TSSI 基盤技術研修コース
— プログラミング言語 — ＃1

久野　靖*

1999.5.7

はじめに

□ 本講座の目的→計算機言語/プログラミング言語に関する概念を整理し、体系的に整理すること

□ 予想される利益:

- 言語の特徴を活かしたソフトウェア/システム設計ができるように
- 開発したシステムの挙動、特徴、弱点を予め予測可能に
- プログラマの能力をよりよく活かす/その限界を超えない

□ 構成:

- 第 1 回→機械語からオブジェクト指向まで+言語のさまざまな実現技術
- 第 2 回→オブジェクト指向
- 第 3 回→並行/並列プログラミング (予定)

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?

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

□ 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?

- 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

*筑波大学大学院経営システム科学専攻

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。
- □ 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　計算機アーキテクチャとプログラミング

2.1　なぜハードウェアアーキテクチャを気にするか?

- □ 速度：　プログラムにおいて重要なものの 1 つ
  - すべてに優先するというわけではないが…
  - ハードウェアの特性を活用することが大切
- □ 命令セットアーキテクチャ：　ソフトとハードのインタフェース
  - コンパイラがどのようなコードを出すかも重要
  - 速度を問題にする場合、命令セットだけでは済まないが…

2.2　命令セットアーキテクチャの変遷

- □ 最初期→アキュムレータ+1 アドレス、3 アドレス等
- □ インデクスレジスタ、さまざまなレジスタの追加
- □ 汎用レジスタ
- □ マイクロコード制御→豊富な命令群→CISC(complex instruction set computers)
  - CISC の特徴→命令の充実、豊富なアドレシングモード、可変長命令、遅い
- □ CISC　批判 →　RISC(reduced instruction set computers)
  - RISC の特徴→ロードストアアーキテクチャ、単一アドレシングモード、単一命令長、ハードワイア制御、高速
- □ CISC の逆襲→現在はどちらも拮抗
  - 高速化技術→ RISC 型コードへの変換実行、簡潔で速い命令は速く

2.3　機械語によるプログラミング

- □ 初期の計算機システム→機械語での開発が当たり前
- □ 計算機の主記憶上のプログラム=機械語
  - 単なるビットの列 (0 と 1 の羅列)
  - 作るのも読むのもすごく大変
- □ 初期の計算機ではすべて機械語によるプログラミングだった
- □ 計算機に押しボタンやランプがついていてそれを操作
  - 主記憶にプログラムを書き込む
  - 主記憶にデータを書き込んだり、データの状況を調べる
  - 実行の様子を調べる

2.4　アセンブリ言語

- □ 機械語では「1 ビットでも間違うと破滅」「読みづらい」
- □ →命令や番地などに名前をつけて書き表すように (アセンブリ言語)
- □ →これを機械語に変換するプログラムを「アセンブラ」
- □ 本質的には機械語と同じ (しかし読み書きしやすさは重要だが)

```
.globl _main
        .type    _main,@function
_main:
        pushl %ebp
        movl %esp,%ebp
        subl $4,%esp
        call ___main
        movl $0,-4(%ebp)
L2:
        cmpl $9999,-4(%ebp)
        jle L4
        jmp L3
        .align 2,0x90
L4:
        incl -4(%ebp)
        jmp L2
        .align 2,0x90
L3:
L1:
        leave
        ret
```

2.5　機械語プログラミングの「素材」

□ CPU

- レジスタ→一時的な作業場所
- ALU →演算命令、条件分岐命令
- PC →プログラムの実行位置、分岐命令

□ 主記憶→変数、配列、制御情報 (戻り番地等)

□ 現在では人間が組む機械語よりコンパイラの方が効率はよい

- 例:　レジスタ割り付け
- 例:　命令スケジューリング
- 例:　ループスケジューリング

2.6　レジスタ割り付け

□ レジスタ割り付け:　変数やテンポラリ (コンパイラが先生する作業変数) をレジスタに対応させること

- すべての変数の生きている範囲を求める
- 生きている変数どうしが同じレジスタに載らないように割り付ける
- 載り切らないものは影響の少ないものからメモリに置く (スピル)

