# TSSI基盤技術研修コース
# — プログラミング言語 — ＃1

久野　靖*

2004.4.16

## はじめに

□ 本講座の目的→プログラミング言語に関する概念を整理し、体系的に理解すること

- オブジェクト指向に特に重点を置いている

□ 予想される利益:

- 同じことをするならより「簡単に」→プログラマの能力をよりよく活かす
- 言語の特徴を活かしたソフトウェア/システム設計
- 「なぜこうなっているのか」「こうだと何が嬉しいのか」が分かる

□ 構成:

- 第1回→プログラミング言語の諸概念
- 第2回→オブジェクト指向のさまざまな側面
- 第3回→並行/並列、スクリプト言語

## 1　計算機言語とは…

□ まず計算機言語とは何かというお話から。

- では計算機とは何をするもの?

□ プログラミング言語…計算機言語の一種。

□ 計算機言語とはどういう言語?　なぜ存在する?

### 1.1　計算機と人間の情報伝達

□ 計算機…情報を取り扱うための装置

□ 人間と計算機はどうやって情報をやりとりするか?

□ ハードウェア…入出力装置

- 入力→キーボード、マウス、マイク、…
- 出力→ディスプレイ、プリンタ、スピーカ、…

□ 具体的にどのような形で情報をやりとり?

□ 計算機に自然言語(日本語)で命令できたら嬉しいと思うか?　(Yes/No)

### 1.2　人工言語

□ 人間どうしの情報交換→自然言語(日本語、英語、…)

- 計算機とのやりとりには不向き
- あいまいさ、処理が複雑
- 人間を相手にするのと計算機を相手にするのでは違って当然(?)

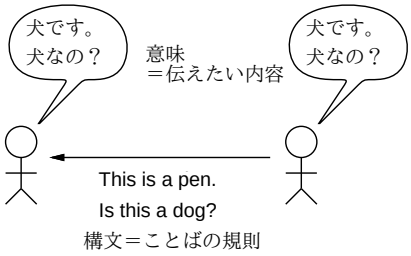
□ 簡単な規則に基づく人工言語→計算機の処理に向く

- 計算機で取り扱うために作った人工言語→計算機言語

### 1.3　構文と意味

□ 構文→言語に現われる語の並び方の規則性

□ 意味→どのような構文であれば何を意味するか




□ 言語の定義→構文と意味のペア(計算機言語も同様)

- ほかに語彙(単語)も必要だけど。

### 1.4　プログラミング言語

□ プログラムを書くための計算機言語

- プログラムとは?

□ プログラミング言語の用途?

*筑波大学大学院経営システム科学専攻

- 計算機に読み込ませる
- 人間が読む (娯楽、仕事、…)
- 自分が考えるための手段→でも動かした方が嬉しい

### 1.5 その他の計算機言語

□ プログラミング言語でない計算機言語も多数ある

□ 楽譜記述言語

□ 図形記述言語

- プリンタ→ページ記述言語→ PostScript…プログラミング言語でもある。

□ 文書記述言語 (マークアップ言語)、データ記述言語

- SGML → HTML → XML、XHTML

□ アプリケーションを動かし画面で見れば十分?

- 「処理」したいときは言語になっている方がよい。

### 1.6 この節のまとめ

□ 計算機言語とは、計算機で扱うための人工言語

- でも人間が読み書きしてもよい。
- 人間が考えるための「手段」でもある。

□ 代表はプログラム言語だが、他の計算機言語もいろいろある。

## 2 プログラミング言語の諸側面

□ プログラム→「人間が書くもの」だが、プログラム以前には無かったさまざまな特徴/特性がある

- 個別のプログラミング言語についても、その特徴/特性についていろいろな選択肢がある→以下で検討

### 2.1 プログラムが持つ特徴/特性

□ 簡単な C プログラムを題材に…

```
int main() {
  int x, y;
  printf("input x> "); scanf("%d", &x);
  printf("input y> "); scanf("%d", &y);
  if(x > y)
    printf("%d is larger.\n", x);
  else
    printf("%d is larger.\n", y);
  return 0;
}
```

- Q. このプログラムは何をするプログラムですか?
- Q. このプログラムにはどんな問題がありますか?
- Q. あなたが認識した「問題」を解決する方法は?

□ 問題かどうかはあくまで目的 (仕様) に照らさなければ分からない

- 仕様: 「x と y が等しければ何が出力されてもよい」だったら?
- 仕様: 「x と y に等しい数を入れてはならない」だったら?
- 仕様: 「x と y に等しい数が入ることはあり得ない」だったら?

### 2.2 プログラムが持つ特徴

□ 動作はきっちり決まっている (言語仕様に曖昧さがなければ)

- それが「プログラマの意図」さらには「ソフトの仕様」と一致しているかどうかはまた全然別の問題

□ 1 つの動作を行うプログラムは何通りも書ける

```
/* A */
if(x > y)
  printf("%d is larger.\n", x);
else
  printf("%d is larger.\n", y);

/* B */
if(x > y) z = x; else z = y;
printf("%d is larger.\n", z);

/* C */
z = (x > y) ? x : y;
printf("%d is larger.\n", z);

/* D */
printf("%d is larger.\n", (x>y)?x:y));

/* E */
z = x;
if(y > z) z = y;
printf("%d is larger.\n", z);
```

- どれがいいと思うか? それはなぜか?

□ もし 3 つの値 x、y、z の最大だったら?

```
/* D' */
printf("%d is larger.\n",
       (x>y)?((x>z)?x:z):((y>z)?y:z));

/* E' */
max = x;
if(y > max) max = y;
if(z > max) max = z;
printf("%d is larger.\n", max);
```

- こんどはどれがいいか?

□ 1 つの動作を行うプログラムは何通りも書ける

- そのどれがいいかは様々な側面を総合して決めるしかない
- プログラミング言語の機能やデザインもまさにそう!

## 2.3 プログラミング言語が持つ「要素」「概念」「機能」

□ また同じ C プログラムを題材に…

```
int main() {
  int x, y;
  printf("input x> "); scanf("%d", &x);
  printf("input y> "); scanf("%d", &y);
  if(x > y)
    printf("%d is larger.\n", x);
  else
    printf("%d is larger.\n", y);
  return 0;
}
```

- これにどのような「要素」「概念」「機能」が含まれている?

□ たとえば…

- 順次実行 (A; B; C)
- 制御構造 (if、while、…)
- 変数
- 型 (値の種別)
- 定数 (リテラル)
- 文字列
- アドレス (ポインタ)
- 関数 (サブルーチン)
- 名前
- 入出力

□ これらがどういう意味を持っているか考えて見よう…

## 2.4 実行順序

□ 「文が順次実行される」という考え方は「当然」か?

```
x = y * z;
y = 3.1416 + 1;
z = 2.7183 + 2;
```

- どういう順番で実行されると「感じられる」?

□ 方程式であれば、「参照関係に従って」計算する。
□ 関数型言語などでも同様。

```
double x() { return y() + z(); }
double y() { return 3.1416 + 1; }
double z() { return 2.7183 + 2; }
```

□ 再帰と組み合わせることで複雑な計算でも可能

```
double fact(n) {
  return (n < 1) ? 1 : n * fact(n-1);
}
```

□ ではなぜ現在の多くの言語は「順次実行」なのか?

- 手続き型言語のモデル←現在の計算機のモデル
- 現在の CPU は「命令を順番に実行して行く」モデルに従っている (実際には全然順番に実行していないことが多いが…)
- プログラム言語でもこれにならった方が自然 (?)

## 2.5 制御構造

□ 「枝分かれ」「繰り返し」などの*実行順序*を表す機構

```
if(x > y) {
  z = x; x = y; y = z;
}

while(n > 0) {
  fact = fact * n; n = n - 1;
}
```

- 実はこのように「制御構造の中に文の集まりが入る」という考え方が一般的になったのは 1970 年代の「構造化プログラミング運動」の結果。
- それ以前はこんな便利な (?) ものはなかった。

□ Fotran 60 (JIS Fortran 5000/7000)

```
   IF(X .LE. Y) GOTO 10
     Z = X
     X = Y
     Y = Z
10 CONTINUE

30 IF(N .LE. 0) GOTO 20
     FACT = FACT * N
     N = N * 1
     GOTO 30
20 CONTINUE
```

□ JIS 3000

```
   IF(X - Y) 20, 10, 10
20   Z = X
     X = Y
     Y = Z
10 CONTINUE
```

□ 構造化プログラミング運動

- GOTO はスパゲティプログラムになるからやめよう
- 構造化構文 (今の if や while) の入れ子を使おう

□ しかし「入れ子」がいいかどうかも疑問はある

```
if(a[low] < a[high]) {
  for(i = 0; i < high-low; ++i)
    for(j = low; j < high; ++j)
      if(a[j] > a[j+1]) {
        z = a[j]; a[j] = a[j+1]; a[j+1] = z;
      }
} else {
  for(i = 0; i < high-low; ++i)
    for(j = high; j > 0; --j)
      if(a[j-1] < a[j]) {
        z = a[j-1]; a[j-1] = a[j]; a[j] = z;
      }
}
```

- このコードが何をしているか読めるか?
- 読めるようになったとすれば「どう考えた」から?

□ 「入れ子」構造は、ある部分 (たとえば「文」が入る部分) に、複雑な構造であっても 1 つの単位として互換性があれば入れられる、という思想によっている。

- 計算機による処理は容易 (やり方が分っていれば)。
- しかし人間の頭はそういう風にはできていない。
- →プログラムの書き手としては、「人間に扱えるように」書く方がよい。
- →具体的にはどうする?

