

# プログラミング言語AIII

2025年度講義資料 (8)

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 データ型
- ② データ型の利用

# 目次

## ① データ型

## ② データ型の利用

## データ型(教7章)

リスト型やオプション型は、より一般的なデータ型とよばれる型の例となっている。データ型は、datatype 宣言によって定義される型のことをいう。

- リスト型の定義

```
1 datatype 'a list = :: of 'a * 'a list | nil
```

- オプション型の定義

```
1 datatype 'a option = NONE | SOME of 'a
```

- ブール型の定義

```
1 datatype bool = false | true
```

# データ型の宣言

datatype 宣言を使って、自分で新しい型を定義することができる。定義した新しいデータ型であっても使い方は組み込みのデータ型と同じ。

- 1 複数の値からなる型を|で区切って列挙する。

```
1 datatype day_of_week = MON | TUE | WED | THU
2 | FRI | SAT | SUN
```

```
# datatype day_of_week = MON | TUE | WED | THU | FRI | SAT | SUN;
datatype day_of_week = FRI | MON | SAT | SUN | THU | TUE | WED
# fun next MON = TUE | next TUE = WED | next WED = THU
> | next THU = FRI | next FRI = SAT | next SAT = SUN
> | next SUN = MON;
val next = fn : day_of_week -> day_of_week
# next FRI;
val it = SAT : day_of_week
#
```

# データ型の宣言

- ② of...によって、パラメータを指定することができる。

```
1 datatype graphic_obj = Circle of real
2   | Square of real | Rectangle of real * real
3   | Triangle of real * real * real
```

```
# datatype graphic_obj = Circle of real | Square of real
  | Rectangle of real * real | Triangle of real * real * real;
datatype graphic_obj = .....
# fun area (Circle r) = Real.Math.pi * r * r
  | area (Square i) = i * i
  | area (Rectangle (i,j)) = i * j
  | area (Triangle (i,j,k)) = let val l = (i + j + k) / 2.0
    in Real.Math.sqrt (l * (l - i) * (l - j) * (l - k)) end;
val area = fn : graphic_obj -> real
# area (Circle 1.0);
val it = 3.14159265359 : real
```

# データ型の宣言

- ③ パラメータにいま定義している型を指定すると帰納的な定義になる。

```
1  datatype expr = NUM of int | PLUS of expr *  
    expr | TIMES of expr * expr
```

```
# datatype expr = NUM of int | PLUS of expr * expr  
    | TIMES of expr * expr;  
datatype expr = .....  
# fun eval (NUM n) = n  
    | eval (PLUS (e1,e2)) = eval e1 + eval e2  
    | eval (TIMES (e1,e2)) = eval e1 * eval e2;  
val eval = fn : expr -> int  
# eval (TIMES (PLUS (NUM 2, NUM 3), NUM 5));  
val it = 25 : int
```

# データ型の宣言

- ④ パラメータに型変数を使うと多相型が定義できる。

```
1 datatype 'a tree = Empty
2           | Node of 'a * 'a tree * 'a tree
```

```
# datatype 'a tree = Empty
      | Node of 'a * 'a tree * 'a tree;
datatype 'a tree = ...
# fun depth Empty = 0
      | depth (Node (x,left,right)) = 1 +
      Int.max (depth left, depth right);
val depth = fn : ['a. 'a tree -> int]
# Node (2, Node (1, Empty, Empty), Empty);
val it = Node (2, Node (1, Empty, Empty), Empty) : int tree
# depth it;
val it = 2 : int
```

# データ型の宣言(まとめ)

## datatype 宣言による型定義

```
1 datatype 型変数列 型名 = 構成子 [of 型]  
2                               { | 構成子 [of 型] }*
```

- 型変数列の部分は、空列、型変数、もしくは、(型変数, ..., 型変数).
- of 以下には、いま定義している型を使うことができる.
- 右辺に用いる型変数は型変数列に入れる必要がある.

構成子がないと複数の種類の値が区別できないが、種類が1つなら、構成子がなくとも問題ない。その場合は type 宣言を用いる(⇒ 次ページ).

