RYMER Julie

Rapport de projet :

Dérivation en Lisp

Énoncé :

« **Lisp** : Dérivées (\*\*\*)

Une fonction numérique peut être dérivée, n'est-ce pas ? Écrire la fonction derive qui dérive une fonction qui lui est donnée en paramètre. Exemple:

? (derive '(+ (\* n n) (+ (\* n 5) 3))) = (+ (\* 2 n) 5)

Vous pouvez commencer par dériver une fonction dont le premier terme est un +, puis un \*...  
Il est important de traiter la question des "variables", par exemple de décider que l'on traite, d'abord, uniquement des fonctions à une seule variable et qu'elle se nomme (par exemple) x.

Ensuite, il faut surtout tout écrire sous une forme générique. »

1. Présentation du programme
2. Fonctions pratiques

*(define (liste a b)*

*(cons a (cons b '())))*

*(define (liste3 a b c)*

*(cons a (cons b (cons c '()))))*

*(define (-cos l)*

*(liste3 '\* -1 (liste 'cos l)))*

*(define (-sin l)*

*(liste3 '\* -1 (liste 'sin l)))*

*(define (o l m)*

*(remplacex l m))*

*(define (remplacex l n)*

*(if (pair? l)*

*(cons (remplacex (car l) n) (remplacex (cdr l) n))*

*(if (equal? l 'x)*

*n*

*l)))*

1. Mode d’emploi

Ce sont juste les fonctions qui nous ont facilitées la tache. Il y a notamment la concaténation à deux ou trois éléments (*liste* et *liste3*), toutes deux beaucoup utilisées. On trouve aussi les fonctions *–sin*, *-cos* et fonction composée (*o*) qui ne sont pas comprises dans les fonctions de base que le langage connait. La fonction *remplacex* va parcourir la liste grâce à la récursivité et à chaque fois qu’elle trouvera *x*, va le remplacer par la valeur *n* qu’on lui donne en argument. Utilisée dans le cas de fonctions composées, l’argument en question sera la deuxième fonction.

1. Traces d’exécution

Concaténation :

*> (liste3 '+ (\* 4 5) 'x)*

*'(+ 20 x)*

-Sinus :

*> (-sin '(expt x 3))*

*'(\* -1 (sin (expt x 3)))*

Composée :

*> (o '(\* 2 x) '(+ 4 x))*

*'(\* 2 (+ 4 x))*

1. Dérivées au cas par cas

*(define (derivepow l)*

*(liste3 '\* (liste3 '\* (car (cdr (cdr l))) (liste3 'expt (car (cdr l)) (- (car (cdr (cdr l))) 1))) (derive (car (cdr l)))))*

*(define (deriveadd l)*

*(liste3 '+ (derive (car (cdr l))) (derive (car (cdr (cdr l))))))*

*(define (derivesous l)*

*(liste3 '- (derive (car (cdr l))) (derive (car (cdr (cdr l))))))*

*(define (derivemult l)*

*(liste3 '+ (liste3 '\* (derive (car (cdr l))) (car (cdr (cdr l)))) (liste3 '\* (car (cdr l)) (derive (car (cdr (cdr l)))))))*

*(define (derivediv l)*

*(liste3 '/ (liste3 '- (derive (car (cdr l))) (derive (car (cdr (cdr l))))) (liste3 'expt (car (cdr (cdr l))) 2)))*

*(define (derivesqrt l)*

*(liste3 '/ (derive (car (cdr l))) (liste3 '\* 2 (liste 'sqrt (car (cdr l))))))*

*(define (derivecomp l)*

*(liste3 '\* (liste3 'o (derive (car (cdr l))) (car (cdr (cdr l)))) (derive (car (cdr (cdr l))))))*

*(define (derivereciproq l)*

*(liste3 '/ 1 (liste3 'o (derive (car (cdr l))) (liste3 'expt (car (cdr l)) -1))))*

*(define (derivecos l)*

*(-sin (car (cdr l))))*

*(define (derivesin l)*

*(liste 'cos (car (cdr l))))*

*(define (derive-cos l)*

*(liste 'sin (car (cdr l))))*

*(define (derive-sin l)*

*(-cos (car (cdr l))))*

*(define (derivetan l)*

*(derive '(/ (sin x) (cos x))))*

*(define (deriveexpo l)*

*(liste3 '\* (liste3 'expt 'exp (car (cdr (cdr l)))) (derive (car (cdr (cdr l))))))*

