プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 2

授業の全体像

今日の 3 限: 講義

- 教科書 1 章: メタ言語と対象言語、式と文、構文と意味
- 教科書 2 章前半: スコープ, 束縛、静的束縛、閉じた式、自由変数
- (来週) 教科書 2 章後半: スタック機械、コンパイル、 PostScript
- (将来) 3 章は飛ばして、4 章へ行く

今日の 4 限: 演習

- まずは OCaml (または F#) に慣れることが必要
- 教科書 pp.245-255 の範囲
- 今日のレポート: 初めての人向けの課題と、「ソフトウェア技法」等で OCaml を勉強したことのある人向けの課題を両方出題して、どちらかを解答 (manaba システムから提出)

Classic な例題

「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： 木があらわしているもの (つまり、実際の木) 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

ほかの例

何がメタ言語に属し、何が対象言語に属すか？

- You are a student. という文の主語は You である。
- あなたは学生であるという文の主語はあなたである。
- 「あなたは学生である」という文の主語は「あなた」である。
- X から Y-Z を引いた式と X+Z から Y を引いた式は同じ値を持つ。
- 「X から Y-Z を引いた式」と「X+Z から Y を引いた式」は同じ値を持つ。
- $[\![X-(Y-Z)]\!]$ と $[\![(X+Z)-Y]\!]$ は同じである。
- $[\![X-(Y-Z)]\!] = [\![(X+Z)-Y]\!]$

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
- 先週の例: 整数定数と足し算等の演算がある言語
- 今後: OCaml や C 言語のサブセット

メタ言語

- 構文や意味を定義「する」言語
- 先週の例 1: BNF
- 先週の例 2: 意味定義 $[\![M+N]\!] = [\![M]\!] \dot{+} [\![N]\!]$ (M と N は対象言語の表現で、それ以外はメタ言語の表現、つまり、$+$ は対言語に、$\dot{+}$ はメタ言語に属する。)

なぜ、こんなことにこだわるのか？

- 今週以降、OCaml (あるいは F#) をメタ言語として使う。
- 対象言語が OCaml サブセットのときがある。

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):

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

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. `CstI(10) : expr`
例 2. `Prim("+",CstI(2),CstI(-3)) : expr`
例 3. `Prim("+",CstI(20),Prim("*",CstI(-3),CstI(5))) : expr`

式と文

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 では、print 文に相当するものも式である (unit 型を返す式)。

OCaml 概要

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

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

C や Java における再帰関数と同様。

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`

BNF に近い。木をあらわしている。
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．

OCaml の代数データ型

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'`

同様にして，自分で木構造を定義できる。

OCaml の代数データ型

BNF 風のものを型として書ける。

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

ここで `int`, `string` は、OCaml がもともと持っている型 (整数型、文字列型)。`expr` は今定義した型。
OCaml の代数データ型:

- 型をユーザが定義できる; 開始キーワードは `type`
- 型名はユーザが選ぶ; 小文字で始まる識別子 (`expr` など)
- 定義の中身は、「ケース」を縦棒で区切って並べる。
- 「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、型が来る。
- 引数の型として、今定義している型を使ってよい (型の定義も再帰的)。

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。

```
type expr2 =   (* NOT GOOD *)
  | CstF of float
  | Prim of string * expr2 * expr2
```


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
```


意味

この小さな言語の意味：

$$[\![\mathtt{Int}(n)]\!] = n$$
$$[\![\mathtt{Prim}(\text{"+"}, e_1, e_2)]\!] = [\![e_1]\!] \dot{+} [\![e_2]\!]$$

OCaml でのインタープリタ：

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


OCamlで書いたインタプリタ

```
let rec eval e =          再帰関数 eval の定義を始める、
 match e with             e に関するパターンマッチ
  | CstI i   → i          e が CstI i の形なら i を返す。
  | Prim("+", e1, e2)     e がこのパターンなら
      → (eval e1) + (eval e2)
                          の計算結果との計算結果を足し e1e2
                          たものを返す
  | _ → failwith "unknown expression"
                          が上記以外の形なら、efail する
```

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
```


(参考) 抽象機械による意味論

CK 機械 (C=constrol string or code, K=continuation or stack)

$$
\begin{aligned}
e &\to \texttt{eval}\langle e \mid \texttt{init}\rangle \\
\texttt{eval}\langle \mathit{CstI}(n) \mid K\rangle &\to \texttt{apply}\langle K \mid n\rangle \\
\texttt{eval}\langle \mathit{Prim}(\text{"+"}, e_1, e_2) \mid K\rangle &\to \texttt{eval}\langle e_1 \mid \mathit{plus1}(e_2)::K\rangle \\
\texttt{apply}\langle \mathit{plus1}(e)::K \mid n\rangle &\to \texttt{eval}\langle e \mid \mathit{plus2}(n)::K\rangle \\
\texttt{apply}\langle \mathit{plus2}(n_2)::K \mid n_1\rangle &\to \texttt{apply}\langle K \mid n\rangle \quad (n_2 \dot{+} n_1 = n) \\
\texttt{apply}\langle \texttt{init} \mid n\rangle &\to n \quad \text{(最終結果)}
\end{aligned}
$$

抽象化されたハードウエアでの処理をあわらす：上記の場合、プログラム C をスタック K だけで処理するインタプリタ
例: $\mathit{Prim}(\text{"+"}, \mathit{CstI}(3), \mathit{CstI}(5)) \to \cdots \to 8$