プログラミング言語処理系論 (2)
Design and Implementation of Programming Language Processors

佐藤周行
（情報基盤センター/電気系専攻融合情報学コース）

今日の予定

- プログラミング言語処理系の論点
  - 何が研究対象になるのか？
  - どのような方法論が使われるのか？

- Web Page

http://www-sato.cc.u-tokyo.ac.jp/SATO.Hiroyuki/PLDI2010/

プログラミング言語の論点

- （プログラミング言語処理系ではなく…）

- 何をどう表現するか？
- Semanticsは？
- 実行モデルは？

- 処理系をどう作るか？

- 規格

論点（1） 何をどう表現するか

- プログラム＝アルゴリズム＋データ構造
（クラシックな見方）
アルゴリズムをどう表現するかが一つの見方
- アルゴリズムとデータ構造 (岩波講座 ソフトウェア科学)

アルゴリズムの表現

- 高いレベルの表現 – Algol系言語
  - 条件分岐と繰り返し
  - If, for, while,
  - 関数呼び出し、再帰
- 低いレベルの表現 -- Fortran
  - フローチャート
  - Compare, goto

アルゴリズムの表現

- (追加)並列性の表現は現代的な言語での課題の一つ
  - 並列DO構文
  - 同期
  - データの配置とアクセス制御
  - データ並列

プログラム例(Fortran)

Fortranで並列性・ループ構文を表現する例

```
REAL :: A(10, 10), B(10, 10) = 1.0
. . .
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
   A(I, J) = REAL (I + J - 2)
   B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

```
FORALL (I = 1:N, J = 1:N)
   A(:, I, :, J) = 1.0 / REAL(I + J - 1)
END FORALL
```

並列性に関しての脱線

- 並列性に関しては、いろいろな方針が錯綜しています。
  - 並列アルゴリズム記述を許す
    - なんでもあり
  - 並列アルゴリズム記述を許さない
    - データを「並列」に配置することを許す
      - 逐次アルゴリズムから並列性を抽出する＝データ並列
      - SPMD
    - データの「並列」配置を許さない
      - 逐次アルゴリズムから並列性を抽出する
      - DO Loopの自動並列化

- 読めますか？(Coarray in Fortran 2008)

- ```
  real :: r[*]
  ! Scalar co-array
  real :: x(n)[*]
  ! Array co-array
  ! Co-arrays always have assumed
  ! co-size (equal to number of images)
  real :: t

  integer :: p
  t = r[p]
  x(:) = x(:)[p]
  x(:)[p] = r
  ```
  ```

プログラム例（ALGOL系）

```
int bsearch(const char *key, const char *base[ ], int bottom, int
top)
{ int middle;
  int tmp;

if (bottom == top)
    return (-1);

middle = (top + bottom) / 2;


if ((tmp = strcmp(key, base[middle])) == 0)
    return (middle);
else if (tmp > 0) {
        if(top == middle + 1)
            bsearch(key, base, middle, top + 1);
        else bsearch(key, base, middle + 1, top);
} else
    bsearch(key, base, bottom, middle);
 }
```

データ構造の表現

- 高いレベルの表現
  - グラフ、リストなどのデータ構造
  - 構造体など、ひとまとまりのデータを表現する
  - 再帰的なデータ構造の定義を許す

  - 抽象データ型
  - データを操作する関数群を「データ」に含める
  - それ以外の操作を禁止する

- 低いレベルの表現
  - マシン内部の構造の表現
  - 整数、実数、ポインタ

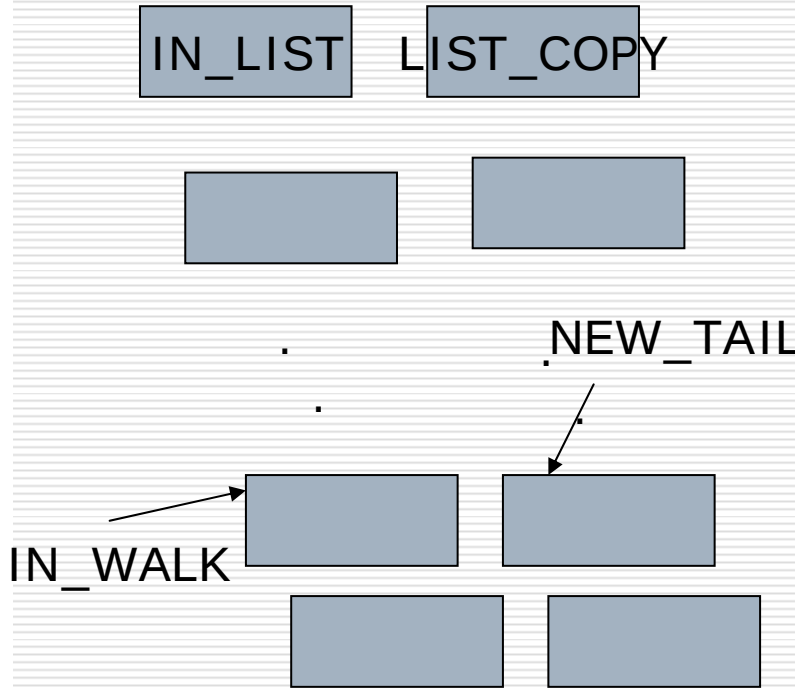
リストの表現

- datatype int list = Cons of int * int list | Nil;

```
typedef  struct int_list {
    int elem;
    struct int_list *next;
} int_list;
```

