1400011948
còpia 1

WALKER
una eina d'analisi del codi escrita en Le-Lisp versió 16*

Luis Valentin

Report LSI–90–28

WALKER
Una eina d'análisi del codi escrita en Le-Lisp versió 16*

Luis Valentin
Facultat d'Informàtica de Barcelona
Departament de Llenguatges i Sistemes Informàtics
Pau Gargallo, 5. Barcelona 08028
flbcls06@clrs.lsi.upc.es
Juliol–Setembre 1990

Abstract

WALKER is a very general code–analysis tool written in Le–Lisp version 16 whose initial purpose was to provide a code analyser for Le–lisp version 16 special forms language.

WALKER is similar to the code walker included in **CLOS** (Common Lisp Object System).

Recursive descent is handled by **patterns** which are data structures containing information about evaluation contexts expressed in terms of a set of specification keywords such as **eval, fcall** or **optional**. So, WALKER does not use **case**–like forms to perform subexpression search as it is usual in other program–analysis tools.

Patterns definition capability and walk function abstraction (the walk function is one of the arguments of WALKER main entry) have permitted to build a code walker with a high degree of generality. WALKER also allows the definition of special functional patterns for any form undescribable in the specification language.

* LE-LISP és una marca registrada per l'INRIA, la versió 16 de LE-LISP es desenvolupa actualment a ILOG S.A. (Gentilly, France). El present treball ha estat realitzat durant el periode juliol-setembre 1990 a ILOG.

Resum

WALKER és una eina general d'anàlisi del codi escrita en le–lisp versió 16 amb l'objectiu inicial d'oferir un analitzador de codi pel conjunt de les formes especials de Le–lisp versió 16.

WALKER és una eina similar al code walker inclòs al CLOS (Common Lisp Object System).

L'anàlisi recursiva s'implementa mitjançant estructures de dades anomenades **patterns** que contenen informació sobre els diversos contextes d'avaluació expressats en termes d'un conjunt de paraules clau (keywords) d'especificació com per exemple **eval, fcall** o **optional**. Aquesta és la raó per la qual WALKER no utilitza expressions tipus **case** per dur a terme la cerca de subexpressions com és habitual en eines d'anàlisi del codi.

La definició de patterns i l'abstracció de la walk function (la walk function és un dels arguments de la funció principal de WALKER) han permés la construcció d'un analitzador extremadament general. WALKER també permet la definició de patterns funcionals especials per qualsevol forma sense descripció possible en el llenguatge d'especificació.

La *Walk function* concentra tot el codi referent a modificacions del codi o a obtenció d'informació (per exemple detecció de variables lèxiques, variables dinàmiques, crides locals o globals ..). Aquesta particularitat permet una còmoda adaptació de WALKER a les necessitats dels usuaris donat que les manipulacions a efectuar no requereixen un coneixement profund del funcionament del motor recursiu de recorregut del codi. Per tant, el ventall de possibles aplicacions de WALKER inclou des de la definició d'un compilador o precompilador fins a eines molt més senzilles dedicades per exemple a l'enregistrament de variables léxiques.

WALKER : un outil d'analyse du code

Luis Valentin

(UPC (LSI (IT)))

ILOG

1 Introduction

`WALKER` est un outil d'analyse du code qui a été conçu pour analyser les formes spéciales de LE-LISP version 16. Cependant il est suffisament général pour être appliqué à d'autres tâches et surtout à d'autres langages.

Un analyseur peut être vu d'un point de vue théorique comme un moteur de parcours recursif de code accompagné d'une fonction dite fonction de parcours (walk function) qui est appliquée à chaque unité de code délivrée par le moteur. Cette fonction peut introduire des modifications ou plus simplement obtenir des informations de chaque unité reçue. Ce sera donc la fonction de parcours qui determinera l'orientation et l'utilité de l'outil.

2 Utilisation

Ainsi donc, un analyseur de code (code analyser ou code walker) pour être général devrait au moins admettre comme arguments la forme que l'on prétend analyser et une fonction de parcours particulière. `WALKER` tient compte de ce fait et ajoute aux deux arguments indiqués un troisième permettant la definition d'un environnement initial d'analyse.

`(walker forme environnement walk-function [macrof [ignoref]])` [*Function*]

Les deux derniers arguments sont optionnels et permettent:

- inhiber le processus de macroexpansion
- indiquer au `WALKER` que certaines parties du code sont à ignorer (nottament les commentaires)

`WALKER` peut s'utiliser aussi moyennant:

`(fast-walk forme)` [*Macro*]

ce qui est équivalent à l'appel

```
(walker forme () #'walk-fn)
```

Où `walk-fn` est un exemple de walk function inclu dans le module `WALKER`.

Les annexes A et B donnent des indications précises sur les modules à charger.

3 Le langage de spécification de formes

L'atout principal de WALKER est sans doute qu'il offre un petit langage de spécification de formes à l'aide duquel il est possible de définir la syntaxe de la plupart des formes que l'analyseur devra être capable de traiter.

En effet, chaque forme sera affectée de ce qu'on appelle un **pattern** défini en termes des mots clés (keywords ou entry types) du langage de spécification (ou langage de patterns). Ce pattern guidera l'analyse de la forme en question.

3.1 Mots clés du langage de spécification

Les mots clés de la version actuelle du langage sont les suivants:

- FCALL identification de fonctions (noms de fonctions ou fonctions anonymes)
- MORE indique l'éventuelle répétition d'une forme
- EVAL indique que l'unité de code correspondante à ce keyword doit être évaluée
- SET l'unité sera affectée d'une valeur
- KEY correspond généralement à l'identificateur de la forme
- QUOTE l'unité ne doit pas être évaluée
- LABEL l'unité correspondante doit être considérée comme une étiquette
- VARIABLE l'unité correspondante doit être considérée comme une variable
- SYMBOL semblable au précédent mais n'impose pas l'évaluation
- OPTIONAL indique que la suite du pattern est optionnelle
- MARK l'unité correspond à une "marque" dans l'environnement

3.2 Définition et manipulation des patterns

Un pattern est une liste formée de mots clés et éventuellement d'un autre pattern (interne) placé à la suite du keyword MORE.

Plus précisement, la syntaxe des patterns peut s'exprimer de la façon suivante:

Soit K l'ensemble des mots clés du langage de spécification. Un pattern est alors de la forme:

([K - {MORE OPTIONAL}]* [MORE ([K - {MORE OPTIONAL}]*)] [K - {MORE}]*)

Voici quelques exemples:

(KEY EVAL)

(MARK OPTIONAL EVAL)

```
(KEY SET EVAL MORE (SET EVAL))
(KEY EVAL EVAL OPTIONAL EVAL)
```

Les opérations pour définir et manipuler les patterns sont les suivantes:

`(define-pattern identificateur pattern)` *[Function]*

> Construit un nouveau pattern associé à **`identificateur`** et retourne un objet de type pattern. Si **`identificateur`** identifiait déjà un pattern celui-ci est modifié, **`define-pattern`** retourne alors `t`. Si **`pattern`** est incorrect [1] l'erreur `ERRPAT` est déclenchée.

`(remove-pattern identificateur)` *[Function]*

> Détruit le pattern associé à **`identificateur`** et retourne l'objet de type pattern supprimé. Si **`identificateur`** ne correspond pas un pattern, `()` est retourné.

`(lookup-pattern identificateur)` *[Function]*

> Retourne l'objet de type pattern associé à **`identificateur`**. Si celui-ci ne correspond pas a un pattern, `()` est retourné.

3.3 Fonctions spéciales

Le langage de spécification ne suffit pas toujours. En effet, certaines formes ont une syntaxe parfois trop complexe ou bien elles obligent à augmenter l'environnement et les mots clés actuels en sont incapables.

Voici quelques exemples:

- la forme spéciale **`tagbody`** a une syntaxe très particulière et en plus elle crée un certain nombre d'étiquettes que `WALKER` devra retenir
- les formes spéciales **`lambda`**, **`labels`**, **`flet`** créent de nouvelles liaisons (variables ou fonctions) qu'il faut ajouter à l'environnement

Pour guider l'analyse de telles formes, `WALKER` permet la définition de fonctions spéciales (ou special handling functions).

Les fonctions spéciales sont des fonctions à trois arguments:

- la forme que la fonction spéciale traite spécifiquement.
- l'environnement correspondant
- la fonction de parcours (walk function)

[1] define-pattern contient un parser pour le langage qui ne traite pas tous les cas décrits par la syntaxe précédente

Le résultat attendu est la forme analysée (a walked form).

`define-pattern` accepte la définition de patterns dits fonctionnels:

`(define-pattern identificateur fonction-speciale)`

4 La fonction de parcours

L'importance de la walk function a déjà été mise en évidence. Son avantage est clair, elle permet de concentrer tout le code concernant les modifications à introduire ou les informations à recueillir, et surtout n'oublions pas qu'elle est un des arguments passés à `WALKER` ce qui fournit un haut degré de flexibilité à l'ensemble fonction de parcours-moteur de parcours.

