1

変数, 整数, 加算, 乗算から構成された式を下のデータ型で示す.

```
type exp =
  Var of string (* 変数 *)
  | Int of int (* 整数 *)
  | Add of exp * exp (* 加算 *)
  | Mul of exp * exp (* 乗算 *)
```

例えば次の OCaml の式は $(x+y)(y+2)$ を示している.

```
Mul (Add (Var "x", Var "y"), Add (Var "y", Int 2))
```

(a) このデータ型で表された式に対して記号的に微分を行う関数をかけ.

まず記号的に微分を行うプログラムをソースコード 1 に示す.

このとき微分はまず整数の時は 0 を返せばよい. そして変数が来た時は例えば x で微分したい時に x のみがくれば 1 を返し,y など x 以外の変数の時は 0 を返すようにする. さらに $(f(x)+g(x))' = f'(x)+g'(x)$ を返す. 積のときはヒントにあるように $(f(x)g(x))' = f'(x)g(x)+f(x)g'(x)$ であるのでそのように計算するように定義すれば記号的な微分を行うことができる.

ソースコード 1: 記号的に微分を行うプログラム

```
(* 変数, 整数, 加算, 乗算から構成された式を定義 *)
type exp =
  Var of string (* 変数 *)
  | Int of int (* 整数 *)
  | Add of exp * exp (* 加算 *)
  | Mul of exp * exp (* 乗算 *)
;;


(* a *)
let rec deriv f v =
  match f with
    Int n -> Int 0
  | Var x -> if x = v then Int 1 else Int 0
  | Add (x1, x2) -> Add (deriv x1 v, deriv x2 v)
  | Mul (x1, x2) -> Add (Mul (x1, deriv x2 v), Mul (x2, deriv x1 v));;
```

ソースコード 2: 記号的微分を行うプログラムを実行した結果

```
# deriv (Mul (Add (Var "x", Var "y"), Add (Var "y", Int 2))) "x";;
- : exp =
Add (Mul (Add (Var "x", Var "y"), Add (Int 0, Int 0)),
 Mul (Add (Var "y", Int 2), Add (Int 1, Int 0)))
# deriv (Mul (Add (Var "x", Int 1), Add (Var "x", Int 2))) "x";; deriv (Mul (
    Add (Var "x", Int 1), Add (Var "x", Int 2))) "x";;
- : exp =
Add (Mul (Add (Var "x", Int 1), Add (Int 1, Int 0)),
 Mul (Add (Var "x", Int 2), Add (Int 1, Int 0)))
```

(b)(a) で定義した微分を行う関数を拡張し単純化した答えを返すようにせよ

単純化するために加算と乗算を行うとき単純化した答えを返す補助関数を定義して微分するようにする.(ソースコード 3)

ソースコード 3: 単純化した答えを返すようにする

```
(* 変数, 整数, 加算, 乗算から構成された式を定義 *)
type exp =
  Var of string (* 変数 *)
  | Int of int (* 整数 *)
  | Add of exp * exp (* 加算 *)
  | Mul of exp * exp (* 乗算 *)
;;


(* 単純化した加算の式を返す *)
let add (x, y) =
  match (x, y) with
    (x, Int 0) -> x
  | (Int 0, y) -> y
  | (x, y) -> Add (x, y);;

(* 単純化した乗算の式を返す *)
let mul (x, y) =
  match (x, y) with
    (x, Int 0) -> Int 0
  | (Int 0, y) -> Int 0
  | (x, Int 1) -> x
  | (Int 1, y) -> y
  | (x, y) -> Mul (x, y);;

(* b *)
let rec deriv f v =
  match f with
    Int n -> Int 0
  | Var x -> if x = v then Int 1 else Int 0
  | Add (x1, x2) -> add (deriv x1 v, deriv x2 v)
  | Mul (x1, x2) -> add (mul (x1, deriv x2 v), mul (x2, deriv x1 v));;
```

ソースコード 4: 実行結果

```
# deriv (Mul (Add (Var "x", Var "y"), Add (Var "y", Int 2))) "x";;
- : exp = Add (Var "y", Int 2)
# deriv (Mul (Add (Var "x", Int 1), Add (Var "x", Int 2))) "x";; deriv (Mul (
    Add (Var "x", Int 1), Add (Var "x", Int 2))) "x";;
- : exp = Add (Add (Var "x", Int 1), Add (Var "x", Int 2))
```

(c) 扱える式を e^n の形の式で拡張せよ.