□ 「if」と「switch」と「while/for」と「do-while」でいいのか? もっとあった方がいいのか? 減らした方がいいのか? (yes/no)

□ 現在あるもので「一応」足りているが「まだ」あった方がよいかも。

- 定理: すべての (含むスパゲティ) プログラムは連接、分岐 (if)、反復 (while) の組合せに書き換えることができる。
- しかし「言語としての使いやすさ」が問題だからもっとあってもよい。
- 後述する「例外」などは新しい制御構造のバリエーション。

## 2.6 変数

□ 変数とは、何ですか? (知らない人に説明するとしたら?)

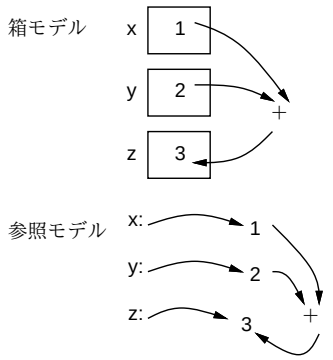
□ 変数の 2 つのモデル (どっちがいい?)

- 説明 A: 値を入れておく「箱」のようなもの。
- 説明 B: 値を表す名前 (値に名前をつけたもの)。

□ 「箱」のモデル…手続き型言語のモデル

- 手続き型言語…Fotran、COBOL、Pascal、C、C++、Java、…
- 変数に代入して値を書き換えて行く
- 計算機ハードウェア (メモリ) の自然な抽象化になっている
- 副作用ベース→分かりにくいバグの原因となることがある

```
while(i > 0) {
  fact = fact * i; i = i = 1;
}
```




□ 「値を表す名前」のモデル…数学に近い感じ

- 関数型/論理型言語…ML、Haskell、Prolog、…
- 変数には値が「束縛」される
- 一度束縛された値が書き変わることはない (単一代入とも言う)
- ただし変数は複数のインカーネーション (実体) を持つことも…

```
int fact(int n) {
  return (n < 1) ? 1 : n * fact(n-1);
}

fact(n:5)
↓ 5 * fact(n:4)
↓ 5 * 4 * fact(n:3)
↓ 5 * 4 * 3 * fact(n:2)
↓ 5 * 4 * 3 * 2 * fact(n:1)
↓ 5 * 4 * 3 * 2 * 1 * fact(n:0)
↓ 5 * 4 * 3 * 2 * 1 * 1
```

□ 結局: どちらのモデルも言語として成り立つ

- 優劣は一概に言えない (言語どうしで優劣が言えないのと同様)

## 2.7 型 (= 値の種別)

□ 「int x」「double x」「char *x」

- 何のために「型」があるのか?

□ さまざまな「型がある理由」(A)

- 入れる「箱」の大きさが種別によって変わるから。
- 種別ごとに演算 (演算命令) が違うから
- →これらは言語の設計次第でどうにでもなる。
- →ただし型がある方が効率はよい
- 型のない言語 (弱い型の言語) → Lisp、Smalltalk、Perl、…

□ さまざまな「型がある理由」(B)

- プログラマに「ここはどういう種類の値」という情報を書かせたい
- おかしなデータの使い型をチェックして教えたい
- →「よりよくプログラムを書くための道具」としての型
- 強い型の言語→ Pascal、C、C++、Java、ML、Haskell、…
- 型の明示 (変数宣言に必ず型を書く) と型推論 (「x = 10」なら x は int、のように使用関係に基づいて決定) とがある。

□ その他の区分

- 型には単純な型 (「整数」「実数」「文字」…基本型) と込み入った型 (「レコード」「配列」…複合型) がある。(オブジェクト指向言語のオブジェクトも複合型に属することが普通。)
- 「標準で用意されている型」「ユーザ定義の型」という区別があることもある。
- できるだけどんな型でも同じように区別なく使える方が望ましい。が。
- 例: C++の演算子定義→ユーザ定義の型でも演算子が使える。が、そのために言語的には複雑になっている。

## 2.8 定数 (リテラル)

□ 標準型の値を書き表す「プログラム言語上の記法/構文」

- 数値リテラル→「1」「3.1416」
- 文字 (列) リテラル→「'a'」「"ABC"」
- 論理値リテラル→「true」「false」
- その他…「null」「undefined」「NaN」「Infinity」

□ 「書き換えられない変数」と「名前のリテラル」の違い?

- 言語の構文に組み込まれているのがリテラル。
- だが言語によってはわりとあいまいな区別。

□ 複合型のリテラルは込み入った構文が必要なのであまりない。

- 「レコードリテラル」「配列リテラル」くらいなら言語によってはある。

□ ユーザ定義型と標準型を差別なく扱いたいが、リテラルが書けないことが障壁になることが多い。

## 2.9 基本型

□ 基本型→「直接 CPU が演算できるような値の型」(たいていは)

- 例: 整数、実数、文字、論理値、列挙値 (Pascal)
- しかし文字は演算しないんだけど…

□ (先の定義)「直接 CPU が演算できるような値の型」

□ 別の定義「マシンレジスタに載るような値の型」(あまりよくない)(なぜ?)

□ 別の定義「それ以上分解できないような値の型」(うーん…)

- 結局、言語仕様として「基本型に決めたから基本型」みたいな…

## 2.10 文字列

□ 文字列は「基本型」か「複合型」か?

□ 文字列リテラルは大抵の言語にあるが…

□ 古い言語だと…

- Fortran 60 →文字列リテラルはあったが文字列型はなかった!
- 文字列リテラルの書き方も恐怖!「3HABC」
- COBOL、PL/I あたりから文字列がある言語に。「文字が並んでいる」ということで配列と同様の扱いを受けることが多かった→複合型

□ 現在では必ずしも複合型とはかぎらない

- JavaScript、Perl の場合: 単一の値→基本型
- C/C++の場合: 「初期化された文字配列の先頭要素へのポインタ」→複合型
- Java の場合: 文字列オブジェクト値→複合型

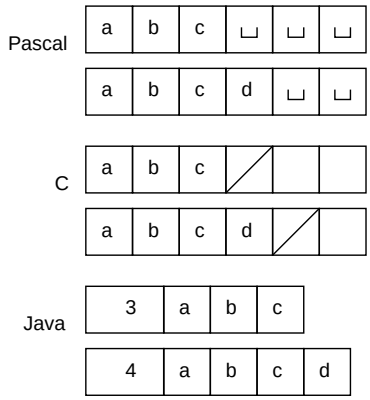
□ 「文字列は配列である」(複合型) の流儀→弱点 (どんな?)

□ 多くの言語で配列が持っている制約

- 大きさが伸び縮みできない→固定長文字列、または長さを別に管理
- 大きさが違うと型が違う (互換性がない!) ← Pascal

□ 文字列は長さ可変でないと不便すぎる→長さをどうするか?

- Pascal 流→終わりまで空白文字を詰める (何が困るか?)
- C 流→特別な「印」で終わりを表す (何が困るか?)
- 長さを別に持つ→これだけでもう配列ではない場合が多い (これが今は主流)




## 2.11 複合型

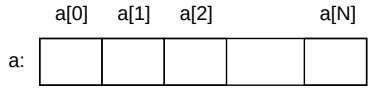
□ 複合型→「中に構造を持つ (別の値が含まれている) ような型」「込み入った型 :-)」

- 言語仕様上は「〜へのポインタ」「〜を返す関数」なども単純な型ではないので復号型に分類してしまうことが多い
- しかし基本は配列、レコード、ユニオン

## 2.12 配列型

□ 最も古くからある複合型 (FORTRAN で、ベクトルや行列演算のため)

- 同じ型の値が連続して並んでいる
- 添字により「何番目」を指定してアクセスする
- 実現上は単にメモリ上に必要なだけの領域を並べるのが基本 (だった) →しかしそれでは不便だと分かってきたので…(どう不便?)




□ 配列の「要素型の種別」は型の一部

- 「整数の配列」と「実数の配列」をごっちゃに使うわけに行かないから

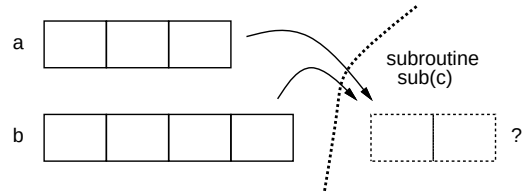
□ 配列の「長さ」ないし「添字範囲」は型の一部かどうか? (どっちがいい?)

- Pascal →型の一部である
- C、C++、Java →型の一部ではない

□ 「添字範囲」を型の一部にしてしまうとチェックはしやすい

- しかし融通が効かなくてすごく不便

□ 「添字範囲」を型の一部に含めないとチェックが難しくなる

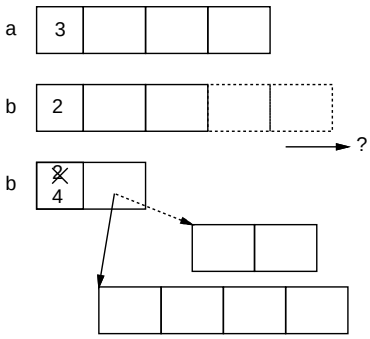



- パラメタで受け取った後、大きさ (添字範囲) が分からないと困る
- 全然困らないもーんという言語もある (C のこと)
- しかし C で 2 次元配列を渡すとすごく困るはず

□ 現在では「先頭に大きさを入れてしまう」が主流 (Java)

□ さらに進んで「大きさを実行中に変更できるように」というのも…

- ただしその場合、1 つのメモリブロックでは済まないという問題が…




## 2.13 レコード型/ユニオン

□ こちらは COBOL が元祖

□ 複数の「違う型の」値を組み合わせたもの

- 非常に基本的な型だった…はずが、オブジェクト指向のせいで影が薄く…

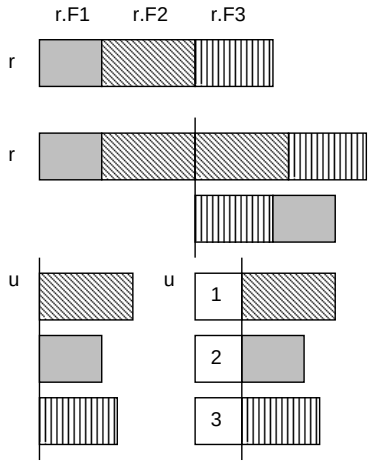
- Java にはレコード型はない

□ 可変レコード…レコードで「途中からフィールドを取り換える」ようにした← Pascal

- 場合によって違う種類の値が保持できるから便利 (?)