La walk function est une fonction à trois arguments:

- l'unité de code correspondante à l'état actuel de l'analyse
- un mot clé determinant le contexte d'évaluation
- l'environnement correspondant

La walk function doit retourner une paire pointée le **`car`** de laquelle correspond au résultat de l'application de la fonction à l'unité de code reçue. Le **`cdr`** de son côté, est un flag indiquant si l'analyse doit se poursuivre.

Le module `WALKER` contient un exemple de fonction de parcours qui peut être utilisé pour en construire d'autres adaptées à chaque tâche particulière.

5 Le moteur de parcours

`WALKER` analyse recursivement le code sans utiliser des structures du type **`case`** qui sont substituées par des patterns. La descente recursive est gérée par la fonction **`real-walker`** [2] au moyen d'un évaluateur pour le langage de spécification

`(real-walker forme contexte environnement walk-function)` *[Function]*

Agit sur les unités de code reçues (en utilisant la walk function) obtient les patterns appropiés et envoit cette information à l'évaluateur. Le résultat final d'un appel à cette fonction doit être la forme analysée.

Pour les appels fonctionnels **`real-walker`** construit des patterns appropiés en profitant toujours des informations disponibles [3]

[2]il ne faut pas attacher trop d'importance aux noms ..

[3]Le pattern fonctionnel type serait `(FCALL MORE (EVAL))` mais si la fonction a été définie localement (`labels` par exemple) le pattern pourrait être beaucoup plus précis, par exemple `(FCALL EVAL EVAL` pour une fonction à deux arguments. La fonction `construct-functional-pattern` est chargée de la définition des patterns fonctionnels

`real-walker` réalise également (si le mécanisme n'est pas inhibé) la macroexpansion des formes qu'elle reçoit.

(walk-eval-pattern forme pattern environnement walk-function**)** *[Function]*

Interprète `pattern` et délivre unités de code et contextes d'évaluation à **`real-walker`**.

(walk-eval-more forme pattern more-pattern end-form env walk-fn**)** *[F]*

La semanthique un peu particulière du keyword `MORE`[4] rend indispensable une fonction qui s'en occupe spécifiquement.

L'annexe C contient quelques explications sur les erreurs liées directement à évaluation des patterns.

6 Annexe A - Le langage des formes spéciales LE-LISP version 16

Le module `WALKER` contient le moteur de parcours, les fonctions de définition et manipulation des patterns et des environnements, des fonctions spécifiques pour chacun des mots clés du langage ainsi qu'un exemple de walk function. Le Module **`WALK16`** contient les definitions des patterns correspondantes au langage des formes spéciales LE-LISP version 16, ainsi que les fonctions spéciales qui ont été définies pour certaines formes.

Les patterns sont les suivants:

- `(define-pattern 'lambda #'lambda-walker)`
- `(define-pattern 'flet #'flet-walker)`
- `(define-pattern 'labels #'labels-walker)`
- `(define-pattern 'tagbody #'tagbody-walker)`
- `(define-pattern 'block #'block-walker)`
- `(define-pattern 'dynamic-let #'dynamic-let-walker)`
- `(define-pattern 'if '(KEY EVAL EVAL OPTIONAL EVAL))`
- `(define-pattern 'dynamic '(KEY SYMBOL))`
- `(define-pattern 'go '(KEY LABEL))`
- `(define-pattern 'return-from '(KEY LABEL EVAL))`

[4]En effet, ce mot clé n'est pas associé à une unité de code, il indique simplement qu'une partie de la forme (et donc une partie du pattern) peut être répétée un certain nombre de fois

- (define-pattern 'function '(KEY FCALL))
- (define-pattern 'unwind-protect '(KEY EVAL EVAL))
- (define-pattern 'multiple-value-prog1 '(KEY EVAL MORE (EVAL)))
- (define-pattern 'multiple-value-call '(KEY EVAL EVAL))
- (define-pattern 'progn '(KEY MORE (EVAL)))
- (define-pattern 'setq '(KEY SET EVAL MORE (SET EVAL)))
- (define-pattern 'dynamic-setq '(KEY SYMBOL EVAL))
- (define-pattern 'quote '(KEY QUOTE))
- (define-pattern 'catch '(KEY EVAL MORE (EVAL)))
- (define-pattern 'throw '(KEY EVAL EVAL))

Aux patterns précédents nous devons ajouter un autre concernant la forme **lambda** utilisée avec &nobind [5].

- (define-pattern 'arg '(MARK OPTIONAL EVAL))

arg est en effet une fonction (?) à 0 ou 1 argument. **WALKER** utilisé avec ce langage devrait être capable de résoudre des cas "pathologiques" comme les suivants:

```
(labels ((arg &nobind (arg)))
(flet ((arg &nobind (arg)))
```

Si la macroexpansion de formes est inhibée, il faut ajouter aux patterns précédents deux autres pour guider l'analyse des let et des let*.

- (define-pattern 'let #'let-special-walker)
- (define-pattern 'let* #'let*-special-walker)

7 Annexe B - Représentation des environnements

Le module WALKERSTR contient toutes les définitions d'objets nécéssaires à **WALKER**. En particulier, les définitions relatives aux **patterns** (**patterns-type**) et aux liaisons (**binding-type**).

Les environnements lexicaux manipulés par **WALKER** sont représentés par des listes d'objets binding-type.

Voici quelques unes des opérations de manipulation, création et accés des environnments et des liaisons:

[5] Lorsque ce symbole prend la place de la liste d'arguments, les références aux arguments se font au moyen de arg

`(construct-bindings names vbinds type args-list)` *[Function]*

`(create-binding-object name vbind type arg)` *[Function]*

`(lookup-name-env name environment)` *[Function]*

`(lookup-variable-p name environment)` *[Function]*

`(lookup-label-p name environment)` *[Function]*

`(lookup-mark-p name environment)` *[Function]*

`(lookup-set-expression-p name environment)` *[Function]*

8 Annexe C - Quelques précisions sur les erreurs de l'évaluateur

Les erreurs liées á l'évaluation des patterns, c'est à dire les erreurs de "pattern matching", sont les suivantes:

- ERRFSH la forme est trop courte par rapport au pattern
- ERRSFSH semblable au précédent mais faisant référence à une forme interne (traitée par MORE)
- ERRFLG la forme est trop longue par rapport au pattern

Ces erreurs sont peu claires surtout parce que WALKER a des difficultés pour déterminer exactement l'endroit où l'erreur s'est produite. C'est pourquoi, une fonction `find-closest-form` a été incorporée à WALKER rien que pour essayer de préciser ce type d'erreurs. [6]

9 Annexe D - Patterns pour certaines formes spéciales Eulisp

Le fichier `walkEu` contient quelques definitions de patterns pour certaines formes spéciales Eulisp, ainsi que le code des fonctions spéciales pour les formes `let/cc` et `with-handler` [7]

- `(define-pattern 'let/cc #'let/cc-walker)`

[6] WALKER ne construit pas l'arbre de la descente recursive, `find-closest-form` utilise pour préciser les erreurs,le parcours en profondeur produit par la fonction `keep-track-context`

[7] Il faudrait plutôt dire "quelques indications concernant les fonctions spéciales .."

- (define-pattern 'with-handler #'with-handler-walker)
- (define-pattern 'progn '(KEY MORE (EVAL)))
- (define-pattern 'if '(KEY EVAL EVAL OPTIONAL EVAL))
- (define-pattern 'multiple-argument-ref '(KEY VARIABLE EVAL))
- (define-pattern 'dynamic '(KEY SYMBOL))
- (define-pattern 'dynamic-setq '(KEY SYMBOL EVAL))
- (define-pattern 'quote '(KEY QUOTE))
- (define-pattern 'apply '(KEY FCALL MORE (EVAL)))

