4.2 Démonstrateurs de théorèmes propositionnels pour G, G*, S4Grz, IL, KD?, KD?*.

```
;-*- Mode:Common-Lisp; Package:USER; Base:10 -*-
;
; Il s'agit de la traduction en LISP de l'algorithme basé sur la méthode des tableaux
; (Smullyan, Beth, Herbrand, Gentzen) présenté et prouvé correct dans Boolos 1979.
;
; Développé sur Explorer et Macintosh.
```

4.2.1 partie propositionnelle

```
(defun rule-and (l)
  (cond ((null l) nil)
        ((atom (car l))(cons (car l)(rule-and (cdr l))))
        ((equal (caar l) '&)
         (cons (cons '* (car l))
               (append (cdr l) (cdar l))))
        (t (cons (car l) (rule-and (cdr l))
                 )) ))

(defun mcons (x ll)
  (cond ((null ll) nil)
        (t (cons (cons x (car ll))
                 (mcons x (cdr ll))))
       ))

(defun rule-or (l)
  (cond ((null l) (list nil))
        ((atom (car l))(mcons (car l)
                              (rule-or (cdr l))))
        ((equal (caar l) 'v)
         (mcons (cons '* (car l))
                (list (append (cdr l) (list (cadar l)) )
                      (append (cdr l) (list (caddar l))) )))
        (t (mcons (car l)
                  (rule-or (cdr l))))
       ))

(defun simp-not-imply (l)
        (cond ((null l)nil)
              ((atom (car l))(cons (car l) (simp-not-imply (cdr l))))
              ((atom (cadar l))(cons (car l)(simp-not-imply (cdr l))))
              ((and (equal (caar l) '-)
                    (equal (caadar l) '->))
               (cons (list '& (cadr (cadar l)) (list '-
                                                     (caddr (cadar l))))
                     (simp-not-imply (cdr l))))
              (t (cons (car l)
                       (simp-not-imply (cdr l))))
             ))

(defun simp-not-or (l)
  (cond ((null l)nil)
        ((atom (car l))(cons (car l)(simp-not-or (cdr l))))
        ((atom (cadar l))(cons (car l)(simp-not-or (cdr l))))
        ((and (equal (caar l) '-)
              (equal (caadar l) 'v))
         (cons (list '& (list '- (cadr (cadar l))) (list '- (caddr (cadar l))))
               (simp-not-or (cdr l))))
        (t (cons (car l)
                 (simp-not-or (cdr l))))
       ))
```

```
(defun simp-not-and (l)
  (cond ((null l)nil)
        ((atom (car l))(cons (car l)(simp-not-and (cdr l))))
        ((atom (cadar l))(cons (car l)(simp-not-and (cdr l))))
        ((and (equal (caar l) '-)
              (equal (caadar l) '&))
         (cons (list 'v
                     (list '- (cadr (cadar l)))
                     (list '- (caddr (cadar l))) )
               (simp-not-and (cdr l))))
        (t (cons (car l)
                 (simp-not-and (cdr l))))
       ))

(defun simp-imply (l)
  (cond ((null l)nil)
        ((atom (car l))(cons (car l)(simp-imply (cdr l))))
        ((equal (caar l) '->)
         (cons (list 'v
                     (list '- (cadar l))
                     (caddar l))
               (simp-imply (cdr l))))
        (t (cons (car l)
                 (simp-imply (cdr l))))
       ))

(defun double-negation (f)
  (and (not (atom f))
       (not (atom (cadr f)))
       (equal (car f) '-)
       (equal (caadr f) '-))
 )

(defun simp-not (l)
  (cond ((null l) nil)
        ((double-negation (car l))
         (cons (cadar (cdar l))
               (simp-not (cdr l))))
        (t (cons (car l)(simp-not (cdr l)) ))
       ))

(defun simp-not-di&tr (l)
  (cond ((null l)nil)
        ((atom(car l))(cons (car l)(simp-not-di&tr (cdr l))))
        ((equal (caar l) '-)(cond((equal (cadar l) 'tr)(cons '(& p (- p)) (simp-not-di&tr (cdr l))))
                                 ((equal (cadar l) 'fa)(cons '(V p (- p)) (simp-not-di&tr (cdr l))))
                                 ((atom (cadar l))(cons (car l)(simp-not-di&tr (cdr l))))
                                 ((equal (caadar l) '<->)(cons (list 'V (list '&
                                                                      (second(second(first l)))
                                                                      (list '- (third(second(first l)))))
