

# プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 2

講義の参考書: Peter Sestoft, “Programming Language Concepts”, Springer, 2012. (ただし、今年度は、講義スライドだけで理解できるようにする予定。講義をこえた advanced な内容はこの本を参考にされたい。)

今日のテーマ:

- メタ言語と対象言語、式と文、構文と意味
- スコープ, 束縛、静的束縛、閉じた式、自由変数

# 目次

- 1 メタ言語と対象言語
- 2 式と文
- 3 OCaml の式と型
- 4 非常に小さな言語の意味論

「木は左右対称だ。」

- 読み方その1：木という文字は、左右対称だ。
- 読み方その2：木が表しているもの（つまり、現実世界の木）は、左右対称だ。

より精密な書き方

- 「木」は、左右対称だ。
- 木は、左右対称だ。

「木は左右対称だ。」

- 読み方その1：木という文字は、左右対称だ。
- 読み方その2：木が表しているもの（つまり、現実世界の木）は、左右対称だ。

より精密な書き方

- 「木」は、左右対称だ。
- 木は、左右対称だ。

「木は左右対称だ。」

- 読み方その1：木という文字は、左右対称だ。
- 読み方その2：木が表しているもの（つまり、現実世界の木）は、左右対称だ。

より精密な書き方

- 「木」は、左右対称だ。
- 木は、左右対称だ。

メタ言語とオブジェクト言語 (対象言語)

- **対象言語**: 語られる対象を記述する言語
- **メタ言語**: 語る側を記述する言語

# この授業での使い方

## 対象言語

- 構文や意味を定義「される」言語
- 例: C, OCaml など

## メタ言語

- 構文や意味を定義「する」言語
- 例: BNF, 意味定義  $[[M + N]] = [[M]] + [[N]]$ , OCaml

今後、OCaml あるいは F# をメタ言語として使う。

# 目次

- 1 メタ言語と対象言語
- 2 式と文
- 3 OCaml の式と型
- 4 非常に小さな言語の意味論

## Expression (式)

17

3-4

7.9 + 10

## Statement (文)

```
printf("The answer is %d\n", 17);  
x = y * 2 + 1;
```

プログラム言語によっては、どちらかしかないこともある。

- C や Java では両方ある。
- OCaml は式のみ (print なども式である)

# 簡単な式 (算術式) の構文

式  $e$  の構文 in BNF ( $i$  は整数定数、 $p$  はプリミティブ演算子):

$e ::= i \mid p(e, e) \mid (e)$  具体構文

$e ::= \text{Int}(i) \mid \text{Prim}(p, e, e)$  抽象構文

上記の抽象構文を (メタ言語としての) OCaml のデータ型を使って、以下のように表現する。

```
type expr =
  | CstI of int           (* CstI(i) *)
  | Prim of string * expr * expr (* Prim(p, e1, e2) *)
```

# 簡単な式 (算術式) の構文

式  $e$  の構文 in BNF ( $i$  は整数定数、 $p$  はプリミティブ演算子):

$e ::= i \mid p(e, e) \mid (e)$  具体構文

$e ::= \text{Int}(i) \mid \text{Prim}(p, e, e)$  抽象構文

上記の抽象構文を (メタ言語としての) OCaml のデータ型を使って、以下のように表現する。

```
type expr =  
  | CstI of int           (* CstI(i) *)  
  | Prim of string * expr * expr (* Prim(p, e1, e2) *)
```

例 1.  $\text{CstI}(10) : \text{expr}$

例 2.  $\text{Prim}("+", \text{CstI}(2), \text{CstI}(-3)) : \text{expr}$

例 3.  $\text{Prim}("+", \text{CstI}(20), \text{Prim}("*", \text{CstI}(-3), \text{CstI}(5))) : \text{expr}$

# 目次

- 1 メタ言語と対象言語
- 2 式と文
- 3 OCaml の式と型
- 4 非常に小さな言語の意味論

## OCaml の基本

- 「式」のみ。(「文」相当の概念はあるが、「文」そのものはない。)
- 「式」は値を返す。また、「式」は型を持つ。
- 「関数」を組み合わせてプログラムを構成する。

## OCaml の型

- とても大事: 型が整合しないプログラムは, コンパイル時にはねられる (実行しない) .
- 基本型: int, bool, float, string など。
- 型構成子: t list や t array など (ここで t は型、たとえば int list は整数リストの型)
- 代数データ型: ユーザが定義する、後述

# OCaml における再帰関数

C や Java における再帰関数と同様。

```
let rec foo n =  
  if (n > 0)  
  then (foo (n / 2)) + 1  
  else 0  
in  
  foo 8  
  
==> 3
```

# OCaml の代数データ型

例: 2分木 (葉に整数, ノードにデータなし)

```
type binary_tree =  
  | Leaf of int  
  | Node of binary_tree * binary_tree
```

例 1: Leaf(13) : binary\_tree

例 2: Node(Leaf(13),Node(Leaf(5),Leaf(7))) : binary\_tree

注意: 変数や型の名前は小文字で始まる. 構成子 (Leaf など) は大文字で始まる.