□ これを「取り換えずに並べるだけ」と「取り換えるだけ」に分離→ユニオン型 (C)

- 言語概念的にはきれいだけど使う時はめんどくさい
- だいたいは複数のものを切替えたいのでレコードのユニオンになる
- もっとひどいとレコードの一部がユニオンでその中にまたレコードが入っている




□ 可変レコードもユニオンも「どの枝が使われているか」ちゃんとチェックしないと危険←しかし野放しが多かった

- オブジェクト指向言語ではほとんどユニオンはない (どうしてるの?)

## 2.14 アドレスとポインタ

□ 式「& x」の結果は何?

□ C 言語の式「& x」の結果として考えられるもの…

- 答 A: 変数 x の主記憶上の番地 (アドレス)
- 答 B: 変数 x を「指す」(参照する) 何らかの値
- 答 C: その他…

□ 言語仕様的には「B」。

- 言語仕様に「番地」は出てこない (実装に依存しすぎる)。
- int *p = …; … p + 1 … ←実際に足されるのは「1」ではない
- それはいいが、なぜこういうもの (ポインタないし参照) が必要なのか?

□ ポインタは何のために必要?

- 変数に値を書き込んで戻してもらうため←言語に参照渡しがない場合
- 動的データ構造 (連結リスト、2 分木、グラフ、…)
- 大きなデータを持ち回る非効率を避けたい場合
- データを複数個所で共有したい場合 (cf. 広域変数)

□ データの共有は副作用を可能にする→注意が必要

□ 言語設計上の問題点

- 任意の変数のポインタを取れるのは危険 (なくなった後も…)
- C/C++→配列の添字式がポインタ演算 (「a[i]」は「*(a+i)」と同じ意味) →非常に特異な言語設計、問題山積み
- 型 T と型*T を使い分けるのは混乱のもと (C++だとさらに & T もある)
- すべての値をポインタとするのも 1 つの選択肢 (Java がこれに近い)

□ 用語: 「参照 (reference)」か「ポインタ (pointer)」か?

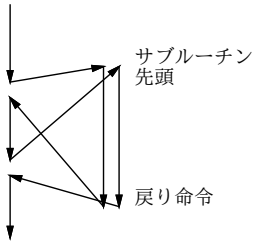
- 基本的には同じ意味 (アドレスを抽象化したもの)
- 「ポインタ」というと C/C++の「演算できるポインタ」の印象が強い
- 「参照」というと、対象を「指している値」というだけで、その値そのものは加工できないという印象が強い (等しいかどうかの判定くらいしか許されない)

## 2.15 関数

□ 関数 (サブルーチン) とは、何ですか?

□ 現象的には、「ある範囲のコードの先頭にジャンプし、そのコードを実行し、終わったら元の (ジャンプする前の) ところに戻って来る」

- 実装的には戻り番地 (ジャンプした命令の次の命令の番地) を保存しておいてそこへ間接ジャンプで戻る
- しかしそれが関数 (サブルーチン) だと言われても釈然としないでしょう?



□ プログラマから見ると→

- ひとまとまりのコードに名前をつけたもの
- その名前を指定して呼び出せる
- それぞれが完結した何らかの仕事をしてくれる (その細部は呼び側では知らなくてもよい)

□ 要するに「抽象化の手段」→最も重要な機能

- さっきの「入れ子をどうするか」の答えも第 1 義的にはここにある。
- 「いちど関数を書いてしまえば、言語にそういう命令が増えたものとして扱える」

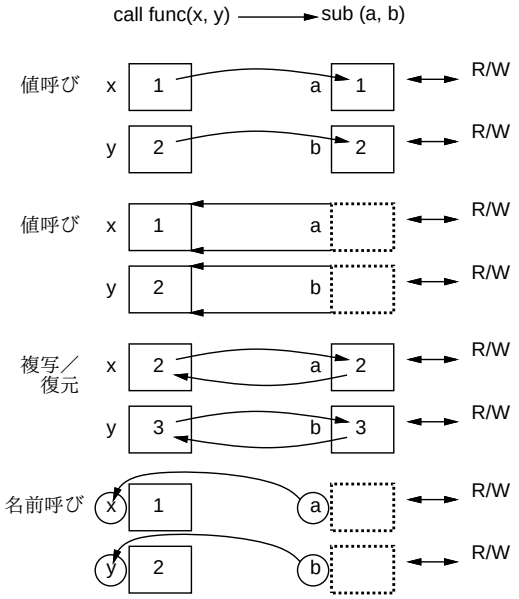
□ 関数の利用が普及したのは構造化プログラミングの時期以降

- それ以前だと「1 万行のメインプログラムだけ」とか珍しくなかった
- やはりプログラミングの技術/スタイルも進化している…

□ そのためには「パラメタを渡す」機能は不可欠

□ 引数渡し機構にもいくつかのバリエーション

- 値渡し (call by value)
- 参照渡し (call by reference)
- 複写復元 (copy-restore linkage)
- 名前呼び (call by name)




## 2.16 名前

□ プログラミング言語において「名前」とは?

- 名前 (name) →プログラムにおいて区別されるべき個々のものを指示するもの
- 識別子 (identifier) →複数の名前を相互に区別するための文字列
- 識別子 (名前文字列) はどういうものを指すことができるか?

□ 典型的な言語において識別子が指すもの

- 変数
- 関数
- 型
- レコードのフィールド (←変数?)
- 名前空間
- (cf. 予約語← if、while、…)

□ 名前は貴重な資源 (分かりやすい名前は限られている)

□ 名前どうしの衝突や混同をなくすには…

- 名前を長くする (いつも長いと大変…)
- コンテキストで区別する (プログラマの負担)
- プレフィクスをつけて指定する (毎回書くのも大変) ←「a.name」
- 正式には長い名前だが短く書けるようにもする (使い分け…) ← with 文、import

## 2.17 入出力

□ 古典的な言語では「入出力命令」が言語の一部に組み込まれていた

- Fortran の read 文/write 文、COBOL の read 命令、write 命令
- Pascal あたりまでやや特別扱い
- C からは「入出力用の関数を呼ぶだけ」に (cf. 可変引数)

□ 入出力は本質的に副作用

- 関数型のように副作用を持たない言語では扱いに工夫が必要

## 2.18 この節のまとめ

□ プログラミング言語には非常に多くの要素が含まれている

□ それぞれの要素について「そこをどのように設計するか」という複数の選択肢がある

- それぞれの選択肢の得失を知っておくと、それぞれの言語の得手不得手が読み取れるようになるはず

□ ここに出て来たのはまだまだ一部分。全部やろうとするといくらやっても終わらない

# 3 Java言語入門

□ 本講座では、以下具体的なプログラミング言語の実例(および課題レポート用)としてJavaを使用します。

- Java言語の処理系は各自で入手し使用してください。バージョンはJDK(J2SE) 1.3.1以降が望ましい。http://java.sun.com/j2se/ から入手可能。
- アプレットの表示もNetscape 6.x/7.x、Mozillaであれば最初からJava 2対応なので。MSIE 5.x/6.0もJDK 1.3.1以降を入れるとJava 2対応になる。
- #3でJavaScriptを使用するがこれも上記Netscape/Mozilla系の方がよいので…

## 3.1 Javaの歴史

□ SunMicrosystems社→Unix WSの老舗。

□ 90年代前半に、セットトップボックスやディジタル家電用のシステムを開発するプロジェクトを開始。

- 最初は言語として(オブジェクト指向は必須と考えたので)C++を予定→しかしC++はアドレス/ポインタが直接触れるため安全でない。
- このため、安全なオブジェクト指向言語としてJavaを開発。
- C++を安全にし、複雑さの原因となる機能を削除→言語専門家には高い支持
- アプレットのための言語として、世の中に急速に普及

□ 現在ではアプレットだけではなく…

- 特定プラットフォームだけでないアプリケーション開発
- ダウンロード可能なコード+小さい実行環境→組み込み、iアプリ等
- Webサーバ内への組み込み→サーブレット、JSP
- サーバ側コンポーネントプログラミング→EJB

## 3.2 オブジェクト指向言語

□ オブジェクト指向言語とは?→いろいろな定義があると思うがここでは一応次のように考える。

- 「オブジェクト指向」→プログラムが扱う対象をそれぞれ自律的な/完結した「もの」(オブジェクト)であるとする「考え方」
- 「オブジェクト指向言語」→オブジェクト指向の考え方をサポートするようなプログラミング言語

□ 抽象的でよく分からない? まあそうだと思いますが…

```
outstream ostream = ...;
outstream_println(ostream, "Hello, World.\n");
↑関数名が長い    ↑操作対象も引数 : 従来のスタイル

outstream ostream = ...;
ostream.println("Hello, World.\n");
↑オブジェクト.メソッド(引数…) : オブジェクト指向ぽい
```

