) (1+ n)))…}
```

5.3.10 Définition de fonctions externes

(**defext** <fic><seg><nom><str> | <par>)	Forme syntaxique	x

Permet de définir une fonction externe. L'écriture du code externe (sous MPW) doit suivre les conventions exposées au §4.5.1.3.1.2, doit être lié à la librairie LibHelp.o …(Cf dossier exemples). Les paramètres sont:

- <fic>----------nom du fichier (ressources CODE);
- <seg>----------nom du segment contenant le code de la fermeture;
- <nom>----------nom à donner à la fermeture
- <str>----------vecteur de bits de "strictness" (Cf setstrict)
- <par>----------les paramètres

5.3.11 Construction de liaisons

(**let** <liaisons> <corps>)	Forme syntaxique	x

Il y a évaluation de <corps> dans l'environnement courant étendu avec les <liaisons>.
La portée des liaisons créées est limitée au <corps>.

<liaisons> est une cellule formée de listes qui peuvent avoir les formes:

- (<ident> <quelconque>): <ident> sera lié (paresseusement) à la valeur de <quelconque>.
- (<closdef> <corpsclos>): permet la définition d'une fonction locale (non récursive du fait de la portée limitée des liaisons) plus aisément qu'en utilisant une lambda expression. <corpsclos> est une séquence d'expressions quelconques, <closdef> est une liste dont le CAR doit être un identificateur (nom de la fermeture ainsi définie) et le CDR une "liste" de paramètres formels (Cf. lambda).

```
(let [(x 1.1)
      ((double n)(+ n n))]
     (double x))                    ➟ 2.200000000000000e+00
```

(**letrec** <liaisons> <corps>)	Forme syntaxique	x

La syntaxe est la même que pour let mais la portée des liaisons créées est égale à l'union de

<liaisons> et <corps>. Il est possible (comme pour let) de définir ainsi des fonctions locales qui ici peuvent être récursives. Du fait de la paresse, il n'y a pas (comme en Scheme par exemple) de problème de définition d'objets faisant immédiatement référence aux identificateurs définis.

```
(let [(r (fib 20))                ;erreur en Scheme
     ((fib n)                     ;(fib fonction non définie)
       (cond (<? n 2) 1
             (+ (fib (1- n))(fib (- n 2))))))]
```

```
r)                                  ➠ 10946
```

5.3.12 Evaluation en séquence

(**begin** <quelconque1>…)	Forme syntaxique	x

Evaluation en séquence des <quelconque1>…. Permet de regrouper les évaluations pour qu'elles se réalisent au même instant. Ceci est particulièrement utile pour les interactions avec l'utilisateur.

```
(let [(x (begin (prin "Entrez x")(read)))]
    (traiter x))                    ➠ …selon traiter
```

5.3.12 Déverminage

Le système de gestion d'erreurs Help, du fait de la paresse (qui permet de construire des objets de façon partielle), n'interrompt pas l'évaluation lorsqu'intervient une erreur. Suivant le mode courant d'évaluation, il peut y avoir:

- passage d'un objet du type erreur à la continuation (mode ƒ);
- passage d'un objet du type erreur à la continuation et affichage d'un message sur stder (mode ());
- appel à un "debugger" (mode †).

(**pause**)	Forme syntaxique	x

Permet de se retrouver dans une boucle READ-EVAL-PRINT locale avec un prompt de la forme {n}† où n représente un niveau d'imbrication des pauses. La sortie de la boucle se fait si il y a lecture du symbole † (du fait de la nature du "prompt", il suffit donc de taper ENTER). Toute action est possible durant la boucle (évaluations diverses…). La valeur retournée est le symbole ?.

(**warn** <ident> <quelconque1>…)	Forme syntaxique	x

Evalue les formes <quelconque1>… dans le mode spécifié par <ident> (Cf § précédent). Il faut noter que les évaluation retardées qui peuvent apparaître du fait de la paresse se feront effectivement dans le mode indiqué, même si alors on est sorti du warn (les suspensions capturent donc le mode courant de gestion d'erreurs).

Le "debugger" consiste (pour le moment) est une simple boucle READ-EVAL-PRINT (Cf pause) dont on peut sortir (pour reprendre l'exécution) en faisant lire le symbole † (vrai). L'expression et la valeur ayant probablement causé l'erreur sont affichées. Les différentes fermetures et formes syntaxiques bindings, envar, stack… permettent alors de consulter l'environnement, la pile, les valeurs de variables…La fermeture break permet de retourner au TopLevel. Il faut noter que la demande de valeurs (sous le debugger) peut déclencher des évaluations qui n'aurait pas du se produire et modifier ainsi le séquencement du programme.

```
(warn ƒ (+ 0 'a))                   ➠     ?:bad-type
                                          et aucun message d'erreur
```

(**step** <booléen> <quelconque1>…)	Forme syntaxique	x

Le <booléen> est d'abord évalué. Sa valeur de vérité va indiquer si l'on doit localement armer ou désarmer le mode pas à pas. Les formes <quelconque1>… .seront donc évaluées dans le mode indiqué. Il faut noter que les évaluation retardées qui peuvent apparaître du fait de la paresse seront effectivement exécutées en pas à pas, même si alors on est sorti du step (les suspensions capturent donc le mode courant d'évaluation). A chaque pas, la fonction step? (définie par

l'utilisateur) est appelée avec la forme qui va être évaluée et l'environnement. Si la valeur retournée est faux (ƒ), il n'y pas de pas à pas pour l'évaluation de cette forme. Sinon, la fermeture `stepin` est appelée avec la forme et l'environnement. La valeur qu'elle retourne est passée avec la valeur de l'expression "steppée" à la fermeture `stepout`.

Exemple:

```
(step † (+ 1 2))                    ➡    3 et par exemple la session suivante:
-> "(+ 1 2)"{1}†
-> "+"{1}†
<- "{Closure:{Code 680xx for +} Env:()}"{1}†
-> "1"{1}†
<- "1"{1}†
-> "2"{1}†
<- "2"{1}†
<- "3"{1}†
{ = 3 }
```

Le mode pas à pas permet une grande souplesse grâce à l'utilisation des fermetures `step?`, `stepin` et `stepout` (en plus de la forme `(step ƒ …)` qui permet de désactiver localement le pas à pas).

Ainsi, après la définition:

```
(define (step? f e)
  (not (number? f)))
```

Le pas à pas précédent devient:

```
(step † (+ 7 12))          ➡    19 et par exemple la session suivante:
1 -> "(+ 7 12)"{1}†
2 -> "+"{1}†
2 <- "{Closure:{Code 680xx for +} Env:()}"{1}†
1 <- "19"{1}†
{ = 19 }
```

5.4 Les fermetures standards Help

5.4.1 Booléens

Les deux constantes † et ƒ sont l'incarnation des booléens. Cependant, dans tous les tests effectués par Help, toute valeur différente de ƒ est considérée comme vrai (si ce n'est pas une erreur, auquel cas, le test échoue). En particulier, la liste vide est considérée comme vraie !

(**not** *quelconque*)	Fermeture:ProcN	1

Retourne † (vrai) si *quelconque* est égal à *ƒ*, sinon retourne ƒ.

```
(not †)                    ➡ ƒ
```

```
(not 3)                    ➠ ƒ
(not (list 1 2))     ➠ ƒ
(not ƒ)                    ➠ †
(not '())            ➠ ƒ
```

(**boolean?** *quelconque*)	Fermeture:ProcN	1

Retourne † (vrai) si *quelconque* est égal à † ou ƒ. Retourne ƒ sinon.

```
(boolean? †)         ➠ †
```

```
(boolean? 3)          ➟ ƒ
```

5.4.2 Prédicats d'équivalence

Les trois prédicats d'équivalence `eq?`, `=?` et `equal?` définissent trois relations d'équivalence (réflexive, symétrique transitive) sur les objets Help. La relation d'équivalence définie par `eq?` est incluse dans celle définie par `=?` elle même incluse dans cell définie par `equal?`.

(**eq?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne † (vrai) si les deux objets sont les mêmes (i.e existent à la même adresse en mémoire). Retourne ƒ autrement. Du fait de leur unicité, ce prédicat peut-être utilisé avec avantage sur les symboles.

```
(eq? 'a 'a)           ➟ †
(eq? 1 2)             ➟ ƒ
(eq? 1 1)             ➟ non défini
```

(**neq?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne ƒ (faux) si les deux objets ne sont les mêmes (i.e existent à la même adresse en mémoire). Retourne † autrement. Du fait de leur unicité, ce prédicat peut-être utilisé avec avantage sur les symboles.

