全学共通科目･工学部専門科目

「計算機科学概論」
アルゴリズムとプログラミング
その4

五十嵐 淳
igarashi@kuis.kyoto-u.ac.jp
大学院情報学研究科
通信情報システム専攻

担当分のメニュー

- 6/21, 6/28: アルゴリズムについて
- 7/5: 出張につき休講
- 7/12, 7/19: プログラミングについて
- (うまい時間が見つからなかったため補講は行いません)

講義情報

http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cs-intro/

今日のメニュー

- JavaScript プログラミング入門(つづき)
  - データ構造
  - 整列アルゴリズム
- 他のプログラミングパラダイム
  - OCaml による関数プログラミング
  - Prolog による論理プログラミング

先週の復習

JS プログラム = 関数･手続き定義と文の列

- 文 … 命令の単位
- 式 … 値を計算するひとまとまり
  - いろいろな値の種類(数値、文字列、真偽値)
- 変数 … (式の)値を格納する箱
  - 宣言･初期化･参照･代入
- 関数･手続定義 … ひとまとめの処理に名前をつける
- 条件判断と繰り返し

プログラムその5(訂正版)

- 機能: キーボードからふたつの数を入力させ、その最大公約数を表示する

```
var x = Math.max(m, n);
var y = Math.min(m, n);
var z = x % y;

while (z != 0) {
   x = y;  y = z;  z = x % y;
}
```

JavaScript における配列

代表的操作

- 配列を作る式: [〈式0〉, 〈式1〉, ..., 〈式n-1〉]
  - 〈式i〉 の値を i 番目の要素とする n 要素配列
- 要素読み出し式: 〈式1〉[〈式2〉]
  - 配列〈式1〉の〈式2〉番目の要素
- 配列要素の更新: 代入文を使う
  - 例: a[0] = 30;
- 配列の長さを求める式: 〈式〉.length

念のためうましかソート再掲

- 入力: 配列X (長さ n)
- i = 0, 1, ..., n-2 まで以下を繰り返す
  - j = i+1, i+2, ..., n-1まで以下を繰り返す
    - X[i] と X[j] を比較し、X[j] が小さかったら X[i] と交換
- X を出力

うましかソート in JS

```
var i = 0;  var j;

while (i < x.length - 1) {
  j = i + 1;
  while (j < x.length) {
    if (x[i] > x[j]) {
      // swap x[i] and x[j]
      var b = x[j];  x[j] = x[i];  x[i] = b;
    }
    j = j + 1;
  }
  i = i + 1;
}
```

ふつうは for 文という while とは別の繰り返し用構文を使います

JavaScriptにおけるリスト(1/2)

- 配列と違い組込み機能として提供されていない
- 他の機能を組み合わせて表現する
  - 今回はレコード(JS用語ではオブジェクト)を使う
  - レコード = 名前(ラベル)が付いた値の集まり

```
var x = {name: “Igarashi”, room: 224};

alert(“My name is ” + x.name);
alert(“My office is at room ” + x.room);
```

JavaScriptにおけるリスト(2/2)

- 空のリストを特殊な定数 null
- 非空リストを
  {head: 〈先頭要素〉, tail: 〈後続リスト〉}
で表現

```
function print_list(l) {
    if (l == null) { return; }
    else {
      alert(l.head);
      print_list(l.tail); }
}
```

リストを使ったマージソート in JS

```
function add(x, l){
  return {head: x, tail: l};
}
function merge(l1, l2) {
  if (l1==null) { return l2; }
  else if (l2==null) { return l1; }
  else if (l1.head < l2.head) {
    return add(l1.head,
               merge(l1.tail,l2));
  } else {
    return add(l2.head,
               merge(l1, l2.tail));
} }
```

```
function partition(lst) {
  if (lst == null) {
    return {left: null, right: null}; }
  else if (lst.tail == null) {
    return {left: lst, right: null}; }
  else {
    var x1 = lst.head;
    var x2 = lst.tail.head;
    var rest = lst.tail.tail;
    var part = partition(rest);
    return {left: add(x1, part.left),
            right: add(x2, part.right)};
    }
}
```

```
function msort(l) {
  if (l == null || l.tail == null) {
    return l; }
  else {
    var part = partition(l);
    var sorted_left = msort(part.left);
    var sorted_right = msort(part.right);
    return merge(sorted_left,
                 sorted_right);
  }
}
```