データ構造例(Fortran polymorphic)

```
TYPE :: LIST ! A list of anything
  TYPE(LIST), POINTER :: NEXT => NULL()
  CLASS(*), ALLOCATABLE :: ITEM
END TYPE LIST
...
TYPE(LIST), POINTER :: IN_LIST, LIST_COPY => NULL()
TYPE(LIST), POINTER :: IN_WALK, NEW_TAIL
! Copy IN_LIST to LIST_COPY
IF (ASSOCIATED(IN_LIST)) THEN
  IN_WALK => IN_LIST
  ALLOCATE(LIST_COPY)
  NEW_TAIL => LIST_COPY
  DO
    ALLOCATE(NEW_TAIL%ITEM, SOURCE=IN_WALK%ITEM)
    IN_WALK => IN_WALK%NEXT
    IF (.NOT. ASSOCIATED(IN_WALK)) EXIT
    ALLOCATE(NEW_TAIL%NEXT)
    NEW_TAIL => NEW_TAIL%NEXT
  END DO
END IF
```



注意

- ここまでFortranの例をそれなりに出してきました
- 実は、佐藤はFortran使いではありません
- Fortranは、数値計算系では今でも有力な言語の一つです（昔言われたような、言語規格上の優位さはすでにありませんが）
- 無視してよい言語ではありません

プログラミングにおける「概念」

- アルゴリズム＋データ構造を表現できれば、プログラミング言語としては一応合格ですが…

- どう考えながら「プログラミング」するか、というスタイルが問題になるようになって来ました
  - オブジェクト
  - ファンクション
  - アスペクト
  - …

プログラミングスタイルの問題

- オブジェクトを考えながらプログラミングする人なら、オブジェクトが自然に表現できる言語を使う
- ファンクションの組み合わせとしてプログラムを組む人なら、ファンクションを自然に表現できる言語を使う
- …
- こうして、プログラミング言語は「何を表現できるか」を競うようになりました

このような「古い」教科書を相手にしてはいけません

- 命令型プログラミング言語
- 関数型プログラミング言語
- 論理型プログラミング言語

- があって、云々

さて、

- プログラミングスタイルの研究は格好良いですが、
- げっぷが出るほどプログラムを書き、しかもそれが賞賛の対象になるようなものでないと、説得力を持たないのも事実です。

- がんばれ

データ構造の表現

- オブジェクトの表現を例にもう少し詳しく見る

- 一般的な構造体の定義との違いを注意しておこう
  - Public/private
  - Inheritance
  - Abstract

C++のクラスの例

```
class Human {
private: int age; // プライベート
         char name[20]; //
public: void SetHuman(int toshi, char
*namae);
void Show(); //
 };
```

抽象クラスの定義

- // 抽象クラス: 硬貨 C++
  class Coin{ public: virtual int value() = 0; };
  // 1円硬貨
  class Coin1: public Coin{ public: int value(){ return 1; } };
  // 10円硬貨
  class Coin10: public Coin{ public: int value(){ return 10; } };

  ...// 抽象クラス: 硬貨 Java
  class Coin{ public abstract int value(); }
  // 1円硬貨 class Coin1 extends Coin{ public int value(){ return 1; } }
  // 10円硬貨 class Coin10 extends Coin{ public int value(){ return 10; } } ...

オブジェクト間のインタラクション

```
class ctest{
    public static void main(String args[])
    {
        Television tv1 = new Television();

        tv1.setChannel(1);
    }
}
```

論点 (2) Semantics

- プログラミング言語はプログラムを書くための言語である
- プログラムは実行されてはじめて意味を持つ
- Semanticsとは、プログラムの持つ(計算上の)意味のことをいう

SyntaxとSemantics

- コンパイラのフェーズとしてある``Semantic Analysis''のSemanticとは別の概念

- （Digression)

CFG、特にBNFは、文法を記述するには非常に便利であるが、CFGだけでプログラミング言語が定義できるわけではない

Semanticsの記述

- 代表的なものは2つ
  - 計算機械をひとつ仮定した上で、プログラムがその機械上でどのように振舞うかを記述する
    - Operational Semantics
  - ある数学的な計算モデルを作った上で、プログラムがその計算モデル上のどのような対象として解釈されるかを記述する
    - Denotational Semantics

計算機械

- 計算はどのように進むか？
  - 変数と、そこに割り当てられている「値」のペアの集合を考える
  - それぞれの集合を「状態」ということにする
  - 「環境」ということもある。環境は名前と値のペアにもっぱら使われる
  - プログラムの各命令は、状態を入力として状態を出力する関数であると思うことにする

例

- プログラム

```
int i,y;
i = 0; y = 0;
while (i < 100) {
   y = y + i;
   i = I + 2;
}
```

- 状態の変化

(i,y) = (0,0)
↓
(0,0)
↓
(2,0)
↓
(2,2)
↓
(2,4)

最初の問題点

- 関数コールはどうか？
  - 仮引数と実引数の名前の対応をどうとるか？
  - 変数のバインディングをどう表現するか？
    - Call by value, Call by Reference, Call by Name, …
  - ローカルな変数とグローバルな変数の違い
- 関数コールごとのフレームはどうか？
  - 特に再帰に関して典型的に出てくる
  - 関数コールごとに異なる実行状態（フレーム）を作り出すことをどう表現するか？