                                                                                (list '&
                                                                      (list '- (second(second(first l))))
                                                                      (third(second(first l)))))
                                                                                (simp-not-di&tr (cdr l))))
                                 (t(cons(car l)(simp-not-di&tr (cdr l))))))
         (t(cons(car l)(simp-not-di&tr (cdr l))))
       ))
```

```
(defun simp-doub-imp (l)
  (cond((null l)nil)
       ((atom (car l))(cons (car l)(simp-doub-imp (cdr l))))
       ((equal (caar l) '<->)
        (append (list(list '->
                           (cadar l)
                           (caddar l))
                     (list '->
                           (caddar l)
                           (cadar l)))
                (simp-doub-imp (cdr l))))
       (t(cons(car l)(simp-doub-imp(cdr l))))
      ))

(defun simp-truth (l)
  (cond((null l)nil)
       ((equal (car l) 'tr)(cons '(V p (- p)) (simp-truth (cdr l))))
       ((equal (car l) 'fa)(cons '(& p (- p)) (simp-truth (cdr l))))
       (t(cons (car l)(simp-truth (cdr l))))
      ))

; plus tard faire les simp pour les formules et n'utiliser qu'une boucle de recursion

(defun simp-chemin (l)

(simp-not
 (simp-imply
  (simp-not-or
   (simp-not-and
    (simp-not-imply
     (simp-not-di&tr
      (simp-doub-imp
       (simp-truth l))))))))   )

(defun simp (ll)
  (cond ((null ll)nil)
        (t (cons (simp-chemin (car ll))
                 (simp (cdr ll))))
       ))

(defun submember (x l)
  (cond ((null l)nil)
        ((atom (car l))(submember x (cdr l)))
        ((equal (caar l)
                x)t)
        (t(submember x (cdr l)))
       ))

(defun submember-arbre (x ll)
  (cond ((null ll)nil)
        ((submember x (car ll)) t)
        (t(submember-arbre x (cdr ll)))
       ))

(defun rule-and-arbre (ll)
  (cond ((null ll)nil)
        ((submember '& (car ll))
         (cons (rule-and (car ll))
               (rule-and-arbre (cdr ll)) ))
        (t(cons (car ll) (rule-and-arbre (cdr ll))))
       ))
```

```
(defun rule-or-arbre (ll)
  (cond ((null ll)nil)
        ((submember 'v (car ll))
         (append (rule-or (car ll))
                 (rule-or-arbre (cdr ll))))
        (t(cons (car ll)(rule-or-arbre (cdr ll))))
       ))

(defun ferme-aux (l1 l2)
  (cond ((null l1)nil)
        ((atom (car l1))(ferme-aux (cdr l1) l2))
        ((equal (caar l1) '-)(or (member (cadar l1) l2) (ferme-aux (cdr l1) l2)))
        (t(ferme-aux (cdr l1) l2))
       ))


(defun ferme (l)
  (ferme-aux l l)
 )

(defun supprime (ll)
  (cond ((null ll)nil)
        ((ferme (car ll))(supprime (cdr ll)))
        (t(cons (car ll)
                (supprime (cdr ll))))
       ))

(defun del*-chemin (l)
  (cond ((null l)nil)
        ((and (not (atom (car l)))
              (equal (caar l) '*))
         (del*-chemin (cdr l)))
        (t(cons (car l) (del*-chemin (cdr l))))
       ))

(defun del*-arbre (ll)
  (cond ((null ll)nil)
        (t(cons (del*-chemin (car ll))
                (del*-arbre (cdr ll))))
       ))

(defun chemin-final (l)
  (cond ((null l)t)
        ((atom (car l))(chemin-final (cdr l)))
        ((and (equal (caar l) '-)(atom (cadar l)))(chemin-final (cdr l)))
        (t nil)
       ))

(defun chemin-final-prop-modal (l)
  (cond ((null l)t)
        ((atom (car l))(chemin-final-prop-modal (cdr l)))
        ((and (equal (caar l) '-)
              (or (atom (cadar l))
                  (equal (caadar l) 'bw)))(chemin-final-prop-modal (cdr l)))
        ((equal (caar l) 'bw)(chemin-final-prop-modal (cdr l)))
        ((equal (caar l) 'f#)(chemin-final-prop-modal (cdr l)))
       ))

(defun arbre-final (ll)
  (cond ((null ll)t)
        ((chemin-final (car ll))(arbre-final (cdr ll)))
        (t nil)))
```

```
(defun arbre-final-prop-modal (ll)
  (cond ((null ll)t)
        ((chemin-final-prop-modal (car ll))(arbre-final-prop-modal (cdr ll)))
        (t nil)))

(defun pl (ll)
  (cond ((null ll)(terpri)(terpri)(quote done!))