今日のメニュー

- JavaScript プログラミング入門(つづき)
  - データ構造
  - 整列アルゴリズム
- 他のプログラミングパラダイム
  - OCaml を使った関数プログラミング
  - Prolog を使った論理プログラミング

プログラミングパラダイム

プログラミング(に対する考え方)・問題をどう捉えるか(モデリング)の規範

- 命令型プログラミング(imperative programming)
- 関数型プログラミング(functional programming)
- 論理プログラミング(logic programming)
- オブジェクト指向プログラミング(object-oriented programming)
  - などなど

命令型プログラミング

- 手続型(procedural)とも
- プログラム = 命令(≒文)の列
  - 状態(変数に格納された値)の更新 as プログラム実行
- 代入と繰り返し構文を多用

- 典型的な命令型プログラミング言語
  - C, FORTRAN, JavaScript

関数(型)プログラミング

- 式がプログラムの主要な構成要素
  - 式の計算 as プログラム実行
- 関数と再帰を多用
  - 数学的な関数定義に近い雰囲気でプログラムが書ける
- 関数も値である
- (変数の初期化以外の)代入はあまり使わない

- 典型的な関数型プログラミング言語
  - Scheme (Lisp の一種), OCaml, Haskell

「○○(型)プログラミング言語」

- 「○○(型)プログラミング」がしやすいように作られている言語
  - JavaScript で関数型プログラミングも可能
    - GCD, マージソートの例はかなり関数型
  - OCaml で命令型プログラミングも可能
- 最近のプログラミング言語には命令型・関数型両方の特徴を兼ね備えたものも多い

漸化式の定義から関数型プログラムへ

- フィボナッチ数列 fib_n の定義

  $fib_1 = 1,\ fib_2 = 1$

  $fib_n = fib_{n-1} + fib_{n-2}$

- フィボナッチ数列の第n項を求める関数定義(in JS)

```
function fib(n) {
    return (n==1 || n==2)?
                    1
                  : (fib(n-1) + fib(n-2));
}
```

フィボナッチ関数 in OCaml

```
let rec fib(n) =
  if n = 1 || n = 2
    then 1
  else fib(n-1) + fib(n-2)
```

“return”すら書かない!

第n項を計算する方法が書いてある、
というより、むしろ
第n項とは何かが書いてある

プログラミング言語OCaml

主な特徴

- 関数も値である
- パターンマッチ
- 静的型システム
- 多相型システム
- 型推論

関数も値である

- 数列の和 Σf(i) = f(1) + ⋯ + f(n) の定義:

```
let rec sum(f,n) =
  if n = 1 then f(1)
  else sum(f, n-1) + f(n);;
  (* 関数を表す引数 f *)

sum(fib, 5)
```

を計算すると 12になる

パターンマッチ

- データの形を指定して場合分けと分解を行う機能
- 例: 整数リストの要素和を計算する関数 list_sum

```
let rec list_sum(lst) =
  match lst with
      [] -> 0
    | x :: rest -> x + list_sum(rest);;

list_sum(4 :: 3 :: 10 :: 94 :: []);;
```

静的型システム

- プログラム実行前に演算と式の整合性を検査をする仕組み
  - 式の計算結果の種類で分類
    - 整数、文字列、リスト…
- 型(type) … 式の分類
  - 式 1+1 は整数型
  - 式 4 :: 3 :: [] は整数リスト型
  - 式 x * “Igarashi” には型が与えられない

    → 検査に通らない

静的型システムの利点

- プログラムの間違いの早期発見
  - 定義を入力するやいなや指摘してくれる!
- 型安全性: 検査に通ったプログラムからは値の種類にまつわるエラーが発生しない!
  - 強い型システム: 型安全性が保証された静的型システム
    - 強くない型システムを持つ言語の例: C言語

型推論

- 静的型システムを持つ言語の多くでは変数宣言にその変数にどんな型の値を格納するかを注記する
  - C 言語での変数宣言の例:
    ```
    int x = 13;
    ```
- 型推論は、その注記を推論する機能
  - 例: list_sum

多相型システム

- ひとつの関数の型がいろいろ変化することを許すような静的型システム

```
let rec length(lst) =
  match lst with
     [] -> 0
   | x :: rest -> 1 + length(rest);;

length(1 :: 2 :: 4 :: []);;
  (* int list を受け取る関数として *)
length("foo" :: "bar" :: []);;
  (* string list を受け取る関数として *)
```