- 同じことを2度言わなくて済む感じ
- 同じことには同じ名前が使える
- 名前空間の節約

□ あと、とりあえず関連する用語を紹介しておく

□ 「クラス」→オブジェクトの種類に対応する構文要素。

- 新しい種類のオブジェクトを定義したければ、クラスを書く。

□ クラス定義に含まれるもの…

- オブジェクトはどういう動作(メソッド)を持っているか←関数
- オブジェクトはどういう属性を持っているか←変数
- そのほかクラスはモジュールとしての役割りも←クラスメソッド、クラス変数

□ 「インスタンス」→クラス定義に基づいて作り出された「もの」(オブジェクト)

## 3.3 Javaはどんな言語?

□ オブジェクト指向言語

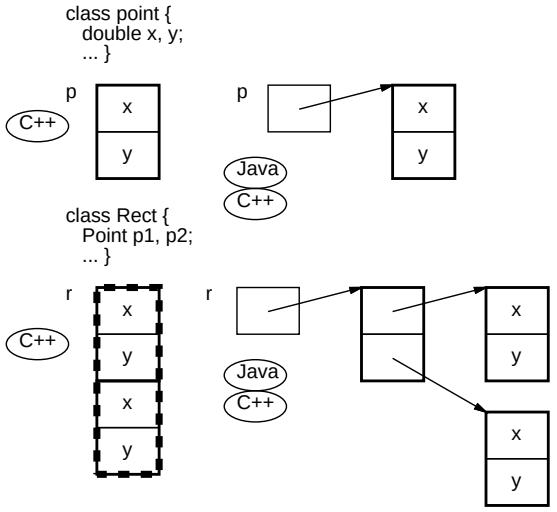
□ 構文はC++に似せているが、中身はまったく別もの

- 構文を似せることにどんな意味があったのか…
- ぶら下がりelse問題みたいなのは現代の言語なら最初から避けるはず

```
if(x > y)
  if(x > z) max = x;
else max = -1;
```

□ C++から多くの (複雑さの原因となる) 機能を取り除いて整理

- ポインタ演算、オペレータオーバーローディング、多重継承がない
- オブジェクトはすべてヒープ上のオブジェクト。オブジェクトはすべて参照で扱う→最も重要な単純化。動的データ構造のためにはどのみち参照が必要。そのため、すべて参照に統一している。




□ きちんとした/大規模なプログラミングに必要な機能を追加

- 「ヘッダファイル」をやめ、コンパイル済みコードに宣言情報を一体化
- パッケージ機能により、名前の衝突や混乱を回避
- クラスの中にクラス定義を入れられる
- プログラムの階層構造: Javaでは「パッケージ」→「クラス」→…→「メソッド、変数」
- Magic Number 7 ± 2…システムが複雑になって来ると階層を増やす必要

□ オブジェクト指向言語としての機能の洗練

- ガベージコレクション (GC、ごみ集め) が前提→メモリ管理で悩まないでいい
- インタフェース機能→実装と界面の分離 (最近のOOPの流れ)

□ その他の特徴→実行環境やライブラリの話題

- write once, run anywhere
- アプレット、セキュリティサンドボックス
- 豊富な標準API、標準拡張API(C++のようにバラバラでない)

□ Java 2 → 3 つの標準 API → J2SE(standart edition)==いわゆる JDK、J2EE(enterprise edition)、J2ME(micro edition)

- Java3D、Java Telephony API, Java Medita Framework, ...

## 3.4 例 1: Hello, World

□ ではとりあえず最初の例題

```
//--- Sample1.java ---

public class Sample1 {
  public static void main(String[] args) {
    System.out.println("Hello, World.");
  }
}
```

□ Java プログラムはクラスの集まり

□ クラスを指定して開始→そのクラスの public static void main(String[] args) というシグニチャ(型仕様) のメソッドから実行開始

- public : このメソッドはクラス外から呼べる
- static : このメソッドはクラスメソッド (オブジェクトを生成しなくても呼べる)
- void : このメソッドの返値はない
- String args[] : 引数が 1 つあり、その型は「String の配列」

□ クラス System のクラス変数 out に格納されているオブジェクト (PrintStream オブジェクト) のメソッド println() を呼ぶ→標準出力に文字列を書き出し改行

## 3.5 簡略化した構文

□ ごく簡略化した構文を示す

```
クラス   ::= [public] class クラス名
               { [変数...] メソッド... }
メソッド ::= [修飾子...] 型 メソッド名 ([引数,..])
               { 文... }
修飾子   ::= public | static | final
型       ::= 基本型 | クラス名 | 型 []
基本型   ::= boolean | byte | char | int | long |
               float | double
変数     ::= 型 (変数名 | 変数名 = 式),.. ;
引数     ::= 型 変数名
式       ::= リテラル | 式 演算子 式 | 演算子 式 |
             式 演算子 | ( 式 ) |
             式 . メソッド名 ([式,..]) |
             式 . 変数名 |
             クラス名 . メソッド名 ([式,..]) |
             クラス名 . 変数名
```

□ コメントは「//」から行末まで、または「/* ... */」(C++と同じ)

### 3.6 動かし方等

□ ソースファイル名の末尾は「.java」

□ クラス名とそれを格納するファイル名は一致している必要がある

□ コンパイラ:「javac」、実行:「java」

```
> javac Sample1.java  ←ファイルを指定
> java Sample1        ←クラス名を指定
Hello, World.
>
```

### 3.7 基本型とオブジェクト型

□ Java では「値」と「オブジェクト」の 2 種類のデータを扱う

- 「値」→基本型→整数、文字など「レジスタに載るようなデータ」
- オブジェクト→クラスによって定義され、ヒープ上に領域が割り当てられるようなデータ

□ 基本型をオブジェクトとして使いたい時には包囲 (wrapper) クラスを使う。int → Integer、char → Character など。

□ それぞれの基本型を操作する便利なメソッドも包囲クラスに入れる (クラスでないと入れる場所が無い)

□ 言語的には「基本型」「オブジェクト」の区別はない方が美しいと考える。

- 効率上やむを得ないから分けているという姿勢←本当にそうするしかないかどうかは疑問?
- 包囲クラスに入れるのを boxing、逆に取り出すのを unboxing と呼ぶことも。これをある程度自動化しようという動き→C # がはじめて、Java でも JDK 1.5 で入った。

### 3.8 例 2: 数値、クラス、配列

□ 数値は C あたりと同様「だが」上述のように数値の「オブジェクト」もある

- ただし演算は値の方でしかできない

□ オブジェクト→「new クラス名 (引数…)」で作る

□ 配列は見た目同じだがだいぶ違う (配列もオブジェクト)

```
//--- Sample2.java ---

public class Sample2 {
  public static void main(String args[]) {
    double sum = 0.0;
    for(int i = 0; i < args.length; ++i) {
      System.out.println(i + ":" + args[i]);
      Double d = new Double(args[i]);
      sum += d.doubleValue();
    }
    System.out.println("sum is: " + sum);
  }
}
```

- 「new Double(...)」で Double「オブジェクト」ができる
- 「d.doubleValue()」でそこから double「値」を取り出す

□ 文字列と何かの「+」→文字列に変換した上で連結

- 文字列への変換と連結の「+」はかなり特殊。便利だけど、何でもこれで表示すればよいというものではない (なんで?)
- 正式にはフォーマット用クラスを用いて行う→今回はすべて「さぼっている」

□ 実行例

```
> javac Sample2.java
> java Sample2 1.5 3 7
0:1.5
1:3
2:7
sum is: 11.5
>
```

□ もし数値でないものを入れると…

```
> java Sample2 1.5 abc 8
0:1.5
1:abc
Exception in thread "main" java.lang.Number
    FormatException: abc
  at
java.lang.FloatingDecimal.readJava
    FormatString(FloatingDecimal.java:1213)
        at java.lang.Double.valueOf(Double.java:183)
        at java.lang.Double.<init>(Double.java:258)
        at Sample2.main(Sample2.java:8)
>
```

- まあエラーになって当然ですが…

### 3.9 オブジェクトの生成とごみ集め

□ 先のコード→どんどん必要なオブジェクトを生成し、使わないものは GC で回収、というスタイル。

- GC のオーバヘッドは織り込み済みという割り切りが必要。
- 自分で領域開放するよりずっとよい。自前でやるとコードが複雑化し、また重大な誤りの原因になりやすい。

- GC は Lisp あたりでは古くから使われているが、普通の手続き型言語で標準としたのは Java がはじめて

□ GC の原理: 変数から参照できるオブジェクト群を順次たどって行き、たどったという印をつける。印がつかなかった領域はごみとして回収

- その他、使っているオブジェクトだけコピーする方式、オブジェクトがそれぞれ何個所から指されているか常に数えおく方式などもある。
- GC で自動的に回収するためには「不要なデータ構造は指さないようにする」(指している所に null を入れるなどしてつながりを切る) ことが必要
- 「もう要らないからこれ回収して」とは言えない (なんで?)

## 3.10 例外処理

□ 最近の言語で加わった新しい制御構造 (Lisp などでは古くからある)

□ もともとの問題: エラー検出はどうやるのがいいのか?

```
status = some_func(...);
if(status != OK) {
  エラー処理…
}
```

- どういう問題があるか?

□ 帰り値が使えなくなってしまう

□ 広域変数にするとマルチスレッドとか問題

□ エラー処理が各所に挿入されると処理の流れが見づらい

- どこか 1 箇所で処理したいが goto はあまり使いたくない
- エラーに飛び出したために後始末が飛ばされたりすると困る

□ エラー処理のための制御構造を導入→「例外」処理

```
try {
  ... 何かあるかも知れない
  ... 処理をいくらでも書く
} catch(Exception ex) { ←エラー情報受け取り
  ここでまとめて処理
}
```

□ 「どの範囲のエラー」という取捨選択が可能

□ 「必ずやりたい後始末」も指定できる

```
try {
  ...
  try {
    ... ←この中で起きたエラーで
  } catch(NumberFormatException e1) {
    ... ←数値書式エラーはここへ来る
  } finally {
    必ずやる後始末
  }
  ...
} catch(Exception ex) {
  ... ←他のエラーは直接ここへ来る
}
```

□ 受け止めなかった例外は?