```
(neq? 'a 'a)          ➟ ƒ
(neq? 1 2)            ➟ †
(neq? 1 1)            ➟ non défini
```

(**=?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne † (vrai) si les deux objets sont les mêmes (i.e existent à la même adresse en mémoire) ou si leur contenu est identique (au sens de `eq?` pour les structures de données référencant d'autres objets). Retourne ƒ autrement. Ce prédicat est avantageusement utilisable sur les nombres, les bit-arrays et les chaînes de caractères et dans certain cas sur les doublets, cellules....

```
(=? 'a 'a)            ➟ †
(=? 1 2)              ➟ ƒ
(=? 1 1)              ➟ †
(=? %01 %01)          ➟ †
```

(**<>?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne ƒ (faux) si les deux objets sont les mêmes (i.e existent à la même adresse en mémoire) ou si leur contenu est identique (au sens de `eq?` pour les structures de données référencant d'autres objets). Retourne † autrement. Ce prédicat est avantageusement utilisable sur les nombres, les bit-arrays et les chaînes de caractères et dans certain cas sur les doublets, cellules....

```
(=? 'a 'a)                          ➠ †
(=? 1 2)                            ➠ ƒ
(=? 1 1)                            ➠ †
(=? %01 %01)                        ➠ †
(=? '(a | b) '(a | b))              ➠ †
```

(**equal?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne † (vrai) si les deux objets sont équivalents (au sens de =?) ou, sinon, si leur contenu est identique (au sens de equal?). Retourne ƒ autrement.

```
(equal? 'a 'a)                    ➠ †
(equal? 1 1)              ➠ †
(equal? '(a b) '(a b))    ➠ †
```

(**nequal?** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne *f* (faux) si les deux objets sont équivalents (au sens de =?) ou, sinon, si leur contenu est identique (au sens de equal?). Retourne † autrement.

```
(equal? 'a 'a)                    ➠ †
(equal? 1 1)              ➠ †
(equal? '(a b) '(a b))    ➠ †
```

5.4.3 Listes et doublets

Un doublet (ou paire pointée) est une structure de données hétérogène à deux champs appelés (pour des raisons historiques que je ne rappellerai pas ...) CAR et CDR. Ces structures sont, à l'heure actuelle, modifiables. L'accès aux champs d'un doublet se fait par les sélecteurs numériques.

L'utilisation principale des doublets est la représentation de listes. Une liste est définie comme étant soit la liste vide () soit un doublet dont le champ CDR est une liste.
La liste vide () est un objet à part, implémenté au moyen d'un symbole de constante. Elle ne contient pas d'élément et sa longueur est 0.

(**list?** *quelconque*)	Fermeture:ProcN	1

Retourne *f* (faux) si *quelconque* n'est pas une liste (i.e. un doublet ou le symbole ()). Retourne † sinon.

```
(list? '())          ➠ †
(list? '(1 2 3))     ➠ †
(list? 3)            ➠ f
```

(**cons?** *quelconque*)	Fermeture:ProcN	1

Retourne † (vrai) si *quelconque* est un doublet. Retourne *f* sinon.

```
(cons? '())          ➠ f
(cons? '(1 2 3))     ➠ †
(cons? 3)            ➠ f
```

(**atom?** *quelconque*)	Fermeture:ProcN	1

Retourne *f* (faux) si *quelconque* est un doublet. Retourne † sinon.

```
(atom? '())          ➠ t
(atom? '(1 2 3))     ➠ f
(atom? 3)            ➠ t
```

(**cons** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Retourne un doublet fraîchement alloué dont le CAR contient *quelconque1* et le CDR *quelconque2*. Le doublet est différent [5] de tout autre objet déjà existant.

```
(cons 1 2)          ➠ (1 | 2)
(cons 'a '())       ➠ (a)
(cons '(a b) 'c)    ➠ ((a b) | c)
```

(**car=!** *doublet quelconque*)	Fermeture:ProcN	2

Permet d'écrire l'objet *quelconque* dans le champ CAR d'un *doublet*. Du fait de la paresse, des effets inattendus peuvent être observés. L'utilisation de cette fermeture est donc fortement déconseillée. Elle devrait disparaître dans les versions futures.

```
(car=! '(1 2) 3)    ➠ (3 2)
(car=!  %1010 3)    ➠ ?:bad-type
```

(**cdr=!** *doublet quelconque*)	Fermeture:ProcN	2

Permet d'écrire l'objet *quelconque* dans le champ CDR d'un *doublet*. Du fait de la paresse, des effets inattendus peuvent être observés. L'utilisation de cette fermeture est donc fortement déconseillée. Elle devrait disparaître dans les versions futures.

```
(cdr=! '(1 2) 3)    ➠ (1 | 3)
(cdr=!  %1010 3)    ➠ ?:bad-type
```

(**null?** *quelconque*)	Fermeture:ProcN	1

Retourne *ƒ* (faux) si *quelconque* n'est pas la liste vide. Retourne † sinon.

```
(null? '())         ➠ †
(null? '(1 2 3))    ➠ ƒ
```

(**list** *quelconque …*)	Fermeture:NProc	0

Retourne la liste de ses arguments.

```
(list 'ts (+ 24 3) '65)    ➠ (ts 27 65)
(list)                     ➠ ()
```

(**length** *list*)	Fermeture:ProcN	1

Retourne la longueur de *list*. La longueur d'une liste est définie récursivement par:

- ▼ la longueur de tout objet qui n'est pas un doublet est 0;
- ▼ la longueur d'un doublet est 1 plus la longueur du CDR de ce doublet.

```
(length '())                    ➠ 0
```

[5] dans le sens de eq?, il n'y a donc pas de "hash-consing".

```
(length '(1 2 3))                ➠ 3
(length '(mcl (1 2 3) (a b)))    ➠ 3
(length 1)                       ➠ ?:bad-type
```

(**append** *list1 quelconque*)	Fermeture:ProcN	2

Retourne une liste constituée des éléments de la première liste *list1* suivi des éléments de la liste *quelconque* .Le second argument peut ne pas être une liste.

```
(append '(1 2 3) '(4 5 6)) ➟ (1 2 3 4 5 6)
(append '() '(a b))        ➟ (a b)
(append '(a b) '(c | d))   ➟ (a b c | d)
```

(∞ *quelconque*)	Fermeture:ProcN	1

Retourne une liste infinie constituée uniquement de la valeur de *quelconque*.

```
(∞ 1)         ➟ (1 1 1 1 1 1…)
(∞ (+ 2 3))   ➟ (5 5 5 5 5 5…)
```

(… *nombre*)	Fermeture:ProcN	1

Retourne une liste infinie constituée des entiers à partir de *nombre*.

```
(… 1)         ➟ (1 2 3 4 5 6…)
(… (- -2 1)) ➟ (-3 -2 -1 0 1 2…)
```

5.4.4 Symboles

Les symboles sont des objets dont l'utilisation principale repose sur le fait que deux symboles sont identiques (dans le sens de eq? ou =?) si ils ont le même nom. Cette propriété étant particulièrement utile pour les identificateurs de variables ou de constantes, Help utilise les symboles pour représenter ces identificateurs ainsi que les erreurs.

(**symbol?** *quelconque*)	Fermeture:ProcN	1

Retourne † (vrai) si *quelconque* est un symbole, sinon retourne l'objet *quelconque*.

(**intern** *symbole quelconque*)	Fermeture:ProcN	2

Retourne un symbole de nom *quelconque* (qui doit être une chaîne ou un symbole) et de père (dans la hiérarchie des symboles) *symbole*.

```
(intern 'père "fils")      ➟ père:fils
(intern 'père 'fils)       ➟ père:fils
```

5.4.5 Nombres

Suivant leur position, les nombres peuvent être interprétés comme fonctions (sélecteurs numériques) ou comme données numériques. Ce double rôle peut être considéré comme une faiblesse sémantique mais représente un grand confort du point de vue pragmatique.

L'utilisation d'un nombre entier de taille fixe en tant que sélecteur numérique sur une liste ou cellule est détaillée dans le §5.3.

(**+** *nombre nombre*)	Fermeture:ProcN	2

Retourne la somme de ses arguments. Si l'un des deux arguments est flottant, le résultat sera flottant.