2.7　命令スケジューリング

□ 現在の CPU はほぼすべてパイプライン実行→命令の実行がオーバラップ

□ 命令の順序を実行時間ができるだけ短くなるように並べる

- 各命令の実行タイミング/どの時点でどの資源を使用するかの情報を持つ
- クリティカルパスの命令から先に、資源が重複しないように並べて行く

□ ハードウェアで命令の並べ替えを行うもの (out of order 実行) もあるが→長いループの内部などでは並べ切れない

2.8　ループスケジューリング

□ ループのピッチができるだけ短くなるように命令を並べる

- そのためには、周回を通じたオーバラップを用いる
- ある変数の i 回目のインスタンスと i+1 回目のインスタンスが共存することも

2.9　分岐予測

□ 条件分岐命令はパイプラインにバブルができてしまい遅くなる要因

□ 分岐予測→行く先を予測し、そちらの命令列を仮に実行開始する (間違っていたらキャンセルする)

- ハードウェアによる分岐予測 (履歴つき)
- コンパイラによる分岐予測 (分岐予測ビット、ない場合はたとえば分岐先のアドレスの大小で表す)

□ ハードによっては予測でなく両側同時実行も (条件分岐の連続だと限界)

2.10　機械語/アセンブラによるシステム開発

□ 最近まで組み込みシステム等ではあった (今でも?)

□ どういうふうに設計?　(実装/低レベルの設計)

- 使用するデータ構造を決める (ここは C などと同様)
- コード、データの配置を決める

- サブルーチンリンケージを決める (標準がある場合も)
- その他、ハードウェアに関わる規約を決める (特に OS がない場合)
- 機能分解 (おおむねサブルーチン等) に従って開発

2.11 配列の実現

□ メモリ上に領域を取る→カンタン

□ 問題は添字によるアクセス。どうする?

- インデクスレジスタ (とは?)
- 番地を計算してレジスタ間接アクセス (とは?)
- 番地を計算してメモリ間接アクセス
- 番地を計算して…命令の番地部を書き換える (!!)

□ 命令のアドレシングモード次第

2.12 アドレシングモード

□ 古典的なアーキテクチャ: 番地のみ

- 少し凝ったもの→間接ビット、インデックスビット
- もっと凝ったもの→間接語にさらにこれらのビットがつけられる

□ CISC アーキテクチャ: 豊富なモード

- (直接, 間接)+レジスタ、レジスタ間接 (+自動加減算)、さらに間接
- 360 アーキテクチャ: さらにベースレジスタ (番地部の短さをカバー)

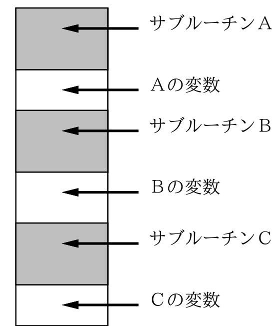
□ RISC アーキテクチャ: 1 種類だけに絞る

- (オフセット+レジスタ) の間接

□ いずれにせよ、「遠い」アドレスは遅い。

2.13 素朴なメモリモデル

□ メモリ上に命令列、変数を一緒に配置

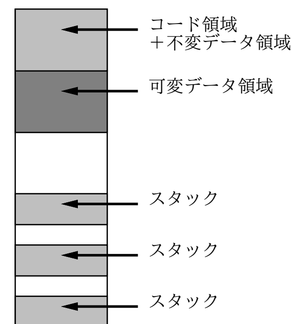


□ 現在では命令列と変数は分けて配置 (なぜ?)

- 命令は書き換えない (なぜ?)
- 命令は複数プロセスが共有 (なぜ?)