- メソッド呼び出し側に返される (そこに try-catch があればそこで受け止められる)
- 最後まで受け止められなければプログラム実行を中止

□ 例外種別の分類→階層構造

```
Throwable ←例外すべて
  Error ←システム上の問題
    OutofMemoryError
    ...
  Exception ←普通の例外
    IOException
    InterruptedException
    RuntimeException ←どこでも起きるような例外
      NumberFormatException
      NullPointerException
      ...
```

□ メソッドは自分が返す例外の種類を「throws 種別, ...」という形で明示しなければならない (これを見ると何が発生し得るか分かる)

□ 自分が呼ぶメソッドで発生する例外は原則として受け止めて処理する

□ 受け止めないでもいいのは自分も同じ例外を throws で明示している場合のみ

□ ただし、すべてのメソッドは「throws Error, RuntimeException」ははじめから指定されているものと見なされる

## 3.11 例 3: 例外を受け止める

□ 先の例題をちょっと直して try-catch を入れる

```
//--- Sample3.java ---

public class Sample3 {
  public static void main(String args[]) {
    double sum = 0.0;
    for(int i = 0; i < args.length; ++i) {
      try {
        System.out.println(i + ":" + args[i]);
        Double d = new Double(args[i]);
        sum += d.doubleValue();
      } catch(Exception ex) {
        System.out.println("!" + ex);
      }
    }
```

```
    System.out.println("sum is: " + sum);
  }
}
```

□ 実行例…

```
> javac Sample3.java
> java Sample3 1.5 abc 8
0:1.5
1:abc
!java.lang.NumberFormatException: abc
2:8
sum is: 9.5
>
```

- 例外を受け止めて処理を続行している

## 3.12 例 4: 入出力ストリーム

□ System.in → InputStream オブジェクト (バイト単位)

□ InputStreamReader → 文字単位で読める

□ BufferedReader → 行 (=文字列) 単位で読める

```
//--- Sample4.java ---

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Sample4 {
  public static void main(String[] args) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    try {
      while(true) {
        System.out.print("N? ");
        String line = br.readLine();
        if(line == null) break;
        int num = (new Integer(line)).intValue();
        System.out.println("add 1: " + (num+1));
      }
    } catch(Exception ex) {
      System.out.println("!" + ex);
    }
  }
}
```

- import はパッケージ中のクラスをクラス名のみで使用できるように取り込む (標準では java.lang パッケージの中だけが取り込まれた状態。必要なら import java.io.*; のようにまとめて取り込める)。
- System.in は InputStream を保持。これはバイト単位の読み込み。
- InputStreamReader は InputStream をもとにして、文字単位の読み込みをサポート
- 「new クラス名 (...);」はクラスのインスタンス (オブジェクト) を生成
- BufferedReader は行単位の読み込みをサポート
- Integer は整数用の包囲クラス。そのメソッド intValue() は Integer オブジェクトをもとにそれと同じ値を持つ int 値を返す。

□ バイトと文字を Stream と Reader/Writer で分けているのは明確。

□ Stream も Reader/Writer も「あるものを元に機能を追加した新しいものを作る」構造は分かりやすい (効率という点では不利?)

□ try-catch を使わないと readLine() の IOException を捕まえてないよ、というエラーになる。捕まえるべきエラーをチェックするのはよい (繁雑だと思う人も?)

## 3.13 フルクラス名と import

□ 自分が「Test」というクラス (関数でもよい) を作ったら、他人も同じ名前のを作っていて困ったことは?

- そのような問題に対してどのような解決策があるか?

□ Java ではクラスは基本的に「長い名前」(フルクラス名) を持つ。java.lang.String、java.io.InputStreamReader など。

□ import しなくても、フルクラス名を書けばよい。import を指定するとそれを短く書くことができるというだけ。

□ パッケージの命名規則→ java、javax で始まるのは標準。各社や個人が公開する場合はドメイン名を逆に書く。たとえば

```
jp.co.toshiba.ssel.mathlib.newMath
```

などとなる。衝突の危険がないという点はメリット。

□ パッケージはそれぞれ独立だが、「.」でつなげた名前がつけられるので分類整理はやりやすい

## 3.14 ライブラリ API

□ プログラミングの生産性を高める方法の 1 つ→再利用 (書かずに済ます)

- 従来の再利用→ライブラリサブルーチン群→いろいろ不自由
- Java →クラス単位 (さらにその集まりであるパッケージ単位) の再利用

□ JDK 1.0.x : 8 パッケージ → JDK 1.1.x : 22 → JDK 1.2 : 58 → JDK 1.4.1 → 135 くらい

□ 特に重要なもの:

- java.lang.* → Java 言語の仕様の一部となるもの (System、Integer 等)
- java.io.* → 各 種 入 出 力 (Stream、Reader/Writer 等)
- java.net.* → ネットワーク関係
- java.awt.* → AWT(abstract windowing toolkit): 画面操作
- java.text.* → テキスト処理、フォーマット
- java.applet.* → アプレット関係

## 3.15 例 5: String オブジェクト

□ 簡単な例題: 文字列の左右反転

```
//--- Sample5.java ---

import java.io.*;

public class Sample5 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    try {
      while(true) {
        System.out.print("Str> ");
        String str = br.readLine();
        if(str == null || str.equals("")) break;
        String res = "";
        for(int i = str.length()-1; i >= 0; --i) {
          res += str.charAt(i);
        }
        System.out.println(res);
      }
    } catch(Exception ex) {
      System.out.println("!" + ex);
    }
  }
}
```

- 文字列リテラルは「唯一」オブジェクトのリテラル
- 「==」は「同じオブジェクトかどうか」判断する演算子。「equals()」はオブジェクトの値 (中身) が同じかどうかを判断するメソッド。

□ 「3 == 1 + 2」(Yes/No?)

□ 「"abcde" == "abcd" + "e"」(Yes/No?)

□ (演習問題) 文字列を 1 行読み込み、それに対して次のような出力を生成する Java プログラムを書け。

- (a) 入力文字列中の小文字をすべて大文字にする。
- (b) 入力文字列中の母音 (a, e, i, u, o) を「*」に変換する。
- (c) 次のような三角形を表示

```
Str> abcd
abcd
 bcd
  cd
   d
```

- (d) 次のような巡回表示

```
Str> abcd
bcda
cdab
dabc
abcd
```

## 3.16 オブジェクトを定義する

□ クラスを定義→オブジェクト種別を定義→新しいデータ種別

□ たとえば「値の対」クラスを定義するとする。

```
//--- Pair.java ---
class Pair {
  double x, y;
  public Pair(double x1, double y1) {
    x = x1; y = y1;
  }
  public Pair(String s) {
    try {
      int i = s.indexOf(' ');
      x=(new Double(s.substring(0, i))).doubleValue();
      y=(new Double(s.substring(i+1))).doubleValue();
    } catch(Exception ex) {
      throw new NumberFormatException(s);
    }
  }
  public double getX() { return x; }
  public double getY() { return y; }
  public String toString() {
    return "(" + x + "," + y + ")";
  }
  public Pair add(Pair p1) {
    return new Pair(x+p1.x, y+p1.y);
  }
  public Pair sub(Pair p1) {
    return new Pair(x-p1.x, y-p1.y);
  }
  public Pair mul(Pair p1) {
    return new Pair(x*p1.x, y*p1.y);
  }
}
```

□ インスタンス変数→このクラスのオブジェクト (インスタンス) がそれぞれ保持する固有のデータ。

- ここでは対なので 2 つの値 x、y を保持する

□ コンストラクタ→クラス名と同名のメソッド、インスタンスを初期化するという役割を担う

- コンストラクタを書いておけば、初期化ルーチンを呼び損なうということはあり得ない

□ コンストラクタでの初期化失敗→例外を返して通知する

- 例外を返すには「throw 例外オブジェクト」という文を使う

□ インスタンスメソッド→その中ではインスタンス変数が参照できる

- どのインスタンスか、はどのインスタンスに対してこのメソッドを呼び出したかによって決まる→カレントインスタンス
- カレントインスタンスは「this」という名前で参照できる→メソッド呼び出し時に「影の引数」として渡されていると考えるとよい

□ アクセサメソッド→内部データにアクセスするためのメソッド

- 外部からインスタンス変数を直接参照するようにも指定できるが、それは避けた方がよい→カプセル化の考え方

□ toString() というメソッド→「文字列にする」時に自動的に使われる

□ 自クラスの他のインスタンスを受け取ると→そのインスタンス変数などはカレントインスタンスと同様にアクセスできる

- カプセル化が破れているような気がする?