```
(+ 10 10)          ➟ 20
(+ 1 2.312)        ➟ 3.3120000000000000e+00
```

```
(+ 9999999999 1)     ➟ 10000000000
(+ 1 'a)             ➟ ?:bad-type
```

(**-** *nombre nombre*)	Fermeture:ProcN	2

Retourne la différence de ses arguments. Si l'un des deux arguments est flottant, le résultat sera flottant.

```
(- 10 10)            ➟ 0
(- 2.312 1)          ➟ 1.312000000000000e+00
(- 9999999999 1)     ➟ 9999999998
(- 1 'a)             ➟ ?:bad-type
```

(**1+** *nombre*)	Fermeture:ProcN	1

Retourne le résultat de l'incrémentation de son argument.

```
(1+ 10)              ➟ 11
(1+ %)         ➟ %00000000000000000000000000000001
(1+ 'a)              ➟ ?:bad-type
```

(**1-** *nombre*)	Fermeture:ProcN	1

Retourne le résultat de la décrémentation de son argument.

```
(1- 10)              ➟ 9
(1- 'a)              ➟ ?:bad-type
```

(***** *nombre1 nombre2*)	Fermeture:ProcN	2

Retourne le produit de ses arguments. Si l'un des arguments est flottant, il y aura conversion en flottant de l'autre argument et calcul du produit en flottant.

```
(* 10 23)      ➟ 230
(* 'a 2)       ➟ ?:bad-type
(* 1e10 2)     ➟ 2.000000000000000e10
```

(**/** *smallnum1 smallnum2*)	Fermeture:ProcN	2

Retourne le quotient (entier si les deux arguments sont entiers) de ses arguments. Si l'un des arguments est flottant, il y aura conversion en flottant de l'autre argument et calcul du produit en flottant.

```
(/ 23 10)            ➟ 2
(/ 'a 2)             ➟ ?:bad-type
(/ 23 10.0)          ➟ 2.300000000000000e0
```

(**modulo** *fix1 fix2*)	Fermeture:ProcN	2

Retourne le reste du quotient de *fix1* par *fix2*.

```
(modulo 23 10)       ➠ 3
(modulo 'a 2) ➠ ?:bad-type
```

(**<?** *nombre1 nombre2*)	Fermeture:ProcN	2

Retourne ƒ si *nombre1* n'est pas strictement inférieur à *nombre2*. Retourne *nombre1* sinon.

```
(<? 23 10)           ➠ f
(<? 'a 2)            ➠ ?:bad-type
```

```
(<? 12 20)          ➠ 12
```

(**>?** *nombre1 nombre2*)	Fermeture:ProcN	2

Retourne ƒ si *nombre1* n'est pas strictement supérieur à *nombre2*. Retourne *nombre1* sinon.

```
(>? 23 10.0)                    ➠ 23
(>? 'a 2)                ➠ ?:bad-type
(>? 20 12345678987654321)  ➠ ƒ
```

(**zero?** *nombre*)	Fermeture:ProcN	1

Retourne ƒ si *nombre* n'est pas égal à zéro. Retourne *nombre* sinon.

```
(zero? 0)           ➠ 0
(zero? 'a)          ➠ ?:bad-type
```

(**float** *nombre*)	Fermeture:ProcN	1

Retourne le résultat de la conversion de *nombre* en flottant.

```
(float 0)           ➠ 0.00000000000e+0
```

(**cos** *nombre*)	Fermeture:ProcN	1

Retourne le cosinus (en flottant) de *nombre* exprimé en radians.

(**sin** *nombre*)	Fermeture:ProcN	1

Retourne le sinus (en flottant) de *nombre* exprimé en radians.

(**tan** *nombre*)	Fermeture:ProcN	1

Retourne la tangente (en flottant) de *nombre* exprimé en radians.

(**acos** *nombre*)	Fermeture:ProcN	1

Retourne l'arc-cosinus (en flottant) de *nombre* .

(**asin** *nombre*)	Fermeture:ProcN	1

Retourne le l'arc-sinus (en flottant) de *nombre*.

(**atan** *nombre*)	Fermeture:ProcN	1

Retourne l'arc-tangente (en flottant) de *nombre*.

(**cosh** *nombre*)	Fermeture:ProcN	1

Retourne le cosinus hyperbolique (en flottant) de *nombre*.

(**sinh** *nombre*)	Fermeture:ProcN	1

Retourne le sinus hyperbolique (en flottant) de *nombre*.

(**tanh** *nombre*)	Fermeture:ProcN	1

Retourne la tangente hyperbolique (en flottant) de *nombre*.

(**atanh** *nombre*)	Fermeture:ProcN	1

Retourne l'arc-tangente hyperbolique (en flottant) de *nombre*.

(**log** *nombre*)	Fermeture:ProcN	1

Retourne le logarithme népérien (en flottant) de *nombre*.

(**exp** *nombre*)	Fermeture:ProcN	1

Retourne l'exponentielle (en flottant) de *nombre*.

(**sqrt** *nombre*)	Fermeture:ProcN	1

Retourne la racine carrée (en flottant) de *nombre*.

(**ibase** *fixpos*)	Fermeture:ProcN	1

Modifie la base de lecture courante du lecteur. Le *fixpos* doit être compris entre 2 et 36. Retourne la valeur de son argument. Voir §5.2.1.

```
(ibase 10)          ➟ 10
```

5.4.6 Fermetures

Les fermetures sont le produit cartésien d'un "code" qui attend une séquence d'expression et d'un environnement. A des fins d'optimisation, chaque fermeture est accompagné d'un tableau de 16 bits indiquant si la fermeture est stricte par rapport aux 15 premiers arguments (15 premiers bits) et aux suivants (dernier bit). Ce tableau est automatiquement généré par le compilateur (dans un certaine mesure seulement) mais pas par l'interprète.

(**apply** *applicable liste*)	Fermeture:ProcN	2

Applique la fermeture ou le sélecteur numérique *applicable* à la liste d'arguments *liste*.

```
(apply 1 '((a)))          ➟ a
(apply + '(1 2))          ➟ 3
```

(**getcode** *fermeture*)	Fermeture:ProcN	1

Retourne le code (interprété ou compilé) de la *fermeture*.

```
(getcode 1+)              ➟ {Code 680xx for 1+}
(getcode (lambda(x)x))    ➟ ((x) x)
```

(**getenv** *fermeture*)	Fermeture:ProcN	1

Retourne l'environnement capturé par la *fermeture*.

```
(getenv 1+)                ➠ ()
(getenv
  (let [(x 2)]
     (lambda(x)x)))        ➠ {Env: «x=2» Then ()}
```

(**getstrict** *fermeture*)	Fermeture:ProcN	1

Retourne le bit-array associée à la *fermeture* en relation avec le caractère strict de la fermeture sur le nième argument.

```
(getstrict cons)          ➟ %00000000000000000000000000000000
(getstrict +)             ➟ %11000000000000000000000000000000
```

(**setstrict** *fermeture bitarray*)	Fermeture:ProcN	2

Permet de fixer le bit-array associée à la *fermeture* en relation avec le caractère strict de la fermeture sur le nième argument. Permet d'améliorer le temps de calcul et la place mémoire consommée si la fonction est stricte par rapport à certains de ses arguments.

5.4.7 Macros

(**expand** *liste*)	Fermeture:ProcN	1

La *liste* doit représenter un appel à une macro "quoté". Retourne le résultat de l'expansion de la macro.

```
(expand '(quasiquote (a (unquote b))))  ➟ (list 'a b)
```

5.4.8 Cellules

Les cellules sont des structures hétérogènes indexées par des entiers. Ces structures (tout comme les doublets) sont, à l'heure actuelle, modifiables après leur création. Du fait de la paresse, il est fortement déconseillé d'utiliser ces mécanisme d'écriture.

Lors de l'accès par les sélecteurs numériques, le premier élément d'une cellule est indexé par zéro et le dernier par la longueur de la cellule moins un. La longueur d'une cellule peut être obtenue en appelant la fermeture `blength` et en soustrayant 1 au résultat obtenu.

(**cell?** *quelconque*)	Fermeture:ProcN	1

Retourne *f* (faux) si *quelconque* n'est pas une cellule, sinon retourne la cellule elle-même.

```
(cell? 23)          ➟ f
(cell? [1 2 3])     ➟ [1 2 3]
```

(**cell** *quelconque …*)	Fermeture:NProc	0

Retourne une cellule dont le contenu est *quelconque* ….

```
(cell 0 1 2)              ➟ [0 1 2]
```

```
(cell 'd 'e 1 '(a) [1])    ➠ [d e 1 (a) [1]]
```

(**makecell** *fixpos*)	Fermeture:ProcN	1

Retourne une cellule de taille *fixpos* et dont les champs sont tous initialisés à ?.