La forme `multiple-argument-values` fournit un bon exemple de l'utilisation du keyword `OPTIONAL`, le pattern est en effet:

```
(KEY VARIABLE OPTIONAL EVAL OPTIONAL EVAL)
```

```
;;;
;;; WALKER:  The code walker
;;;
;;; ---------------------------------------------------------------------
;;; This file is part of Le-Lisp version 16, developped by ILOG and INRIA.
;;; Inquiries to ILOG S.A.
;;;              2 Avenue Gallie'ni, BP 85,
;;;              94253 Gentilly Cedex, France.
;;;
;;; (c) 1989,1990 Le-Lisp is a trademark of INRIA.
;;;
;;; ---------------------------------------------------------------------
;;;


;;;
;;; WALKER
;;; ***********************
;;;
;;; patterns manipulation
;;; walk-functions
;;; walker motor
;;; patterns evaluator

(defglobal :state (#:walker-data-type:make))

(defun define-pattern (key pattern)
  ;;  define-pattern allows new patterns definition or update
  ;;  of patterns previously defined
  ;;  define-pattern includes a parser for patterns
  ;;  language
  (let ((pattern-flag (parse-pattern pattern))
        (obj (current-pattern (parse-key key)
                              (#:walker-data-type:patterns
                                          (global :state)))))
         (cond ((null obj)
                (update-patterns-list
                    (create-pattern-object key pattern pattern-flag)))
               (t (#:patterns-type:pattern obj pattern)
                  (#:patterns-type:special-handler? obj pattern-flag)
                  (print " pattern updated ")
                  obj))))

(defun update-patterns-list (obj)
  (#:walker-data-type:patterns
   (global :state)
   (cons obj (#:walker-data-type:patterns (global :state))))
  (print " new pattern ")
  obj)

(defun create-pattern-object (key pattern flag)
  (let ((new-obj (#:patterns-type:make)))
    (#:patterns-type:key new-obj key)
    (#:patterns-type:pattern new-obj pattern)
    (#:patterns-type:special-handler? new-obj flag)
    new-obj))

(defun parse-key (key)
  (if (symbolp key)
      key
    (:error 'parse-key ':errkey key)))

(defun parse-pattern (pattern)
  ;;  parser definition is trivial
  ;;  at the moment parse-pattern only verifies
  ;;  one of the restrictions that should be required
  ;;  i.e. that at each level of parenthesis
  ;;  nesting only one  more  entry-type is alowed
  (cond ((functional-object-p pattern) t)
        ((internal-parse-pattern pattern
                                 (#:walker-data-type:keywords (global :state))
                                 ())
         ())
        (t (:error 'parse-pattern ':errpat pattern))))

(defun functional-object-p (obj)
  ;; \\ What's wrong with FUNCTIONP?
  ;; \\too short
  (eq (type-of obj) 'function))

(defun internal-parse-pattern (pattern keywords one-more)
    (cond ((null pattern) t)
          ((atom pattern)
           (memq pattern keywords))
          ((eq (car pattern) 'more)
           (when (null one-more)
             (internal-parse-pattern (cdr pattern) keywords t)))
          (t
           (and (internal-parse-pattern (car pattern) keywords one-more)
                (internal-parse-pattern (cdr pattern) keywords one-more)))))


(defun remove-pattern (key)
  (let ((patterns (#:walker-data-type:patterns (global :state))))
    (#:walker-data-type:patterns
      (global :state)
      (remove (current-pattern key patterns) patterns))))

(defun lookup-pattern (key)
  (current-pattern key
                   (#:walker-data-type:patterns (global :state))))


(defun lambdap (form)
  (and (listp (car form)) (eq (caar form) 'lambda)))

(defun walk-variable (name env)
  ;; \\lexref is really pretty
  (if (lookup-variable-p name env)
      `(lexref ,name)
    `(dynamic ,name)))

(defun lookup-variable-p (name env)
  ;; looks for variables
  ;; modifies environment indicating a new variable reference
  ;; arguments
  ;;  a variable name
  ;;  an environment
  (let* ((remaining-env (lookup-name-env name env))
         (binding-object (car remaining-env)))
    (if remaining-env
        (case (get-type binding-object)
          ((variable immutable)
           (set-state binding-object 'used))
          (t (lookup-variable-p name (cdr remaining-env)))))))

(defun lookup-mark-p (name env)
  (let* ((remaining-env (lookup-name-env name env))
         (binding-object (car remaining-env)))
    (cond ((null remaining-env) ())
          ((eq (get-type binding-object) 'mark)
           (set-state binding-object 'used))
          (t (lookup-mark-p name (cdr remaining-env))))))
```

```
(defun lookup-set-expression-p (name env)
  (let* ((remaining-env (lookup-name-env name env))
         (binding (car remaining-env)))
    (cond ((null remaining-env) ())
          ((eq (get-type binding) 'immutable)
           (:error 'lookup-set-expression-p ':erribm name))
          ((eq (get-type binding) 'variable)
           (set-state binding 'used))
          (t (lookup-set-expression-p name (cdr remaining-env))))))

(defun walk-mark-expression (name env)
  (if (lookup-mark-p name env)
      name
      (:error 'walk-mark-expression ':errnls name)))

(defun walk-set-expression (name env)
  (cond ((not (variablep name))
         (:error 'walk-set-expression ':errsiv name))
        ((lookup-set-expression-p name env)
         `(lexref ,name))
        (t (:error 'walk-set-expression ':errdbm name))))

(defun walk-label-expression (label env)
  (cond ((not (symbolp label))
         (:error 'walk-label-expression ':errgns label))
        ((lookup-label-p label env)
         label)
        (t (:error 'walk-label-expression ':errgul label))))

(defun walk-symbol-expression (name)
  ;; () is not considered a symbol in this context
  (if (variablep name)
      name
    (:error 'walk-symbol-expression ':errnsn name)))

(defun walk-variable-expression (name env)
  (if (not (variablep name))
      (:error 'walk-variable-expression ':errsiv name)
    (walk-variable name env)))

(defun walk-fn (form entry-type env)
  ;; WALK-FN is the second really useful walker function.
  ;; performs code modifications
  ;; arguments
  ;;  a form
  ;;  an entry-type (see patterns language description)
  ;;  an environment
  ;; results
  ;;  a mapped-form
  ;;  a flag (walk end)
  ;; side effects
  ;;  modifies environments
  (terpri)
  (print form)
  (print entry-type)
  (case entry-type
    (eval (cond ((variablep form)
                 `(,(walk-variable form env) . t))
                (t `(,form))))
    (fcall `(,(walk-functional-call form env #'walk-fn) t))
    (set `(,(walk-set-expression form env) . t))
    (label `(,(walk-label-expression form env) . t))
    (symbol `(,(walk-symbol-expression form) . t))
    (variable `(,(walk-variable-expression form env) . t))
    (mark `(,(walk-mark-expression form env) . t))
    (quote `(,form . t))
    (t  `(,form))))

(defun walk-functional-call (form env walk-function)
  (prog1 (cond ((variablep form)
                (if (get-current-local-call)
                    `(loccall ,form)
                    `(globcall ,form)))
               ((eq (car form) 'lambda)
                (lambda-function-walker form env walk-function))
               (t (:error 'walk-functional-call ':errifc form)))
         (set-current-local-call ())))

(defun lookup-call-p (name env)
  (let* ((remaining-env (lookup-name-env name env))
         (functional-binding (car remaining-env)))
    (cond ((null remaining-env) ())
          ((eq (get-type functional-binding) 'functional-object)
           (set-current-local-call functional-binding)
           (set-state functional-binding 'used))
          (t (lookup-call-p name (cdr remaining-env))))))

(defun set-current-local-call (val)
  (#:walker-data-type:current-local-call (global :state) val))

(defun get-current-local-call ()
  (#:walker-data-type:current-local-call (global :state)))

(defun get-macro-disabled ()
  (#:walker-data-type:macro-disabled (global :state)))

(defun get-str-list ()
  (#:walker-data-type:str-list (global :state)))

(defun get-name (binding-object)
  (#:binding-type:name binding-object))

(defun get-type (binding-object)
  (#:binding-type:type binding-object))

(defun get-state (binding-object)
  (#:binding-type:state binding-object))

(defun get-args (binding-object)
  (#:binding-type:args binding-object))

(defun set-args (binding-object args)
  (#:binding-type:args binding-object args))

(defun set-name (binding-object name)
  (#:binding-type:name binding-object name))

(defun set-type (binding-object type)
  (#:binding-type:type binding-object type))

(defun set-vbind (binding-object vbind)
  (#:binding-type:vbind binding-object vbind))

(defun set-state (binding-object state)
  (#:binding-type:state binding-object state))

;;;no more eval

(defun improper-list-p (list)
  (and (listp list) (cdr (last list))))
```

```
(defun make-proper-list (list)
  (append (firstn (length list) list)
          `(,(cdr (last list)))))
;;;

(defun lambda-expression-walker (form env walk-function)
 ;;LAMBDA-EXPRESSION-WALKER walks lambda expressions invoking
 ;;appropiate patterns for lambda function and body
  (let ((new-lambda-function (lambda-function-walker (car form)
                                                      env
                                                      walk-function))
        (new-vbind (walk-vbind (cdr form)
                               (get-current-local-call)
                               env
                               walk-function)))
    `(,new-lambda-function
      ,@new-vbind)))

(defun lambda-function-walker (form env walk-function)
 ;;LAMBDA-FUNCTION-WALKER finds a pattern for lambda forms
 ;;and then walks form
 ;;eventually raises an error if there is not a pattern
 ;;for lambda forms (:ERRIFC)
  (let* ((template (get-pattern form env))