## 3.17 例 6: 定義したオブジェクトの利用

□ 先の Pair オブジェクトを用いた「電卓」

```
//--- Sample6.java ---

import java.io.*;

public class Sample6 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    Pair p = new Pair(0.0, 0.0);
    try {
      while(true) {
        System.out.print("> ");
        String str = br.readLine();
        if(str == null || str.equals("")) break;
        char c = str.charAt(0);
        String s = str.substring(1);
        if(c == '=') {
          p = new Pair(s);
        } else if(c == '+') {
          p = p.add(new Pair(s));
        } else if(c == '-') {
          p = p.sub(new Pair(s));
        } else if(c == '*') {
          p = p.mul(new Pair(s));
        } else {
          System.out.println("?" + str);
        }
        System.out.println("=" + p);
      }
    } catch(Exception ex) {
      System.out.println("!" + ex);
    }
  }
}
```

- メインクラス内で他のクラスメソッド (=単なる関数/サブルーチン) を作って利用している

□ マニュアル

- 「=値」→その値をセット
- 「+値」/「-値」/「*値」→加算、減算、乗算
- ただリターンだけ→終る

□ 実行例:

```
% javac Sample6.java
% java Sample6
> =1 3
=(1.0,3.0)
> +5 7
=(6.0,10.0)
> *0.5 0.5
=(3.0,5.0)
>
%
```

- 文字列表現にするところでは自動的に定義した toString() が使われている
- このように「ペア」というデータ種別に対して 1 つのクラスを定義することで、それに関わるデータや操作をひとまとめにできる

□ このような考え方→抽象データ型

## 3.18 より複雑なオブジェクト定義

□ 「文字列の集合」型を作る→内実は配列

- 配列は「new 型名 [要素数]」という特別な new でオブジェクト生成

```
//--- StrSet.java ---

class StrSet {
  String[] arr;
  public StrSet() { arr = new String[0]; }
  public StrSet(String[] a) { arr = a; }
  public StrSet(String s1) {
    int p = 0, n = 0; s1 = s1 + " ";
    for(int i = 0; i < s1.length(); ++i)
      if(s1.charAt(i) == ' ') ++n;
    arr = new String[n];
    for(int i = 0, j = 0; i < s1.length(); ++i) {
      if(s1.charAt(i) == ' ') {
```

```
        arr[j++] = s1.substring(p, i); p = i+1;
      }
    }
  }
  public boolean contains(String s) {
    for(int i = 0; i < arr.length; ++i) {
      if(arr[i].equals(s)) return true;
    }
    return false;
  }
  public StrSet addOne(String s) {
    if(contains(s)) return this;
    String[] a = new String[arr.length+1];
    System.arraycopy(arr, 0, a, 0, arr.length);
    a[a.length-1] = s;
    return new StrSet(a);
  }
  public StrSet add(StrSet set) {
    for(int i = 0; i < arr.length; ++i)
      set = set.addOne(arr[i]);
    return set;
  }
  public StrSet sub(StrSet set) {
    StrSet s = new StrSet();
    for(int i = 0; i < arr.length; ++i)
      if(!set.contains(arr[i])) s = s.addOne(arr[i]);
    return s;
  }
  public StrSet mul(StrSet set) {
    StrSet s = new StrSet();
    for(int i = 0; i < arr.length; ++i) {
      if(set.contains(arr[i])) s = s.addOne(arr[i]);
    }
    return s;
  }
  public String toString() {
    if(arr.length <= 0) return "{}";
    String str = "{" + arr[0];
    for(int i = 1; i < arr.length; ++i)
       str += (" "+arr[i]);
    return str + "}";
  }
}
```

□ このクラスの設計…

- 複数のコンストラクタ→引数で区別可能 (オーバロード)。
- contains(s) →文字列 s が集合に含まれるかどうか返す。
- addOne(s) →文字列 s を集合に (既に入っていなければ) 追加。
- add(set) → 2 つの集合の和集合
- sub(set) → 2 つの集合の差集合
- mul(set) → 2 つの集合の積集合
- toString() →文字列表現

□ この実現はかなり naive(効率が悪い)。しかし簡単は簡単。

## 3.19 例 7: 文字列集合電卓

□ 先の StrSet を利用して文字列集合電卓にしてみた

- 実はさっきのペア電卓とほとんど変わらない (なぜそうできるのか?)

```
//--- Sample7.java ---
import java.io.*;

public class Sample7 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    StrSet p = new StrSet();
    try {
      while(true) {
        System.out.print("> ");
        String str = br.readLine();
        if(str == null || str.equals("")) break;
        char c = str.charAt(0);
        String s = str.substring(1);
        if(c == '=') {
          p = new StrSet(s);
        } else if(c == '+') {
          p = p.add(new StrSet(s));
        } else if(c == '-') {
          p = p.sub(new StrSet(s));
        } else if(c == '*') {
          p = p.mul(new StrSet(s));
        } else {
          System.out.println("?" + str);
        }
        System.out.println("=" + p);
      }
    } catch(Exception ex) {
      System.out.println("!" + ex);
    }
  }
}
```

## 3.20 書き換え可能/書き換え不能

□ 書き換え不能 (immutable) →オブジェクトの状態が変化しないこと

- Pair オブジェクトも StrSet オブジェクトも内部的には変数を持っていて値が変化することは可能だが、外からメソッド経由で使っているぶんには値を変化させることはできない。
- 書き換え不能だと副作用の心配がない
- その代わりちょっとでも変更したいときは新しいオブジェクトを作ることになる

□ 書き換え可能 (mutable) →オブジェクトの状態が変化すること

- 例えばRectで「moveTo(x,y)で位置移動」StrSetで「addOne(s)で直接集合に文字列sを追加」などとすると書き換え可能になる
- 配列オブジェクトなどが書き換え可能なものの典型例
- CLUなど「書き換えられない配列」を持つ言語もあった
- 書き換え可能だと変更する時の効率はよい
- その代わり思わぬ副作用が起きないか注意する必要がある

□ 自分がクラスを作る時はどちらにするか十分検討して考える

### 3.21 この節のまとめ

□ Javaは C++などの反省点に立って設計されたオブジェクト指向言語

- しかしまだまだ理想的とは言えないところもある

□ この節ではオブジェクト指向機能についてはあまり触れていない。これについては次回＃2で詳しく取り上げる

□ この節では前節で取り上げなかった機能を中心に述べた

□ 特に重要な考え方→抽象データ型

## 4 さまざまなプログラミング言語/言語の歴史

### 4.1 プログラミング言語と実現技術

□ 言語によって固有の実現技術

□ とくに「通常の手続き型以外」

- ただし他の場面で応用可能
- あるいは言語Aで言語Bが得意とする処理を行う場合なども

□ まずは手続き型との対比から

### 4.2 低水準言語と高水準言語

□ 低水準言語→アセンブリ言語、機械語

- 言い替えれば、ハードウェアに依存した言語
- 移植性がない→現在ではめったに使われない

□ 高水準言語→ハードウェアに依存しない

- 最初の高水準言語→FORTRAN→「数式が楽に書ける」ことが目的
- 同じ高水準でも「より高水準」への変遷
- 高水準言語なら移植性はあるのか? →並列、その他…
- コンパイラの移植の手間、チューニングの手間…
- APIの互換性(cf. POSIX)

### 4.3 手続き型言語

□ 手続き型言語→命令型言語ともいう。変数、代入文、1文ずつの順次実行モデル

□ (一般的な)計算機ハードウェアの抽象化と思ってよい

```
main() {
  int i = 0;
  while(i < 10000) i = i + 1;
}
```

- 変数→「主記憶上の領域」の抽象化
- 式→演算命令の抽象化
- 条件→比較命令や条件分岐命令の抽象化
- 制御構造→分岐命令の抽象化

□ 手続き型→ハードウェアとの対応性→性能的には有利(だった?)→現在でも主流

### 4.4 Lispと記号処理

□ 記号→「互いに区別のつくもの」

□ 記号やその集まりに基づくプログラミング→記号処理

- 代表→Lisp、Prolog、(ML、Smalltalk-80、…)

### 4.5 記号の実現はどうするのがよい?

□ 伝統的な手続き型では…

- 文字列として表す→「等しいかどうか」の判定が遅い
- 数値に符号化して表す→扱いにくい(読みにくい)

□ Lispでは…

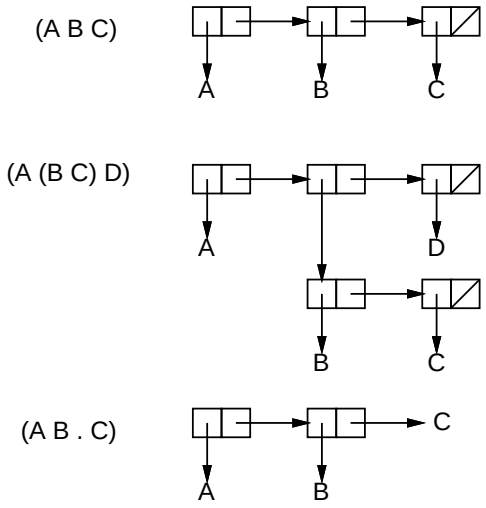
- 「文字列を格納したアドレス」で表す
- プログラムを読み込むとき、同じ名前の記号は同じアドレスで表す
- プログラムの実行中はポインタの比較で「等しいかどうか」がわかる

□ 手続き型言語でも「記号」という機能はあってもよいはず

## 4.6 リスト

□ 一般的には「…のならび」を表すことば

□ Lisp のリスト→葉にしかデータを置かない 2 分木




- きわめて汎用性が高い
- 記憶領域は多く必要→かっては CDR コーディングなどの最適化が行われたりした→最近はあまり問題でない、素直な実現が多い
- ごみ集めは必須 (最近はさまざまな改良)

□ リスト (のセル) とそうでないもの (アトム) の区別が必要

## 4.7 記憶域のタグづけ

□ 「これはポインタ」「これは数値」「これは記号」といった区別がつけられないと困る (タグづけ)

□ 素直な方法→そのためのビットを用意する→使える空間が減る、のろい

□ リスト、セルなどをアドレス範囲で区別→範囲検査は遅い

□ 下位ビットによるタグ

- リストセルなどは 8 バイト境界→下 3 ビットは通常 0 → 0 以外の値をタグに使う
- 整数の加減算などはタグがついたまま実行してもよい (補正すればよい)
- 3 ビット 8 種類では不足のことも→補助タグを使って拡張