        (t (print (car ll))
           (pl (cdr ll)))
       ))

(defun pp (ll)
  (cond (t (terpri)(terpri)(pl ll))
       ))

(defun boucle-prop (ll)
  (cond ((null ll)nil)
        ((arbre-final-prop-modal ll) (supprime ll))
        (t (boucle-prop (del*-arbre(supprime(rule-and-arbre(supprime
            (rule-or-arbre (simp ll))))))))
       ))

;
;
;
```

4.2.2 partie modale

```
;
;
;
(defun last-cons (x l)
  (append l (cons x nil))
 )

(defun rule-not-bew (l)
  (cond((null l)nil)
       ((and(not(atom(car l)))
            (equal(caar l) '-)
            (not(atom(cadar l)))
            (equal(caadar l) 'bw))
        (cons(cons '* (car l))
             (last-cons (list 'f#
                        (list(list
                             (list '-
                                   (cadr (cadar l)))
                             (list 'bw
                                   (cadr(cadar l)))
                            ))
                      )
                      (rule-not-bew (cdr l))
                     )
          ))
       (t(cons(car l)(rule-not-bew (cdr l))))
      ))

(defun rule-not-bew-arbre (ll)
  (cond((null ll)nil)
       (t(cons (rule-not-bew (car ll))
               (rule-not-bew-arbre (cdr ll))) )
      ))
```

```
(defun rule-bew (l)
  (cond((null l)nil)
       ((and (not(atom(car l)))
             (equal(caar l) 'bw)
             (not(without-window l)))
        (cons(cons '* (car l))(add-window (cadar l) (rule-bew (cdr l)))))
        (t(cons (car l)
                (rule-bew (cdr l))))
     ))

(defun m-last-cons (a ll)
  (cond ((null ll)nil)
        (t(cons (last-cons a (car ll)) (m-last-cons a (cdr ll))))
       ))

(defun add-window (a l)
  (cond((null l) nil)
       ((atom(car l))(cons (car l)
                           (add-window a (cdr l))))
       ((equal (caar l) 'f#)
        (cons (list 'f#
                    (m-last-cons (list 'bw a)
                                 (m-last-cons a (cadar l)))
                    )
              (add-window a (cdr l)) ))
       (t(cons (car l)
               (add-window a (cdr l))
              ))
     ))

(defun rule-bew-arbre (ll)
  (cond ((null ll)nil)
        (t(cons (rule-bew (car ll))
                (rule-bew-arbre (cdr ll))) )
       ))

(defun mr (ll)
  (rule-bew-arbre (rule-not-bew-arbre ll))
  )

(defun dmb (ll)
  (del*-arbre (mr (boucle-prop ll)))
  )

(defun member-fenetre-vide-chemin (l)
  (cond ((null l) nil)
        ((atom(car l))(member-fenetre-vide-chemin (cdr l)))
        ((and (equal (caar l) 'f#)
              (equal (cadar l) nil)) t)
        (t(member-fenetre-vide-chemin (cdr l)))
       ))

(defun supprime-chemin-avec-fenetre-vide (ll)
  (cond ((null ll) nil)
        ((member-fenetre-vide-chemin (car ll))
         (supprime-chemin-avec-fenetre-vide (cdr ll)))
        (t(cons (car ll)
                (supprime-chemin-avec-fenetre-vide (cdr ll))))
       ))

(defun nettoie-arbre (ll)
  (cond ((null ll)nil)
        (t(cons (nettoie-chemin (car ll))
                (nettoie-arbre (cdr ll))))       ))
```

```
(defun nettoie-chemin (l)
  (cond((null l)nil)
       ((atom (car l))(cons(car l)(nettoie-chemin (cdr l))))
       ((equal(caar l) 'bw)(nettoie-chemin (cdr l)))
       (t(cons(car l)(nettoie-chemin(cdr l))))
      ))

(defun boucle-mod-chemin (l)
  (cond((null l)nil)
       ((atom(car l))(cons (car l)(boucle-mod-chemin (cdr l))))
       ((equal (caar l) 'f#)
        (cons (list 'f#
                    (bma (cadar l)))
              (boucle-mod-chemin (cdr l)) ))
       (t (cons (car l)(boucle-mod-chemin (cdr l))))
      ))


;  boucle-mod-arbre a ete modernisee par bma

; (defun boucle-mod-arbre (ll)
; (cond((equal (boucle-prop ll)
;              (mr (boucle-prop ll)))
;        (nettoie-arbre (boucle-prop ll)))
;       (t (supprime-chemin-avec-fenetre-vide