         (pattern (car template))
         (special-handler-p (cdr template)))
    (cond (special-handler-p
            (funcall pattern form env walk-function))
          (pattern
            (walk-eval-pattern form pattern env walk-function))
          (t (:error 'lambda-function-walker ':errifc form)))))

(defun walk-vbind (vbind functional-object env walk-function)
 ;;WALK-VBIND walks lambda expressions body and also check
 ;;(if it is possible) the number of arguments
  (let* ((lambda-list (when functional-object (get-args functional-object)))
         (nargs (car lambda-list))
         (nvbind (length vbind)))
    (case (cdr lambda-list)
      (proper
        (when (gt nargs nvbind) (:error 'walk-vbind ':errfa nvbind))
        (when (lt nargs nvbind) (:error 'walk-vbind ':errma nvbind)))
      (improper
        (when (gt nargs nvbind) (:error 'walk-vbind ':errfa nvbind)))))
    (set-current-local-call ())
    (walk-eval-pattern vbind
                       '(more (eval))
                       env
                       walk-function))

(defun lambda-list-type (lambda-list)
  (case (cdr lambda-list)
    (undefined
      (list (car lambda-list)))
    (improper
      (make-proper-list (car lambda-list)))
    (t (car lambda-list))))

(defun construct-typed-lambda-list (lambda-list)
  (cond ((variablep lambda-list)
         (cons lambda-list 'undefined))
        ((listp lambda-list)
         (if (improper-list-p lambda-list)
             (cons lambda-list 'improper)
             (cons lambda-list 'proper)))
        (t (:error 'construct-typed-lambda-list ':errlsl lambda-list))))
```

```
;;;

(defun create-binding-object (name vbind type args)
   (let ((new-obj (#:binding-type:make)))
      (set-name new-obj name)
      (set-vbind new-obj vbind)
      (set-type new-obj type)
      (set-args new-obj (cons (length (car args)) (cdr args)))
      (set-state new-obj 'unused)
      new-obj))

(defun construct-bindings (names vbinds type args-list)
   (mapcar #'create-binding-object
               names
               vbinds
               (cirlist type)
               args-list))


;;;  *****************
;;;      WALKER
;;;  *****************


;;; general walker motor
;;; uses a special pattern language to perform a recursive
;;; analysis of given forms

;;; pattern-language definition
;;;  entry-types
;;;    fcall  function-call
;;;    more   denotes repetition of sub-forms
;;;    eval   evaluation
;;;    set
;;;    key    for keyword
;;;    quote
;;;    label
;;;    variable
;;;    optional
;;;    mark   marking lexical environment

;;;  patterns include a special-handler? flag
;;;  active when a special purpose handling function
;;;  is provided for a given form
;;;
;;; e.g.  a progn form
;;;           (progn form1 form2 .. ..)
;;;       could be represented
;;;           (key more (eval))
;;;       a setq form
;;;           (setq var1 form1 var2 form2 .. ..)
;;;       could be represented
;;;           (key set eval more (set eval))
;;;       an if form
;;;           (if form1 form2 [form3])
;;;       could be represented
;;;           (key eval eval optional eval)

(defun get-pattern (keyform env)
  ;; returns an appopiate pattern
  ;; () if there is no pattern defined for key
  ;; if keyform is a lambda-expression an special purpose function
  ;; is returned
  (and (not (atom keyform))
       (let ((pattern (find-pattern (car keyform) env)))
          (if (lambdap keyform)
            (cons #'lambda-expression-walker t)
          pattern))))

(defun find-pattern (key env)
  (let ((obj (current-pattern key
                              (#:walker-data-type:patterns (global :state)))))
    (and (not (lookup-call-p key env))
         obj
         (cons (#:patterns-type:pattern obj)
               (#:patterns-type:special-handler? obj)))))

(defun current-pattern (key patterns)
  (cond ((null patterns) ())
        ((eq (#:patterns-type:key (car patterns)) key) (car patterns))
        (t (current-pattern key (cdr patterns)))))

(defun :init-state (macrof)
  (#:walker-data-type:str-list (global :state) ())
  (#:walker-data-type:current-local-call (global :state) ())
  (when macrof
        (#:walker-data-type:macro-disabled (global :state) t)
        (print " macroexpansion disabled ")))

(defun cleanup-form (form cleanup)
  (cond ((atom form) form)
        (cleanup
          (print "cleaning .. ")
          (labels ((local-cleanup (form clean-form)
                     (let ((sub-form (car form))
                           (rem-form (cdr form)))
                       (cond ((null form) clean-form)
                             ((listp sub-form)
                                `(,@clean-form
                                  ,(local-cleanup sub-form ())
                                  ,@(local-cleanup rem-form ())))
                             ((:ignore-form-p sub-form)
                                (local-cleanup rem-form clean-form))
                             (t (local-cleanup rem-form
                                               `(,@clean-form ,sub-form))))))
            (local-cleanup form ())))
        (t form)))


;;;  WALKER motor
;;; **************
;;; WALKER main entry, pattern language evaluator
;;; general messages

(defun walker (form env walk-function &rest side-effects)
  ;; main entry
  ;; arguments
  ;;  a form
  ;;  an environment
  ;;  the walk function
  ;;  a side effects list
  ;;   allows to disable macroexpansion and to ignore
  ;;   certain parts of form
  ;; result
  ;;  a walked form
  (:init-state (car side-effects))
  (real-walker (cleanup-form form (cadr side-effects))
               'eval
               env
               walk-function))

(defmacro fast-walk (form)
  `(walker ',form () #'walk-fn))

(defun real-walker (form entry-type env walk-function)
  ;; recursive walker
  ;; arguments
  ;;  a form
  ;;  an entry type defined previously initially eval
  ;;  an environment
  ;;  the walk function
  ;; result
  ;;  a walked form
  (keep-track-context form env entry-type)
  (let* ((wf (funcall walk-function form entry-type env))
         (mapped-form (car wf))
         (walk-end-p (cdr wf))
         (ptt (get-pattern mapped-form env))
         (pattern (car ptt))
         (special-handler-p (cdr ptt))
         (expanded-form (macro-walker mapped-form pattern)))
    (cond (walk-end-p mapped-form)
          ((not (eq form mapped-form))
           (real-walker mapped-form entry-type env walk-function))
          ((atomp mapped-form) mapped-form)
          (special-handler-p (funcall pattern mapped-form env walk-function))
          (pattern (walk-eval-pattern mapped-form pattern env walk-function))
          ((eq mapped-form expanded-form)
           (walk-eval-pattern mapped-form
                              (construct-functional-pattern)
                              env
                              walk-function))
          (t (real-walker expanded-form entry-type env walk-function)))))

(defun construct-functional-pattern ()
  ;; constructs an appropiate functional pattern
  ;; whenever some information is available
  (let* ((functional-object (get-current-local-call))
         (arguments (when functional-object (get-args functional-object))))
    (case (cdr arguments)
      (improper
        `(fcall ,@(makelist (car arguments) 'eval) more (eval)))
      (proper
        `(fcall ,@(makelist (car arguments) 'eval)))
      (t '(fcall more (eval))))))

(defun macro-walker (form pattern)
  ;; handling eventual macroexpressions redefinition
  (cond ((atom form) form)
        (pattern form)
        ((get-current-local-call) form)
        ((get-macro-disabled) form)
        (t (car (macroexpand-1 form)))))



;;; pattern-language evaluator
;;; **************************
;;; recursive descent is not handled by case forms
;;; patterns including the  more  entry-type
;;; are evaluated by walk-eval-more

(defun walk-eval-pattern (form pattern env walk-function)
  ;; performs the recursive descent
  ;; arguments
  ;;  a form
  ;;  a pattern see get-pattern
  ;;  an environment
  ;;  the walk function
  ;; result
  ;;  walks form according to pattern
  (cond ((atomp pattern)
         (back-to-real-walker form pattern env walk-function))
        ((eq (car pattern) 'more)
         (walk-analyse-more form pattern env walk-function))
        ((eq (car pattern) 'optional)
         (when form
           (walk-eval-pattern form (cdr pattern) env walk-function)))
        ((atomp form)
         (focalize-error ':errfsh form))
        (t (cons (walk-eval-pattern (car form)
                                    (car pattern)
                                    env
                                    walk-function)
                 (walk-eval-pattern (cdr form)
                                    (cdr pattern)
                                    env
                                    walk-function)))))

(defun walk-analyse-more (form pattern env walk-function)
  ;;calls walk-eval-more
  ;;computes end form
  ;;eventually raises an error :ERRFSH
  (let* ((more-form-length (sub (length form)
                                (length (cddr pattern))))
         (end-form (nthcdr more-form-length form)))
    (if (lt more-form-length 0)
        (focalize-error ':errfsh form)
        (walk-eval-more form
                        (cdr pattern)
                        ()
                        end-form
                        env
                        walk-function))))


(defun back-to-real-walker (form pattern env walk-function)
  ;; calls real-walker
  ;; eventually returns form
  (case pattern
    (key form)
    ((eval fcall set quote label symbol variable mark)
     (real-walker form pattern env walk-function))
    (t (and form (focalize-error ':errflg form)))))

(defun focalize-error (msg off)