2.14 現在の標準的なメモリモデル

□ コード/不変データ、書き換えるデータ、スタックに分かれる



- コード/不変データ: 共有可能
- 書き換えるデータ: プロセス単位 (ヒープも含む)
- スタック: 自動的なメモリ割り当て、スレッド単位

2.15 サブルーチンリンケージ

□ 必要なこと: サブルーチンを呼んだ箇所に戻って来ること

□ 命令書き換え型のリンケージ

- 呼び出し命令はなくても可能 (戻り命令をジャンプ命令にしておき、その番地部に戻り番地を直接書き込む
- これではあまりに不便 (戻り番地も書き込む場所も予め計算) →呼び出し命令
- 「i 番地に戻り番地を書き込み、i+1 番地へ分岐」という仕様もあった。戻り命令は「サブルーチン先頭番地を参照する間接ジャンプ」

- しかし結局、命令部を書き換えるのはまずい

□ 命令を書き換えない→別のデータ構造に格納 (リエントラントにできる)

□ スタックに積むのが主流に

- 再帰呼び出しが行える
- CISC →自動的にスタックに積む命令仕様→引数渡しも処理
- RISC →単に戻り番地を特定レジスタに入れるだけ
- この方がどうにでもなるので便利。葉の手続きならメモリに格納しなくてもよい。

2.16 引数渡し

□ 引数モデル→値渡し、参照渡し、複写復元。どれでも似たようなもの (複写復元はコピーし戻すところが余分だが)。

□ 素朴には、特定番地に入れておく

- 広域変数だと→副作用などであぶない
- 手続きローカル→リエントラントでない。アドレシングが遠い。

□ スタックで受け渡す→標準的

- スタックポインタ (またはフレームポインタ) レジスタ起点のアドレシングで効率よくアクセスできる。
- CISC ではスタックの積み降ろし命令がある。

□ レジスタ渡し→高速→現在では、最初のいくつかはレジスタ渡しとするのが普通

- しかし、そのレジスタが必要になったとき入れる場所がいる→スタックの場所もこれまで通り用意
- 手続き間レジスタ割り付け→そのレジスタに置いたままにできるように割り付ける

□ しかし究極の高速化手法は→その場展開

- あらゆる最適化手法が使えて有利
- コードサイズが大きくなるという弱点→ヒューリスティックで展開を制御

2.17 レジスタリネーミング/レジスタウィンドウ

□ VLSI 集積度の向上→ CPU チップ上にはレジスタに使える領域が十分ある

- しかしアーキテクチャを変えられないのでレジスタ数は増やせない

□ レジスタリネーミング (名前替え): ハード側で自動的にレジスタをマッピングする

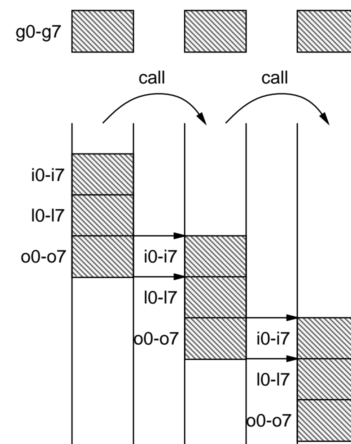
```
load r1,Mxxxxx
add r1,r2,r1
store r1,Myyyyy
load r1,Mzzzzz
add r1,r2,r1
store r1,Mttttt
```

□ 最初の r1 の使用と 2 回目の r1 の使用はオーバラップ可能→内部的には別のレジスタに割り当てる

- out of order 実行 (命令の並び順と発行順が一致しない)

□ レジスタウィンドウ→手続きの呼び/戻りごとにレジスタセットを切り替える

- 呼び/戻りに際してレジスタを退避回復しないでよい
- 引数/返値のため、オーバラップがある
- SPARC アーキテクチャで採用されている
- 切り替わらないレジスタ (グローバルレジスタ) × 7、ウィンドウレジスタ× 24、うち 8 が呼び側とオーバラップ (引数を受ける)、8 が呼ばれ側とオーバラップ (引数を渡す)、残り 8 が手続きローカル



□ もちろん、無限にあるわけではない→物理レジスタが足りなくなると割り込みが起こってスタックに退避/回復

- プロセス切り替えの際も退避/回復が必要

2.18　この節のまとめ

□ 計算機アーキテクチャには非常にさまざまな特性がある

□ プログラミング言語の実行→このアーキテクチャに適切にマッピングできれば高速に実行が可能

- 基本的にはコンパイラの仕事
- しかしコンパイラの仕事が何であるか知っておくと無駄な努力が避けられる

3　さまざまなプログラミング言語

3.1　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

3.2　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)

