IA01-Compte_rendu_TD2

Objectif TD

Le but de ce TD est d'écrire une fonction deriv permettant de dériver une expression par rapport à une de ses variables.

Pour débuter

Nous pouvons donc en déduire, par l'énoncé, que l'entête de fonction sera de la forme suivante :

```
(defun deriv (expression variable)
        ; ...
)
```

En effet dérive a besoin de l'expression ainsi que de la variable de dérivation en paramètres.

Décomposition du problème

Cas d'un seul terme

Pour la dérivation d'un seul terme, la logique est assez triviale ; où il s'agit de la variable A correspondant à la variable de dérivation dA, auquel cas la dérivée vaut 1. Ou bien la dérivée vaut donc 0.

Nous avons donc :

```
(defun deriv-term (A dA)
 (if (eq a DA)
    1
  0
  )
 )
```

Traitement des opérateurs

Une fois que nous savons dériver un terme, il faut dériver les expressions linéaires contenant un ensemble de termes associés par des opérateurs.

- Nous allons commencer par l'addition :

On sait que a + b –'> a' + b'

On a donc :

```
(defun deriv-sum (exp var) ; (+ a b)
 (if (listp exp) ; verification traitement d'une liste
    (list (car exp) (deriv (cadr exp) var) (deriv (caddr exp) var)) ; retour (+ a' b')
  (deriv-term exp var)
  )
 )
```

De la même manière, nous traitons les autres opérateurs

- Soustraction

```
(defun deriv-dif (exp var) ; (- a b)
 (if (listp exp) ; verification traitement d'une liste
    (list (car exp) (deriv (cadr exp) var) (deriv (caddr exp) var)) ; retour (- a' b')
  (deriv-term exp var)
  )
 )
```

Remarque, pour la dérivation d'une différence (- a b), il s'agit de l'équivalent d'une dérivation d'une somme (+ a (-b)). Par conséquent, nous pourrions condenser notre code en appelant deriv-sum en ayant fait au préalable la modification du second terme et opérateur.

- Multiplication

```
(defun deriv-mult (exp var) ; (* u v)
 (if (listp exp) ; verification traitement d'une liste
    (list
     '+
     (list '* (deriv (cadr exp) var) (caddr exp))
     (list '*  (cadr exp) (deriv (caddr exp) var))
     )  ; retour (+ (* u' v) (* u v'))
  (deriv-term exp var)
  )
 )
```

- Division

```
(defun deriv-div (exp var) ; (/ u v)
 (if (listp exp) ; verification traitement d'une liste
    (list
     '/
     (list
      '-
      (list '* (deriv (cadr exp) var) (caddr exp))
      (list '*  (cadr exp) (deriv (caddr exp) var))
      )
     (list
      '*
      (caddr exp)
      (caddr exp)
      )
     )  ; retour (/ (- (* u' v) (* u v')) (* v v))
  (deriv-term exp var)
  )
 )
```

Dérivation d'une expression linéaire :

```
(defun deriv (exp var)
 (if (listp exp) ; verification traitement d'une liste
    (cond
     ((eq (first exp) '+) (deriv-sum exp var))
     ((eq (first exp) '-) (deriv-dif exp var))
     ((eq (first exp) '*) (deriv-mult exp var))
     ((eq (first exp) '/) (deriv-div exp var))
     (T (print '"error operator"))
     ) ; branchement sur la bonne formule de dérivation en fonction de l'opérateur
  (deriv-term exp var)
  )
 )
```

Avant de passer à la suite de l'exercice, il parait cohérent de vérifier nos fonctions.
Nous pouvons pour cela implémenter une fonction de test unitaire. Nous pouvons utiliser la fonction suivante qui permet de comparer fx(data) = expectedResult et s'assurer que l'égalité est vraie.

```
(defun unitTest (fx data expectedResult)      ; fonction de test unitaire
  (setq result (apply fx data))          ; on calcule le résultat de fx(data) avec les données envoyées
  (if (equal result expectedResult)          ; on compare avec le résultat attendu indiqué
    (setq test 'pass)
    (setq test 'error)
  )
  (format t "~% ~S unitTest : ~S ; result = ~S | expected = ~S" fx test result expectedResult)
; on affiche le résultat du test
  )
```

Exemples d'utilisation :

```
CG-USER(288): (unitTest 'deriv-sum '((+ 3 x) x) '(+ 0 1))
 DERIV-SUM unitTest : PASS ; result = (+ 0 1) | expected = (+ 0 1)

CG-USER(292): (unitTest 'deriv-dif '((- x 1) x) '(- 1 0))
DERIV-SUM unitTest : PASS ; result = (- 1 0) | expected = (- 1 0)

CG-USER(294): (unitTest 'deriv-mult '((* x 5) x) '(+ (* 1 5) (* x 0)))
 DERIV-MULT unitTest : PASS ; result = (+ (* 1 5) (*  X 0)) | expected = (+ (* 1 5) (*  X 0))

CG-USER(295): (unitTest 'deriv-div '((/ x 1) x) '(/ (- (* 1 1)(* x 0)) (* 1 1)))
 DERIV-DIV unitTest : PASS ; result = (/ (- (* 1 1)(* x 0)) (* 1 1))) | expected = (/ (- (* 1 1)(* x 0)) (* 1 1)))

CG-USER(297):  (unitTest 'deriv '((- (* 3 x) (/ (+ 5 2) x)) x)
'(- (+ (* 0 X) (* 3 1))
  (/ (- (* (+ 0 0) X) (* (+ 5 2) 1)) (* X X))))

DERIV unitTest : PASS ; result ; ...
```

Nous pouvons donc voir que nos fonctions sont correctes avec ces quelques tests unitaires. Bien sur, il est conseillé d'en faire bien plus afin de gérer un maximum et s'assurer de l'exactitude des résultats fournis pour des opérations élémentaires.