  (:warn 'focalize-error
         ':meval
         (get-form (car (get-str-list))))
  (:warn 'find-closest-form
         ':min
         (find-closest-form))
  (:error 'focalize-error msg off))

;;;now comes a pretty function very helpful in focalizing errors

(defun find-closest-form ()
  ;;when an unfocalized error raises (:ERRFSH :ERRSFSH :ERRFLG)
  ;;finds the closest form (super-form)
  (let* ((context-objects-list (get-str-list))
         (current-form (get-form (car context-objects-list))))
    (labels ((local-find (objects-list alias)
               (let ((super-form (get-form (car objects-list))))
                 (cond ((null objects-list) current-form)
                        ((eq current-form super-form)
                           (local-find (cdr objects-list) (add1 alias)))
                        ((listp super-form)
                           (local-check objects-list
                                        (sub alias
                                            (local-memq super-form))
                                        super-form))
                        (t (local-find (cdr objects-list) alias)))))
                (local-check (objects-list alias form)
                  (if (eqn alias 0)
                     (cond ((eq (car form) 'quote)
                              (setq current-form form)
                              (local-find (cdr objects-list) 1))
                            (t form))
                     (local-find (cdr objects-list) alias)))
                (local-memq (super-form)
                  (let ((form (memq current-form super-form)))
                    (if (null form)
                        0
                        (add1 (local-memq (cdr form)))))))
      (local-find (cdr context-objects-list) 1))))

(defgeneric :ignore-form-p (form))

(defmethod :ignore-form-p (form)
  (ignore form)
  ())


(defun get-form (obj)
  (#:context:form obj))


(defun walk-eval-more (form pattern more-pattern end-form env walk-function)
  ;; evaluates more patterns
  ;; returns a partially walked form
  (cond ((null form)
         (walker-end-on-more more-pattern end-form))
        ((eq form end-form)
         (back-to-walk-eval form (cdr pattern) more-pattern env walk-function))
        ((null more-pattern)
         (walk-eval-more form pattern (car pattern) end-form env
                         walk-function))
        (t (cons (walk-eval-pattern (car form)
                                    (car more-pattern)
                                    env
                                    walk-function)
                 (walk-eval-more (cdr form)
                                 pattern
                                 (cdr more-pattern) end-form
                                 env
                                 walk-function)))))

(defun walker-end-on-more (more-pattern end-form)
  ;; stops more evaluation
  ;; eventually raises an error
  (if (and (null more-pattern)
           (null end-form))
      ()
    (focalize-error ':errsfsh ())))

(defun back-to-walk-eval (form remaining-pattern
                               more-pattern env walk-function)
  ;;evaluation process returns to walk-eval-pattern
  ;;eventually raises an error
  (if (null more-pattern)
```

```lisp
      (walk-eval-pattern form remaining-pattern env walk-function)
    (focalize-error ':errsfsh form)))

;;;

(defun lookup-name-env (name remaining-env)
  (cond ((atom remaining-env) ())
        ((eq (get-name (car remaining-env)) name) remaining-env)
        (t (lookup-name-env name (cdr remaining-env)))))

;;;

(defun lookup-label-p (label env)
  (let ((remaining-env (lookup-name-env label env)))
    (cond ((null remaining-env) ())
          ((eq (get-type (car remaining-env)) 'label)
           (set-state (car remaining-env) 'used))
          (t (lookup-label-p label (cdr remaining-env))))))

;;;

;;; objects for context information storage
;;; we keep track of form environment and entry-type
;;; each time keep-track-context is called

(defun keep-track-context (form env entry-type)
  ;; KEEP-TRACK-CONTEXT creates a new instance of CONTEXT and sets
  ;; fields with its arguments.
  (let ((new-obj (get-context-object (#:context:make))))
    (#:context:form new-obj form)
    (#:context:environment new-obj env)
    (#:context:entry-type new-obj entry-type)
    new-obj))

(defun get-context-object (obj)
  ;; get-context-object keeps track of the new instance
  ;; (global :state) instance of  (global :state)-type is updated
  (#:walker-data-type:str-list
   (global :state)
   (cons obj (#:walker-data-type:str-list (global :state))))
  obj)

(defun internal-display-context (str-list)
  (cond ((null str-list) ())
        (t  (pprint (car str-list))
            (internal-display-context (cdr str-list)))))

(defun #:context:pretty (x)
  ;; special pretty-print format for context type
  (print 'form)
  (pprint (#:context:form x))
  (print 'environment)
  (pprint (#:context:environment x))
  (print 'entry-type)
  (pprint (#:context:entry-type x)))

(defun display-context-data (key)
  ;; DISPLAY-CONTEXT-DATA follows the list STR-LIST from (GLOBAL :STATE)
  ;; and displays context information stored on each instance of
  ;; context.
  (case key
    (patterns
     (internal-display-context
      (#:walker-data-type:patterns (global :state))))
    (keep-track
     (internal-display-context
      (reverse (#:walker-data-type:str-list (global :state)))))
    (t 'the-reklaw-walker)))

;;;

;;; Messages
;;; greetings from Catalonia

(record-language 'catalan)

(defun :error (&rest args)
  (apply #'error args))

(defun :warn (&rest args)
  (apply #'warn args))


;;; walker errors and warnings

(defmessage :errkey
   (english "key must be a symbol")
   (french  "cle doit etre un symbole")
   (catalan "clau ha d'esser un simbol"))

(defmessage :errpat
   (english "incorrect pattern")
   (french "pattern incorrect")
   (catalan "pattern incorrecte"))

(defmessage :errfsh
   (english "the form is too short")
   (french "la forme est trop courte")
   (catalan "la forma es massa curta"))

(defmessage :errsfsh
   (english "incomplete repeat sub-form")
   (french "sub-forme de repetition incomplete")
   (catalan "sub-forma de repeticio incompleta"))

(defmessage :errifc
   (english "incorrect function call")
   (french "appel fonctionnel incorrect")
   (catalan "crida funcional incorrecta"))

(defmessage :errsiv
   (english "illegal variable")
   (french "variable illegale")
   (catalan "variable il.legal"))

(defmessage :errnsn
   (english "non symbolic argument")
   (french "argument non symbolique")
   (catalan "argument no simbolic"))

(defmessage :erribm
   (english "attempt to modify an immutable binding")
   (french "tentative de modification d'une liaison immuable")
   (catalan "intent de modificacio d'un vincle immutable"))

(defmessage :errdbm
   (english "attempt to modify the binding of a dynamic variable")
   (french "tentative de modification de la liaison d'une variable dynamique")
   (catalan "intent de modificacio del vincle d'una variable dinamica"))

(defmessage :errflg
   (english "the form is too long")
   (french "la forme est trop longue")
   (catalan "la forma es massa llarga"))

(defmessage :errma
   (english "too many arguments")
   (french "trop d'arguments")
   (catalan "excessius arguments"))

(defmessage :errfa
   (english "too few arguments")
   (french "nombre d'arguments insuffisant")
   (catalan "manquen arguments"))

(defmessage :errnls
   (english "not lexical scope")
   (french "pas de portee lexicale")
   (catalan "sense abast lexic"))

;;;

(defmessage :meval
   (english "after evaluation of")
   (french "apres l'evaluation de")
   (catalan "despres de l'evaluacio de"))

(defmessage :min
   (english "error detected in")
   (french "erreur dans")
   (catalan "error a"))

;;;
```

```
;;;
;;; WALKERSTR:  Structures for the walker
;;;
;;; -----------------------------------------------------------------------
;;; This file is part of Le-Lisp version 16, developped by ILOG and INRIA.
;;; Inquiries to ILOG S.A.
;;;              2 Avenue Gallie'ni, BP 85,
;;;              94253 Gentilly Cedex, France.
;;;
;;; (c) 1989,1990 Le-Lisp is a trademark of INRIA.
;;; $Header: /nfs/work16/lelispv16/llib/RCS/walkerstr.ll,v 1.1 90/08/13 14:22:48 davis
xp Locker: valentin $
;;; -----------------------------------------------------------------------
;;;

(defstruct walker-data-type
  ;; WALKER-DATA-TYPE includes a list of pattern objects,
  ;; a list of context objects used to store relevant data (context),
  ;; and finally two fields indicating which is the current local call
  ;; (if there is one) and macroexpand behaviour
  ;; There is only one instance of this class :STATE, which is
  ;; a global record of the walker's state.
  patterns
  str-list
  (keywords '(key
              eval
              more
              fcall
              set
              label
              quote
              symbol
              optional
              variable
              mark))
  current-local-call
  macro-disabled)

(defstruct context
  ;; CONTEXT instances will be created to keep track of context
  ;; information
  form
  environment
  entry-type)

(defstruct patterns-type
  ;; each instance of PATTERNS-TYPE will be used to define a pattern.
  ;; SPECIAL-HANDLER? flag indicates that a special handling function
  ;; is provided for this pattern type.
  key
  pattern
  special-handler?)

(defstruct binding-type
  ;; each instance of BINDING-TYPE will be a binding in some
  ;; environment, which is a list of binding objects.
  name
  vbind
  type
  state
  args)
```

```
;;;
;;; WALK16:  WALKER functions and patterns for Le-lisp v16
;;;
;;; ---------------------------------------------------------------------