□ いずれの方法でも整数値の範囲は少なくなる→即値の整数と参照先に格納する整数を併用する方法も

## 4.8 文字列処理

□ 初期の言語 (Fortran、Algol) には文字列はなかった

- しかし文字列が扱えることは重要

□ 文字列のさまざまな表現

- COBOL、Pascal →固定長文字列 (空白を詰める)
- PL/I →可変長文字列 (先頭に長さ)
- C →可変長文字列 (?)(末尾に 0)

□ いずれでも格納する最大長ぶんの領域を割り当てる

## 4.9 動的文字列

□ ヒープから文字列を割り当てる (通常 GC を前提とする)

- 文字列をリストで実現する方法も使われたことがある (連結、部分列に有利)
- 文字列本体とディスクリプタを分離する方法 (連結、部分列に有利)
- 現在はメモリも多量にあるので素直な実現が多い

## 4.10 日本語文字列

□ 旧来→ 1 文字は 8 ビット→多バイト表現→ EUC が多い

- EUC だと 2 言語 (英語+日本語) まで。

□ ANSI C の国際化→ wchar 型→ 16 ビットの文字 (固定長)

□ 現在の流れ→ Unicode (Java、Perl、tcl/tk、…)

- Unicode であっても別途言語情報をペアで持たないと困る…→しかし実際には一見国際化実は局地化なコードが多い?

## 4.11 文字列処理用言語

□ 古典→ SNOBOL 属 (SNOBOL 3、SNOBOL 4) → Icon

□ 現在→スクリプト言語で扱う→ tcl、awk、Perl

□ 今後→オブジェクト指向スクリプト言語→ Python、Ruby、…

### 4.12 パターンマッチング

□ 文字列をパターンと照合→多くの用途

□ 以前→文字列処理言語に組み込み

□ 現在→汎用言語+パターン処理ライブラリでも OK

- まずパターンをデータ構造にコンパイル→左から右へスキャン

□ 高速なマッチングアルゴリズムも研究されている

```
aaaaaaaaaaaaaaaaaaabcd ~ aaaabcd
```

の場合、naive な方法 (2 次マッチング) だと

```
aaaaaaaaaaaaaaaaaaab
aaaab
 aaaab
  aaaab
   ...
```

現在では「右から」マッチさせ、失敗したら「いくつずらしてよいか」を計算しておく方法

```
aaaaaaaaaaaaaaaaaaabcd
aaaabcd
   aaaabcd
      aaaabcd
         ...
```

### 4.13 関数型言語

□ 関数型言語→副作用を持たない関数の呼び出しに基づく

- 基本操作: 変数の*束縛*(代入ではない)
- 代表的なもの→ ML、Haskell、Miranda、…

□ なぜ「副作用を持たない」か?

- 最適化において「副作用は敵」←一度計算した式を再利用したくても、変数の値が変わっている*可能性*があると再度計算し直し
- 並列実行も副作用がなければ容易。副作用があると並行制御が面倒
- 変数の値の変化を追跡する必要/手間がない (変数は一度値が入ったらそれ以上変化しない) ←慣れないと不自由ではある

□ では入出力はどうするのか?

- 「getchar()」を 2 回実行したら値は当然違う!

□ 「副作用がある」というのは、呼んで戻って来た時状態が変化している、ということ→戻って来なければよい!

- つまり「getchar(channel, function)」という形で呼び、getchar は channel から 1 文字読んでそれを関数に渡す。渡された関数の中でさらに次の文字を読むために getchar を呼ぶ→ファイルの終わりまでそうやって呼び続ける。

□ この渡す関数は「入力ができたら私の仕事の『続き』はこれね」という意味を持っている。これを継続 (continuation) と呼ぶ。

- このように継続を渡して呼び続けるスタイル→ CPS (continuation passing style) という。
- CPS を使うと素直に実現できるシステムもいろいろある→ただし C などでは無限に呼び続けるとスタックオーバフローするので何らかの手当が必要

### 4.14 論理型言語

□ 元祖は Prolog。さまざまなものが派生している。

- 基本操作: 節のマッチングと変数の単一化 (unification)
- 単一化は片方が定数のとき代入に近い (しかしどちら側からでもよいので代入より強力
- バックトラックにより単一化は解除されることも←Prolog 属でもバックトラックを持たない言語もある…特に並列ものの場合

### 4.15 この節のまとめ

□ 高水準言語は最初はハードウェアの抽象化だった

□ その後さまざまな計算モデルに基づく言語が生まれている

□ 問題の記述に適したモデルを言語がサポートしていれば記述が楽

□ つまり問題とハードウェアのギャップを、問題と言語、言語とハードウェアの 2 つのギャップで置き換える→しかも後者のギャップは専門家が 1 回実装すれば済む

□ 結局、計算機言語はプログラマにプログラムを作りやすくするためにある (あたりまえの結論)。

## 5 手続き型言語の進化

□ 結局、現在でもメジャーな (プログラマ人口が多い、ソフトウェア製品の開発に使われることが多い) 言語は手続き型言語の末裔。

- ただし、手続き型言語の系列もかなり変化してきている
- その系譜を眺めてみよう

## 5.1 構造化プログラミング

□ 初期の手続き型言語はGOTO文化(COBOL、FORTRAN、…)

- スパゲティプログラムになりがち

□ 構造化プログラミング(1960年代末〜) → GOTOを使わず、もっと整った制御構造(if-then、while、repeat-until)を使おう

□ 段階的詳細化(トップダウン開発)

□ いずれも、教条的にやってもうまく行かないという結論

□ 実現上の技術は、特にない(コンパイラとしては当たり前)

## 5.2 手続きと引数機構

□ アセンブリ言語開発での手続き→広域変数に依存することが多かった

□ 構造化設計(モジュール間の存関係を減らす)→広域変数より引数

□ 引数渡し機構…値渡し、名前渡し(Algol)→参照渡し、copy/restore(Fortran)→オブジェクト渡し(オブジェクト指向)

- (参照渡しの)引数として渡した変数をそのまま作業変数としてさまざまな計算にも利用するのはよくない

```
call subr(x)
...
x = 1.0
y = .... x ....
```

## 5.3 例外

□ 例外とは→通常の制御の流れと違う制御の移行(例外的状況/エラー)

- 通常の制御構造でまかなえないgotoの代替機能として…
- エラー処理の統一的な枠組みとして…

□ Lispなどで古くからある→最近はC++、Java、…

```
try {
  ...
  int i = Integer.parseInt(s).intValue();
  ...
} catch(NumberFormatException e) { ... }
```

□ 利点:

- 同種のエラーを1箇所でまとめて処理できる
- 異種のエラーをそれぞれ別の場所で処理できる
- コードの主要部分がエラー処理でごちゃごちゃしない
- 手続きに無理矢理「エラーコード」を返させなくてすむ
- 受け止められていない例外をコンパイラがチェック

□ 例外機構のバリエーション

- 手続き内/手続き間
- 例外の伝播…そのまま/汎用の例外に変換/必要なら変換
- 例外からの復帰/後ろへ抜けるだけ

□ 例外機構はどうやって実装する?

- 自明な選択肢: 手続きの戻りコードにして戻り側で検査→わかりやすいが通常の呼び/戻りがのろくなる→避けたい
- 一般的な選択肢: 例外スコープの表を作っておき、例外が起きた時に自前で戻り処理をやりながら戻り先を検索する→例外が起きた時の速度は遅くなるが、通常の実行は遅くならない→例外はそう頻繁におきないはず→よく使われる手法

## 5.4 型

□ 型とは?

- 値の集合→古典的な定義。例: 整数型→{1, 2, 3, ...}
- 一群の操作を提供する値の集合→抽象データ型/オブジェクト指向に合う。例: 整数型→加算、減算、…

□ プログラミング言語における型…

□ ごく初期の型

- 機械語、アセンブリ言語(原始的な場合)には型がない
- (旧)Fortran→演算命令の区別のための型(整数、実数)→文字型がない、配列も「単なる繰り返し」であり型ではない
- COBOL→レコード定義も「データの集まり」であり型ではない
- Algol68→ref T型(C言語でいう「*」)→ポンインタを型にした

□ Pascal→これらさまざまなものを型として整理し、「ユーザ定義の型」を大幅に導入した。

- 基本型 (int、real、char、boolean、列挙型)
- 部分範囲型 (「1..100」)
- 配列型、ポインタ型、レコード型、集合型
- 範囲型、列挙型、集合型は現在の言語ではすたれている
- 集合型の実装→ビットベクター。いくつかの主要な Pascal 処理系では 32 ビットに限定されていたので「set of char」が使えなかった (今でも日本語だと無理がある…)

□ C → Pascal と同時代のライバル

- 範囲、列挙、集合に加え、論理型がなく、配列とポインタが混同されている (わざと) →アドホックな設計
- 「int sub(int *a)」と「int sub(int a[])」の a の型は同じ? (Y/N)
- 「int a[100]」と「int *a」は意味は違うが…a の型は同じ? (Y/N)

□ 結局、型は何のためにある?