*(define (deriveln l)*

*(liste3 '/ (derive (car (cdr l))) (car (cdr l))))*

1. Mode d’emploi

On retrouve rangé ici les fonctions qui permettent de dériver les différents types de fonctions. Chaque cas particulier à sa fonction. Il est ainsi possible de dériver une puissance (*expt*), une addition (*+*), une soustraction (*-*), une multiplication (*\**), un quotient (*/*), une racine carré (*sqrt*), une composée (*o*), une réciproque (*expt x -1*), un cosinus positif et négatif (*cos et –cos*), un sinus positif et négatif (*sin et –sin*), une tangente (*tan*), une exponentielle (*expt e x*) et enfin un logarithme (*log*). À noter que les paramètres d’entrée et de sortie seront toujours tous deux des listes. Les fonctions seront en effet à chaque fois travaillées sous cette forme afin que le programme n’essaie pas d’interpréter les inconnues et nous rende donc la fonction telle quelle, de la même forme qu’on l’écrirait pour effectivement effectué le calcul. On remarque également la présence de la fonction *derive* dans le corps de certaines fonctions, qui n’a pas encore été vue. Il s’agit en réalité la fonction qui va regrouper tous ces cas par cas et que nous montrons juste après.

1. Traces d’exécution

Dérivée de la soustraction :

*> (derivesous '(- (expt x 2) (\* 2 x)))*

*'(- (\* (\* 2 (expt x 1)) 1) (+ (\* 0 x) (\* 2 1)))*

Dérivée de la racine carrée :

*> (derivesqrt '(sqrt (\* 2 x)))*

*'(/ (+ (\* 0 x) (\* 2 1)) (\* 2 (sqrt (\* 2 x))))*

Dérivée de la tangente :

*> (derivetan '(tan (\* 8 x)))*

*'(/ (- (cos x) (\* -1 (sin x))) (expt (cos x) 2))*

1. Dérivation générale

*(define (derive x)*

*(if (pair? x)*

*(if (equal? (car x) 'expt)*

*(if (equal? (car (cdr (cdr x))) -1)*

*(derivereciproq x)*

*(if (equal? (car (cdr x)) 'exp)*

*(deriveexpo x)*

*(derivepow x)))*

*(if (equal? (car x) '+)*

*(deriveadd x)*

*(if (equal? (car x) '-)*

*(derivesous x)*

*(if (equal? (car x) '\*)*

*(derivemult x)*

*(if (equal? (car x) '/)*

*(derivediv x)*

*(if (equal? (car x) 'sqrt)*

*(derivesqrt x)*

*(if (equal? (car x) 'o)*

*(derivecomp x)*

*(if (equal? (car x) 'cos)*

*(derivecos x)*

*(if (equal? (car x) '-cos)*

*(derive-cos x)*

*(if (equal? (car x) 'sin)*

*(derivesin x)*

*(if (equal? (car x) '-sin)*

*(derive-sin x)*

*(if (equal? (car x) 'tan)*

*(derivetan x)*

*(if (equal? (car x) 'log)*

*(deriveln x)*

*x)))))))))))))*

*(if (equal? x 'x)*

*1*

*0)))*

1. Mode d’emploi

Bien que la longueur de cette fonction *derive* puisse inquiéter, elle est en réalité assez simple. Elle est là, comme dit plus tôt, pour rassembler les différents types de dérivation possible. Ainsi, elle va regarder si la fonction est une liste et si le *car* de la fonction correspond à l’une des entités qu’on lui fourni (par exemple *\**) et appeler sa fonction dérivée correspondante (toujours dans l’exemple *derivemult*). Dans le cas où la fonction n’est pas une liste, elle reverra 1 si le paramètre est *x* (dérivée d’une inconnue) ou 0 dans le cas contraire (dérivée d’une constante, la seule possibilité restante). Le fait que *derive* appelle des fonctions qui elles-mêmes l’utilisent crée donc une récursivité indirecte qui permettra de dériver tous les éléments qui doivent l’être dans une même fonction. C’est la fonction principale de ce programme et celle qui finalement nous permet d’avoir la dérivée de n’importe quelle fonction.