;            (apply-boucle-mod-arbre
;                (del*-arbre
;              (mr (boucle-prop ll))))))
;       ))

(defun bma (ll)
  (let*((x (boucle-prop ll))
        (y (mr x)))
    (cond ((equal x y)(nettoie-arbre x))
          (t(supprime-chemin-avec-fenetre-vide
             (apply-boucle-mod-arbre
              (del*-arbre y))))
        ))
 )

(defun apply-boucle-mod-arbre (ll)
  (cond ((null ll)nil)
        (t(cons(boucle-mod-chemin (car ll))
               (apply-boucle-mod-arbre (cdr ll))))
       ))

(defun without-window (l)
  (cond ((null l) t)
        ((atom (car l))(without-window (cdr l)))
        ((equal (caar l) 'f#) nil)
        (t(without-window (cdr l)))
       ))
```

4.2.3 G, G*, S4Grz, IL, KD?, KD?*

```
(defun g (f)
  (bma (list(list (list '- f))))
 )

(defun tf (ll)
  (cadr (caar ll))
 )

;                       G*
;
;
;
(defun subew-aux (f mem)
  (cond ((atom f)mem)
        ((null f)mem)
        ((2-atom (car f))(union (subew-aux (second f) mem)
                                (subew-aux (third f) mem)))
        ((equal(car f) '-)(subew-aux (second f) mem))
        ((equal(car f) 'bw)(subew-aux (second f)(cons f mem)))
        (t mem)
        ))

(defun subew (f)
  (subew-aux f nil)
 )

(defun 2-atom (a)
  (member a '(-> <-> & V))
 )

;  subew donne la liste de toutes les sous-formules de f de la forme (bw X) .

(defun make-conj (l)
  (cond((null l)(list 'V 'ppp '(- ppp)))
       ((null(cdr l))(car l))
       (t(list '&
               (car l)
               (make-conj (cdr l))))
      ))

(defun list-reflection (l)
  (cond((null l)nil)
       (t (cons (list '->
                      (car l)
                      (cadar l))
                (list-reflection (cdr l))))
      ))

;  list-reflection ((bw 1)(bw 2) ...) --> ((->(bw 1) 1)(->(bw 2) 2) ...)

(defun sol (f)
   (list '->
         (make-conj (list-reflection (subew f)))
         f)
 )


(defun g* (f)
  (g (sol f))
 )
```

```
;                    S4Grz
;
(defun BGKM (f)
  (cond((atom f)f)
       ((equal (car f) '-)(list '- (BGKM (cadr f))))
       ((2-atom (car f))(list (car f)
                         (BGKM (cadr f))
                         (BGKM (caddr f))))
       ((equal (car f) 'bw)(let ((x (BGKM (cadr f))))
                         (list '& (list 'bw x) x)))
      ))

(defun s (f)
  (g (BGKM f))
 )

;
;                    IL
;

; il s'agit d'une translation due à McKinsey & Tarski, celle utilisée par
; Goldblatt c168 page 42.

(defun make-il (f)
  (cond ((atom f) (list 'bw f))
        ((equal (car f) '&)
         (list '& (make-il (cadr f)) (make-il (caddr f)) ))
        ((equal (car f) 'v)
         (list 'v (make-il (cadr f)) (make-il (caddr f)) ))
        ((equal (car f) '-) (list 'bw
                              (list '- (make-il (cadr f)))))
        ((or (equal (car f) '->) (equal (car f) '<->))
         (list 'bw
               (list (car f)
                     (make-il (cadr  f))
                     (make-il (caddr  f)) )))
      )
 )

(defun il (f)
 (s4grz (make-il f))
)


; problème (non arrêt ?) avec (ilip '(- - - p <-> - p))

#|
; translation de Gödel 33 (erreur, progrès, encore erreur sur --p->p)

(defun G33 (f)                ; ancien make-il-2
  (cond ((atom f)  f)           ; plus efficace qu'avec (list 'bw f), mais équivalent
        ((equal (car f) '&)
         (list '&  (G33 (cadr f))  (G33 (caddr f)) ))
        ((equal (car f) 'v)
         (list 'v (list 'bw (G33 (cadr f))) (list 'bw (G33 (cadr f))) ))
        ((equal (car f) '-)  (list 'bw (list '-  (G33 (cadr f)))))
        ((or (equal (car f) '->) (equal (car f) '<->))
         (list 'bw
               (list (car f)
                     (G33 (cadr  f))
                     (G33 (caddr  f)) )))
      )
 )
```

```
(defun il (f)
 (s4grz (G33 f))
)

? (G33 (ip '(p v - p)))
(V (BW P) (BW (- P)))
?
