

プログラミングAIII

2023年度講義資料 (6)

新潟大学 工学部工学科 知能情報システムプログラム

青戸等人

目次

- ① パターンによる定義
- ② リスト構造による再帰

目次

- ① パターンによる定義
- ② リスト構造による再帰

リストパターンを用いた束縛 (1)

valによる変数束縛は、対について使えたことを思い出そう。

```
# val (x,y) = (2,3);  
val x = 2 : int  
val y = 3 : int  
# val (x::xs) = [1,2,3];  
(interactive):105.5-105.20(119) Warning: ...  
val x = 1 : int  
val xs = [2, 3] : int list  
#
```

valは、変数や変数の対だけでなく、リストパターンを使っても、値を束縛することができる。

リストパターンを用いた束縛 (2)

さまざまなリストパターンが使えるが、...

```
# val (x::y::xs) = [1,2,3,4];  
(interactive):108.5-108.25(75) Warning: ...  
val x = 1 : int  
val y = 2 : int  
val xs = [3, 4] : int list  
# val (x::y::xs) = [1];  
uncaught exception Bind at (interactive):110.5
```

リストパターンを使った束縛は、パターンが適合できない場合は失敗(例外で中断)してしまう。

パターンを用いた定義

対によるパターンは、関数の定義でも用いた。同様のことが、リストのパターンについても可能。

```
# fun addPlusOne (x,y) = x + y + 1;
val addPlusOne = fn : int * int -> int
# addPlusOne (2,3);
val it = 6 : int
# fun sum [] = 0
    | sum (x::ys) = x + sum ys;
val sum = fn : int list -> int
# sum [1,2,3];
val it = 6 : int
```

リストの場合は、対の場合と違って、複数のパターンを考慮する必要がある。

⇒ 次ページ

パターンを用いたリスト関数の定義(教p.87)

前ページのsum関数の定義をよく見てみよう。

```
1 fun sum [] = 0
2   | sum (x::xs) = x + sum xs;
```

- 場合分けのところには、|を挟む。引数のパターンを書くことで場合を識別する。
- 関数定義であるから、先頭にfunが必要。
- 関数適用の方が ::や+ よりも結合力が強い。したがって、(x::xs)の括弧は必要で、sum xsの括弧は必要ない(つけてもよい)。
- 途中の改行はなくてもよい。逆に、途中に;を入れると、そこで関数の定義が終了を表わすことになるのでダメ。

さまざまなパターンの利用

[]と $x::xs$ とによるパターン区別を用いた定義をみたが、その他にも、さまざまなパターンの区別のやり方がある。

```
fun isWeaklyIncreasing [] = true
  | isWeaklyIncreasing (x::[]) = true
  | isWeaklyIncreasing (x::y::ys) =
      (x <= y) andalso isWeaklyIncreasing (y::ys);
fun myZip ([],ys)= []
  | myZip (xs,[]) = []
  | myZip (x::xs,y::ys) = (x,y) :: myZip (xs,ys);
```

なお、パターンがすべての場合を網羅していない場合は、警告(Warning)が表示される。

パターンマッチの順序

前ページのzipのパターンは、 $([], [])$ の場合に重複している。

このようにパターンに重複がある場合は、先にマッチしたパターンが用いられる。

```
fun is0or3or7 0 = true
  | is0or3or7 3 = true
  | is0or3or7 7 = true
  | is0or3or7 x = false;
```

この場合、is0or3or7の最後の行にマッチするのは、0,3,7以外の数の場合だけになる。

このように、整数の場合も、パターンを用いた定義が可能。

特殊なパターン"_"

前ページの最後の行で，引数のxは用いられていない．このような場合は，引数に変数ではなく _ 記号を用いるとよい．

```
fun is0or3or7 0 = true
  | is0or3or7 3 = true
  | is0or3or7 7 = true
  | is0or3or7 _ = false;
```

_ 記号は，何にでもマッチするが，値は捨てられる．引数を右辺で用いない場合に用いる．

目次

- ① パターンによる定義
- ② リスト構造による再帰

リスト構造による再帰

数学的には集合 A 上のリストは帰納的に定義される。

Definition

- (1) $[]$ はリストである。
- (2) xs がリストで $x \in A$ ならば、 $x :: xs$ はリストである。

したがって、リストに関する演算の定義は再帰的に定義できる。例えば、リスト xs の長さ $\text{myLength } xs$ は、以下のように再帰的に定義できる。