ここで e は任意の式であり n は非負の整数とする. なお私はこの部分では (b) で実装した単純化はしていない. 冪乗の微分の定義は

$$x^n = nx^{n-1}$$

であるからこれを追加すれば良い.(ソースコード 5)

ソースコード 5: 冪乗を追加したもの

```
(* 変数, 整数, 加算, 乗算から構成された式を定義 *)
type exp =
  Var of string (* 変数 *)
  | Int of int (* 整数 *)
  | Add of exp * exp (* 加算 *)
  | Mul of exp * exp (* 乗算 *)
  | Exp of exp * int (* 冪乗 *)
;;


(* c *)
let rec deriv f v =
  match f with
    Int n -> Int 0
  | Var x -> if x = v then Int 1 else Int 0
  | Add (x1, x2) -> Add (deriv x1 v, deriv x2 v)
  | Mul (x1, x2) -> Add (Mul (x1, deriv x2 v), Mul (x2, deriv x1 v))
  | Exp (x, n) -> Mul (Int n, Mul (Exp (x, n-1), deriv x v))
;;
```

ソースコード 6: 実行結果

```
# deriv (Exp (Add (Var "x", Int 1), 2)) "x";;
- : exp =
Mul (Int 2, Mul (Exp (Add (Var "x", Int 1), 1), Add (Int 1, Int 0)))
```

2

上で考えた式 (e^n は含めない) を多項式に展開する問題を考える. 下の形の式を変数 $x_1, x_2, \ldots, x_k$ 上の単項式と呼ぶ.

$$cx_1^{n_1}x_2^{n_2}\cdots x_k^{n_k}$$

ここで c は変数, $n_1, n_2, \ldots, n_k$ は非数の整数とする. 単項式は和の形の式をここでは多項式とする.

2 番はすべて 1 つのソースコードにまとめてしまったのでまず先にソースコード 7 に 2 番の課題すべてのソースコードを示す.

ソースコード 7: 2 番のソースコード

```
(* 上級編 *)
type exp =
    Var of string
  | Int of int
  | Add of exp * exp
  | Mul of exp * exp;;

(* 2-a *)
(* 2つの単項式の積を返す *)
let rec mul_mono (x, xlst) (y, ylst) =
  match (xlst, ylst) with
    ([], []) -> (x*y, [])
  | (xlst, []) -> (x*y, xlst)
  | ([], ylst) -> (x*y, ylst)
```

```
  (* List.map2 は f [a, b] [c, d] -> [f a c, f b d] のようなことをしてくれる *)
  | (xlst, ylst) -> (x * y, List.map2 (fun x -> fun y -> x + y) xlst ylst);;

(* 2-b *)
(* 単項式と多項式の積を返す *)
let rec mul_mono_poly mul poly =
  List.map (mul_mono mul) poly;;

(* 多項式をリストで表現するときに係数を無視した単項式の順番を比較し小さい順に並べる関数 *)
let rec mono_le m1 m2 =
  match (m1, m2) with
    ([], []) -> true
  | (n1::m1', n2::m2') -> n1 < n2 || (n1 = n2 && mono_le m1' m2')
  | _ -> false;;

(* 2-c *)
(* 2つのこの順序で並べられた多項式の和を返す関数 *)
let rec add_poly m1 m2 =
  match (m1, m2) with
    ([], []) -> []
  | (m1, []) -> m1
  | ([], m2) -> m2
  | ((x, xl)::restx, (y, yl)::resty) ->
    if xl = yl then (x + y, xl) :: add_poly restx resty
    else
      if mono_le xl yl then
        (x, xl) :: add_poly restx m2
      else
        (y, yl) :: add_poly m1 resty;;

(* 2-d *)
(* 2つの多項式の積を返す関数 *)
let rec mul_poly m1 m2 =
  match (m1, m2) with
    (m1, []) -> []
  | ([], m2) -> []
  | (x::restx, m2) ->
    (* 単項式と多項式の積と残りの多項式同士の積を足せば良い *)
    add_poly (mul_mono_poly x m2) (mul_poly restx m2);;

(* 2-e *)
(*
 * 課題 1の型exp で表された式を展開し多項式に変換する関数 exp2poly exp xs
 * xs は式 exp に現れる変数をリストとして並べたものである
 *)
let rec initialize_list n =
  match n with
    0 -> []
  | n -> 0 :: initialize_list (n - 1);;

(* var2poly "x" ["x"; "y"; "z"] - : int list = [1; 0; 0] *)
let rec var2mono v xs =
  match (v, xs) with
    (v, []) -> []
  | (v, x::rest) ->
```

```
70      if (List.mem v [x]) then 1:: initialize_list (List.length rest)
71      else 0::var2mono v rest;;
72
73  let rec exp2poly exp xs =
74    match exp with
75      Int x -> [(x, initialize_list (List.length xs))]
76    | Var x -> [(1, var2mono x xs)]
77    | Add (x, y) -> add_poly (exp2poly x xs) (exp2poly y xs)
78    | Mul (x, y) -> mul_poly (exp2poly x xs) (exp2poly y xs);;
```

(a)

n 変数の単項式を整数と長さ n の非負整数のリスト組として表す. たとえば以下は $5x^2y^0z^4$ を表している.

```
1  # (5, [2; 0; 4])
```

ここで 2 つの単項式に関してその積を表す単項式を関数を書く.