1. Traces d’exécution

Dérivation 1 :

*> (derive '(/ (+ x (expt x 2)) (expt x -1)))*

*'(/ (- (+ 1 (\* (\* 2 (expt x 1)) 1)) (/ 1 (o 1 (expt x -1)))) (expt (expt x -1) 2))*

Dérivation 2 :

*> (derive '(expt exp (\* (sin x) (log (- x 1)))))*

*'(\* (expt exp (\* (sin x) (log (- x 1)))) (+ (\* (cos x) (log (- x 1))) (\* (sin x) (/ (- 1 0) (- x 1)))))*

1. Simplification de fonctions

*(define (effaceinutile l)*

*(if (pair? l)*

*(if (equal? (car l) '\*)*

*(if (equal? (car (cdr l)) 0)*

*0*

*(if (equal? (car (cdr (cdr l))) 0)*

*0*

*(if (equal? (car (cdr l)) 1)*

*(car (cdr (cdr l)))*

*(if (equal? (car (cdr (cdr l))) 1)*

*(car (cdr l))*

*l))))*

*(if (equal? (car l) '+)*

*(if (equal? (car (cdr l)) 0)*

*(car (cdr (cdr l)))*

*(if (equal? (car (cdr (cdr l))) 0)*

*(car (cdr l))*

*l))*

*(if (equal? (car l) 'expt)*

*(if (equal? (car (cdr (cdr l))) 0)*

*1*

*(if (equal? (car (cdr (cdr l))) 1)*

*(car (cdr l))*

*l))*

*(if (equal? (car l) 'o)*

*(if (pair? (car (cdr l)))*

*l*

*(if (equal? (car (cdr l)) 'x)*

*(car (cdr (cdr l)))*

*(car (cdr l))))*

*l))))*

*l))*

*(define (simplifie l)*

*(if (pair? l)*

*(effaceinutile (cons (simplifie (car l)) (simplifie (cdr l))))*

*l))*

*(define (deriver l)*

*(simplifie (derive l)))*

1. Mode d’emploi

Une chose qui est embêtante avec notre fonction dérivée est que, puisqu’elle utilise des listes et ne calcule rien, les fonctions dérivées seront souvent remplies d’éléments inutiles à l’arrivée. Pour pallier à cela, nous avons créée la fonction *effaceinutile* qui va se charger de détecter si la liste appartient à l’un des cas ou on pourrait enlever des éléments. Ainsi elle va regarder le *car* de cette liste puis les élément du *cdr* et si la liste possède des éléments superflus, les retirer, sinon renvoyer la liste de départ. Les conditions qui entrainent la présence d’éléments superflus sont une multiplication par zéro ou par un, une addition de zéro, une puissance de zéro ou de un et enfin la composée d’une constante ou d’une inconnue et d’une fonction. Mais comme cette fonction ne s’occupe que de la liste principale et non des sous-listes qui la composent, nous avons aussi créée la fonction *simplifie* qui va utiliser la fonction *effaceinutile* sur chaque sous-liste en utilisant la récursivité et les regrouper la liste à la fin.

Enfin nous avons créer la fonction *deriver* qui va tout simplement utiliser la fonction *simplifie* sur la fonction *derive* afin d’avoir directement la version simplifiée de la dérivée d’une fonction.

1. Traces d’exécution

Simplification :

*> (simplifie '(/ (- (+ 1 (\* (\* 2 (expt x 1)) 1)) (/ 1 (o 1 (expt x -1)))) (expt (expt x -1) 2)))*

*'(/ (- (+ 1 (\* 2 x)) (/ 1 1)) (expt (expt x -1) 2))*

Dérivation simplifiée :

*> (deriver '(sqrt (\* 2 x)))*

*'(/ 2 (\* 2 (sqrt (\* 2 x))))*

Comparaison dérivation basique puis simplifiée :

*> (derive '(/ (+ (\* 6 (expt x 3)) (+ (expt x 2) x)) (- (\* 3 x) 3)))*

*'(/ (- (+ (+ (\* 0 (expt x 3)) (\* 6 (expt x 3))) (+ (expt x 2) 1)) (- (\* 3 x) 3)) (expt (- (\* 3 x) 3) 2))*

*> (deriver '(/ (+ (\* 6 (expt x 3)) (+ (expt x 2) x)) (- (\* 3 x) 3))))*

*'(/ (- (+ (\* 6 (expt x 3)) (+ (expt x 2) 1)) (- (\* 3 x) 3)) (expt (- (\* 3 x) 3) 2))*