3.3　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

3.4　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、…)

3.5　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…

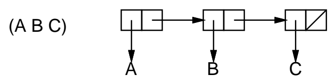
- 「文字列を格納したアドレス」で表す
- プログラムを読み込むとき、同じ名前の記号は同じアドレスで表す
- プログラムの実行中はポインタの比較で「等しいかどうか」がわかる

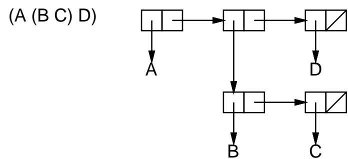
□ 手続き型言語でも「記号」という機能はあってもよいはず

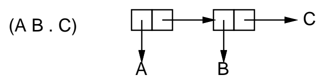
3.6　リスト

□ 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→かつては 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

3.7 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
- 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

3.8 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

3.9 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

3.10 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)

□ 今後→ 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…

3.11 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、…

3.12 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

```
aaaaaaaaaaaaaaaaaaaabcd ~ 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

```
aaaaaaaaaaaaaaaaaaaabcd
aaaabcd
   aaaabcd
      aaaabcd
         ...
```

3.13 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「副作用は的」←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

3.14 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

3.15 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)。

4 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

4.1 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)

4.2 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 ....
```

4.3 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

4.4 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

4.5 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)

4.6 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。

4.7 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++

□ 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。

4.8 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)。

4.9 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)

5 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 1.1.x 相当以上、JDK 1.2(Java 2) ももちろん OK。
- アプレットの表示も JDK 1.1.x 対応以上のブラウザで (MSIE なら 5.x、Netscape Navigator なら 4.5 以降が安全)。ただし JDK 付属の appletviewer でも OK。

5.1 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

5.2 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

□ 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、ポインタ演算がない
- オペレータオーバローディングがない
- オブジェクトはすべてヒープ上のオブジェクト

□ 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

□ 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 (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)

5.3 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 (単独の関数はない)(とりあえずクラス 1 個だけでやる)

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

- public : このメソッド (C でいう関数) はクラス外から呼べる
- static : このメソッドは (インスタンス=オブジェクトでなく) クラスに付属。つまり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

□ 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
```

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

□ すべてクラス、という設計はどうか? → 言語を簡潔に

□ オブジェクト指向ではインスタンスメソッドとクラスメソッド、インスタンス変数とクラス変数の区別がある

□ クラスメソッドはどのみち必要 → 単独メソッドの代替

□ static というキーワードは誤解を招きやすい

□ 定数 → クラス変数に final(変更不能) 指定をつける

□ すべてクラス→責任の所在が明確

□ 配列だけが特別な型 (型生成子) → C++のようにテンプレートをやると言語が複雑になるからという選択。やはりテンプレートは欲しいと考えている人は多いようだが、それなりの見識。

5.4 基本型とオブジェクト型

□ 基本型→整数、文字など「レジスタに載るような」値

□ オブジェクト型→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値

□ 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)

5.5 例題 2: 数を読み込んで 1 足す

□ 簡単な例題ですがクラスが多数

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

public class Sample2 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    try {
      System.out.print("N? "); System.out.flush();
      String line = br.readLine();
      int num = (new Integer(line)).intValue();
      System.out.println("> "+num+" + 1 = "+(num+1));
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}
```

- 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
- try ... catch(...) ... は例外処理の構文
- 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。
- 「文字列 + 何か」は「何か」のメソッド toString() を呼んで「何か」を文字列に変換し、それと左辺の文字列を連結する。

□ import についてはどうか?

- import しなくても、フルクラス名を書けばよい。あくまでもそれを短縮させてくれるのが import。明快でよい。
- 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
  ```

  などとなるわけ。衝突の危険がないという点はメリット。

□ 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?)

□ 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、というスタイル。C++では GC がないからやりにくかった。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。

□ 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現在はちょっと困る? toString() はあくまでもデバッグ用で、正式にはフォーマット用クラスを用いて言語依存に行う、ということになっている。

5.6 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

□ 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

5.7 例: java.lang.String

□ java.lang.String の API を見てみよう。

- クラス名と同名のメソッド→「コンストラクタ」(オブジェクトの初期設定を行うためのメソッド)
- 同じ名前のコンストラクタ/メソッドが多数ある→オーバーローディング(多重定義)
- 文字列をさまざまに操作する方法一式が提供されている→完全性
- == と equals() の違い→同一オブジェクトか、値が等しいか
- intern() → シンボルの実現と同様 → equals() を==で代替可能

□ 簡単な例題: 文字列の左右反転