- 計算機械をもとにしたSemanticsにおいて、関数コールでは「変数のバインド」と「環境の生成・巻き戻し」を処理しなければならない

- 言語の定義において、この部分はもっとも重要な部分の一つである

関数コールのSemanticsの例

- Javaの規格
  - 15.12.4 Runtime Evaluation of Method Invocation
  - 15.12.4.1 Compute Target Reference (If Necessary)
  - 15.12.4.2 Evaluate Arguments
  - 15.12.4.3 Check Accessibility of Type and Method
  - 15.12.4.4 Locate Method to Invoke
  - 15.12.4.5 Create Frame, Synchronize, Transfer Control
  - 15.12.4.6 Example: Target Reference and Static Methods
  - 15.12.4.7 Example: Evaluation Order
  - 15.12.4.8 Example: Overriding
  - 15.12.4.9 Example: Method Invocation using super
- Fortranの規格229ページ
  - 実引数、仮引数および引数結合(association)

Semanticsのご利益

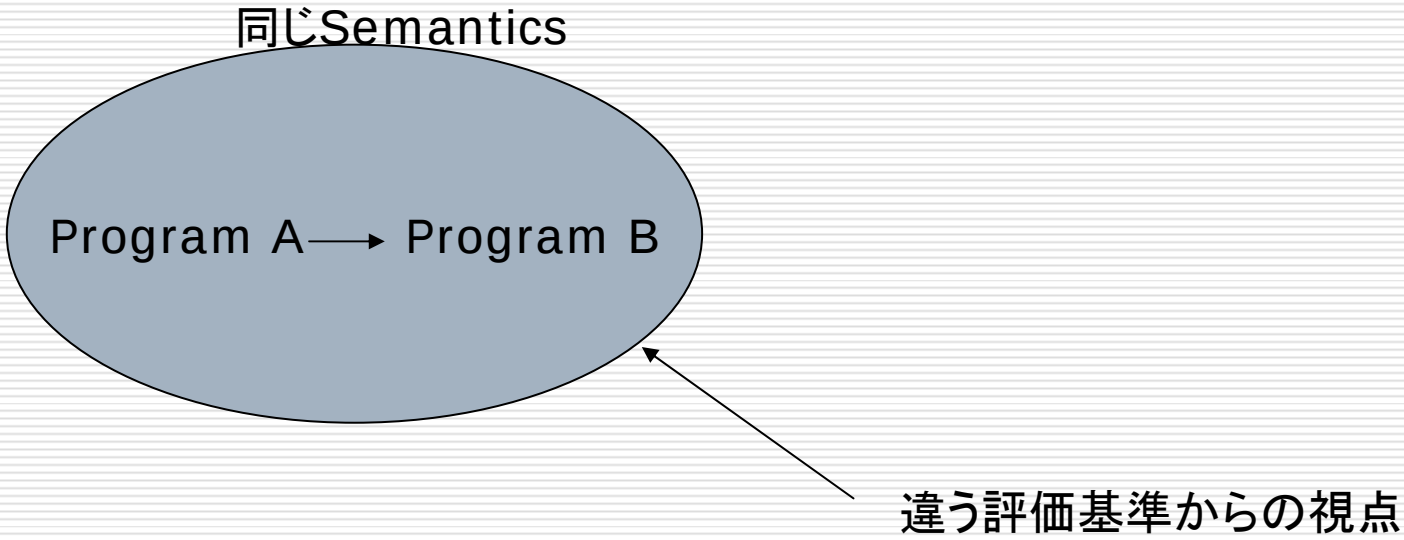
- Semanticsを　関数（状態→状態）　と決めると以下の議論が進む
  - 二つのプログラムが同じ関数をSemanticsにもつならば、
  - どちらを採用してもプログラムのSemanticsは変わらない
  - 「最適化」とはSemanticsを変化させないという制約下でプログラムを変換させること

コンパイラ最適化の約束

- Semanticsは変更しません
  - 特に変数の読み書きの順番

- 変更しないことを守った上で、任意のコードに置き換えます。変換後はスピードが速くなったり、コードサイズが小さくなっているかもしれません
- 性能が改善されたことは、実行マシン上で検証できるかもしれませんが、一般にはSemanticsがよってたつ実行モデルの上では検証できません

- Semanticsを変えないでプログラムを変換する



最適化の例

```
int x, y, z, i;

    x = 0; y = 0; z = 1;
    x = x + z;
    y = y * 2;
    z = x;

    for (i=0;i<10;i++) {
       y = y + i;
    }
```

```
int x, y, z,i;

    x = 1;
    z = 1;


    y = 45;
```

ところで

- 「値」とはなんだろうか？
  - 整数
  - 実数
  - 複素数（ここらあたりまでは直感的に明らか）
  - 関数
  - オブジェクト
  - ポインタ（ここらへんにくると、計算機械（状態→状態）が、何を対象に動くのかを明らかにしなければならなくなる）

数学的な計算モデル

- コンピュータの発明以前から計算をする数学モデルはたくさん考えられてきた
  - 帰納関数論
  - λ計算
- Denotational Semanticsは、プログラムをそれら数学モデルに直接変換しようとする試みである