## type 宣言による型定義

データ型の宣言では構成子が必要だった。構成子がない場合は、type 宣言を使う。この場合、型の種類は1つだから、単なる型の別名と考えてよい。

### type 宣言による型定義

```
1 type 型変数列 型名 = 型
```

- 型変数列の部分は、空列、型変数、もしくは、(型変数, ..., 型変数)。

```
- type 'a stack = 'a list;  
type 'a stack = 'a list  
- type ('a,'b) dict = ('a *'b) list;  
type ('a,'b) dict = ('a * 'b) list
```

# 実習課題 (1)

スライドp.8の'a tree型を宣言して、以下の関数を与えよ。

- ① 整数 $x$ と整数型をノードにもつ二分木が与えられたとき、含まれるノードの値がすべて $x$ 未満であるかどうかを返す関数 `ltAll`

```
# val t = Node(1,Node(3,Empty,Node(2,Empty,Empty)),Node(4,Empty,Empty));
val t = ...
# ltAll 5 t;
val it = true : bool
```

- ② 整数型をノードにもつ二分木が与えられたとき、含まれるノードの値の合計を返す関数 `sum`

```
# val t = Node(1,Node(3,Empty,Node(2,Empty,Empty)),Node(4,Empty,Empty));
val t = ...
# sum t;
val it = 10 : int
```

# 実習課題 (1)

- ③ データの対 $(x, y)$ と二分木が与えられたときに、その木の中の $x$ を $y$ に、 $y$ を $x$ に交換する関数 `swap`

```
# val t = Node(1, Node(3, Empty, Node(2, Empty, Empty)), Node(4, Empty, Empty));  
val t = ...  
# swap (1,2) t;  
val it = Node(2, Node(3, Empty, Node(1, Empty, Empty)), Node(4, Empty, Empty))  
: int tree
```

- ④ 整数型の二分木が与えられたときに、その木に含まれる最大値を (あれば)返す関数 `getMax`

```
# val t = Node(1, Node(3, Empty, Node(2, Empty, Empty)), Node(4, Empty, Empty));  
val t = ...  
# getMax t;  
val it = SOME 4 : int option
```

# 目次

① データ型

② データ型の利用

## データ型の利用例(1) 命題論理式

命題論理式はBNFを用いて次のように定義できた。

$$\begin{aligned} P \in \text{Var} & ::= P_0 \mid P_1 \mid \dots \\ A, B \in \text{Prop} & ::= P \mid (\neg A) \mid (A \wedge B) \mid (A \vee B) \mid (A \rightarrow B) \end{aligned}$$

データ型を用いて、ほぼ同じように、命題論理式の型を定義することが出来る。

```
1  datatype prop = VAR of int | NOT of prop
2      | AND of prop * prop | OR of prop * prop
3      | IMP of prop * prop
```

- 命題論理式のデータ型を宣言せよ。
- 命題論理式  $P_0 \wedge P_1 \rightarrow P_0$  を表わす prop 型の式を与えよ。

# 命題論理式の付値と解釈(復習)

命題変数集合 $\text{Var}$ から真理値集合 $\text{Bool}$ への関数のことを**付値**とよび、付値 $v$ のもとでの命題論理式 $A$ の**解釈**  $\llbracket A \rrbracket_v$ は以下のように与えられた。

$$\begin{aligned}\llbracket P \rrbracket_v &= v(P) \quad (\text{ただし, } P \text{ は命題変数}) \\ \llbracket \neg A \rrbracket_v &= \begin{cases} \text{F} & (\llbracket A \rrbracket_v = \text{T} \text{ のとき}) \\ \text{T} & (\llbracket A \rrbracket_v = \text{F} \text{ のとき}) \end{cases} \\ \llbracket A \wedge B \rrbracket_v &= \begin{cases} \text{T} & (\llbracket A \rrbracket_v = \llbracket B \rrbracket_v = \text{T} \text{ のとき}) \\ \text{F} & (\text{それ以外}) \end{cases} \\ \llbracket A \vee B \rrbracket_v &= \begin{cases} \text{F} & (\llbracket A \rrbracket_v = \llbracket B \rrbracket_v = \text{F} \text{ のとき}) \\ \text{T} & (\text{それ以外}) \end{cases} \\ \llbracket A \rightarrow B \rrbracket_v &= \begin{cases} \text{F} & (\llbracket A \rrbracket_v = \text{T}, \llbracket B \rrbracket_v = \text{F} \text{ のとき}) \\ \text{T} & (\text{それ以外}) \end{cases}\end{aligned}$$