```
import java.io.*;

public class Sample3 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    try {
      while(true) {
        System.out.print("Str> "); System.out.flush();
        String str = br.readLine();
        if(str.equals("")) break;
        String res = "";
        for(int i = str.length()-1; i >= 0; --i) {
          res += str.charAt(i);
        }
        System.out.println(res);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}
```

□ 無限ループと break の制御構造。true は boolean 型の値

□ 文字列は配列ではなく1つのオブジェクト。長さは任意

□ a = a + x と a += x は等価(数値に限らない)

- 本当は str[i] と書けると嬉しい? また文字列以外の「+」もあるといい?
- 言語によっては添字構文や「+」などの演算子をオーバーロード可能。Java では言語仕様を簡潔にするため採用しなかった。

□ (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
```

5.8 クラスをモジュールとして使う

□ クラスの先頭部分に記述する変数→クラス変数、インスタンス変数。

□ クラス変数→クラス1つにつき1つだけ存在。

□ クラス変数とクラスメソッド→「モジュールのようなもの」が作れる。

□ 例題: 文字列プール

□ 名前は StrPool。要件は次の通り

- void StrPool.add(String) → プールに文字列を登録
- String StrPool.intern(String) → intern する（なければ登録）
- boolean StrPool.isIn(String) → 登録されているか調べる

□ テストドライバ

```
import java.io.*;

public class Sample4 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    try {
      while(true) {
        System.out.print("Str> "); System.out.flush();
        String str = br.readLine();
        if(str.equals("")) break;
        System.out.println("isIn: "+StrPool.isIn(str));
        StrPool.add(str);
      }
    } catch(Exception e) { System.err.println("!!"+e); }
  }
}
```

□ StrPool クラス

```
--- StrPool.java ---

public class StrPool {
  static int size = 0;
  static String[] arr = new String[100];
  static String lookup(String s) {
    for(int i = 0; i < size; ++i) {
      if(arr[i].equals(s)) return arr[i];
    }
    return null;
  }
  public static boolean isIn(String s) {
    return lookup(s) != null;
  }
  public static void add(String s) {
    if(!isIn(s) && size < arr.length) arr[size++] = s;
  }
  public static String intern(String s) {
    add(s); return lookup(s);
  }
}
```

- クラス 1 つにつき 1 つのファイルに入れた方がよい（public なクラス、つまりパッケージ外から参照できるクラスはファイル名とクラス名を一致させなければならないので、1 つのファイルに最大 1 つしか入れられない）。
- 配列もオブジェクト。変数 length に要素数。
- クラスの冒頭部分にクラス変数の定義が書ける
- public でないメソッドはクラス内部（正確には同一パッケージ内、厳密にクラス内のみとするには private と指定）からのみ参照可能
- 同一クラス内のメソッドは名前だけで呼べる

□ データ構造（arr、size）はクラス内でのみ操作可能→安全性

□ 内部のデータ構造は外部インタフェースに影響せず変更可能

□ 外部からモジュールを使うとき必要な情報→インタフェースのみ

□（演習問題）配列が満杯になったとき自動拡張するよう改良せよ

5.9 クラスを抽象データ型として使う

□ クラスはそれ自体が型であり、そのクラスのインスタンス（オブジェクト）はいくつでも作れる→つまり抽象データ型の機能を持つ

□ インスタンスは「new クラス名 (...)」で作る

- すると対応するコンストラクタが呼び出されて初期設定が行える
- 1 つもコンストラクタを定義しないと、システムが用意したデフォルトのコンストラクタ（引数なしのコンストラクタ）が呼ばれる。