Definition

$$\text{myLength } xs = \begin{cases} 0 & (xs = [] \text{ の場合}) \\ 1 + (\text{myLength } ys) & (xs = y :: ys \text{ の場合}) \end{cases}$$

再帰を用いたリスト関数の定義

```
# fun myLength [] = 0
> | myLength (x::xs) = 1 + myLength xs;
val myLength = fn : 'a. 'a list -> int]
# myLength [1,2,3];
val it = 3 : int
```

リストの連結を表わす関数は以下のように定義される。

$$\text{myAppend}(xs, ys) = \begin{cases} ys & (xs = [] \text{の場合}) \\ z :: (\text{myAppend}(zs, ys)) & (xs = z :: zs \text{の場合}) \end{cases}$$

```
# fun myAppend ([],ys) = ys
> | myAppend (x::xs,ys) = x :: myAppend (xs,ys)
val myAppend = fn : 'a. 'a list * 'a list -> 'a list]
# myAppend ([1,2,3],[4,5]);
val it = [1, 2, 3, 4, 5] : int list
```

実習課題(1)

- ① リストの反転を表わす関数は以下のように定義される。

$$\text{myReverse } xs = \begin{cases} [] & (xs = [] \text{の場合}) \\ (\text{myReverse } zs)@[z] & (xs = z :: zs \text{の場合}) \end{cases}$$

関数myReverseを定義せよ。

- ② 要素 x とリスト xs が与えられたときに、 x が xs に入っているならばtrueを、 x が xs に入っていないならばfalseを返す関数memberを定義せよ。

$$\text{member } (x, xs) = \begin{cases} \text{false} & (xs = [] \text{の場合}) \\ \text{if } x = y \text{ then true else } (\text{member } (x, ys)) & (xs = y :: ys \text{の場合}) \end{cases}$$

実習課題(1)

- ③ 要素 x とリスト xs が与えられたときに、 xs から x をすべて取り除いたリストを返す関数`delete`を定義せよ。

$$\text{delete}(x, xs) = \begin{cases} [] & (xs = [] \text{の場合}) \\ \text{if } x = y \text{ then delete}(x, ys) & \\ \text{else } y :: (\text{delete}(x, ys)) & (xs = y :: ys \text{の場合}) \end{cases}$$

- ④ リスト $[a_1, \dots, a_n]$ が与えられたときに、リストの要素の合計 $a_1 + \dots + a_n$ を返す関数`sum`を定義せよ。
- ⑤ リストのリスト $[xs_1, \dots, xs_n]$ が与えられたときに、リスト $xs_1 @ \dots @ xs_n$ を返す関数`concatList`を定義せよ。

実習課題(2)

以下の関数を定義せよ。

(ヒント：引数のパターンの与え方を工夫することで、定義が簡単になる。)

- ① $[(x_1, y_1), \dots, (x_n, y_n)]$ のリストに対して、
 $[(y_1, x_1), \dots, (y_n, x_n)]$ のリストを返す `exchangePairs`.

```
# exchangePairs [(1,2),(3,4),(5,6)];  
val it = [(2, 1), (4, 3), (6, 5)] : (int * int) list
```

- ② 与えられたリストの偶数番目の要素からなるリストを返す
関数 `eventhElems`.

```
# eventhElems [1,2,3,4,5,6,7];  
val it = [2, 4, 6] : int list
```


実習課題(2)

- ③ リストのリストが与えられたときに、各要素リストの先頭要素からなるリストを返す関数 `getHeads`. (ただし、要素リストが空の場合、先頭要素はないとしてとばす.)

```
# getHeads [[1,2,3],[4,5],[],[6,7,8]];
val it = [1, 4, 6] : int list
```

- ④ $[x_1, \dots, x_n]$ が与えられたときに、 $[(x_1, x_2), (x_2, x_3), \dots, (x_{n-1}, x_n)]$ を返す `mkSucPairs`.

```
# mkSucPairs [1,2,3,4];
val it = [(1, 2), (2, 3), (3, 4)] : (int * int) list
```

- ⑤ 2つのソートされたリストをマージする関数 `merge`. 例えば、`merge [1,3,6] [2,3,4] = [1,2,3,3,4,6]` となる.

```
# merge ([1,3,5],[0,4,7]);
val it = [0, 1, 3, 4, 5, 7] : int list
```