さて実装の方法としてはまず上の例で言う 5 の部分 (つまり係数) は積の場合かける事を行う. 次に冪乗の部分に関しては各値のときで足し算を行う.OCaml では基本的にリストをインデックスでアクセスすることができないのでなかなか難しかったが List に map2 というものがありこれを用いることで実現できた.map2 は例えば 2 つのリスト `[a; b; c], [d; e; f]` とすると関数 `g` を用いてリスト`|[g a d; g b e; g c f]|`のような処理を行ってくれるのでここで足し算を行えば良い.

ソースコード 8: 実行結果

```
1  # mul_mono (2, [2; 0; 3]) (3, [1; 2; 1]);;
2  - : int * int list = (6, [3; 2; 4])
```

これは $2x^2y^0z^3 \cdot 3x^1y^2z^1 = 6x^3y^2z^4$ を示している.

(b)

次に単項式のリストで表すようにして単項式と多項式の積を多項式として返す関数を考える. 実装の方針としては単項式は多項式に属するすべての単項式にかけることになるのでそのような処理を行えば良い. それには単項式と単項式の積を返す関数 (a で作成した `mul_mono`) を使えば良い.

以下は $2x^2y^0z^3(2x^2y^0z^3 + 3x^1y^2z^1) = 4x^4y^0z^6 + 6x^3y2z^4$ の結果を示している.

ソースコード 9: 実行結果

```
1  # mul_mono_poly (2, [2; 0; 3]) [(2, [2; 0; 3]); (3, [1; 2; 1])];;
2  - : (int * int list) list = [(4, [4; 0; 6]); (6, [3; 2; 4])]
```

(c)

次に多項式をリストで表現するときに (係数を無視した) 単項式の順番を `mono_le` という関数で定義してその定義順で並べられた多項式の和を返す関数を書く.

処理の概要としてはまず係数を無視した単項式の順番が一致する場合は係数のみを足し算すればよい. 一致しない場合は `mono_le` で順番を調べ順番で計算されるように処理を行うことでこの関数を実現することができる.

ソースコード 10: 実行結果

```
1  # add_poly [(2,[0; 0]); (1, [1; 0]); ] [(2, [1; 0]); (3, [1; 2]); ];;
2  - : (int * int list) list = [(2, [0; 0]); (3, [1; 0]); (3, [1; 2])]
```

これは $(2x^0y^0 + x^1y^0) + (2x^1y^0 + 3x^1y^2) = 2x^0y^0 + 3x^1y^0 + 3x^1y^2$ を示している.

(d)

(c) と同じ単項式の順序を仮定し,2 つの多項式の積を表す多項式を返す関数を作成する. これには (c) で作成した関数を利用する.2 つの多項式ではたとえば $(x+y)^2$ の場合 $(x+y)(x+y) = x(x+y) + y(x+y)$ のように単項式と多項式の積の足し算のように分解される. つまり (b) で作成した単項式と多項式の積を求める関数と (c) で作成した多項式同士を加算する関数を組み合わせればよい.

ソースコード 11: 実行結果

```
# mul_poly [(1, [0; 1]); (1, [1; 0])] [(1, [0; 1]); (1, [1; 0])];;
- : (int * int list) list = [(1, [0; 2]); (2, [1; 1]); (1, [2; 0])]
```

(e)

最後に課題 1 の型 exp で表された式を展開して多項式に変換する関数を書く.

これを実現するには課題 1 の型 exp にあるうち Int と Var を処理する関数を用意する必要がある. まず Int は係数である. もしも係数だけの場合は次のように表されなければならない.

$$ax^0y^0(a \text{ は係数}, x, y \text{ は変数})$$

なのでこのような変換を行う関数 initialize_list を定義した.

次に Var の場合は与えられる Var が変数リスト内に含まれていれば 1, 含まれていなければ 0 を返し先ほどと同じようなリストを返すような関数を定義する. これを行うことで Var を処理できる.

以上のような関数を用意すればあとの Add, Mul に関しては (c), (d) で作成した関数を組み合わせて利用すれば実現することができる.

ソースコード 12: 実行結果

```
exp2poly (Int 3) ["x"; "y"];;
- : (int * int list) list = [(3, [0; 0])]
exp2poly (Var "y") ["x"; "y"; "z"];;
- : (int * int list) list = [(1, [0; 1; 0])]
exp2poly (Mul (Add (Var "x", Var "y"), Add (Var "x", Var "y"))) ["x"; "y"];;
- : (int * int list) list = [(1, [0; 2]); (2, [1; 1]); (1, [2; 0])]
```