```
(makecell 2)               ➠ [? ?]
```

(**cell=!** *cellule fixpos quelconque*)	Fermeture:ProcN	3

Ecrit quelconque dans cellule à la position fixpos. Retourne la cellule ainsi modifiée. Cette instruction ayant des effets de bords importants est d'un usage difficile et déconseillé du fait de la paresse.

```
(cell=! '[a b c] 0 0)      ➟ [0 b c]
(cell=! [0 1 2] 3 1)       ➟ ?:indx-out
```

5.4.9 Environnements

La forme syntaxique `bindings` permet de référencer l'environnement courant. Une série de fermetures permet de manipuler cet environnement, ou d'en créer de nouveaux... Les sélecteurs numériques permettent de manipuler les environnements (non vides). A l'index zéro se trouve l'environnement inférieur, ensuite on trouve les valeurs puis les variables.

(**environment?** *quelconque*)	Fermeture:`ProcN`	1

Retourne *f* (faux) si *quelconque* n'est pas un environnement, sinon retourne l'environnement lui-même. La forme syntaxique `bindings` retournant `()` si l'environnement courant est l'environnement global, la liste vide est considérée comme environnement.

```
(environment? 23)                          ➟ f
(environment? '())                         ➟ ()
(environment (let [(x 2)] (bindings)))     ➟ {Env: «x=2» Then ()}
```

(**binding=?** *ident environ*)	Fermeture:`ProcN`	2

Retourne la valeur de *ident* dans *environ* . Si la variable n'est pas définie dans l'environnement, il y retour de l'erreur `?:varundef`.

```
(binding=? '1+ '())                        ➟ {Code6800x for 1+}
(binding=? 'x (let [(x 2)] (bindings)))    ➟ 2
(binding=? 'a '())                         ➟ ?:varundef
```

(**binding=!** *ident environ quelconque*)	Fermeture:`ProcN`	3

Modifie la valeur de *ident* dans *environ*. Retourne la valeur de *environ*.

```
(binding=! '1+ '() 2)                      ➟ ()
(binding=! 'a '() 2)                       ➟ ()
```

(**makeenv** *ident1 …*)	Fermeture:`NProc`	0

Retourne un environnement dans lequel les *ident* sont liés à `?`. Cet environnement est automatiquement étendu avec l'environnement courant.

```
(makeenv 'a 'b)      ➟ {Env: «a=?» «b=?» Then ()}
(makeenv 1)          ➟ ?:bad-type
```

(**envar** *environ*)	Fermeture:ProcN	1

Retourne une cellule contenant tous les identificateurs de l'environnement *environ*. Utile essentiellement durant le déverminage (sous debugger) pour examiner l'environnement sans forcer le calcul des valeurs.

```
(envar (makeenv 'a 'b))     ➟ [a b]
```

5.4.10 Tableau de bits

Les tableaux de bits permettent une économie de mémoire importante dans la représentation des booléens. Ils permettent également une représentation efficace des ensembles et une manipulation aisée (intersection, union...). Les opérations sur les tableaux sont destructives (effets de bords) pour des raisons d'efficacité, une utilisation fonctionnelle peut aisément être réalisée au moyen de la fermeture `bcopy`.

La lecture d'un tableau de bits se fait au moyen des sélecteurs numériques

(**bitarray?** *quelconque*)	Fermeture:ProcN	1

Retourne ƒ (faux) si *quelconque* n'est pas un tableau de bits, sinon retourne le tableau lui-même.

```
(bitarray? 23)      ➟ f
(bitarray? %) ➟ %
```

(**makebitarray** *posfix*)	Fermeture:ProcN	1

Retourne un tableau de bits de longueur au moins égale à *posfix* où tous les bits sont initialisés à 0.

```
(makebitarray 0)     ➟ %
(makebitarray 10)    ➟ %00000000000000000000000000000000
```

(**bitand!** *bitarray1 bitarray2*)	Fermeture:ProcN	2

Effectue le ET logique entre *bitarray1* et *bitarray2*. Le résultat est stocké dans le second tableau. Si les tableaux sont de tailles différentes, l'opération est limitée au plus petit. Cette opération ayant des effets de bords, il est conseillé de faire une copie du second tableau pour appliquer cette fermeture.

```
(bitand! %0011 %0101)      ➟ %00010000000000000000000000000000
```

(**bitor!** *bitarray1 bitarray2*)	Fermeture:ProcN	2

Effectue le OU logique entre *bitarray1* et *bitarray2*. Le résultat est stocké dans le second tableau. Si les tableaux sont de tailles différentes, l'opération est limitée au plus petit. Cette opération ayant des effets de bords, il est conseillé de faire une copie du second tableau pour appliquer cette fermeture.

```
(bitor! %0011 %0101)       ➟ %01110000000000000000000000000000
```

(**bitxor!** *bitarray1 bitarray2*)	Fermeture:ProcN	2

Effectue le ou exclusif logique entre *bitarray1* et *bitarray2*. Le résultat est stocké dans le second tableau. Si les tableaux sont de tailles différentes, l'opération est limitée au plus petit. Cette opération ayant des effets de bords, il est conseillé de faire une copie du second tableau pour appliquer cette fermeture.

```
(bitor! %0011 %0101)        ➠ %01100000000000000000000000000000
```

(**bitnot!** *bitarray*)	Fermeture:ProcN	1

Effectue la négation logique de *bitarray*. Le résultat est stocké dans *bitarray*. Cette opération ayant des effets de bords, il est conseillé de faire une copie du tableau pour appliquer cette fermeture.

```
(bitnot! %01) ➠ %10111111111111111111111111111111
```

(**bitcount** *bitarray*)	Fermeture:ProcN	1

Retourne le nombre de bits à 1 dans *bitarray*. Temps d'exécution proportionnel au nombre de 1.

```
(bitcount %0101)     ➠ 2
```

(**bitfind** *bitarray*)	Fermeture:ProcN	1

Retourne la position du premier bit à 1 dans *bitarray* si il y en a un. Retourne *f* sinon.

```
(bitfind %0001)      ➠ 3
(bitfind %)          ➠ f
```

(**bitset!** *bitarray fixpos*)	Fermeture:ProcN	2

Positionne (met à 1) le bit à la position *fixpos* du *bitarray* . Retourne le tableau modifié.Cette opération ayant des effets de bords, il est conseillé de faire une copie du tableau pour appliquer cette fermeture.

```
(bitset! %0 0)       ➠ %10000000000000000000000000000000
```

(**bitclr!** *bitarray fixpos*)	Fermeture:ProcN	2

Efface (met à 0) le bit à la position *fixpos* du *bitarray* . Retourne le tableau modifié.Cette opération ayant des effets de bords, il est conseillé de faire une copie du tableau pour appliquer cette fermeture.

```
(bitclr! %1 0)       ➠ %00000000000000000000000000000000
```

(**bitchg!** *bitarray fixpos*)	Fermeture:ProcN	2

Change la valeur du bit à la position *fixpos* du *bitarray* . Retourne le tableau modifié.Cette opération ayant des effets de bords, il est conseillé de faire une copie du tableau pour appliquer cette fermeture.

```
(bitchg! %0 0)       ➠ %10000000000000000000000000000000
```

(**zero?** *bitarray*)	Fermeture:ProcN	1

Retournera *f* si un des bits de *bitarray* est positionné. Retourne *bitarray* sinon.