マージソート in OCaml

```
let rec merge(l1, l2) =
  match l1, l2 with
      _, [] -> l1
  | [], _   -> l2
  | x :: rest1, y :: rest2 ->
      if x < y
        then x :: merge(rest1, l2)
      else y :: merge(l1, rest2);;
```

```
let rec partition(lst) =
  match lst with
        [] -> ([], [])
    | x :: [] -> (x :: [], [])
    | x :: y :: rest ->
      let (l2, l3) = partition(rest) in
      (x :: l2, y :: l3)
```

```
let rec msort(lst) =
  match lst with
       [] -> []
     | [x] -> [x]
     | _  ->
       let (l1, l2) = partition(lst) in
       merge(msort(l1), msort(l2))
```

論理プログラミング

プログラム = 「事実」と(既知の事実から新しい事実を導く論理的な)「規則」による述語・関係の定義の集まり

- 事実の例:
  - 「篠田麻理子はチームAのメンバである」
  - 「板野友美はチームKのメンバである」
- 規則の例:
  - 「x がチームAのメンバであるならば x はAKB48のメンバである」
  - 「x がチームKのメンバであるならば x はAKB48のメンバである」
- 事実に関する問い合わせ as プログラムの実行
  - 「板野友美はAKB48のメンバか?」→ yes
  - [X はAKB48のメンバである。X は何か?」
    → X = 篠田麻理子, X = 板野友美

Prolog

- 論理プログラミング言語
- 人工知能分野への応用で盛んに用いられた
- 「どう答えを求めるか」より「何が答えか」を記述
  - 宣言的(declarative)プログラミングと呼ばれることも
    - 関数型プログラミングも宣言的な風味が強い
- 対称的な入出力

例題 in Prolog

```
teamA(marikoShinoda).      % facts
teamK(tomomiItano).
akb(X) :- teamA(X).        % rules
akb(X) :- teamK(X).
akb(X) :- teamB(X).

?- akb(tomomiItano).       %queries
   Yes
?- akb(X).
   X=marikoShinoda
   X=tomomiItano
```

例題: 子孫関係

- 関係: X は Y の親である･祖父母である･先祖である
- 規則:
  - X が Z の親であり Z が Y の親ならば X は Y の祖父母である
  - X が Y の親であるならば、X は Y の先祖である
  - X が Z の親であり、かつ、Z が Y の先祖であるならば、X は Y の先祖である

```
grandparent(X, Y) :-
    parent(X, Z), parent(Z, Y).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :-
    parent(X, Z), ancestor(Z, Y).
```

子孫関係の問い合わせ

- 綱吉の先祖は誰か？

  ?- ancestor(X, tsunayoshi).

- 家康の子孫は誰か？

  ?- ancestor(ieyasu, X).

マージソート in Prolog

```
% merge(X,Y,Z): X と Y のマージは Z である
merge([], X, X).
merge(X, [], X).

merge([X|L1],[Y|L2],[X|L3]) :-
    X =< Y, merge(L1,[Y|L2],L3).

merge([X|L1],[Y|L2],[Y|L3]) :-
    X > Y, merge([X|L1],L2,L3).
```

```
% partition(X, Y, Z):
%   Y と Z は X の分割である
partition([], [], []).
partition([X], [X], []).
partition([X,Y|Rest], [X|L1], [Y|L2])
  :- partition(Rest, L1, L2).

% msort(X, Y): Y は X をソートしたものである
msort([], []).
msort([X], [X]).
msort([X, Y | L1], L2) :-
  partition([X, Y | L1], L3, L4),
  msort(L3, L5),
  msort(L4, L6),
  merge(L5, L6, L2).
```

まとめ

- 様々なプログラミングパラダイム
  - 命令・手続型、関数型、論理プログラミング
- 言語の選択 = 思考の選択
  - “... if thought corrupts language, language can also corrupt thought.”
    (George Orwell, “Politics and the English language”, 1946)

参考URL

- Try OCaml: http://try.ocamlpro.com
  - ブラウザ上でOCaml プログラムを走らせることができる
- Prolog in JavaScript: http://ioctl.org/logic/prolog-latest
  - ブラウザ上で Prolog プログラムを走らせることができる