IL
? (ilip '(p v - p))
(((F\# ((P))) (F\# (((- P))))) (P (F\# (((- P))))) ((- P) (F\# ((P)))))
? (pl (ilip '(p v - p)))


((F\# ((P))) (F\# (((- P)))))

(P (F\# (((- P)))))

((- P) (F\# ((P))))
NIL
? (ilip '(- - - p <-> - p))
NIL


|#

;============

(defun pl (l)
  (cond ((null l) nil)
        (t (terpri) (print (car l)) (pl (cdr l)))
       )
 )

;
;                    KD?
;
(defun deon (f)
  (cond((atom f)f)
       ((equal (car f) '-)(list '- (deon (cadr f))))
       ((2-atom (car f))(list (car f)
                         (deon (cadr f))
                         (deon (caddr f))))
       ((equal (car f) 'bw)(let ((x (deon (cadr f))))
                         (list '& (list 'bw x)
                                  (list '- (list 'bw (list '- x))))))
      ))

(defun KD? (f)
  (g (deon f))
 )
```

4.2.4 parseurs

```
;-------------------------------------------------------------------------------
;
; IP  tranforme expression infixée en préfixée.
;
;-------------------------------------------------------------------------------



; (-> p ( -(bw(- p)) )) doit être bicalculablement associé à    "p -> - B - p"
; ou de suite "p -> C p".
; ordre de précédence:
```

```
;       1) <->
;       2) ->
;       3) <-
;       4) v
;       5) &
;       6) -
  (setq précédence '((<-> 6)(-> 5)(<- 4)(v 3)(& 2)))

  (defun précède (x y)
   (>= (value x précédence) (value y précédence))
  )

  (defun value (x env)
   (cond ((null env) 0)
         ((equal (caar env) x) (cadar env))
         (t (value x (cdr env)))
  )
 )
(defun val (x)
 (value x précédence)
)
;  string -> arbre
;  on lui donne

(setq L8 '(- p & q -> - B p v q))

;   on va conser le symbole d'ordre le plus haut sur list (ce programme
;   appliqué à tout ce qui précède ce symbole, ce programme appliqué à
;   tout ce qui suit.
;   Et autant le faire en une passe.
;
;    p & q v r    ====>    (v (& p q) r)
;    p v q & r    ====>    (v p (& q r))
;     règle = se donner (comme argument) au plus fort...
;  mais d'abord en deux passes

(defun higher-symbol (L)
    (cond ((null L) 0)
          ((précède (car L) (setq x (higher-symbol (cdr l)))) (car l))
          (t x)
 )
)
;  à transformer plus tard en lambda ou let convenable.

(setq L1 '(p & q v r))
(setq L2 '(p -> q v r))
(setq L3 '((- p) v q v r))
(setq L4 '(- (p & q)))
(setq L5 '(- p & q))
(setq L6 '(- - p -> q))
(setq L7 '(Bw (Bw p -> p) -> Bw p))

(defun tra (E)
  (cond ((atom E)  E)
        ((null (cdr E)) (tra (car E)))
        (t (list  (higher-symbol E)  (tra (avant (higher-symbol E) E))
                          (tra (après (higher-symbol E) E)) )  ) ) )

(defun tra2 (E)
 (let ((h (higher-symbol E)))
  (cond ((atom E)  E)
        ((null (cdr E)) (tra2 (car E)))
        (t (list h (tra2 (avant h E))
                          (tra2 (après h E)) )  ) ) )            )
```

```
;  ip2 utilise tra2 qui est la même chose que tra excepté qu' il utilise un let
;  (et ça ne marche pas !..., j'y reviendrai quant j'aurai trouvé le manuel.