- 効率のよいコードを出すため…△ (ごく特別な場合) → cf. ML の型推論
- 値の種別に応じた操作を人間がいちいち指定しなくて済む…◎
- 値の種別の勘違いを検出してくれる (強い型の場合) …◎
- プログラムの構造を設計/記述する手段…◎

□ 強い型による保護

- 型検査で許された操作しかできない→メモリ保護として利用可。この方式の代表格: Java
- 型検査を抜ける (escape) 方法があると役に立たない。代表: C や C++のキャスト
- マシン語を直接書けばどのような操作でも可能なのでやはり問題
- Java では仮想マシン語の検証器によってこのような操作をはねる

## 5.5 モジュール

□ 手続きが単位では「カタマリ」が小さすぎる

□ 複数の手続き群が協調して 1 つのデータ構造を維持する →よくある

□ これらの機能を言語機構として取り入れ→モジュール→「データ構造と、一群の手続きとをひとまとめにして、外からアクセスできるもの/できないものを定める機能」

- Modula、ConcurrentPascal、Modula-2、…
- C などでも「ファイル」はモジュール的に使える (static 変数/関数はそのファイル内だけで参照可能)

□ モジュールの機能は「隠すこと」。なぜ「不自由さ」が重要なのか?

- 外部から「見えてしまう」と「使ってしまう」。「使ってしまう」と「依存関係が増えてしまう」(それで後から変更しようとするとはまる)。
- たとえ変数が見えても「型定義が見えない」ようにできる言語も。そうすると「操作する方法がない」ので「その型の変数は作れるが、そのモジュールの手続きを呼ぶ以外のことは何もできない」状態になる →情報隠蔽 (encapsulation) →抽象データ型につながる考え。
- モジュールの実装→「隠すこと」だけだから特になし (コンパイラで正しくないアクセスをはねればよい)

## 5.6 抽象データ型

□ 1 つのモジュールに 1 つのデータ構造、ではなく、あるモジュールが定義するデータ構造を「いくつも作りたい」場合も→つまり、新しい「値の種類」が増える→型→「抽象データ型」

- 抽象データ型をサポートする言語→ 1970 年代に抽象データ型が脚光を浴びた時にいくつか作られた。CLU、Alphard、…、Ada、…
- 現在ではオブジェクト指向言語にその機能が引き継がれていると言える (由来はちょっと違っているが…)

□ CLU による「整数の集合」型

```
intset = cluster is create, insert, is_in
  node = record[i:int, l,r:intset]
  rep = variant[n:null, r:node]
  create = proc() returns(cvt)
    return(rep$make_n(nil)) end create
  insert = proc(r:cvt, i:int)
    tagcase r
    tag n : rep$change_r(node$[i:i,
      l:intset$create(), r:intset$create()])
    tag r(n:node) :
      if n.i = i then     % do nothing
      elseif n.i < i then intset$insert(n.l, i)
      else                intset$insert(n.r, i)
      end
    end
  end insert
  ...
end intset
```

□ 抽象データ型の実装方法…

- すべてヒープ上のデータ構造へのポインタとするなら難しくはない (CLU)

- スタック上のデータなどにしたい場合は、データ構造の大きさを知らなければならない→抽象データ型の定義において「外から見える」部分と「見えない」部分に分けて記述し、「見えない」部分は「見えないふり」をする→ Ada、C++→しかし「見えない」部分の変更でも再コンパイルが必要だったりしていまひとつ使いやすくない。現在のオブジェクト指向言語ではすべてポインタとするのが主流。

### 5.7 型パラメタ

□ たとえば上のようなコードで、格納される内容は int でなくてもよいはず→「型」を「パラメタ」として扱えるような抽象データ型 (型生成子) が使えるとよい。

- たとえば通常の言語の「配列」なども要素型をパラメタとする型生成子だと言える。
- ものによっては、パラメタ型が「どのような操作を提供しているか」を指定する必要。

□ 型パラメタを持つ言語→ CLU、Ada、C++、Java(JDK1.5〜)

□ CLU による「任意の型 T の集合」型

```
intset = cluster[t:type]
      where t has lt:proctype(t,t) returns(bool)
  is create, insert, is_in
 node = record[e:t, l,r:intset]
 rep = variant[n:null, r:node]
 create = proc() returns(cvt)
  return(rep$make_n(nil)) end create
 insert = proc(r:cvt, e:t)
  tagcase r
  tag n : rep$change_r(node$[e:e,
    l:intset$create(), r:intset$create()])
  tag r(n:node) :
    if n.e = e then     % do nothing
    elseif n.e < e then intset[t]$insert(n.l, e)
    else                intset[t]$insert(n.r, e)
    end
  end
 end insert
 ...
end intset
```

□ 型パラメタの実現方法

- 自明な方法: コンパイル時にマクロ展開する→コードが大きくなる、厳密なチェックが抜けることがある、再帰的な型指定はできない (無限に展開)、その反面最適化はやりやすい。
- パラメタ型の情報を実行時に保持する方法→利点と欠点は上記のほぼ反対。

### 5.8 オブジェクト指向

□ オブジェクト指向については、第 2 回でまとめて取り上げますが、ちょっとだけ。

□ オブジェクト指向の定義→「プログラムが扱う対象を『もの』として取り扱う見かた/考えかた」

- だから、「intset[real]$insert(s, 3.14)」では抽象データ型だが、「s!insert(3.14)」と書ければそれはオブジェクト指向、という考え方も成り立つ (CLU にそういう拡張を施したものがあった)。

### 5.9 この節のまとめ

□ アセンブラでも高水準言語でもプログラマの生産能力 (lines/日) は同じ→高水準であるほど効率はよい

□ 考えなければならないこと (依存関係、関連性) が多いほど生産効率は悪くなる

□ そのため、次のことが重要

- 大きな「かたまり」で考える
- 「かたまりの中」と「かたまりどうし」を分けて考える
- 見なくてよいことは「見えない」ようにすることが大切

□ オブジェクト指向まで、言語はそのような方向で進化してきた

- オブジェクト指向ではまた新たな側面が…(次回をお楽しみに)

## 6 第 1 回レポート課題

□ 下記 (1)〜(4) のうちから 1 つ以上を選び実験を行い、結果をレポートせよ。やったことの説明や書いたコードだけでなく、計測方法を説明し、きちんと考察まで書くこと。期限はリーダから指定。

□ (1) Pair オブジェクトや StrSet オブジェクトを参考に、自分でも何か簡単な抽象データ型のクラスを作り、その「電卓」を動かしてみよ。たとえば「複素数クラス」「分数クラス」などが考えられる。これらの自前クラスや例題クラスの「演算性能」を計測し、Java 言語の基本型 (int や double) との性能比較を行え。また、その性能比はどのように説明されるか考察せよ。

- ヒント: 時間計測には System クラスのクラスメソッド System.currentTimeMillis() を利用するとよい。

```
long t1 = System.currentTimeMillis();
計測したい処理
long t2 = System.currentTimeMillis();
long dt = t2 - t1; ←所要時間 (msec)
```

□ (2) 例題の StrSet オブジェクトにおいて、メソッド addOne() の実行時間は集合に入っている要素数に比例するはずである。このことを実験により確かめよ。またこの性質を改善することを (できれば複数の方法で) 試みよ。どのような方法でどれくらい改善できたか検討すること。

- ヒント: 多数の文字列を生成するにはたとえば次のようにする

  ```
  String s = "a" + (++count);
  ```

  これで「"a1"」「"a2"」…という文字列ができる。名前の長さを一定に保った方がよいなら count を 100000 とかから始める。
- ヒント: 改善する簡単な方法の 1 つは、配列 arr を「きっちり」用意する代わりに多めに用意し、add() を「要素を直接追加」にする。ただしこの場合、StrSet は書き換え可能になってしまう (つまりクラスの仕様が変化する)
- ヒント: クラスの仕様が変化しないで済むようにも少し工夫すればできる。考えてみて欲しい。

□ (3) StrSet のメソッド add() や mul() の所要時間は、和や積を取る 2 つの集合の大きさをそれぞれ M、N とすると、M、N を用いたどのような式に比例するか。まず予想し、続いて実験により確かめよ。次にこの時間を改善する方法を検討せよ。

- 注記: 課題 (1) で addOne() の時間を改善する方法を実装した場合はその改良も含めた状態でやってよい。
- ヒント: 配列 arr 内で文字列が大小順に整列されるようにしておくと、和集合や積集合の計算が速くできる。
- ヒント: 別の方法として、配列に順に詰めるのではなくハッシュ表などを使ってもよい。API ドキュメントで調べて Java 標準ライブラリに含まれるハッシュ表のクラスを活用してもよい。

□ (4) 2 つの StrSet どうしが同じ集合であるかどうかを判断するメソッド equals() を実装せよ。また、その性能について前 2 問と同様に所要時間のモデルを考え、計測により確認し、さらに改良を試みよ。

- 注記: 前 2 問で行なった改良が含まれた状態でやってもよい。
- ヒント: 集合に含まれる文字列群全体のハッシュ値を計算し、集合が異なればハッシュ値も異なるようにしておくと、異なる場合の判断はすごく速くできる。

# 7 おまけ: 久野の所属について

□ 筑波大学東京地区→文京区大塚 (丸の内線茗荷谷駅徒歩 2 分、JR 東京駅から 20 分)

□ ビジネス科学研究科→夜間専門の大学院 (社会人のみ対象)

- 授業はすべて夜間 (18:20〜) または土曜日

□ 経営システム科学専攻 (修士課程) →標準年限 2 年で修士 (経営学、経営システム科学)

- ビジネスマネジメントコース→修士論文を書くコース
- プロジェクトマネジメントコース→教官主催のプロジェクトに参加

□ 企業科学専攻 (博士課程)

- システムズマネジメントコース→標準年限 3 年で博士 (経営学、経営システム科学)

□ 教官は共通→経営学、数理科学、コンピュータサイエンス→計算機がお仕事の人にもぜひ入学して頂きたい

□ Webpage: http://www.gssm.otsuka.tsukuba.ac.jp/

□ 大学院説明会: 7 月 3 日 (上記 webpage でも広報)