;;; This file is part of Le-Lisp version 16, developped by ILOG and INRIA.
;;; Inquiries to ILOG S.A.
;;;              2 Avenue Gallie'ni, BP 85,
;;;              94253 Gentilly Cedex, France.
;;;
;;; (c) 1989,1990 Le-Lisp is a trademark of INRIA.
;;;
;;; ---------------------------------------------------------------------
;;;


;;;
;;; WALKER
;;; ***********************
;;; special handling functions for Le-Lisp v16 special forms
;;; general pattern definition

;;; functions that manipulate environments

;;; each form is walked in a lexical environment
;;; deep representation is used to
;;; handle environments


(defun lambda-walker (form env walk-function)
  ;; LAMBDA-WALKER walks a lambda form and manipulates environments
  ;; lambda body is walked in a environment extended with lambda
  ;; arguments.arguments list may be a proper or improper list a
  ;; symbol and even &nobind which has a very particular
  ;; interpretation
  ;; arguments:
  ;;  a lambda form
  ;;  an environment
  ;;  the walk-function
  ;; result
  ;;  a walked lambda form
  (let* ((lambda-list (parse-names (cadr form)))
         (body (cddr form))
         (lambda-bindings (nobind-hook lambda-list))
         (new-body (walk-eval-pattern body
                                      '(more (eval))
                                      (extend-environment env
                                                          lambda-bindings)
                                      walk-function)))
    (when (not (nobind-binding-p lambda-bindings))
          (find-unused-objects lambda-bindings 'lambda))
    (set-current-local-call (create-binding-object () () 'functional-object
                                                         lambda-list))
    `(lambda ,(car lambda-list)
             ,@new-body)))

 (defun nobind-hook (lambda-list)
   ;;constructs lambda bindings
   ;;includes a hook for lambda forms with &nobind
   ;;as arguments list
   (cond ((eq (car lambda-list) '&nobind)
          (construct-bindings '(arg) () 'mark ()))
         (t (construct-bindings (lambda-list-type lambda-list)
                                () 'variable ()))))

 (defun nobind-binding-p (lambda-bindings)
   (eq (get-type (car lambda-bindings))
       'mark))
```

```
(defun extend-environment (env lambda-bindings)
  (if (nobind-binding-p lambda-bindings)
      (append env lambda-bindings)
      (append lambda-bindings env)))

(defun parse-names (names)
  (let ((lambda-list (construct-typed-lambda-list names)))
    (cond ((duplicated-argument (lambda-list-type lambda-list))
           (:error 'parse-names ':errlda names))
          (t lambda-list))))

;;;

;;; the lambda form returned has been walked
;;; and appropiate environments have been
;;; modified


(defun labels-walker (form env walk-function)
  ;;LABELS-WALKER special purpose handling function for labels forms
  (let* ((bindings (cadr form))
         (parsed-bindings (parse-bindings bindings))
         (names (mapcar #'car
                        parsed-bindings))
         (lambda-bodies (mapcar #'cddr
                                parsed-bindings))
         (body (cddr form))
         (lambda-lists (mapcar #'cadr parsed-bindings))
         (lambda-bindings-list (make-lambda-bindings lambda-lists))
         (labels-bindings (make-labels-bindings names lambda-lists))
         (new-lambda-definitions
          (walk-lambda-bodies lambda-bodies
                              lambda-bindings-list
                              (append labels-bindings env)
                              walk-function))
         (new-labels-bindings
          (update-labels-bindings labels-bindings
                                  new-lambda-definitions))
         (new-body (walk-eval-pattern body
                                      '(more (eval))
                                      (append new-labels-bindings env)
                                      walk-function)))

    (mapcar (lambda (lambda-bindings)
              (when (not (nobind-binding-p lambda-bindings))
                (find-unused-objects lambda-bindings 'labels-definition)))
            lambda-bindings-list)
    (find-unused-objects labels-bindings 'labels-body)
    `(labels ,(mapcar (lambda (name binding lambda-definition)
                        `(,name ,(cadr binding) ,@(cdr lambda-definition)))
                      names
                      bindings
                      new-lambda-definitions)
       ,@new-body)))


(defun walk-lambda-bodies (lambda-bodies lambda-bindings-list
                                         env walk-function)
  (mapcar (lambda (lambda-body lambda-bindings)
            `(,(mapcar #'get-name lambda-bindings)
              ,@(walk-eval-pattern lambda-body
                                   '(more (eval))
                                   (extend-environment env
                                                       lambda-bindings)
                                   walk-function)))
          lambda-bodies
          lambda-bindings-list))

(defun make-lambda-bindings (lambda-lists)
  (mapcar #'nobind-hook
          lambda-lists))


(defun update-labels-bindings (labels-bindings lambda-definitions)
  (mapcar (lambda (labels-binding lambda-definition)
            (set-vbind labels-binding
                       `(lambda ,(car lambda-definition)
                          ,@(cdr lambda-definition))))
          labels-bindings
          lambda-definitions)
  labels-bindings)



;;; labels-walker walks lambda bodies extending environments with
;;; names taken from lambda lists (vbinds are undefined) labels-walker
;;; also walks labels body extending environment with names whose
;;; vbind are the result of constructing lambda functions from
;;; lambda lists and lambda bodies

;;; labels functions names must be known in lambda bodies
;;; bodies are undefined but names and number of arguments
;;; are recorded in the environment
;;; so recursive references to local functions are allowed
;;; and we also check the number of arguments given in each
;;; local call

(defun make-labels-bindings (names lambda-lists)
  (construct-bindings names
                      ()
                      'functional-object
                      lambda-lists))

(defun flet-walker (form env walk-function)
  ;; arguments
  ;;  a flet form
  ;;  an environment
  ;;  the walk-function
  ;; results
  ;;  a walked flet form
  ;; The only difference between FLET-WALKER and LABELS-WALKER is that
  ;; lambda bodies are walked in an environment that in the second
  ;; case allows recursive and mutually recursive references
  (let* ((bindings (cadr form))
         (parsed-bindings (parse-bindings bindings))
         (names (mapcar #'car parsed-bindings))
         (lambda-bodies (mapcar #'cddr parsed-bindings))
         (lambda-lists (mapcar #'cadr parsed-bindings))
         (body (cddr form))
         (lambda-bindings-list (make-lambda-bindings lambda-lists))
         (new-lambda-definitions
          (walk-lambda-bodies lambda-bodies
                              lambda-bindings-list
                              env
                              walk-function))
         (flet-bindings (make-flet-bindings names
                                            new-lambda-definitions
                                            lambda-lists))
         (new-body (walk-eval-pattern body
                                      '(more (eval))
                                      (append flet-bindings env)
```

```
                                             walk-function)))

    (mapcar (lambda (lambda-bindings)
              (when (not (nobind-binding-p lambda-bindings))
                    (find-unused-objects lambda-bindings 'flet-definition)))
            lambda-bindings-list)
    (find-unused-objects flet-bindings 'flet-body)
    `(flet ,(mapcar (lambda (name binding lambda-definition)
                      `(,name ,(cadr binding) ,@(cdr lambda-definition)))
                    names
                    bindings
                    new-lambda-definitions)
           ,@new-body)))

(defun parse-bindings (bindings)
  (if (listp bindings)
      (internal-parse-bindings bindings ())
      (:error 'parse-bindings ':errlbflsbs bindings)))


(defun internal-parse-bindings (bindings parsed-bindings)
  (let* ((binding ((lambda (binding)
                     (if (listp binding)
                         binding
                        (:error 'parse-bindings ':errlbflsb binding)))
                   (car bindings)))
         (lambda-list (construct-typed-lambda-list (cadr binding)))
         (name (car binding)))
    (cond ((null bindings) (reverse parsed-bindings))
          ((improper-list-p binding)
           (:error 'internal-parse-bindings ':erriil binding))
          ((not (variablep name))
           (:error 'internal-parse-bindings ':errlbflnsn name))
          ((duplicated-argument (lambda-list-type lambda-list))
           (:error 'internal-parse-bindings ':errlbflda lambda-list))
          (t (internal-parse-bindings (cdr bindings)
                                      (cons `(,name
                                              ,lambda-list
                                              ,@(cddr binding))
                                            parsed-bindings))))))


(defun duplicated-argument (lambda-list)
  (let ((name (car lambda-list)))
    (cond ((atom lambda-list) ())
          ((not (variablep name))
           (:error 'duplicated-argument ':errlnsn name))
          ((memq name (cdr lambda-list)) t)
          (t (duplicated-argument (cdr lambda-list))))))

(defun make-flet-bindings (names lambda-definitions lambda-lists)
  (mapcar (lambda (name lambda-definition lambda-list)
            (create-binding-object name
                                   `(lambda ,@lambda-definition)