例

```
int fact(int i)
{
   return i*fact(i-1);
}
```

$$(\mu\,\mathrm{fact}.\ \lambda\,\mathrm{i}.\ \mathrm{i}\times\mathrm{fact}(\mathrm{i}\text{-}1))\ 3$$

$$\rightarrow(\lambda\,\mathrm{i}.\mathrm{i}\times(\mu\,\mathrm{fact}.\lambda\,\mathrm{j}.\mathrm{j}\times\mathrm{fact}(\mathrm{j}\text{-}1))(\mathrm{i}\text{-}1))3$$

$$\rightarrow 3\times(\mu\,\mathrm{fact}\ldots)(2)$$

$$\rightarrow 3\times(2\times(\mu\,\mathrm{fact}\ldots)(1))$$

$$\rightarrow\ldots$$

問題点

- 数学的な議論のしやすさから、Operational Semanticsを駆逐するものとしての期待が高まっていた
- さっきの図式でいえば、「最適化」の候補が広がることを意味している
- 実際、「関数型プログラミング」の分野では大きな成果をあげた
  - より高次の「最適化」の表現-プログラム変換理論
- 難しい点はOperational Semanticsとほとんど変わらない
  - 関数コールにおける変数バインディング
  - 「環境」の生成とまき戻し

論点 (3) 実行モデル

- 今後、プログラムの実行というときには「計算機械」を意識的、無意識的に考える
- Semanticsを記述するために、この「計算機械」の上でどういう振る舞いをするか、ということを記述することが良くある。

- 実行モデルとは、この「計算機械」のことをいうことにする。

最近の傾向

- Informalな実行モデル
  - 高度な基本概念を最初に持ってくる
    - オブジェクト、アスペクト、エージェント、…
  - 「計算の単位」がインタラクトしながら状態が変化することが「計算」である
    - オブジェクト指向、アスペクト指向、エージェント指向、…
  - プログラミング言語は、この「計算の単位」を表現しなければならない

最近の傾向(Cont’d)

- 数学的にいろいろな性質を証明しやすくするための枠組ではない
  - エージェントどうしがインタラクトすることでいったい何が明らかになるのか？
  - 人間界と同じように何かが進行することはわかるが、ではそれは、数学的に何かが証明できるか？
- よって、少し違和感があるが、そんなことを気にしない人が増えていることも事実である

最近の傾向(Cont‘d)

- そのかわり、
  - プログラム解析の様々な方法論が発展してきた
    - (データ | コントロール)フロー解析、スライシング、型理論、types and effect systems
  - 必要な性質は、それら方法論を使って証明しよう

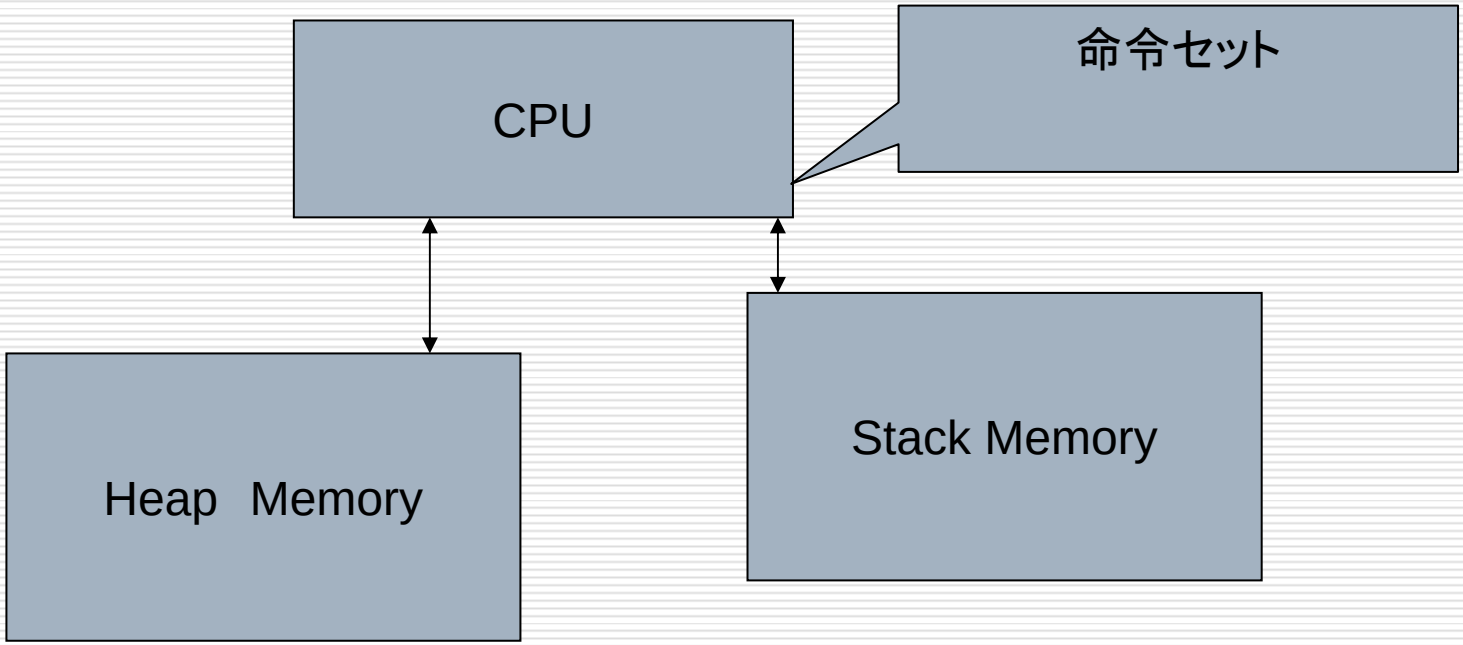
- この動きは、「プログラミング言語理論」と「プログラム解析理論」が分離し始めたことを表している