例: 別の2分木 (葉とノードに浮動小数点数)

```
type binary_tree' =  
  | L of float  
  | N of float * binary_tree' * binary_tree'
```

例 1: L(3.14) : binary\_tree'

例 2: N(3.1,L(13.5),N(4.1,L(5.3),L(7.0))) : binary\_tree'

# OCaml の代数データ型

BNF 風のを「代数データ型」を使って書く。

```
type expr =  
  | CstI of int                (* CstI(i) *)  
  | Prim of string * expr * expr (* Prim(p, e1, e2)  
  *)
```

int, string は整数型、文字列型 (OCaml にもともとある)。  
expr は今、定義した型。

# OCaml の代数データ型

BNF 風のを「代数データ型」を使って書く。

```
type expr =  
  | CstI of int (* CstI(i) *)  
  | Prim of string * expr * expr (* Prim(p, e1, e2) *)
```

int, string は整数型、文字列型 (OCaml にもともとある)。

expr は今、定義した型。

OCaml の代数データ型:

- 型定義の開始キーワードは type
- 型の名前は、小文字で始まる (expr など)
- 定義の中身は、「ケース」を縦棒で区切る。
- 「ケース」は「構成子」から始まる; 大文字で始まる識別子 (CstI など)
- 引数があるときはキーワード of のあとに型が来る。
- 型の定義は再帰的でよい。

# OCaml の代数データ型

代数データ型の「作り方」(その型をもつデータをどうやって構成するか)

```
type expr =  
  | CstI of int  
  | Prim of string * expr * expr  
  
CstI(7) ;;  
Prim("+", CstI(7), CstI(10)) ;;  
Prim("+", CstI(7), Prim("*", CstI(10))) ;;
```

型の構成子が、0引数あるいは1引数の場合、かっこは省略してもよい。

```
CstI 7 ;;  
Prim "*" (CstI 10) (Cst 20);; (* ERROR *)
```

型の構成子は、他の型の定義で使ってはいけない。

# OCaml の代数データ型

代数データ型の「使い方」(その型をもつデータを使ってどう計算するか)

(\* リーフかどうか判定する関数 \*)

```
let goo e =  
  match e with  
  | CstI(i) → true  
  | Prim(s,e1,e2) → false
```

(\* 違う書き方 \*)

```
let rec goo' e =  
  match e with  
  | CstI(_) → true  
  | _ → false
```

(\* ノードの個数を数える関数 \*)

```
let rec foo e =  
  match e with  
  | CstI(i) → 0  
  | Prim(s,e1,e2) → foo(e1) + foo(e2) + 1
```

- 1 メタ言語と対象言語
- 2 式と文
- 3 OCaml の式と型
- 4 非常に小さな言語の意味論**

整数式と加算式だけをもつ言語の意味：

$$\begin{aligned} \llbracket \text{Int}(n) \rrbracket &= n \\ \llbracket \text{Prim}(" + ", e_1, e_2) \rrbracket &= \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket \end{aligned}$$

- $e$  がこの言語の式するとき、 $\llbracket e \rrbracket$  で、「その式の意味」を与える。
- この場合は、「式の意味」は整数値である。
- 式の意味の定義も再帰的。
- どんな式も、意味が一意的に定まる。

# OCaml で書いたインタプリタ

```
let rec eval e =  
  match e with  
  | CstI i    → i      (* e が CstI i の形なら i を返す。 *)  
  | Prim("+", e1, e2)  
    → (eval e1) + (eval e2)  
    (* e がこのパターンなら e1 の計算結果と e2 の計算  
      結果を足したものを返す。 *)  
  | _ → failwith "unknown expression"
```

`let rec` は再帰関数の定義に使うキーワード (`rec` は recursive の意味)。  
`match e with |p1 → e1|... |pn → en` はパターンマッチ。  
`failwith "..."` は、エラーメッセージを出して異常終了。

# OCaml で書いたインタプリタ

```
let rec eval e =  
  match e with  
  | CstI i      → i  
  | Prim("+", e1, e2) → (eval e1) + (eval e2)  
  | _          → failwith "unknown expression"
```

実行例:

```
eval (Prim("+", CstI(3), CstI(5)))  
==> (eval (CstI(3))) + (eval (CstI(5)))  
==> (eval (CstI(3))) + 5 ==> 3 + 5 ==> 8  
eval (CstI (10)) ==> 10  
eval (Prim("+", CstI (10), CstI(20))) ==> 30  
eval (Prim("*", CstI (10), CstI(20))) ==> 例外発生
```