プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 3

整数定数と加算を持つ言語の構文

式 e の抽象構文: BNF による表現

$$e ::= \mathtt{Int}(i) \mid \mathtt{Prim}(p, e, e)$$

例: $\mathtt{Prim}(" + ", \mathtt{Int}(5), \mathtt{Int}(3))$
式 e の抽象構文: OCaml のデータ型による表現

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

例: `Prim("+",CstI(5), CstI(3))`
→ OCaml を少し勉強しよう。

OCaml における再帰関数の定義

再帰関数そのものは、言語によらない。(C,Java,Scheme,OCaml ですべて同じ考えかた)。

以下の関数を書きなさい。入力は正の整数 1 つ。出力は，それが，2 で何回割り切れるか，という回数.
答えの一例

```
let rec f n =
  if (n mod 2 > 0) then 0
  else
     (f (n / 2)) + 1
```

注. この関数定義は、$n \geq 1$ であることを仮定している。(`f 0` は停止しない。)

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 =   (* ERROR *)
  | 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
```


整数定数と足し算を持つ言語の意味

意味を (OCamlでの) インタプリタで記述する。

```
let rec eval e =     再帰関数 eval の定義を始める、仮引数は e
 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 Prim(...) 形なら
      -> (eval e1) + (eval e2)
                        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"
                        e が上記以外の形なら、fail する

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


この章の目的

対象言語を拡張; 変数と変数束縛
その意味を記述する
自由/束縛変数, スコープ、出現、代入
コンパイルとは？
スタック機械

一日でできる量だろうか？
Yes, we can! (もし、皆さんが授業後に、プログラムをさわっていろいろ試してくれるなら。。。)

変数と変数スコープ

スコープ (scope):

```
let x = 10 in
  (let x = 20 in
    x + 10)
  * (x + 3);;
let x = 10 in
 let f x = x + 3 in
  f x ;;
```

キーワード:

- 静的スコープと動的スコープ
- ブロック構造言語
- スコープの入れ子 (nest)

変数の出現と束縛

項 e における変数 x の出現は、x をスコープに持つ束縛 (binding) があるとき、「e において束縛されている」と言う。
そうでないとき「e において自由」と言う。
複数の束縛下にあるとき (入れ子のとき)、一番内側の束縛を取る。

どの x が、自由/束縛 出現か考えなさい。

```
let x = 10 in
 let f x = x + 3 in
  f x ;;
```


束縛のある言語の構文

以前の expr の定義をやめて、新たに定義しなおす。

```
type expr =
| CstI of int
| Var  of string  (* 変数 *)
| Let  of string * expr * expr (* 変数束縛 *)
| Prim of string * expr * expr
```

例: `Let("x", CstI(10), Prim("+",Var("x"),Cst(20)))`

束縛のある言語の意味

```
let rec eval e env =
 match e with
 | CstI i   -> i
 | Var x    -> lookup env x
 | Let(x, erhs, ebody)
     -> let xval = eval erhs env
        let env1 = (x, xval) :: env
         eval ebody env1
 | Prim("+", e1, e2) -> (eval e1 env) + (eval e2 env)
 | _ -> failwith "unknown expression"
```

lookup 関数:

```
lookup [("a", 3); ("x", 7); ("c", 10)]  "x"
  ==> 7
("x", 10) :: [("a", 3); ("x", 7); ("c", 10)]
  ==> [("x", 10); ("a", 3); ("x", 7); ("c", 10)]
```

演習: let x = 10 in x + 20 という式を頭の中で評価してみなさい

自由変数

式の中に自由に出現する変数のリストを求める。

```
let rec freevars e =
 match e with
 | Cst I -> []
 | Var x -> [x]
 | Let(x,erhs, ebody) ->
     union(freevars erhs, minus (freevars ebody, [x]))
 | Pri(ope, e1, e2) ->
     union(freevars e1, freevars e2)
```

完成したプログラムは、当然、自由な変数出現があっては困るので、`freevars e = [ ]` となることが (きちんとコンパイルできることの) 必要条件である。

代入

```
subst e1 [("x", e2)]
```

$e1$ の中の変数 x の自由な出現をすべて $e2$ にする変換。(ただし、$e2$ が別の変数の自由な出現をもっているときは、ややこしいので注意)
→ 次回の演習できちんとやる予定。

スタック機械

Instruction	before	after	Explanation
RCst i	s	s,i	Push
RAdd	x,i1,i2	s,(i1+i2)	Add
RDup	x,i	s,i,i	Duplicate
RSwap	x,i1,i2	x,i2,i1	Swap

宿題

スタック機械があるとき、整数定数と足し算だけの言語はどう処理できるか考えてみてください。