```
(zero? %00010)       ➠ f
(zero? %000) ➠ %00000000000000000000000000000000
```

5.4.11 Entrées-Sorties

Les entrées sorties sont faites via des unités d'entrée/sortie. Ces unités peuvent représenter un fichier ou une fenêtre de l'éditeur. Comme en C, les variables `stdi`, `stdo` et `stder` dénotent (normalement) les unités (resp.) d'entrée, de sortie et d'erreur. Au démarrage, `stdi` et

stdo sont associées à la fenêtre courante et stder à la fenêtre “Transcript”. Il est possible de modifier le contenu de ces variables afin de détourner les flots d’entrée/sortie vers ou en provenance d’un fichier…

(**read**)	Fermeture:ProcN	0

Lit une expression Help sur le flot d’entrée courant (contenu dans la variable stdi). Retourne la représentation interne de l’expression lue.

(**print** *quelconque*)	Fermeture:ProcN	1

Imprime la représentation externe de *quelconque* sur le flot de sortie courant (contenu dans la variable stdo) et passe à la ligne suivante (émission d’un retour chariot). Il faut noter que l’imprimeur Help va forcer toutes les suspensions nécessaires afin de pouvoir afficher l’objet dans sa complétude. Si l’on désire afficher l’objet dans son état courant, utiliser la fonction printdebug. Retourne ?.

(**prin** *quelconque*)	Fermeture:ProcN	1

Imprime la représentation externe de *quelconque* sur le flot de sortie courant (contenu dans la variable stdo). Il faut noter que l’imprimeur Help va forcer toutes les suspensions nécessaires afin de pouvoir afficher l’objet dans sa complétude. Si l’on désire afficher l’objet dans son état courant, utiliser la fonction printdebug. Retourne ?.

(**prinlength** *fixpos*)	Fermeture:ProcN	1

Permet de fixer la longueur maximale d’impression en nombre d’objets imprimés. Particulièrement utile lors de l’impression d’objets infinis. Retourne la valeur de son argument.

(**prindepth** *fixpos*)	Fermeture:ProcN	1

Permet de fixer la profondeur maximale d’impression en nombre d’objets imprimés. Particulièrement utile lors de l’impression d’objets infinis. Retourne la valeur de son argument.

(**openi** *chaîne*)	Fermeture:ProcN	1

Ouvre en lecture seulement le fichier dont le chemin d'accès est indiqué dans *chaîne* . Le chemin d'accès est spécifié en séparant les dossiers par des “:”. Le dossier d’où a été lancé l’évaluateur est le dossier par défaut. Retourne une unité d’entrée associée à ce fichier.

```
(openi “bob”) ➠ «IO-Unit»
```

(**openo** *chaine*)	Fermeture:ProcN	1

Ouvre en lecture-écriture le fichier dont le chemin d'accès est indiqué dans *chaîne* . Le chemin d'accès est spécifié en séparant les dossiers par des “:”. Le dossier d’où a été lancé l’évaluateur

est le dossier par défaut. Retourne une unité d'entrée associée à ce fichier.

```
(openo "deux-pierre")      ➠ «IO-Unit»
```

(**close** *iounit*)	Fermeture:ProcN	1

Ferme le fichier associé à *iounit* . Retourne ?. Toute tentative de lecture ou d'écriture sur une unité d'entrée/sortie fermée se soldera par une erreur ?:bad-type.

(**prinio** *quelconque iounit*)	Fermeture:ProcN	2

Imprime la représentation externe de *quelconque* sur l'unité de sortie *iounit*. Il faut noter que l'imprimeur Help va forcer toutes les suspensions nécessaires afin de pouvoir afficher l'objet dans sa complétude. Si l'on désire afficher l'objet dans son état courant, utiliser la fonction printdebug. Retourne ?.

(**readio** *iounit*)	Fermeture:ProcN	1

Lit une expression Help sur le flot d'entrée *iounit*. Retourne la représentation interne de l'expression lue. Retourne l'erreur ?:eof-error lors que la fin d'un fichier est atteinte.

(**flushio** *iounit*)	Fermeture:ProcN	1

Vide les "buffers" associés à *iounit*. Permet en particulier de mettre à jour l'affichage lorsqu'une unité d'I/O est associée à une fenêtre.

(**load** *chaine*)	Fermeture:ProcN	1

Ouvre le fichier dont le chemin d'accès est indiqué dans *chaîne* , évalue son contenu et ferme le fichier.

5.4.12 Erreurs et gestion d'erreurs

(**printdebug** *quelconque*)	Fermeture:ProcN	1

Affiche sur l'unité d'entrée/sortie référencée par stder la forme externe de *quelconque* sans forcer les suspensions que peut contenir l'objet suivie d'un retour chariot. Une suspension est affichée avec le code suspendu et l'environnement capturé. Retourne ?.

```
(printdebug (cons a b))    affichera ({Susp: a in ()} | {Susp: b in ()})
```

(**error** *erreur quelconque*)	Fermeture:ProcN	2

Emet l'erreur *erreur* avec le message *quelconque*. Suivant le mode courant de traitement de l'erreur, il peut y avoir impression, appel du "debugger" ou juste retour d'une valeur...

(**error?** *quelconque*)	Fermeture:ProcN	1

Retourne † si la valeur de *quelconque* est du type "erreur" (Cf la fermeture type). Retourne *ƒ* sinon.

```
(error? 1)          ➟    ƒ
(error? (1+ 'a))    ➟    †
```

(**explain** *erreur*)	Fermeture:ProcN	1

Retourne le message d'erreur associé à l'*erreur*.

```
(explain '?:varundef)      ➠ "Variable non définie"
```

5.4.13 Contrôle

Les fermetures de contrôle sont peu nombreuses en Help. Comme nous le verrons au §6.2.2 l'introduction d'échappement ou de continuations (à la Scheme) n'apporte que peu de choses dans un contexte d'évaluation paresseuse non parallèle. Cependant, la fermeture `force` permet de redonner aux échappements toutes leurs fonctionnalités, la primitive `call/ep` permettant de réaliser de tels échappements (efficace mais d'utilisation très limitée); la fermeture `call/cc` permettant de capturer la continuation courante (à la Scheme).

(**call/ep** *fermeture*)	Fermeture:ProcN	1

Passe à la *fermeture* (qui doit pouvoir accepter un seul argument) une fermeture de type ProcN, d'arité 1 qui permettra de s'échapper en retournant la valeur passée en argument (comme call/cc en Scheme). Cependant, cette "continuation chronologique" a une durée de vie limitée[6] (écrasement des piles) et ne permet pas de faire du "backtracking" non chronologique.

```
(call/ep (lambda(k)(k 1) 2))      ➠ 1
```

(**call/cc** *fermeture*)	Fermeture:ProcN	1

Passe à la *fermeture* (qui doit pouvoir accepter un seul argument) une fermeture de type ProcN, d'arité 1 qui passera la valeur argument à la continuation capturée lors du `call/cc` (comme call/cc en Scheme). Cette continuation a une durée de vie illimitée mais peut occuper une importante quantité de place mémoire et sa capture comme son appel peuvent nécessiter un temps machine important.

```
(call/cc (lambda(k)(k 1) 2))      ➠ 1
```

(**force** *quelconque*)	Fermeture:ProcN	1

D'un point du vue fonctionnel, c'est la fonction identité. Cependant, elle accède récursivement à toutes les suspensions que pourraient contenir ou référencer *quelconque*. Elle permet essentiellement de simuler la transformation des suspensions Help en "futures" d'une machine parallèle et de redonner à certains traits sales de Help (affectation, échappements…) toutes leurs fonctionnalités.

```
(=! x 2)                          ➠ 2
(=! x (force (cons 1 x)))         ➠ (1 | 2)     ;et non (1 1 1 1 1 1 1 1…)
(force
  (call/cc
    (lambda(k)
      (cons 1 (k 'Sortir)))))     ➠ Sortir      ;et non (1 | <teratos>)
```

(**if** *quelconque1 quelconque2 …*)	Fermeture:NProc	0

[6] c'est le seul objet Help ayant une durée de vie limitée.

Comme l'indique le §3.4.2, la conditionnelle peut être implémentée sous forme de fermeture en Help. La fermeture Help représente l'incarnation sous forme de fermeture de la forme syntaxique `cond` (qui est plus efficace et conservée à ce seul titre dans l'interprète).

```
(if (eq? 'a 'b) 1
   (=? 1 2)    2
```

```
3)                          ➠ 3
```

(**eval** *quelconque environ*)	Fermeture:ProcN	2

Dans la version “interprétée” de Help, cette fermeture réalise l’évaluation de la forme *quelconque* dans l’environnement *environ*. Dans la version “compilée”, elle devrait compiler la forme *quelconque* dans l’environnement *environ* et lancer l’exécution du code compilé généré. A noter que si *quelconque* est déjà une forme compilée (objet de type CODE) il y a exécution immédiate du code dans l’environnement fourni. Ceci permet une utilisation de EVAL rapide si il y a eu précompilation.

```
(eval 1 '())                                    ➠ 1
(eval (cons x x) ((lambda(x) (bindings)) 'a))   ➠ (a | a)
```

(**or** *quelconque1…*)	Fermeture:NProc	0

Evalue les formes *quelconque1…* en séquence jusqu'à ce que une des formes retourne une valeur vraie. Dans ce cas, retourne cette valeur, sinon retourne *f*.