実行モデルの記述

- 仮想的なマシンの記述
  - 「状態」の記述
    - メモリの状態
    - スレッドの状態
  - 「状態遷移」の記述
    - 主だった命令がマシンの状態をどう変化させるかを記述

実行モデルの例

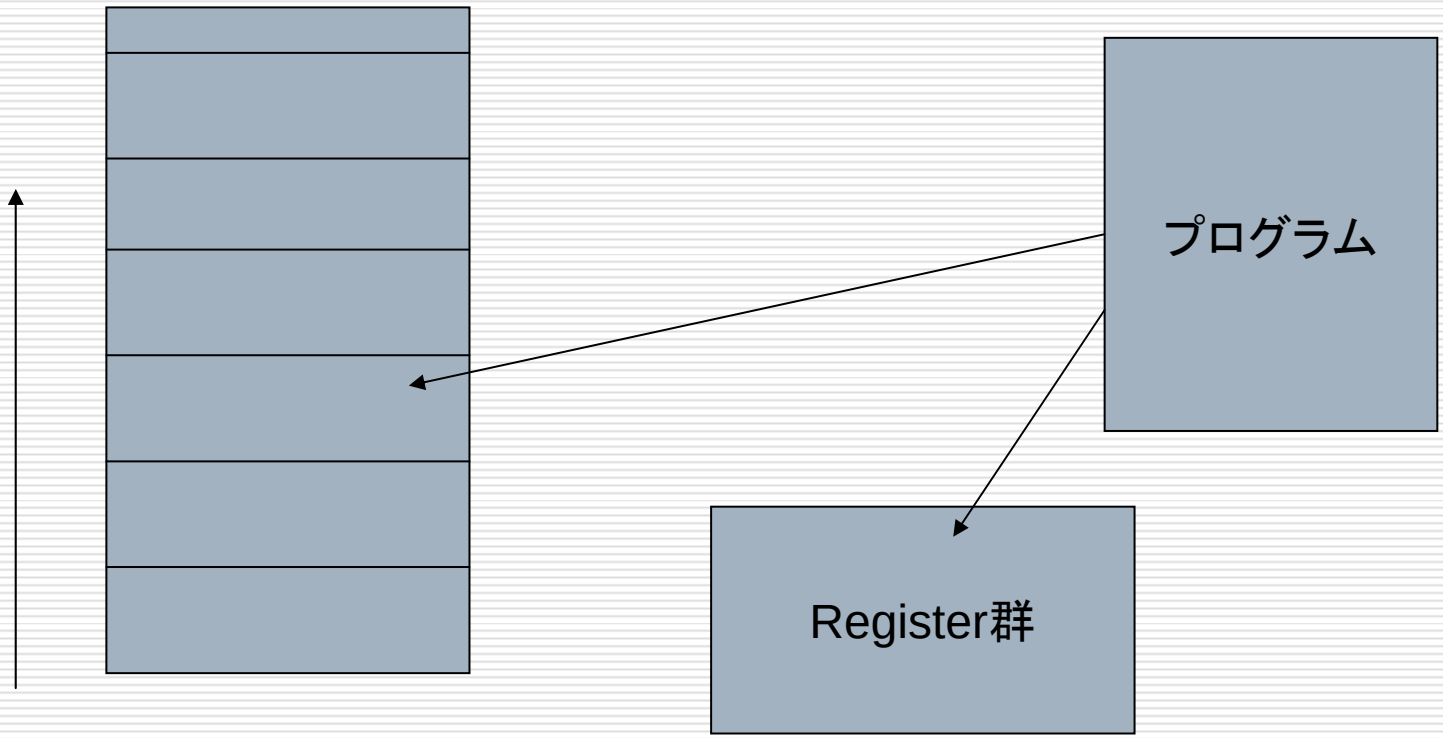
- 「普通」の計算機を反映した実行モデル



Java VMの定義

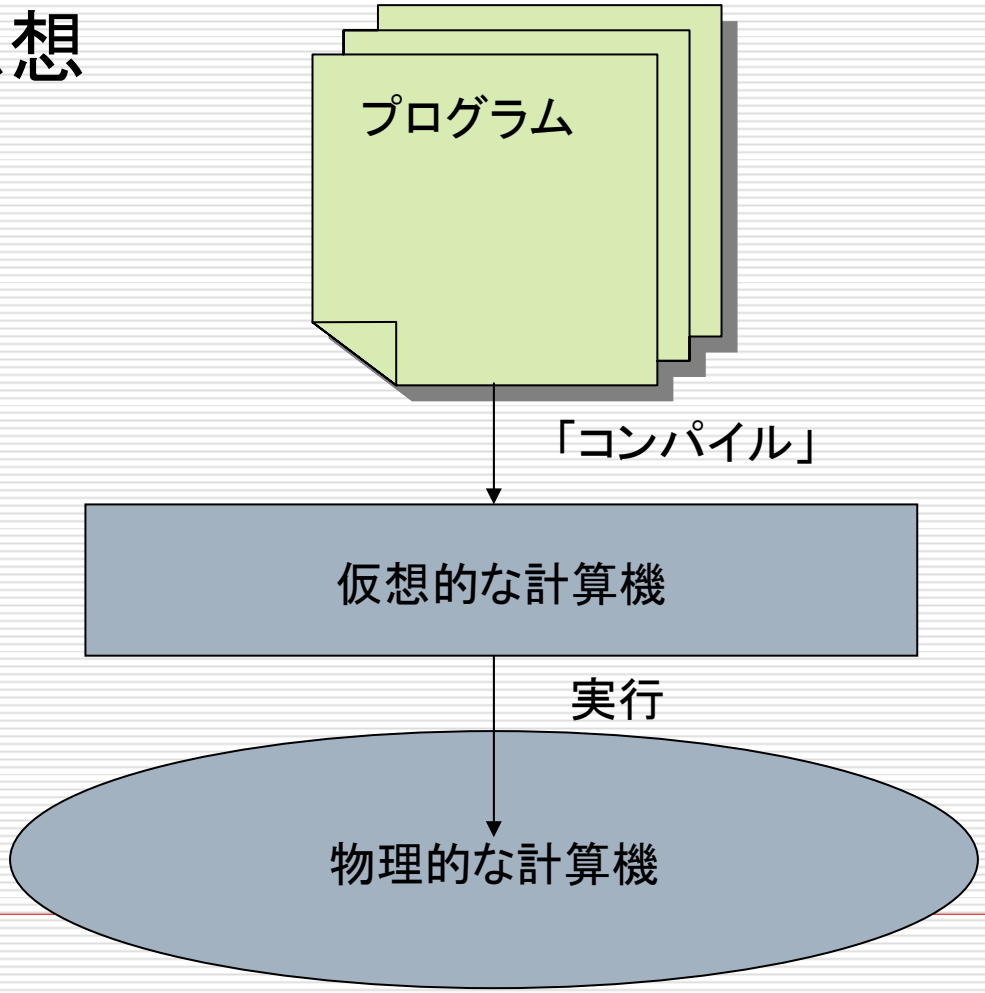
- 典型例としてJava VMがある
  - 3 The Structure of the Java Virtual Machine
    - 3.1 The class File Format
    - 3.2 Data Types
    - 3.3 Primitive Types and Values
      - 3.3.1 Integral Types and Values
      - 3.3.2 Floating-Point Types, Value Sets, and Values
      - 3.3.3 The returnAddress Type and Values
      - 3.3.4 The boolean Type
    - 3.4 Reference Types and Values
    - 3.5 Runtime Data Areas
      - 3.5.1 The pc Register
      - 3.5.2 Java Virtual Machine Stacks
      - 3.5.3 Heap
      - 3.5.4 Method Area
      - 3.5.5 Runtime Constant Pool
      - 3.5.6 Native Method Stacks
    - 3.6 Frames
      - 3.6.1 Local Variables
      - 3.6.2 Operand Stacks
      - 3.6.3 Dynamic Linking
      - 3.6.4 Normal Method Invocation Completion
      - 3.6.5 Abrupt Method Invocation Completion
      - 3.6.6 Additional Information
