プログラミングAIII

2024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

目次

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

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$\mathrm{myLength}\ xs$は，以下のように再帰的に定義できる.

Definition

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

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.

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

```
# 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に入っていればtrueを，xがxsに入っていなければfalseを返す関数 `member` を定義せよ.

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

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．

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

4. リスト$[a_1, \ldots, a_n]$が与えられたときに，リストの要素の合計$a_1 + \cdots + a_n$を返す関数sumを定義せよ．
5. リストのリスト$[xs_1, \ldots, xs_n]$が与えられたときに，リスト$xs_1 @ \cdots @ xs_n$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. $[(x_1, y_1), \ldots, (x_n, y_n)]$のリストに対して，
   $[(y_1, x_1), \ldots, (y_n, x_n)]$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. $[x_1, \ldots, x_n]$が与えられたときに，$[(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n)]$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
```