Lisibilité du résultat

A ce stade, nous avons un résultat correct pour notre dérivée. Néanmoins nous pouvons remarquer que certaines opérations peuvent être simplifiées.

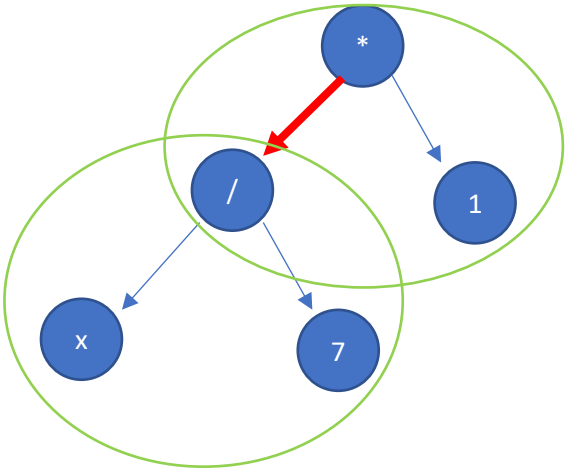
Nous pouvons lister quelques cas :

- (opérateur a b) peut être directement calculé
- (+ a 0) = (+ 0 a) = a
- (- a 0) = a
- (/ 0 a) = 0
- (* 0 a) = (* a 0) = 0
- (* 1 a) = (* a 1) = a

Nous allons donc créer une fonction qui va permettre de simplifier ces expressions.

```
(defun simplifyExp (exp)
 (if (list exp)
    (cond
     ((and (numberp (caddr exp))(numberp (cadr exp)))  ; Opération a signe b = évalution (a signe b)
      (eval exp))

     ((and (eq (car exp) '+)(eq (cadr exp) 0))    ; Somme 0 + a = a
      (caddr exp))

     ((and (eq (car exp) '+)(eq (caddr exp) 0))   ; Somme a + 0 = a
      (cadr exp))


     ((and (eq (car exp) '-)(eq (caddr exp) 0))   ; Somme a - 0 = a
      (cadr exp))

     ((and (eq (car exp) '*)(or (eq (cadr exp) 0)(eq (caddr exp) 0)))  ; Produit (a * 0) ou (0 * a) = 0
      0)

     ((and (eq (car exp) '*)(eq (cadr exp) 1)) ; Produit 1 * x = x
      (caddr exp))

     ((and (eq (car exp) '*)(eq (caddr exp) 1)) ; Produit x * 1 = x
      (cadr exp))

     ((and (eq (car exp) '/)(eq (cadr exp) 0))   ; Somme 0 / a = 0
      0)

     (T
      exp) ; aucune simplification trouvée, on renvoit l'expression
     )

   NIL)
 )
```

Nous avons donc la fonction précédente qui permet de simplifier une expression constituée de 'feuilles', nous allons maintenant créer une fonction qui permet de parcourir notre expression linéaire en la traitant comme un arbre binaire.



Encerclé en vert, il s'agit de la simplification d'une expression simple.

La flèche rouge correspond à l'appel de la simplification d'un sous arbre.

Nous avons donc la fonction suivante :

```
(defun simplify (exp)
 (if (listp exp)
    (cond
     ((and (not (listp (cadr exp))) (not (listp (caddr exp))))  ; les deux fils sont des feuilles
      (simplifyExp exp))

     ((and (listp (cadr exp)) (not (listp (caddr exp))))  ; seul un fils est un terme
      (simplifyExp (list (car exp) (simplify (cadr exp))(caddr exp))))

     ((and (listp (caddr exp)) (not (listp (cadr exp)))) ; seul un fils est un terme
      (simplifyExp (list (car exp) (simplify (caddr exp))(cadr exp))))

     (T  ; sinon, les deux fils sont des sous-arbres à parcourir
      (simplifyExp (list (car exp) (simplify (cadr exp)) (simplify (caddr exp)))))
     )
  exp
  )
 )
```

Ainsi nous pouvons simplifier des expressions comme par exemple :

```
CG-USER(298): (simplify '(/ (- (* (+ 0 0) X) (* (+ 5 2) 1)) (* X X)))

(/ -7 (* X X))
```

Pour aller plus loin

Il est possible d'ajouter d'autres opérateurs fonctions tel que exp, log, cos, sin, etc.

Néanmoins cela demanderait très certainement une restructuration du code qui admet jusqu'à présent l'hypothèse que chaque liste contient 3 termes. (opérateur a b).

Or avec l'introduction de fonctions, nous aurions des listes à 2 termes.

Il en est de même avec l'introduction des puissances.