1. Calcul de la dérivée avec n

*(define (listf nom fonction)*

*(liste3 'define (liste nom 'x) (derive fonction)))*

*(define (crefonction nom fonction)*

*(eval (listf nom fonction)))*

1. Mode d’emploi

Avoir des dérivées de fonctions c’est bien, mais pouvoir les utiliser dans des calculs c’est mieux ! Ainsi nous avons écrit une méthode qui utilise les fonctions *listf* et *crefonction*. Elle sera elle très utile également si on veut pouvoir trouver le résultat d’une dérivée de fonction avec de multiples *x* et donc la réutiliser. Il s’agira pour cela de créer une fonction qui elle-même créer une fonction qu’on pourra reprendre pour lui attribuer n’importe quel argument. On commence par utiliser la fonction *listf* qui, pour un nom et un corps de fonction donné, va créer une liste sous la forme *‘(define (nom\_de\_fonction x) (corps\_de\_fonction\_derivee))*. On va ensuite utiliser la fonction *eval* sur cette même liste grâce à la fonction *crefonction*. Cela nous permet de faire en sorte que celle-ci devienne une fonction à part entière avec x en argument, qu’on pourra ainsi remplacer par la valeur que l’on veut dans l’expression de la fonction. Techniquement, on transforme la liste *‘(fonction)* dont on veut le résultat en corps de fonction utilisable, en la faisant d’abord devenir une sous-liste puis en évaluant le tout. En utilisant le nom qu’on a ainsi attribué à cette nouvelle fonction on peut maintenant calculer la dérivée de telle fonction pour *x=1*, *x=3*, …

1. Traces d’exécution

Création d’une liste *‘(define (nom\_de\_fonction x) (corps\_de\_fonction\_derivee))* :

*> (listf 'fonction1 '(expt exp (\* (sin x) (log (x-1)))))*

*'(define (fonction1 x) (\* (expt exp (\* (sin x) (log (x-1)))) (+ (\* (cos x) (log (x-1))) (\* (sin x) (/ (x-1) (x-1))))))*

Création de *fonction2* et utilisation :

*> (crefonction 'fonction2 '(+ (\* 2 x) (expt x 2)))*

*> (fonction2 7)*

*16*

*> (fonction2 9)*

*20*

*> (fonction2 83)*

*168*

1. Processus de réflexion

La toute première difficulté fut de se mettre d’accord sur le projet à traiter ! En fonction de ses affinités, l’un de nous voulez travailler sur Shell quand l’autre préférait Lisp. À tel point que jusqu’au moment de la décision finale nous avions travaillé sur deux projets en simultané pour nous garder le choix au maximum. Se fut ce projet qui fut finalement choisi car c’était celui qui valait le plus de points, qui avait le plus de choses à faire et nous l’avions déjà bien entamé.

Après nous être remémoré nos cours de maths d’un passé relatif, nous nous somme lancé dans ce projet qui semblait déjà beaucoup plus massif qu’il n’y paraissait au premier abord. Suivant, les indices laissés dans l’énoncé du projet, nous avons tout d’abord commencé par écrire les fonctions dérivées d’une constante et d’une inconnue. Puis nous avons compris qu’il faudrait écrire chaque cas particuliers séparément si nous voulions les rassembler ensuite dans une fonction qui ne partirait pas dans tous les sens. Nous avons écrit les deux fonctions de concaténation *liste* et *liste3* à ce moment en voyant le nombre de *cons* qu’il aurait fallu utiliser sinon. De même, toutes les fonctions d’opération présentes en début de programme (*-sin*, …) furent également écrite là en voyant qu’elle n’était pas dans le vocabulaire du langage et qu’on en aurait besoin. Cette partie, bien que pas vraiment compliquée vu qu’il suffisait de retranscrire en Lisp les définitions des dérivées qu’on pouvait trouver sur internet, fut laborieuse dans sa longueur et sa répétition, non seulement par l’écriture rempli de *car* et de *cdr* *du* *cdr*, mais également par les tests auxquels on procédait à chaque très régulièrement.

L’écriture de la fonction principale *derive* s’est finalement fait en simultané car on avait besoin d’appeler celle-ci pour beaucoup de dérivées de fonction qui contenait elles-mêmes des dérivées. Elle s’est donc faite petit à petit à mesure qu’on écrivait chaque sous fonction. Là aussi se fut un travail surtout répétitif et une indentation soignée fut de mise pour ne pas se perdre parmi les nombreux *if* (17 !).