    - 3.7 Representation of Objects
    - 3.8 Floating-Point Arithmetic
      - 3.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754
      - 3.8.2 Floating-Point Modes
      - 3.8.3 Value Set Conversion
    - 3.9 Specially Named Initialization Methods
    - 3.10 Exceptions
    - 3.11 Instruction Set Summary
      - 3.11.1 Types and the Java Virtual Machine
      - 3.11.2 Load and Store Instructions
      - 3.11.3 Arithmetic Instructions
      - 3.11.4 Type Conversion Instructions
      - 3.11.5 Object Creation and Manipulation
      - 3.11.6 Operand Stack Management Instructions
      - 3.11.7 Control Transfer Instructions
      - 3.11.8 Method Invocation and Return Instructions
      - 3.11.9 Throwing Exceptions
      - 3.11.10 Implementing finally
      - 3.11.11 Synchronization
    - 3.12 Class Libraries
    - 3.13 Public Design, Private Implementation

Java VM Illustrated



VMの思想

- VMの思想



VMの思想(cont'd)

- VMは、実行マシンの変化を吸収するための共通のインタフェースとして出現した
  - UCSD Pascal
- 「複雑な実行モデル」を持つプログラミング言語の処理系としても利用されるようになった
  - Smalltalk
  - 多くの関数型言語
  - 多くのスクリプト系言語