```
(or (null? 'a) (null? '()))              ➠ t
(or (number? 'a) (number? 1.1))   ➠ 1.100000000000000e00
```

(**and** *quelconque1…*)	Fermeture:NProc	0

Evalue les formes *quelconque1…* en séquence jusqu'à ce que une des formes retourne une valeur fausse. Dans ce cas, retourne *f*, sinon retourne la dernière valeur retournée.

```
(and (null? 'a) (null? '()))      ➠ f
(and (number? 10) (number? 1.1)) ➠ 1.100000000000000e00
```

5.4.14 Système

Nous regroupons ici toutes les fermetures ayant un rapport avec le système de gestion de mémoire ou avec le système d’exploitation et le matériel.

(**type** *quelconque*)	Fermeture:ProcN	1

Retourne le type (contenu dans le “tag” du bloc référencé) de *quelconque* sous forme numérique (*fix*). La table suivante, ou plus simplement l’application de la fermeture type à un objet constant permettent les comparaisons.

entier taille fixe	1	nombre flottant	2
tableau de bits	3	chaîne	4
unité d’entrées/sorties	5	identificateur de variable	6
identificateur de constante	7	erreurs	8
macros	9	mot clé	10

code	11	fermeture	12
doublet	13	cellule	14
indirection[7]	15	environnement	16
environnement court	17	nombre entier en précision illimitée	19
forme suspendue mémoïzable[8]	20	forme suspendue[9]	21

[7]type invisible pour l'utilisateur.

[8]type invisible pour l'utilisateur.

[9]type invisible pour l'utilisateur.

```
(type 'a)                          ➠ 6
(type (+ 9999999999 1))            ➠ 19
```

(**coerce** *quelconque fixpos*)	Fermeture:ProcN	2

Change le type de *quelconque* en *fixpos* . Utiliser la fonction type pour connaître les types possibles. Cette fonction peut tromper le Glaneur de Cellules en le forçant à libérer des blocs référençables ou à libérer des blocs qui n'existent pas ceci pouvant entraîner des erreurs terminales (Bus Error, Adress Error...). Les conversions de type sont possibles sans danger entre les différents éléments de chacun des groupes de types suivants:

- ▼ 1 2 3 4 19---------- nombres et bit-arrays
- ▼ 6 7 8 9 10---------- identificateurs...
- ▼ 16 17 14---------- environnements et cellules

(**runtime**)	Fermeture:ProcN	0

Retourne le temps écoulé en 1/60 secondes depuis le démarrage du système.

(**chrono** *quelconque*)	Fermeture:ProcN	1

Retourne une cellule de trois éléments. Le premier élément est la valeur de *quelconque*. Le second représente le temps pris (en secondes, et à 1/60 de secondes près) pour évaluer la forme *quelconque* (attention, une impression peut être nécessaire pour forcer un objet dans sa complétude) et le dernier représente le temps passé en Glanage de Cellules. Noter que chrono effectue un Glanage de Cellules compacifiant AVANT d'évaluer la forme *quelconque* afin d'avoir une plus grande régularité dans ses résultats (le temps pris pour effectuer ce GC n'est évidemment pas pris en compte...). Si la forme *quelconque* déclenche un GC via l'utilisation des fermetures compgc ou masgc, les résultats retournés ne tiendront pratiquement pas compte du temps pris par ce GC.

```
(chrono (fib 20))     ➠ [10946 6.45000000000e+0 0.000000000000e+0]
```

(**masgc**)	Fermeture:ProcN	0

Fait appel au Glaneur de Cellules afin qu'il effectue un ramassage de type "Mark and Sweep". Ce GC très rapide ne diminue pas la fragmentation. Il supprime cependant tous les blocs de type "indirection" que pourraient avoir installé les suspensions forcées ou la fermeture replace. Les blocs isolés de taille 1 ou 2 ne sont pas récupérés. Les symboles dont la valeur est indéfinie et qui ne sont plus référencées sont récupérés. La valeur retournée est ?.

(**compgc**)	Fermeture:ProcN	0

Fait appel au Glaneur de Cellules afin qu'il effectue un ramassage de type "Break Table modifié". Ce GC compacifiant est plus lent et diminue la fragmentation. Il supprime tous les blocs de type "indirection" que pourraient avoir installé les suspensions forcées ou la fermeture replace. Les symboles dont la valeur est indéfinie et qui ne sont plus référencées sont récupérés. La valeur

retournée est `?`.

(**blength** *quelconque*)	Fermeture:`ProcN`	`1`

Retourne la quantité de place mémoire occupé par *quelconque* en mot longs (32 bits).La place occupé par le “tag” associé à chaque bloc n’est pas prise en compte.

```
(blength 'a)        ➟ 6
(blength 2)         ➟ 1
```

(**bcopy** *quelconque*)	Fermeture:ProcN	1

Retourne une copie de l’objet *quelconque* . Cette copie est surfacique (copie de l’objet seulement et non des objets référencés par *quelconque*). Cette fonction ne doit pas être employée sur les objets dont l’unicité est préservée par le système (symboles...).

```
(bcopy 1)           ➟ 1
(eq? (bcopy x) x)   ➟ ƒ   ;(en supposant x ≠ symbole)
(=?  (bcopy x) x)   ➟ †    ;idem
```

(**replace** *quelconque1 quelconque2*)	Fermeture:ProcN	2

Remplace physiquement toutes les occurrences (au sens de eq?) de *quelconque1* par *quelconque2*. Cette fermeture utilise le mécanisme des indirections pour effectuer le remplacement. A utiliser avec précaution sur les identificateurs de variable et a fortiori sur les identificateurs de constante (en particulier ƒ, †...). Son utilité reste à découvrir...

```
(define x '(a b c)) ➟ (a b c)
(replace 'b 'k)     ➟ k
x                   ➟ (a k c)
```

(**≈**)	Fermeture:ProcN	0

Obtenu au clavier par Option-X. Retourne la dernière valeur obtenue au top-level.

```
(openi "palmipède") ➟ «IO-Unit»
(readio (≈))        ➟ (define (palmipede v)…)
```

(**where** *quelconque*)	Fermeture:ProcN	1

Retourne l’adresse de *quelconque* en mémoire dans un entier de taille fixe (32 bits).

(**find** *quelconque*)	Fermeture:ProcN	1

Retourne la liste des identificateurs de variables dont la valeur dans l’environnement global est égale (au sens de eq?) à *quelconque*. Particulièrement utile pour retrouver le symbole associé à une fermeture...