Nous avons ensuite voulu nous débarrasser des valeurs nulles qui empêchait une bonne lisibilité des fonctions-résultat. Nous avons donc commencé une fonction *effaceinutile* qui venait là encore faire de l’identification afin de parcourir toutes les situations où un ménage pouvait être fait. Nous avons eu plus de mal à faire en sorte que cette fonction s’applique à toutes les sous-listes de la liste principale. Ayant utilisé des *when* plutôt que des *if* , l’utilisation de la récursivité ici ne nous permettait que d’avoir une simplification incomplète ou un résultat nul. Après avoir corrigé cette fonction avec uniquement des *if*, nous avons pu créer une fonction chapeau *simplifie* qui elle utilise la récursivité et s’occupe d’appliquer la première à chaque sous-liste si besoin. Il paraissait ensuite plus simple de créer une nouvelle fonction chapeau *deriver* pour appliquer cette opération de simplification à la fonction *derive* que de l’ajouter à chaque résultat que pouvait nous donner cette dernière. Nous avons ainsi enfin obtenu une fonction qui dérive ce qu’on lui donne de manière efficace.

Finalement, il ne restait plus qu’à faire en sorte de pouvoir utiliser ces fonctions dérivées en les calculant. Nous avons tout d’abord pensé par faire une fonction *remplacex* qui viendrait parcourir la liste et remplacer tous les *x* qu’elle croise par une valeur prédéfinie. Après s’être renseigné sur la fonction *eval* il ne nous restait plus qu’à écrire la fonction qui viendrait transformer cette liste en véritable suite d’opérations afin de pouvoir obtenir un résultat, puis la fonction qui viendrait faire la même chose mais avec la dérivée de la fonction et le tour était joué.

*(define (remplacex l n)*

*(if (pair? l)*

*(cons (remplacex (car l) n) (remplacex (cdr l) n))*

*(if (equal? l 'x)*

*n*

*l)))*

*(define (calcul l n)*

*(eval (remplacex l n)))*

*(define (calculderive l n)*

*(calcul (derive l) n))*

Cette méthode avait le désavantage de ne permettre de ne choisir qu’une seule valeur. Si l’on voulait recommencer il fallait réécrire la fonction à utiliser en intégralité. Malgré les copier-coller cela restait peu pratique. Suivant vos conseils, nous avons finalement opté pour la méthode qui crée une nouvelle fonction à chaque fois qu’on la réutilise et qui met la fonction dérivée à calculer directement dans le corps de cette nouvelle fonction. Ainsi il suffit de rappeler la fonction avec le nom et une valeur qu’on a choisi et le calcul de la fonction dérivée avec *x = valeur* sera fait.

Programme complet :

*#lang racket*

*#|---------------- Fonctions utiles -----------------|#*

*(define (liste a b)*

*(cons a (cons b '())))*

*(define (liste3 a b c)*

*(cons a (cons b (cons c '()))))*

*(define (-cos l)*

*(liste3 '\* -1 (liste 'cos l)))*

*(define (-sin l)*

*(liste3 '\* -1 (liste 'sin l)))*

*(define (o l m)*

*(remplacex l m))*

*(define (remplacex l n)*

*(if (pair? l)*

*(cons (remplacex (car l) n) (remplacex (cdr l) n))*

*(if (equal? l 'x)*

*n*

*l)))*

*#|------------- Derivees au cas par cas -------------|#*

*(define (derivepow l)*

*(liste3 '\* (liste3 '\* (car (cdr (cdr l))) (liste3 'expt (car (cdr l)) (- (car (cdr (cdr l))) 1))) (derive (car (cdr l)))))*

*(define (deriveadd l)*

*(liste3 '+ (derive (car (cdr l))) (derive (car (cdr (cdr l))))))*

*(define (derivesous l)*

*(liste3 '- (derive (car (cdr l))) (derive (car (cdr (cdr l))))))*

*(define (derivemult l)*

*(liste3 '+ (liste3 '\* (derive (car (cdr l))) (car (cdr (cdr l)))) (liste3 '\* (car (cdr l)) (derive (car (cdr (cdr l)))))))*