                                   'functional-object
                                   lambda-list))
          names
          lambda-definitions
          lambda-lists))

(defun block-walker (form env walk-function)
  (let* ((body (cddr form))
         (label (cadr form))
         (block-bindings (if (symbolp label)
                             (construct-bindings `(,label) () 'label ())
                           (:error 'block-walker ':errbns label)))
```

```
        (new-body (walk-eval-pattern body
                                     '(more (eval))
                                     (append block-bindings env)
                                     walk-function)))
    `(block ,label
       ,@new-body)))

(defun dynamic-let-walker (form env walk-function)
  ;; special handling function for dynamic-let forms
  (let* ((bindings (parse-let-bindings (cadr form)))
         (body (cddr form))
         (names (mapcar #'(lambda (binding)
                            (cond ((variablep binding)  binding)
                                  ((not (variablep (car binding)))
                                   (:error 'dynamic-let-walker
                                           ':errdlnsn (car binding)))
                                  (t (car binding))))
                        bindings))
         (new-names (if (duplicated-argument names)
                        (:error 'dynamic-let-walker ':errdldn names)
                      names))
         (new-bindings
          (mapcar
           #'(lambda (binding)
               (if (symbolp binding)
                   binding
                 `(,(car binding)
                   ,(real-walker (cadr binding) 'eval env walk-function))))
           bindings))
         (new-body
          (walk-eval-pattern body
                             '(more (eval))
                             (remove-dynamic-names new-names env ())
                             walk-function)))
    `(dynamic-let ,new-bindings
       ,@new-body)))

(defun parse-let-bindings (bindings)
  (cond ((not (listp bindings))
         (:error 'parse-let-bindings ':errdlibs bindings))
        ((internal-parse-let bindings) bindings)))

(defun internal-parse-let (bindings)
  (let ((binding (car bindings)))
    (cond ((null bindings) t)
          ((variablep binding) (internal-parse-let (cdr bindings)))
          ((and (consp binding) (not (improper-list binding)))
           (internal-parse-let (cdr bindings)))
          (t (:error 'internal-parse-let ':errdlib binding)))))

(defun remove-dynamic-names (names env new-env)
  (cond ((null env) (reverse new-env))
        ((memq (get-name (car env)) names)
         (remove-dynamic-names names (cdr env) new-env))
        (t (remove-dynamic-names names (cdr env)
                                 (cons (car env) new-env)))))

(defun find-unus d-objects (bindings where)
  ;; arguments
  ;;  a bindings list
  ;;  a where-flag
  ;;  \\ Not a where-wolf?
  ;;  \\ No fotis, de debo?
  ;; Follows bindings looking for unused objects and could raise an
  ;; appropiate error helped by WHERE
  (mapc
   #'(lambda (binding)
       (when (eq (get-state binding) 'unused)
         (let ((bind-name (get-name binding)))
           (case where
             (lambda
               (:warn 'find-unused-objects ':errluo bind-name))
             (labels-body
              (:warn 'find-unused-objects ':errlbuo bind-name))
             (labels-definition
              (:warn 'find-unused-objects ':errlduo bind-name))
             (flet-body
              (:warn 'find-unused-objects ':errfbuo bind-name))
             (flet-definition
              (:warn 'find-unused-objects ':errfduo bind-name))
             (tagbody
              (:warn 'find-unused-objects ':errtul bind-name))
             (t (:warn 'find-unused-objects 'unused-object where))))))
   bindings))


(defun tagbody-walker (form env walk-function)
  ;; Walks a TAGBODY form.  Constructs an unlabeled form and walks it
  ;; in an environment extended with labels found in the original
  ;; TAGBODY body.
  (let* ((body (cdr form))
         (labels-list (get-labels body ()))
         (unlabeled-body (get-forms body ()))
         (tagbody-bindings
          (construct-bindings labels-list () 'label ()))
         (new-unlabeled-body
          (walk-eval-pattern unlabeled-body
                             '(more (eval))
                             (append tagbody-bindings env)
                             walk-function)))
    (find-unused-objects tagbody-bindings 'tagbody)
    `(tagbody ,@(construct-new-labeled-body new-unlabeled-body
                                             body ()))))

(defun get-labels (body labels-list)
  (let ((sub-form (car body)))
    (cond ((null body) labels-list)
          ((symbolp sub-form)
           (if (memq sub-form (cdr body))
               (:error 'get-labels ':errtdl sub-form)
             (get-labels (cdr body) (cons sub-form labels-list))))
          (t (get-labels (cdr body) labels-list)))))

(defun get-forms (body forms-list)
  (let ((sub-form (car body)))
    (cond ((null body) forms-list)
          ((consp sub-form)
           (get-forms (cdr body) (cons sub-form forms-list)))
          (t (get-forms (cdr body) forms-list)))))

(defun construct-new-labeled-body (unlabeled-body body new-body)
  (cond ((null body) (reverse new-body))
        ((symbolp (car body))
         (construct-new-labeled-body unlabeled-body
                                     (cdr body)
                                     (cons (car body) new-body)))
        (t (construct-new-labeled-body (cdr unlabeled-body)
                                       (cdr body)
                                       (cons (car unlabeled-body)
                                             new-body)))))

(defun let-special-walker (form env walk-function)
  (let* ((expanded-form (car (macroexpand-1 form)))
         (lambda-form (real-walker expanded-form
                                   'eval
                                   env
                                   walk-function))
         (bindings (mapcar (lambda (name val)
                              (if (null val)
                                  name
                                  `(,name ,val)))
                        (cadar lambda-form)
                        (cdr lambda-form))))
    `(let ,bindings
          ,@(cddar lambda-form))))

(defun let*-special-walker (form env walk-function)
  (let* ((nbindings (length (cadr form)))
         (expanded-form (car (macroexpand-1 form)))
         (let-form (real-walker expanded-form
                                'eval
                                env
                                walk-function))
         (bindings-body
            (labels ((make-bindings (let-form bindings)
                       (cond ((eqn nbindings 1)
                              `(,(append bindings (cadr let-form))
                                ,@(cddr let-form)))
                             (t (setq nbindings (decr nbindings))
                                (make-bindings (caddr let-form)
                                               (append bindings
                                                       (cadr let-form)))))))
              (make-bindings let-form ()))))
      `(let* ,@bindings-body)))



(defun if-walker (form env walk-function)
  ;; IF-WALKER walks an IF form.
  ;; \\ IF could be walked using a pattern
  ;; \\ like (key eval eval optional eval)
  ;; arguments
  ;;  an if form
  ;;  an environment
  ;;  the walk-function
  ;; result
  ;;  a walked form
  (and (lt (length form) 3) (:error 'if-walker ':errfa (cdr form)))
  (let* ((test (cadr form))
         (action (caddr form))
         (alternative-actions (cdddr form))
         (new-test (real-walker test 'eval env walk-function))
         (new-action (real-walker action 'eval env walk-function))
         (new-alternative-actions
          (walk-alternative-actions alternative-actions
                                    env
                                    walk-function)))

    `(if ,new-test
         ,new-action
       ,@new-alternative-actions)))

(defun walk-alternative-actions (actions env walk-function)
  (cond ((null actions) ())
        ((gt (length actions) 1)
         (:warn ':erripn actions)
         `(,(real-walker `(progn ,@actions)
                          'eval
                        env
                        walk-function)))
        (t '(,(real-walker (car actions)
                           'eval
                           env
                           walk-function)))))

;;;
;;; patterns definition covers
;;; Le-lisp version 16 special forms language
;;;

(define-pattern 'lambda                  #'lambda-walker)
(define-pattern 'flet                    #'flet-walker)
(define-pattern 'labels                  #'labels-walker)
(define-pattern 'tagbody                 #'tagbody-walker)
(define-pattern 'block                   #'block-walker)
(define-pattern 'dynamic-let             #'dynamic-let-walker)
(define-pattern 'if                      '(key eval eval optional eval))
(define-pattern 'dynamic                 '(key symbol))
(define-pattern 'go                      '(key label))
(define-pattern 'return-from             '(key label eval))
(define-pattern 'function                '(key fcall))
(define-pattern 'unwind-protect          '(key eval eval))
(define-pattern 'progv                   '(key eval eval more (eval)))
(define-pattern 'multiple-value-prog1 '(key eval more (eval)))
(define-pattern 'multiple-value-call  '(key eval eval))
(define-pattern 'progn                   '(key more (eval)))
(define-pattern 'setq                    '(key set eval more (set eval)))
(define-pattern 'dynamic-setq            '(key symbol eval))