## 実習課題 (2)

- ① 命題論理式  $expr$  が与えられて、 $expr$  に含まれる命題変数(の添字)の整数リストを返す関数 `vars`

```
# vars;  
val it = fn : prop -> int list  
# vars (IMP (OR (VAR 2, VAR 1), NOT (VAR 2)));  
val it = [1, 2] : int list
```

- ② 命題論理式を文字列に直す関数 `propToString` (命題論理式の表わし方は適当で構わない。)

```
# propToString;  
val it = fn : prop -> string  
# propToString (IMP (OR (VAR 2, VAR 1), NOT (VAR 2)));  
val it = "(p2 | p1) -> (!p2)" : string
```

これとは逆に、文字列が与えられたときに、データ型の値に直す作業を「**構文解析**」とよぶ。

## 実習課題 (2)

- ③ 付値  $v$  を  $\text{int} \rightarrow \text{bool}$  型の関数によって与えるものとする。命題論理式  $expr$  と付値  $v$  が与えられて、付値  $v$  のもとでの  $expr$  の解釈  $\llbracket expr \rrbracket_v$  を返す関数 `eval` を与えよ。

```
# eval;
val it = fn : (int -> bool) -> prop -> bool
# fun v x = (x mod 2 = 0);
val v = fn : int -> bool
# eval v (AND (OR (VAR 0, VAR 1), VAR 2));
val it = true : bool
```

(ヒント)     `fun eval v (VAR i) = v i (* 引数にとった関数を使って計算 *)`  
              `| eval v (NOT e) = not (eval v e)`  
              `| eval v (AND (e1,e2)) = (eval v e1) andalso ...`  
              `.....`

このように、関数を引数にとるような関数(高階関数)を用いることができる。高階関数については後半で学習する。

## データ型の利用例(2) 二分探索木

二分木において、どの木Node  $(v, l, r)$  においても、

- $v$  は  $l$  に含まれるどのノードの値よりも大きい
- $v$  は  $r$  に含まれるどのノードの値よりも小さい

という性質が成立するとき、その木を**二分探索木**という。

うまく二分探索木を作ると、効率良くデータを検索できる。例えば、二分探索木のなかにデータが含まれているかどうかを判定する関数 `member` は以下のように書ける。

```
fun member Empty x = false
  | member (Node (v, l, r)) x =
    if x < v then member l x
    else x > v then member r x
    else true
```

## 実習課題 (3)

以下の関数を与えよ。

- ① 二分探索木に(その値がまだなければ)値を追加する関数

`insert`

```
# val t = insert 5 Empty;
val t = Node (5, Empty, Empty) : int tree
# val t2 = insert 3 t;
val t2 = Node (5, Node (3, Empty, Empty), Empty) : int tree
```

- ② 二分探索木の性質を使って、二分探索木に含まれる値をソートされたリストで返す関数 `btreeToList`

```
# val t = insert 7 (insert 8 (insert 2 (insert 6 Empty)));
val t = ...
# btreeToList t;
val it = [2, 6, 7, 8] : int list
```

## 実習課題 (3)

- ③ 与えられた木が二分探索木の制約を満たしているかを、ブール型で返す関数 `isBinSearchTree`

```
# val t = Node(1,Node(3,Empty,Node(2,Empty,Empty)),Node(4,Empty,Empty));  
val t = ...  
# isBinSearchTree t;  
val it = false : bool  
# val t2 = Node(6,Node(3,Empty,Node(4,Empty,Empty)),Node(8,Empty,Empty));  
val t2 = ...  
# isBinSearchTree t2;  
val it = true : bool
```

## 実習課題 (4)

- ① 変数, 定数, 加算, 乗算をもつ多項式をデータ型で定義せよ.
- ② 与えたデータ型に関して, どのような関数があるとよいかを自分で考え, それらの関数を実現せよ.