Java VM

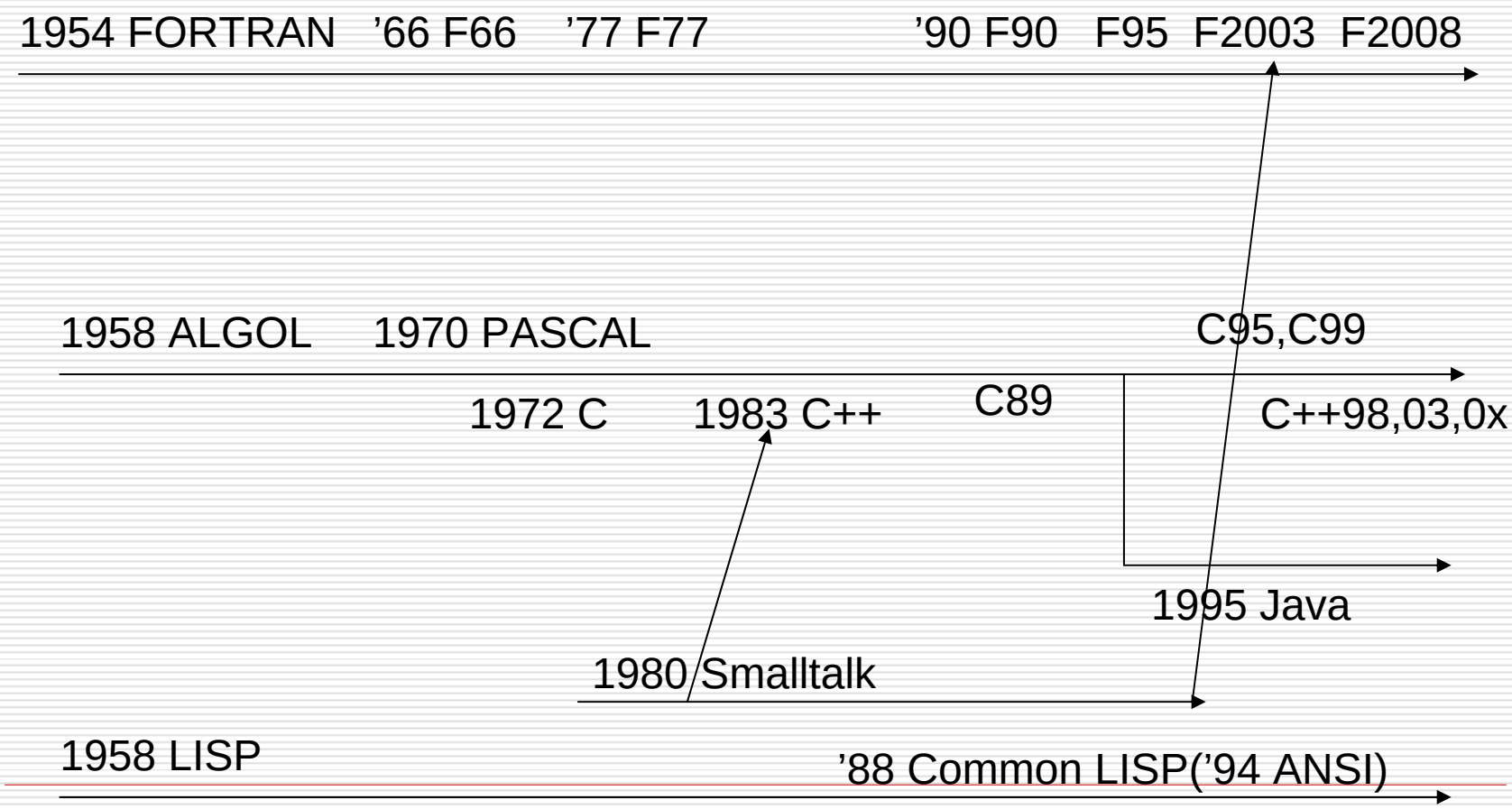
- Java VMは、スタックマシンとして登場した
- 目的は、安全な形でコードの流通性を高めることであった（特にアプレット）

- JavaのSemanticsはJava VMの上で定義することができる
- JITが出現した

プログラミング言語の歴史

- Fortran, Algol系
- 規格、言語仕様の重要性の認識
- スクリプト言語
- JAVAの登場
- VMの登場

Programming Languages at a Glance



Fortranの登場

- 高級言語の概念の提示
- BNFを使った定義
- 数式（変数を使った数学の式）を直接表現することに主眼があった
- Assembly言語（GOTOが主たる実行制御方式）＋数式