(defun suppressnil (l)
 (cond ((null l) nil)
       ((equal (car l) nil) (suppressnil (cdr l)))
       ((listp (car l)) (cons (suppressnil (car l)) (suppressnil (cdr l))))
       (t (cons (car l) (suppressnil (cdr l))))
 )
)
;  pas très beau...

(defun ip (l)
 (suppressnil (tra l))
)
(defun ip2 (l)
 (suppressnil (tra2 l))
)

(defun avant (x l)
 (cond ((null l) nil)
       ((equal (car l) x) nil)
       (t (cons (car l) (avant x (cdr l)))))
)

(defun après (x l)
 (cond ((null l) nil)
       ((equal (car l) x) (cdr l))
       (t (après x (cdr l))))
)

; (higher-symbol l1)
; (higher-symbol l2)
; (avant (higher-symbol l1) l1)
; (après (higher-symbol l2) l2)


; ------------------------------------------------------------------------
;
;  En une passe: (plus tard si je trouve le temps...)
;
; ------------------------------------------------------------------------
;   (defun transform (l)
;    (tra-aux l nil)
;   )
;
;   (defun tra-aux (l mem)
;    (cond ((null l) mem)
;          ((précède (car mem) (car l))
;            (tra-aux (cdr l) (add (car mem) mem)))
;          (t (tra-aux (cdr l) (add-last (car mem) mem)))
;    )
;   )
;
;   (defun add (x l)
;    (cond ((null l) (list x))
;          (t (list x mem))
;    )
;   )
;
```

```
;   (defun add-last (x l)
;    (cond ((null l) (list x))
;          ((null (cddr l)) (list (car l) (add-last x (cadr l))))
;          (t (list (car l) (cadr l) (add-last x (caaddr l))))
;    )
;   )

; ------------------------------------------------------------------------------
;
;  PI  transforme préfixe en infixe
;
; ------------------------------------------------------------------------------

; PI transforme par exemple (-> (B (-> (B P) P)) (B P)) en
;                           (B (B P -> P) -> B P)
;
; ------------------------------------------------------------------------------


;  On suppose toujours <->, &, v, binaire et -, b, ou n'importe quoi, unaire.

;  1) sans simplification

(defun dart (l)              ;  trad correspondant
  (cond ((atom l) l)
        ((member (car l) '(<-> -> v &))
         (cons (dart (cadr l))
               (cons (car l)
                     (cons (dart (caddr l)) nil))))
        ((atom (car l)) (list (car l) (dart (cadr l))))
        (t (cons (car l) (dart (cadr l))))
       )
 )

; ? r
; (-> (BW P) P)
; ? (dart r)
; ((BW P) -> P)
; ? (ip (dart r))
; (-> (BW P) P)




; ----------------------------------------------------------------
;
;              Gip  G*ip   S4Grzip    ilip      KDip
;
; ----------------------------------------------------------------

;  necessite G G* S4Grz, ainsi que PARSEUR


(defun gip (x)
 (g (ip x)))

(defun g*ip (x)
 (g* (ip x)))

(defun S4Grzip (x)
  (S4Grz (ip x)))

(defun ilip (x)
  (il (ip x)))
```

```
(defun kdip (x)
  (kd? (ip x)))

(defun make-t-ip (x)
  (dart (make-t (ip x)))
  )

(defun make-d-ip (x)
  (dart (make-d-ip (ip x)))
  )

#|

? (s4grzip '(p <-> p))
NIL
? (g*ip '(p <-> p))
NIL
? (gip '(p <-> p))
NIL
? (gip '(p <-> q))
((P (- Q)) ((- P) Q))
? (g*ip '(p <-> q))
((PPP P (- Q)) (PPP (- P) Q) ((- PPP) P (- Q)) ((- PPP) (- P) Q)) ; ppp?
? (s4grzip '(p <-> q))
((P (- Q)) ((- P) Q))


;  gip marche sur b140 !!!  (contrairement à mes rumeurs !)