(define-pattern 'quote                   '(key quote))
(define-pattern 'catch                   '(key eval more (eval)))
(define-pattern 'throw                   '(key eval eval))

;;;
;;;arg is not really a special form (in fact don't know what arg is)
;;;but its definition is necessary to handle &nobind
;;;

(define-pattern 'arg '(mark optional eval))

;;; Messages
;;; greetings from Catalonia
;;; WALKER errors and warnings

(defmessage :errtdl
   (english "duplicated label in tagbody")
   (french "duplication d'etiquette dans tagbody")
   (catalan "etiqueta de tagbody duplicada"))

(defmessage :errgul
   (english "unknown label")
   (french "etiquette inconnue")
   (catalan "etiqueta desconeguda"))

(defmessage :errgns
   (english "non-symbolic expression")
   (french "expression non symbolique")
   (catalan "expressio no simbolica"))

(defmessage :errbns
   (english "non-symbolic block name")
   (french "nom de bloc non symbolique")
   (catalan "nom de bloc no simbolic"))

(defmessage :errluo
   (english "unused variable in lambda")
   (french "variable non utilisee dans lambda")
   (catalan "variable no utilitzada a lambda"))

(defmessage :errlbuo
   (english "unused function in labels")
   (french "fonction non utilisee dans labels")
   (catalan "funcio no utilitzada a labels"))

(defmessage :errlduo
   (english "unused argument in labels functions definition")
   (french "argument de fonction labels non utllise")
   (catalan "argument de funcio labels no utilitzat"))

(defmessage :errfduo
   (english "unused argument in flet functions definition")
   (french "argument de fonction flet non utillise")
   (catalan "argument de funcio flet no utilitzat"))

(defmessage :errfbuo
   (english "unused function in flet")
   (french "fonction non utilisee dans flet")
   (catalan "funcio no utilitzada a flet"))

(defmessage :errtul
   (english "unused label in tagbody")
   (french "etiquette non utilisee dans tagbody")
   (catalan "etiqueta no utlitzada a tagbody"))

(defmessage :erripn
   (english "if is not n-ary")
   (french "if n'est pas n-aire")
   (catalan "if no es n-ari"))

(defmessage :errlbflsbs
   (english "incorrect labels/flet bindings")
   (french "liaisons labels/flet incorrectes")
   (catalan "vincles labels/flet incorrectes"))

(defmessage :errlbflsb
   (english "incorrect labels/flet binding")
   (french "liaison labels/flet incorrecte")
   (catalan "vincle labels/flet incorrecte"))

(defmessage :errlbflnsn
   (english "non symbolic local function name")
   (french "nom de fonction locale non symbolique")
   (catalan "nom de funcio local no simbolic"))

(defmessage :errlbflsl
   (english "incorrect labels/flet lambda list")
   (french "liste lambda labels/flet incorrecte")
   (catalan "llista lambda labels/flet incorrecta"))

(defmessage :errrlbflda
   (english "duplicated argument in labels/flet lambda list")
   (french "duplication d'argument dans lambda liste labels/flet")
   (catalan "argument duplicat a llista lambda labels/flet"))

(defmessage :errlda
   (english "duplicated argument in lambda list")
   (french "duplication d'argument dans lambda liste")
   (catalan "argument duplicat a llista lambda"))

(defmessage :errlnsn
   (english "non symbolic lambda argument")
   (french "argument lambda non symbolique")
   (catalan "argument lambda no simbolic"))

(defmessage :errlsl
   (english "incorrect lambda list")
   (french "liste lambda incorrecte")
   (catalan "llista lambda incorrecta"))

(defmessage :errdlnsn
   (english "non symbolic name in dynamic-let bindings")
   (french "nom non symbolique dans dynamic-let")
   (catalan "nom no simbolic a dynamic-let"))

(defmessage :errdlibs
   (english "incorrect dynamic-let bindings")
   (french "liaisons de dynamic-let incorrectes")
   (catalan "vincles de dinamic-let incorrectes"))

(defmessage :errdlib
   (english "incorrect dynamic-let binding")
   (french "liaison de dynamic-let incorrecte")
   (catalan "vincle de dynamic-let incorrecte"))

(defmessage :errdldn
   (english "duplicated name in dynamic-let bindings")
   (french "duplication de nom dans les liaisons de dynamic-let")
   (catalan "nom duplicat als vincles de dynamic-let"))

(defmessage :errdnsn
   (english "non symbolic argument in dynamic")
   (french "argument non symbolique dans dynamic")
   (catalan "argument no simbolic a dynamic"))

(defmessage :erriil
   (english "illegal improper list")
   (french "liste impure illegale")
   (catalan "llista impropia il.legal"))

;;;
```

```
                    ll                          t         i
                     l                          t
                     l
v     v   aaaa       l      eeee    n nnn    ttttt     ii     n nnn
v     v       a      l     e    e   nn   n     t        i     nn   n
v     v   aaaaa      l     eeeeee   n    n     t        i     n    n
 v   v   a     a     l     e        n    n     t        i     n    n
  v v    a    aa     l     e    e   n    n     t  t     i     n    n
   v      aaaa a    lll     eeee    n    n      tt     iii    n    n



                    ll      k        EEEEEEE                             ll       ll
                     l      k        E                                    l        l
                     l      k        E                                    l        l
w     w   aaaa       l      k   k    E        u    u                      l        l
w  w  w       a      l      k  k     EEEEEE   u    u                      l        l
w  w  w   aaaaa      l      k k      E        u    u                      l        l
w  w  w  a     a     l      kk k     E        u    u                      l        l
w  w  w  a    aa     l      k   k    E        u   uu      ..              l        l
 ww ww    aaaa a    lll     k    k   EEEEEEE   uuu u      ..             lll      lll



                              i

v     v   aaaa   v     v     ii     n nnn
v     v       a  v     v      i     nn   n
v     v   aaaaa  v     v      i     n    n
 v   v   a     a  v   v       i     n    n
  v v    a    aa   v v        i     n    n
   v      aaaa a    v        iii    n    n



                                                    Job:  walkEu.ll
                                                    Date: Sun Sep 16 13:50:59 1990
```

```
;;;
;;; some Eulisp special forms
;;;

;;;let/cc creates an immutable binding
;;;let/cc body is walked in an environment
;;;extended with a new immutable binding
;;;(see error :ERRIBM, WALKER module)

(defun let/cc-walker (form env walk-function)
   (let* ((variable ((lambda (var)
                        (if (symbolp var) var
                            (walk-error ':errsiv var))) (cadr form)))
          (let/cc-binding (create-binding-object variable ()
                                                  'immutable ()))
          (new-body (walk-eval-pattern (cddr form)
                                        '(more (eval))
                                        (cons let/cc-binding env)
                                        walk-function)))
      (find-unused-objects `(,let/cc-binding) 'let/cc)
      `(let/cc ,variable
               ,@new-body)))

;;;with-handler-walker
;;;the first argument must be a function ?

(defun with-handler-walker (form env walk-function)
  (let ((func (cadr form))
        (body (cddr form)))
    `(with-handler ,(cond ((atom func)
                           (and (lookup-call-p func env)
                                (neqn (get-args (get-current-local-call))
                                      2)
                                (:error 'with-handler-walker ':errhfa func))
                           (car (funcall walk-function
                                         func
                                         'fcall
                                         env)))
                          ((eq (car func) 'lambda)
                           (real-walker func 'eval env walk-function))
                          (t (:error 'with-handler-walker ':errihf funv)))
                  ,(if (null body)
                       (:error 'with-handler-walker ':errfa form)
                       (real-walker (car body) 'eval env walk-function)))))


;;;labels is not considered a special form in Eulisp as
;;;in Le-Lisp v16
;;;in Eulisp labels is a macro :
;;;
;;; (labels ((<var1> <lambda-list1> <body1>)
;;;          (<var2> <lambda-list2> <body2>) ... )
;;;   <body-labels>)
;;;
;;;    [macroexpansion] ==>
;;;
;;; (lambda (<var1> <var2> ... )
;;;    (setq <var1> (lambda <lambda-list1> <body1>))
;;;    (setq <var2> (lambda <lambda-list2> <body2>))
;;;     ...
;;;    <body-labels>) () () ... )
;;;
;;;   (!!!!!!)
```

```
;;;patterns definition
;;;

(define-pattern 'let/cc #'let/cc-walker)
(define-pattern 'dynamic-let #'dynamic-let-walker)
(define-pattern 'lambda #'lambda-walker)
(define-pattern 'with-handler #'with-handler-walker)

;;;in Eulisp lambda-list could be a single variable bound to the number
;;;of arguments of a multiple argument object
;;;this binding has lexical scope

(define-pattern 'progn '(key more (eval)))
(define-pattern 'if '(key eval eval optional eval))
(define-pattern 'multiple-argument-values
       '(key variable optional eval optional eval))
(define-pattern 'multiple-argument-ref '(key variable eval))
(define-pattern 'dynamic '(key symbol))
(define-pattern 'dynamic-setq '(key symbol eval))
(define-pattern 'quote '(key quote))
(define-pattern 'apply '(key fcall more (eval)))

;;;

(defmessage :errihf
  (english "invalid handler function")
  (french "handler incorrect")
  (catalan "handler incorrecte"))

;;;

(defmessage :errhfa
  (english "a handler function needs two arguments")
  (french "une handlerfunction a besoin de deux arguments")
  (catalan "una handler function necessita dos arguments"))

;;;
```