*(define (derivediv l)*

*(liste3 '/ (liste3 '- (derive (car (cdr l))) (derive (car (cdr (cdr l))))) (liste3 'expt (car (cdr (cdr l))) 2)))*

*(define (derivesqrt l)*

*(liste3 '/ (derive (car (cdr l))) (liste3 '\* 2 (liste 'sqrt (car (cdr l))))))*

*(define (derivecomp l)*

*(liste3 '\* (liste3 'o (derive (car (cdr l))) (car (cdr (cdr l)))) (derive (car (cdr (cdr l))))))*

*(define (derivereciproq l)*

*(liste3 '/ 1 (liste3 'o (derive (car (cdr l))) (liste3 'expt (car (cdr l)) -1))))*

*(define (derivecos l)*

*(-sin (car (cdr l))))*

*(define (derivesin l)*

*(liste 'cos (car (cdr l))))*

*(define (derive-cos l)*

*(liste 'sin (car (cdr l))))*

*(define (derive-sin l)*

*(-cos (car (cdr l))))*

*(define (derivetan l)*

*(derive '(/ (sin x) (cos x))))*

*(define (deriveexpo l)*

*(liste3 '\* (liste3 'expt 'exp (car (cdr (cdr l)))) (derive (car (cdr (cdr l))))))*

*(define (deriveln l)*

*(liste3 '/ (derive (car (cdr l))) (car (cdr l))))*

*#|---------------- Derivees generales ----------------|#*

*(define (derive x)*

*(if (pair? x)*

*(if (equal? (car x) 'expt)*

*(if (equal? (car (cdr (cdr x))) -1)*

*(derivereciproq x)*

*(if (equal? (car (cdr x)) 'exp)*

*(deriveexpo x)*

*(derivepow x)))*

*(if (equal? (car x) '+)*

*(deriveadd x)*

*(if (equal? (car x) '-)*

*(derivesous x)*

*(if (equal? (car x) '\*)*

*(derivemult x)*

*(if (equal? (car x) '/)*

*(derivediv x)*

*(if (equal? (car x) 'sqrt)*

*(derivesqrt x)*

*(if (equal? (car x) 'o)*

*(derivecomp x)*

*(if (equal? (car x) 'cos)*

*(derivecos x)*

*(if (equal? (car x) '-cos)*

*(derive-cos x)*

*(if (equal? (car x) 'sin)*

*(derivesin x)*

*(if (equal? (car x) '-sin)*

*(derive-sin x)*

*(if (equal? (car x) 'tan)*

*(derivetan x)*

*(if (equal? (car x) 'log)*

*(deriveln x)*

*x)))))))))))))*

*(if (equal? x 'x)*

*1*

*0)))*

*#|----------- Simplification de fonctions -----------|#*

*(define (effaceinutile l)*

*(if (pair? l)*

*(if (equal? (car l) '\*)*

*(if (equal? (car (cdr l)) 0)*

*0*

*(if (equal? (car (cdr (cdr l))) 0)*

*0*

*(if (equal? (car (cdr l)) 1)*

*(car (cdr (cdr l)))*

*(if (equal? (car (cdr (cdr l))) 1)*

*(car (cdr l))*

*l))))*

*(if (equal? (car l) '+)*

*(if (equal? (car (cdr l)) 0)*

*(car (cdr (cdr l)))*

*(if (equal? (car (cdr (cdr l))) 0)*

*(car (cdr l))*

*l))*

*(if (equal? (car l) 'expt)*

*(if (equal? (car (cdr (cdr l))) 0)*

*1*

*(if (equal? (car (cdr (cdr l))) 1)*

*(car (cdr l))*

*l))*

*(if (equal? (car l) 'o)*

*(if (pair? (car (cdr l)))*

*l*

*(if (equal? (car (cdr l)) 'x)*

*(car (cdr (cdr l)))*

*(car (cdr l))))*

*l))))*

*l))*

*(define (simplifie l)*

*(if (pair? l)*

*(effaceinutile (cons (simplifie (car l)) (simplifie (cdr l))))*

*l))*

*(define (deriver l)*

*(simplifie (derive l)))*

*#|---------- Calcul de la fonction derivee ----------|#*

*(define (listf nom fonction)*

*(liste3 'define (liste nom 'x) (derive fonction)))*

*(define (crefonction nom fonction)*

*(eval (listf nom fonction)))*