```
(define x 1+) ➟ {Closure: {Code 6800xx for 1+} in {Env: ()}}
(find x)      ➟ (x 1+)
```

5.5 L’interface Help

Nous décrirons ici l'implémentation réalisée sur MacIntosh II. Cette réalisation reste incomplète à deux points de vue: le compilateur n'est pas encore "utilisable", l'environnement reste encore embryonnaire (pas d'impression, taille limitée des fichiers...).

5.5.1 Configuration

Rappelons que l'interprète Help ne tourne que sur 68020+, il ne peut donc être utilisé que sur MacSE30 ou MacII. Il peut tourner sur très peu de mémoire (500 Ko) si l'on est prêt à supporter des GCs fréquents. Une taille mémoire de 1 Mo commence a être confortable. Help "tourne" sous MultiFinder ou système 7 sans problème.

La taille mémoire "demandée" par Help est tout simplement la taille affichée en sélectionnant l'application "Paresseux" et en demandant les Infos (Command-I). sous le Finder.On peut alors la fixer à sa convenance. Par défaut, elle est ajustée à 512 Ko. Modifiez la si vous avez plus (je j(espère pour vous).

Pour modifier d'autres caractéristiques de Help, un ensemble de ressources TEMPLATES pour RESEDIT (à insérer dans votre Resedit avec Resedit) est fournie avec Help. Il permet d'éditer aisément les différentes ressources intéressantes.

5.5.1.1 La ressource CONF (Id 0, "Configuration")

Contient quatre champs modifiables par l'utilisateur:

- Font Id: permet de fixer la fonte utilisée par défaut par l'éditeur Help. La valeur initiale (22) correspond à la fonte Courrier (fonte non proportionnelle, permettant une indentation de qualité).
- Font Size: permet de fixer la taille de la fonte utilisée par défaut par l'éditeur Help. Valeur par défaut: 9.
- Block Visu: permet de fixer la durée (en soixantièmes de secondes) pendant laquelle l'éditeur Help visualise le bloc lexical courant (parenthèses, crochets et accolades correspondantes) lors de l'utilisation de la souris ou des touches de déplacement. Valeur initiale: 6 soit 1/10 seconde.
- Stack Memory %Age: permet de fixer le pourcentage de mémoire réservée aux pile. Valeur initiale :10% (très largement suffisant).

5.5.1.2 Les ressources STC#

Elles contiennent un ensemble de chaînes (codée "à la C"). La première (Id 0, "SynF Names") contient le nom de toutes les formes syntaxiques du langage et sont modifiables à la convenance de l'utilisateur (qui devra quand même modifier tous ses sources...).

La seconde (Id 1, "Erreurs") contient alternativement le message d'erreur correspondant à l'erreur dont le nom suit. Là encore, l'utilisateur peut si il le désire modifier les messages d'erreur (sans aucun problème) et les noms des erreurs (avec modification des sources si il font références à des symboles d'erreur).

5.5.1.3 La ressource STCL (Id 0, "Startup File")

Elle contient le nom du fichier évalué par l'interprète au démarrage. Modifiable au gré de l'utilisateur (nom par défaut: "Start"). Il est possible d'indiquer un chemin d'accès complet en utilisant la syntaxe précisée aux fermetures `openi` et `openo`.

5.5.1.4 La ressource STCN (Id 0, "Closures Names")

Elle contient un ensemble d'informations afférentes à chacune des fermetures prédéfinies. Pour chacune des fermetures, il est possible de préciser:

- Strictness: nombre en hexadécimal qu'il faut considérer comme un tableau de 16 bits, dont les 15 bits de poids faible indique (quand le bit n est positionné) que la fermeture est stricte par rapport à son n^{ième} argument et dont le bit de poids fort indique si la fermeture est stricte par rapport à tous les autres arguments. Il est évidemment assez facile de transformer Help en un "Pseudo-Scheme" en mettant tous ces champs à $FFFF et en utilisant la macro `defkap` (au lieu de `define`) du fichier de démarrage.

- Arity: Si la fermeture est d'arité fixe (type 0), ce champ indique son arité. Si elle est d'arité variable (type 1), il indique son arité minimale. La modification de ces champs par l'utilisateur n'est ni utile, ni conseillée.

- Type: Désigne le type de la procédure (arité fixe:0, arité variable: 1). La modification de ces champs par l'utilisateur n'est ni utile, ni conseillée.

- The string: Désigne tout simplement le nom du symbole qui désignera la fermeture. Si un nom est modifié, il faudra bien évidemment modifier tous les sources utilisant cette fermeture (y compris le fichier de démarrage !).

5.5.1.5 La ressource CART (Id 200, "Reader conf")

Contient 256 octets désignant au lecteur le type de chacun des caractères ASCII. Pour modifier convenablement ce tableau afin de modifier la micro-syntaxe de Help, se reporter au §4.2.6.

5.5.1.6 Les ressources WIND

Afin de s'adapter parfaitement tant à l'utilisateur qu'au type d'écran utilisé. Il est possible de choisir la répartition des fenêtres à l'écran. Pour ceci il suffit de modifier au choix:

- La taille et position de fenêtre "Transcript", ID 1000;
- Les tailles et positions des cinq fenêtres d'édition qu'utilise cycliquement l'éditeur à chaque création de fenêtre (ID 1001 à 1005).

5.5.2 Utilisation de l'éditeur

L'éditeur utilisant intensivement les routines de la ROM, il possède de ce fait certaines limitations, la plus importante étant que toutes les fenêtres d'édition ne peuvent contenir un texte qui fasse plus de 32K caractères. Il arrive des choses indéfinies (mais sans danger) en cas de dépassement.

La fenêtre "Transcript" reçoit les messages d'erreurs (elle correspond à l'unité d'entré sortie dénotée au démarrage par le symbole "`stder`"). Etant donnée qu'elle a la même limitation en taille que toutes les fenêtres… il est indispensable de la "nettoyer" de temps à autre.

Toutes les fenêtres sont à la fois des fenêtres d'édition et d'évaluation. La touche de validation employée permet de choisir entre les deux modes:

- La touche "retour chariot" permet d'utiliser les fenêtres en mode "éditeur". L'indentation est automatique ainsi que le "matching" des parenthèses, crochets et accolades.

- La touche ENTER (clavier numérique) permet d'évaluer la forme sélectionné (si une sélection a été faite) ou de procéder à la sélection automatique de la S-expression supérieure et à son évaluation.

Dans ce cas, il est possible de jouer sur l'impression de la valeur (boucle READ-EVAL-PRINT):

- Si on appuie sur Shift en même temps qu'ENTER, il n'y aura pas impression de la valeur. Ceci permet naturellement d'éviter le "forçage" par l'imprimeur de toutes les suspensions contenues dans la valeur.

- Si on appuie sur Option en même temps qu'ENTER, la valeur sera imprimée sur l'unité d'entrée sortie dénotée par le symbole `stder`. Ceci est particulièrement utile lorsque l'on désire les valeurs sans abîmer le fichier évalué.

Le "click" souris permet de vérifier le "matching" des "`()`", "`[]`" et "`{}`". Si il n'y a pas de correspondance, il y a émission d'un bip sonore. Si on maintient la touche Option appuyée durant le click, la sélection restera, permettant de copier, effacer…. Le double click permet de sélectionner un "mot".

5.5.2 L'évaluateur et les "bugs"…

En cas de bug pendant une évaluation (icône en forme de petit Mac+) ou durant un GC (icône de Mark, Sweep, Compacifieur (1,2,3)):

- Si vous utilisez un "Debugger", il suffit de sauter à l'adresse contenue dans le registre D6 (Taper `G D6` sous MacsBug). Il y aura alors retour au Top-Level, dans les meilleures conditions possibles (piles et registres de la machine restaurés…), le système peut être instable si le Tas MacIntosh ou Help ont subi des dommages….

- Sinon, le "System Error Manager" va tenter de dessiner un dialogue contenant un message d'erreur et deux boutons. Il se peut que le dessin soit incomplet…. Le bouton de gauche est le classique "Redémarrer", celui de droite permet de "Reprendre" dans les conditions signalées ci-dessus… (rafraîchir ensuite l'écran en "zoomant" puis "dézoomant" une fenêtre…).

Bibliographie

[Abelson 85] : Harold Abelson et Gerald Jay Sussman avec Julie Sussman
Structure and Interpretation of Computer Programs
M.I.T. Press, Cambridge, 1985

[Aho 83]: A. Aho, J. Hopcroft et J. Ullmann
Structures de données et Algorithmes
InterEditions 87 (orig. Addison & Wesley 83)

[Aho 86]: A. Aho, R. Sethi et J. Ullmann
Compilateurs: Principes, Techniques et Outils
InterEditions 89 (orig. Addison & Wesley 86)

[Allen 78]: John Allen
Anatomy of LISP
McGraw-Hill Inc., 1978

[Allison 86]: Lloyd Allison
A Practical introduction to denotational semantics
Cambridge University Press, 1986

[Ashcroft 85]: Edward A. Ashcroft, William W. Wadge
Lucid, the Dataflow Programming Language
Academic Press, 1985

[Avenhaus 90] J. Avenhaus & K. Madlener
Term Rewriting and Equationnal Reasoning
Elsevier Science Publishers - North Holland, 1990

[Barendregt 84] Barendregt H.P.
The Lambda calculus, Its syntax and semantics
North Holland, 1984

[Bloss 88] A. Bloss, P. Hudak, J. Young
Code Optimisations for Lazy Evaluation
Lisp ans Symbolic Computation ,Vol. 1, N° 2,p 147-164 (1988)

[Briot 86] J.P. Briot, P. Cointe & E. Saint-James
Réécriture et récursion dans une fermeture
Journées Langages Orientés Objet - p90-100

[Cayrol 85] : Cayrol Michel
Conception, Formalisation et Expérimentation d'un modèle pour le traitement d'objets finis ou infinis dénombrables.
Thèse d'Etat , Université Paul Sabatier, 1985

[Cayrol 87] : Schiex Thomas, Cayrol Michel
Psil: L'infini en programmation
AFCET-RFIA 1987

[Cayrol 90] Cayrol Michel, Palmade Olivier, Schiex Thomas
ATMS: A new semantics.
En soumission (New generation computing), 1990

[Chailloux 80]: Jérôme Chailloux
Le Modèle VLisp: Description, Implémentation et Evaluation
Thèse de troisième cycle, Université P. et M. Curie (Paris VI), 1980

[Chailloux ??]: Jérôme Chailloux & ??

Manuel Le_Lisp version 15.21
INRIA - 19??

[Clinger 82]: William Clinger
NonDeterministic Call by Need is Neither Lazy Nor by Name
ACM Symposium on Lisp and Functionnal Programming, 1982

[Clinger 87]: William Clinger, Jonathan Rees (Editors)
Revised^3 Report on the Algorithmic Language Scheme
M.I.T. Artificial Intelligence Memo.

[CM2 87]: Thinking Machines Company
Connection Machine - Model CM-2 - Technical Summary
Thinking Machines technical report HA87-4 , 1987

[Cohen 83]: Comparison of Compacting Algorithms for Garbage Collection
Jacques Cohen & Alexandru Nicolau
ACM Transactions on Prgramming Languages and Systems Vol5, N°4, Octobre 1983 - p532-553

[Cousineau 89]: Guy Cousineau et Gérard Huet
The CAML Primer - Projet Formel
INRIA-ENS- 1989

[Dybvig 90]: R. Kent Dybvig & Robert Hieb
A New Approach to Procedures with Variable Arity
Lisp & Symbolic computation, Vol.3, N°3, p229-244 (1990)

[Field 88]: Anthony J. Field, Peter G. Harisson
Functionnal Programming
Addison Wesley Publishing company - 1988

[Gabriel 85]: Richard P. Gabriel
Performance and Evaluation of Lisp Systems
The MIT Press - 1985

[Girardot 85]: Jean Jacques Girardot
Les langages et les systèmes LISP
EdiTests, 1985

[Halstead 85]: Halstead R.H.
MultiLisp: A Language for concurrent symbolic computation.
ACM Transactions on Prgramming languages and systems 7(4),
(p 501-538) (Octobre 1985).

[Hillis ??]: Hillis ??
The Connection machine

[Hindley 86]: J. Roger Hindley & Jonathan P. Seldin
Introduction to Combinators and λ-Calculus
Cambridge University Press, 1986
Addison Wesley Publishing Company, 1968

[Jaulent 87]: P. Jaulent & L. Baticle
µ-processeurs 68020, 68030 et leurs coprocesseurs.
Eyrolles (1987)

[Knuth 68]: Donald E. Knuth
The Art of Computer Programming. Vol.1. Fundamental Algorithms
Addison Wesley Publishing Company, 1968

[Mac 85-86]: Apple Computer Inc.
Inside MacIntosh, Vol I à V
Addison Wesley Publishing Company, 1985-86

[Schiex 87] : Schiex Thomas
Psil: Manipulation d’objets infinis dénombrables
Rapport de D.E.A., Université Paul Sabatier, 1987

[Schiex 88] Schiex Thomas
Psil et la Connection Machine
Rapport pour le C.N.R.S et le Conseil Régional (1988)

[Schiex 89] : Schiex Thomas
Psil: Un héritier de Scheme
BIGRE: Special Issue : “Putting Scheme to work” , 1989

[Schiex 91] Schiex Thomas
Interprétation et Compilation d'un dialecte paresseux de Scheme: Help
Phd. Thesis
Université Paul Sabatier, Toulouse, FrancePhd. Thesis

[Steele 90] Guy L. Steele Jr.
Common Lisp: the language (2nd edition)
Digital Press - 1990

[Tarski 55] Tarski A.
A Lattice-theorical FixPoint Theorem and its Applications
Pacific J. Math. (p285-309), 1955

Exemples

