# プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 3



## 目次





## 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。



## 目次

## OCaml における再帰関数の定義

「再帰関数」の概念は、特定の言語のものではない。
(C,Java,Scheme,OCaml,Haskell etc.)

例: 与えられた正の整数に対して、「偶数なら 2 で割り、奇数なら 3 倍して 1 を足す」ことを無限に繰返す関数 f. (ただし f(1)=1)

```
let is_even n = (n mod 2 = 0)
let rec f n =
  if n = 1 then 1
  else if is_even n then f (n / 2)
  else f (n * 3 + 1)
```



## OCaml における再帰関数の定義

「100 回まわったところで答えを返す」方法は？ 帰納法 (?)

- f(n) を f(n-1) であらわす。。。といっても無理。
- f(n,k) ... k 回まわったところでの答えを返す。
- f(n,k) を f(X,k-1) であらわす、、、、ことは可能。

```
let rec f n k =
  if k <= 0 then n
  else if n = 1 then 1
  else if is_even n then f (n / 2) (k - 1)
  else f (n * 3 + 1) (k - 1)
let f0 n = f n 100
```

新しい f は (実質的に)2 つの引数を持つ関数。

```
val f : int → int → int = <fun>
```

この型が意味するところ (カリー化) は、後日説明。



## 再帰関数のポイント

再帰関数といえども、停止するためには、パラメータの値がどんどん「減って」いく必要がある。
しかし、しばしば、与えられたパラメータとは別のパラメータが「減って」いく。

- 「m が素数であるかどうかを返す関数」...m を 1 ずつ減らしても意味がない。m を固定して、「2 から m-1 までのすべての整数で m を割る」演算をしたい。つまり、m とは別に、「2 から m-1 までを動く変数 k」をパラメータにもつ関数 g(m,k) を作って、k をどんどん動かすコードにすればよい。

⇒ウェブにある「再帰関数」の説明を読んでください。



## 目次

## この章の目的

- 対象言語を拡張; 変数と変数束縛
- 自由/束縛変数, スコープ、出現、代入
- スタック機械へのコンパイル



## 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 (ここでは静的スコープのみ扱う)
- 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
| Prim of string * expr * expr
| Var  of string   (* 変数 *)
| Let  of string * expr * expr  (* 変数束縛 *)
```

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

## 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)
```

式 e に自由変数がないこと (`freevars e = [ ]`) が、e がまともな式 (コンパイルできる式) であ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。



## 変数と変数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 in
        let env1 = (x, xval) :: env in
          eval ebody env1
  | Prim("+", e1, e2) → (eval e1 env) + (eval e2 env)
  | _ → failwith "unknown expression"
```

env == 環境 (次のページ)
補助関数として lookup を使っている。



## 実行時の環境 (environment)

変数の束縛を表す (変数とその値の対応): 例: x=10, y=20, z=30
ここでは、OCaml のリストを使って表現する。

空の環境 ... []
x=10, y=20 ... ["x",10); ("y",20)]

OCaml の組 (タプル) とリスト

- 組 (a, b, c)
- リスト: [ a; b; c ]

組とリストの違いは、型にある。

(10,"abc",true) と [10;20]=[20;30;40] は OK
[10;"abc";true] と (10,20)=(20,30,40) は駄目

## 環境の操作

初期値 (空の環境) `[ ]`
環境の延長 (extend) `a :: b`

```
 ("x",10) :: [("y",20); ("z",30)]
-> [("x",10); ("y",20); ("z",30)]
```

環境からの値の取り出し (lookup)

```
 lookup [("x",10); ("y",20); ("x",30)]  "x"
-> 10    (30 ではない)
```

後から入れたものが優先される.



## 評価の例

- let x=10 in x+20
- let x=10 in let x=30 in x+20
  - `x+20` を評価する時の環境は `[("x",30); ("x",10)]`
- let x=10 in (let x=30 in x+20) + x * 5
  - `x+20` を評価する時の環境は `[("x",30); ("x",10)]`
  - `x*5` を評価する時の環境は `[("x",10)]`



## 演習問題

今回の eval (変数と変数束縛のある言語に対する評価器) のもとで、以下の式を実行した過程 (1 ステップずつの実行) を手動でやってみなさい。

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

(この式は `let x=10 in (let x=20 in x + 20) + x` を表す。)



## 目次

## 簡単なスタック機械

- 1 本のスタックをもつ抽象機械:
  - プログラムは命令列として与える。
  - スタック以プ外にはメモリはない。
- 命令 (in BNF):

$$r ::= \texttt{RCstI}\ i \mid \texttt{RAdd} \mid \texttt{RDup} \mid \texttt{RSwap}$$

  $i$ は整数定数とする.

このスタック機械の命令をあらわす OCaml の型:

```
type rinstr =
   | RCstI of int | RAdd | RDup | RSwap
```

例: `RCstI(10)`



## 簡単なスタック機械の意味

| 命令 | 実行前 | 実行後 | 説明 (参考) |
|---|---|---|---|
| RCst i | s | s,i | Push |
| RAdd | s,i1,i2 | s,(i1+i2) | Add |
| RDup | s,i | s,i,i | Duplicate |
| RSwap | s,i1,i2 | s,i2,i1 | Swap |

例 1 `[RCst 10; RCst 20; RCst 30; RAdd; RAdd]`
- 10+(20+30) の計算に相当する.

例 2 `[RCst 10; RCst 20; RAdd; RCst 30; RAdd]`
- (10+20)+30 の計算に相当する.

逆ポーランド記法 (reverse Polish form)



## 簡単なスタック機械の評価器

主要部分のみ (詳細は教科書またはコードを参照のこと)

```
let rec reval inss stack =
 match (inss, stack) with
 | ([]       , v :: _)     → v
 | (RCst i::r, s)          → reval r (i::s)
 | (RAdd  ::r, i2::i1::s)  → reval r ((i1+i2)::s)
 | (RDup  ::r, i1::s)      → reval r (i1::(i1::s))
 | (RSwap ::r, i2::i1::s)  → reval r (i1::(i2::s))
 | _ → failwith "undefined transition"
```

`inss` は命令列 (rinstr 型のリスト)
`stack` はスタック (リストで表現する)
例: `reval [RCst 10; RCst 20; RAdd] [] ==> [30]`



## 課題

整数定数と足し算だけの言語で書かれたプログラム (式) は、スタック機械の命令列としては、どういう風にあらわされるか?
例 1. 10+(20+30) は、以下の命令列に翻訳するとよい。
- `[RCst 10; RCst 20; RCst 30; RAdd; RAdd]`

例 2. (10+20)+30 は、以下の命令列に翻訳するとよい。
- `[RCst 10; RCst 20; RAdd; RCst 30; RAdd]`

翻訳=プログラム変換=コンパイル (Compilation)

## 課題の答

「整数定数と足し算だけの言語」から「スタック機械の命令列」への翻訳をする関数：

```
let rec comp e =
 match e with
 | CstI i    →  [(RCstI i)]
 | Prim("+", e1, e2) → (comp e1) @ (comp e2) @ [RAdd]
```

例 1: `comp (Prim("+", CstI 10, Prim("+", CstI 20, CstI 30)))`

- `(comp (CstI 10))@(comp (Prim("+", CstI 20, CstI 30)))@[RAdd]`
- `[(RCstI 10)]@((comp (CstI 20))@(comp (CstI 30))@[RAdd])@[RAdd]`
- `[(RCstI 10)]@([(RCstI 20)]@[(RCstI 30)]@[RAdd])@[RAdd]`
- `[(RCstI 10);(RCstI 20);(RCstI 30);RAdd;RAdd]`



## 課題の答

例 2: `comp (Prim("+", Prim("+",CstI 10, CstI 20), CstI 30))`

- `(comp (Prim("+", CstI 10, CstI 20)))@(comp (CstI 30))@[RAdd]`
- `((comp (CstI 10))@(comp (CstI 20))@[RAdd])@[(RCstI 30)]@[RAdd]`
- `([(RCstI 10)]@[(RCstI 20)]@[Radd])@[(RCstI 30)]@[RAdd]`
- `[(RCstI 10);(RCstI 20);RAdd;(RCstI 30);RAdd]`



## コンパイラ= 翻訳系

ここでは、以下の 2 言語間の変換:

- ソース: 整数と足し算からなる言語の項
- ターゲット: スタック機械の命令列

多くの場合,

- ソース: 高レベル言語のプログラム
- ターゲット: 低レベル言語 (機械語など) のプログラム
- コンパイルの過程で，プログラムを解析して，高性能プログラムへ変換することが多い (最適化)



## 発展課題

テキスト第 2 章では,「整数定数，四則演算，変数，変数束縛をもつ言語のプログラム」から「拡張されたスタック機械の命令列」への変換についても論じているが，具体的な実装は示していない．これをどう実現すればいいか，考えよ．

例: `let z = 17 in z + z` を以下の命令列の翻訳したい。

- `[SCstI 17; SVar 0; SVar 1; SAdd; SSwap; SPop]`
- `SVar n` は、スタックトップから $n$ 番目の値を取ってきて、スタックにプッシュする命令。
- 注意点: `z` が 2 回出現するが、翻訳された命令列では SVar 0 と SVar 1 という異なるインデックスで表現。

命令の構文:

```
type sinstr =
     | SCstI of int | SVar of int | SAdd | SPop | SSwap
```