? b140
(-> (& (BW (-> (- (BW (- (-> PP PP)))) (& (- (BW P)) (- (BW (- P)))))) (& (BW (->
 (- (BW P)) P)) (- (BW (- (- (BW (- (-> PP PP))))))))) (& (& (& (BW (-> (- (BW (-
 (-> PP PP)))) P)) (- (BW (-> P (- (BW (- (-> PP PP)))))))) (- (BW (-> (- P) (- (
BW (- (-> PP PP)))))))) (- (BW (-> (- (BW (- (-> PP PP)))) (- P))))))
? (equal b140 (ip (dart b140)))
T
? (gip (dart b140))
NIL

? (ilip '(p -> (q -> p)))
NIL
? (ilip '(q -> (p v q)))
NIL
? (ilip '(q -> (p & q)))
(((F\# (((BW (-> (& (BW Q) Q) (& (& (BW P) P) (& (BW Q) Q)))) (BW Q) Q (- P)))))
(Q (F\# (((- P) Q)))) ((BW Q) Q (- P)))
? (length (ilip '(q -> (p & q))))
3
? (car (ilip '(q -> (p & q))))
((F\# (((BW (-> (& (BW Q) Q) (& (& (BW P) P) (& (BW Q) Q)))) (BW Q) Q (- P)))))
? (cadr (ilip '(q -> (p & q))))
(Q (F\# (((- P) Q))))
? (ilip '(p v (- p)))
(((F\# ((P))) (F\# (((- P))))) ((- P) (F\# ((P)))))
? (car (ilip '(p v (- p))))
((F\# ((P))) (F\# (((- P)))))
? (length (car (ilip '(p v (- p)))))
2
? (ilip '(- p -> (p -> q)))
NIL
? (ilip '(p -> - - p))
NIL
```

```
? (ilip '(- - p -> p))
(((F\# (((- P) (F\# (((BW (- (& (BW P) P))) (BW (- (& (BW (- (& (BW P) P))) (- (&
 (BW P) P))))) (BW (-> (& (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P))))) (-
(& (BW (- (& (BW P) P))) (- (& (BW P) P))))) (& (BW P) P))) (BW P) P (BW P) P (BW
 P) P))))))) ((F\# (((BW (- (& (BW P) P))) (BW (- (& (BW (- (& (BW P) P))) (- (&
(BW P) P))))) (BW P) P (BW P) P))) (F\# (((- P) (F\# (((BW (- (& (BW P) P))) (BW
(- (& (BW (- (& (BW P) P))) (- (& (BW P) P))))) P (BW P) (BW P) P (BW P) P)))))))
 ((- P) (F\# (((BW (- (& (BW P) P))) (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P)
 P))))) (BW P) P (BW P) P)))))
? (pprint (ilip '(- - p -> p)))

(((F\#
  (((- P)
    (F\#
     (((BW (- (& (BW P) P)))
       (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P)))))
       (BW
        (->
         (& (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P)))))
          (- (& (BW (- (& (BW P) P))) (- (& (BW P) P)))))
         (& (BW P) P)))
        (BW P) P (BW P) P (BW P) P)))))))
 ((F\#
  (((BW (- (& (BW P) P))) (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P)))))
    (BW P) P (BW P) P)))
  (F\#
   (((- P)
     (F\#
      (((BW (- (& (BW P) P)))
        (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P))))) P (BW P) (BW P) P
        (BW P) P)))))))
 ((- P)
  (F\#
   (((BW (- (& (BW P) P))) (BW (- (& (BW (- (& (BW P) P))) (- (& (BW P) P)))))
    (BW P) P (BW P) P)))))

|#

(setq lab '(                    ; Liste d' ABbréviations
            (f (& p (- p)))
            (t (-> p p))
            (con (- (bw (& p (- p)))))
            (ref ((bw p) -> p))
            (k1 (bw (p -> q) -> (bw p) -> (bw q)))
           ))

(defun defin (x y)
  (cons (list x y) lab)
 )

(defun value-nil (x env)
   (cond ((null env) nil)
         ((equal (caar env) x) (cadar env))
         (t (value x (cdr env)))
  )
 )

; ------------------------------------------------------------

#|
```

```
? (g* '(& (-> (bw p)
              (& (bw p) p))
          (-> (& (bw p) p)
              (bw p))))
NIL
? (make-t '(bw p))
(& (BW P) P)
? (g '(& (-> (bw p)
              (& (bw p) p))
          (-> (& (bw p) p)
              (bw p))))
(((BW P) (- P)))
? (s4grz '(& (-> (bw p)
              (& (bw p) p))
          (-> (& (bw p) p)
              (bw p))))
NIL
? (s4grz '(-> p (bw p)))
((P (F\# (((- P))))))

|#
```