```
{Composition de fonctions}
{••••••••••••••••••••••••}
(define (rond f g)
       (lambda x (f (apply g x))))

{Les entiers naturels}
{••••••••••••••••••••}
(define n (… 0))

{Les fibonacci}
{•••••••••••••}
(define (fibn n1 n2) (cons n1 (fibn (+ n1 n2) n1)))
(define fibl (cons 1 (cons 1 (map (∞ +) fibl (-1 fibl)))))

{La suite bizarre}
{•••••••••••••••}
(define (entrelace l1 l2)
     (cons (0 l1) (cons (0 l2) (entrelace (-1 l1) (-1 l2)))))
(define biz (entrelace (… 0) biz))

{flot de lecture}
{•••••••••••••••}
(define (in)(cons (read)(in)))
(define input (in))

{Factorielle CPS…}
{•••••••••••••••}
(define (fact x k)
    (cond (zero? x)(k 1)
    (fact (- x 1)(lambda(n)(k (* x n))))))
(define factl (cons 1 (map (∞ *) (… 1) factl)))

{Le crible d'Erathostène}
{••••••••••••••••••••••}
(define (erat l)
    (cons (0 l)
          (erat (diff (-1 l)
                      (map (∞ *) (∞ (0 l)) l)))))

{Décomposition en nombres premiers}
{••••••••••••••••••••••••••••••••}
(define (dec n l)
 (cond (=? n 1) ()
       (zero? (modulo n (0 l))) (cons (0 l) (dec (/ n (0 l)) l))
       (< n (* (0 l)(0 l))) (list n)
       (dec n (-1 l))))

{Church numbers}
{•••••••••••••••}
(define plus
 (lambda(n1)
        (lambda(n2)
               (lambda(f)
                      (lambda(x)
                             ((n1 f) ((n2 f) x)))))))

(define zero (lambda(f) (lambda(x) x)))

(define mul
 (lambda(n1)
        (lambda(n2)
               (lambda(f)
```

```
                        (lambda(x)
                                ((n1 (n2 f)) x))))))
(define suc
 (lambda(n)
        (lambda(f)
                (lambda(x) ((n f)(f x))))))

(define un (suc zero))
(define deux (suc un))
(define trois (suc deux))
(define quatre ((mul deux)deux))

(define exp
 (lambda(n1)
        (lambda(n2)
                (lambda(f)
                        (lambda(x)
                                (((n2 n1) f) x))))))

{The paradoxical combinators}
{••••••••••••••••••••••••••}

;by Church Y0
(define (Y0 g)
 ((lambda(x) (G (x x)))(lambda(x)(G (x x)))))

{The fixed point for fixed point combinator}
(define (G y)
 (lambda(f) (f (y f))))

;by Turing Y1=Y0 G
(define Y1
 ((lambda(a)
   (lambda(b) (b ((a a) b)))) (lambda(a)
                                (lambda(b) (b ((a a) b))))))
;un autre fpc délirant par Klop
(define £
(lambda(a)
 (lambda(b)
  (lambda(c)
   (lambda(d)
    (lambda(e)
     (lambda(f)
      (lambda(g)
       (lambda(h)
        (lambda(i)
         (lambda(j)
          (lambda(k)
           (lambda(l)
            (lambda(m)
             (lambda(n)
              (lambda(o)
               (lambda(p)
                (lambda(q)
                 (lambda(s)
                  (lambda(t)
                   (lambda(u)
                    (lambda(v)
                     (lambda(w)
                      (lambda(x)
                       (lambda(y)
                        (lambda(z)
                         (lambda(r)
(r ((((((((((((((((((((((((((
            (t h)i)s)i)s)a)f)i)x)e)d)p)o)i)n)t)c)o)m)b)i)n)a)t)o)r)
                          ))))))))))))))))))))))))))
```

```
(define $ ((((((((((((((((((((((
          (£ £)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)£)

;un autre fpc délirant "à la Klop" en VF
(define £
(lambda(a)
 (lambda(b)
  (lambda(c)
   (lambda(e)
    (lambda(f)
     (lambda(g)
      (lambda(h)
       (lambda(i)
        (lambda(j)
         (lambda(k)
          (lambda(l)
           (lambda(m)
            (lambda(n)
             (lambda(o)
              (lambda(p)
               (lambda(q)
                (lambda(r)
                 (lambda(s)
                  (lambda(t)
                   (lambda(u)
                    (lambda(v)
                     (lambda(w)
                      (lambda(x)
                       (lambda(y)
                        (lambda(z)
                         (lambda(d)
(d ((((((((((((((((((((((
             (H e)l)p)E)s)t)T)e)r)r)i)b)l)e)m)e)n)t)F)l)e)m)m)a)r)d)
                          )))))))))))))))))))))))

;la fonctionnelle associée à factorielle
(define (FF f)
 (lambda(x)
   (cond (zero? x) 1
         (* (f (1- x)) x))))

{calcul avec des inconnues: remplacer par max en 1 passe}
{••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••}
(define (rpm l)
 (letrec [((f n rest&maxc)
              (cons (cons max (0 rest&maxc))
                    (cond (>? n (-1 rest&maxc)) n (-1 rest&maxc))))
          (rest (reduce f '(() | 0) l))
          (max (-1 rest))]
         (0 rest)))
```

Index