□ 1 個のインスタンスごとにインスタンス変数を一式ずつ持つ。

□ インスタンスメソッドを呼ぶとその中ではインスタンス変数が使える（またはインスタンスを表す擬変数 this が使える）。

5.10 例題: 文字列集合の抽象データ型

□ テストドライバ: 「集合電卓」

```
import java.io.*;

public class Sample5 {
  public static void main(String args[]) {
    BufferedReader br =
      new BufferedReader(
        new InputStreamReader(System.in));
    int top = 0;
    StrSet[] a = new StrSet[100];
```

```
    a[top] = new StrSet();
    while(true) {
      try {
        System.out.print("> "); System.out.flush();
        String str = br.readLine();
        if(str.equals("q")) {
          break;
        } else if(str.equals("&")) {
          if(top == 0) throw new Exception();
          a[top-1] = a[top-1].intersection(a[top]);
          System.out.println("{"+a[--top]+"}");
        } else if(str.equals("|")) {
          if(top == 0) throw new Exception();
          a[top-1] = a[top-1].union(a[top]);
          System.out.println("{"+a[--top]+"}");
        } else if(str.equals(";")) {
          System.out.println("{"+a[top]+"}");
          a[++top] = new StrSet();
        } else if(str.equals("")) {
          throw new Exception();
        } else {
          a[top] = a[top].add(str);
        }
      } catch(Exception e) { System.err.println("?"); }
    }
  }
}
```

- 「電卓」なので集合を作る、intersection、union の 3 操作を提供
- 1 行の処理中にエラーがあれば例外を投げて 1 箇所で処理
  - 例外処理で投げるものは Exception のインスタンス → new で作る
- 「文字列集合」はあるものとして考える→問題の分解、抽象データ型の利点
- 「文字列集合」の実装: 単純素朴版 (分かりやすさ優先)

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

public class StrSet {
  String[] arr;
  public StrSet() { arr = new String[0]; }
  public StrSet(String s1) { arr = new String[]{ s1 }; }
  public StrSet(String[] a) { arr = a; }
  public boolean contains(String s) {
    for(int i = 0; i < arr.length; ++i) {
      if(arr[i].equals(s)) return true;
    }
    return false;
  }
  public StrSet add(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 union(StrSet set) {
    for(int i = 0; i < arr.length; ++i)
      set = set.add(arr[i]);
    return set;
  }
  public StrSet intersection(StrSet set) {
    StrSet s = new StrSet();
    for(int i = 0; i < arr.length; ++i) {
      if(set.contains(arr[i])) s = s.add(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;
  }
}
```

- 「自分」は内部表現を操作するが「他の StrSet オブジェクト」はあくまで外部から操作するスタイルを採っている。
- 直接「set.arr」のように他の StrSet の内部を操作することも可能ではあるが、間違いを犯しやすくなる (効率は上げられるが)。
- StrSet は「書き換え不能」(immutable) な抽象データ型。たとえば add は 1 つ要素を追加した StrSet を返す。
  - これを「書き換え可能」にするとどうなる? 何がいい? 何が悪い?

6 第 1 回レポート課題

- 下記 (1)〜(3) 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。きちんと考察まで書くこと。期限は 5 月末日。
- (1) 自分の手元に用意した Java の処理系と他の言語 (好きに選んでよい) の処理系の性能比較を行え。まず仕事のどのような場面で比較が必要とされているかのシナリオを定め、その条件下ではどのような方法で比較するのが適切かを検討し、最終的に選択した方法で比較を行え。レポートには比較した処理系 (双方とも) とハード、シナリオ、検討内容、比較方法、および比較結果とその分析を記すこと。
- (2) 上の「文字列集合」抽象データ型の代替実装を作成せよ。上のコードとの性能比較を行え。(1) と同様の記述を行うこと。

☐ （3） Java ではオブジェクトの領域は自動ごみ集め（GC）によって回収される。自動ごみ集めにどれくらいのオーバヘッドが取られているか、できる範囲で実験を行い計測せよ。もちろん、オーバーヘッドは処理系内部のデータの使われ方によって影響される。どのように影響されるかもできる範囲で検討せよ。

- ヒント： java.lang.Runtime などに情報を取得するメソッドが多少ある。また多くの java 実行系では実行開始時にヒープの大きさを指定できる。