- 言語規格を制定するときに大きな議論の場を提供した

ALGOL系言語の登場

- アルゴリズムの記述に主眼
- 制御構造その他で、「ALGOL系言語」にくくられるタイプをごく初期から決定した

- 子孫がたくさんいる
  - Pascal, C, C++,…

現在のプログラミング言語の流れ

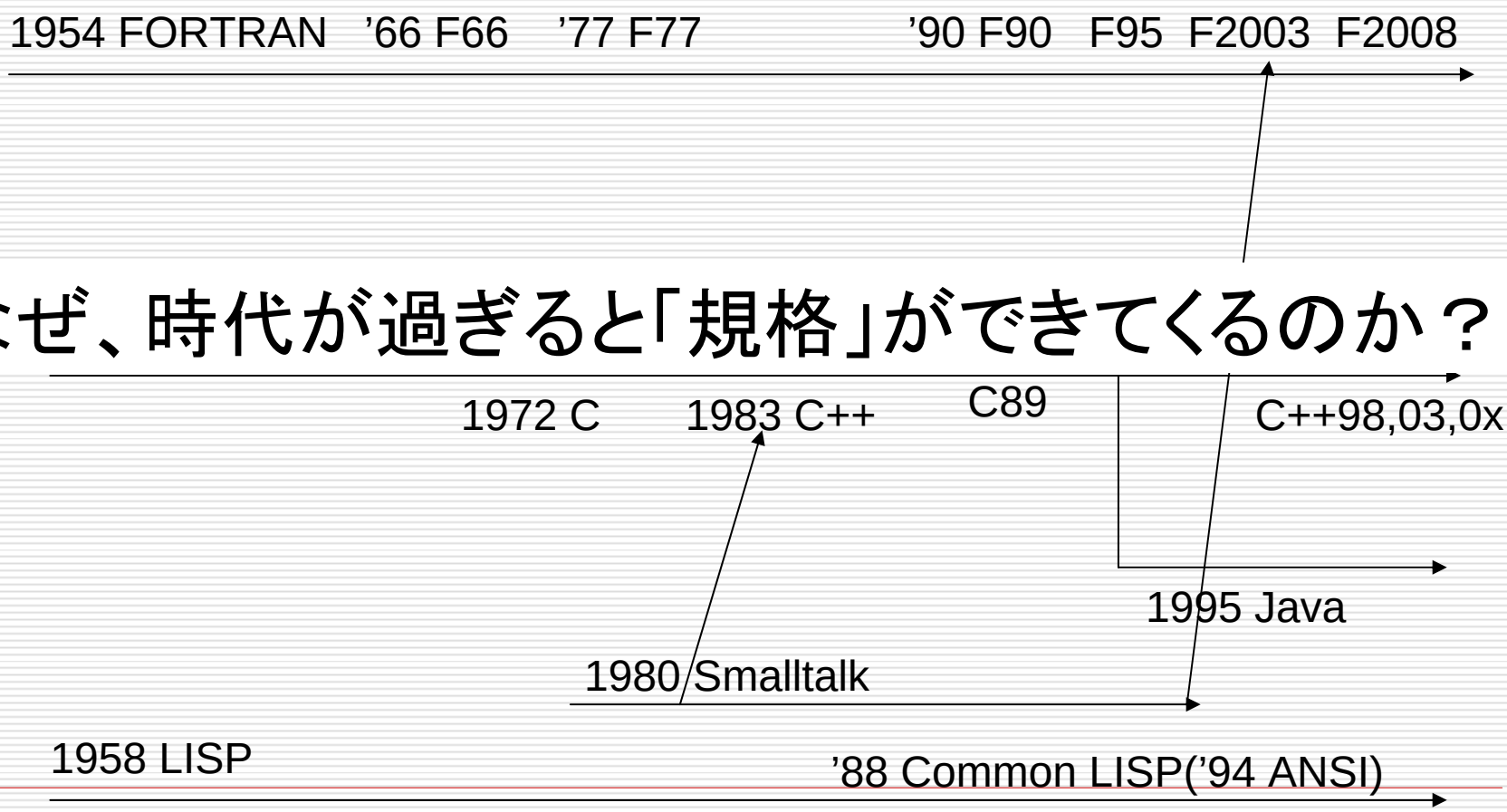
- オブジェクトはあたりまえ

- 総合的なプログラミング環境の提供
  - プログラミング方法論の拡張
    - テンプレート、継承、仮想関数
  - モジュールなどの提供
    - Module, namespace
  - ぶあついライブラリ群の提供

その他（落穂ひろい）

- スクリプト言語とインタープリタ
- システムとのインターフェイス、動的な環境変化への対応

最初の表に戻る


1954 FORTRAN
’66 F66
’77 F77
’90 F90
F95
F2003
F2008
なぜ、時代が過ぎると「規格」ができてくるのか？
1972 C
1983 C++
C89
C++98,03,0x
1995 Java
1980 Smalltalk
1958 LISP
’88 Common LISP(’94 ANSI)

プログラミング言語の規格(I)

- 初期のFortran
  - IBM Proprietary
  - Fortranが「使えるソフト」としてデファクトに
  - 各メーカがこぞってサポートを開始
  - 実装ごとに言語仕様を拡張 （そのうちのいくつかはよいアイデアとして他も採用）
  - 「Fortranプログラム」がコンパイルできるシステムとできないシステムがでてきた
- 規格の重要性の認識
  - ANSI (ISO)を主戦場とするか（C#）
  - 仲間を作って管理するか(各種コンソーシアム)
  - 一社（一者）で厳しく管理するか(Ada, Java)

プログラミング言語の規格(II)

- 規格の重要性の認識
- 規格を形式的に記述する技術の向上
  - SyntaxとSemanticsの分離
  - Syntaxは形式言語で(BNF)
  - Syntaxの足りないところは、BNFに対する注釈で
  - Semanticsはプログラムの実行の意味を決める
  - Semanticsは自然言語で記述

プログラミング言語の規格(III)

- Semanticsを記述する技術の向上が、言語の規格を厳格に定義することに大きく貢献した
  - 残念ながら、現在特定の形式主義に基づいたSemanticsの定義は行なわれていない（W3で無駄な試みがいくつか…）
  - Semanticsは自然言語で厳格に定義できる（数学が自然言語で展開されていることを考えればこれは驚くに足りない）
  - 必要だったのは、「形式主義」の理解と、それを遵守する能力（現在ではSemanticsを定める人間に大きな負担がかかっている）
- Fortranの規格を読んでみようか(冗談！)

プログラミング言語の規格

- 国際・国内機関
  - ISO
  - JIS
- コンソーシアム
  - IETF
  - W3
- その他
  - RSA他、ガリバーが保守する規格

次回の予定

- 「規格を読む」訓練をしましょう

- テキストはXMLの定義文書
  - http://www.w3.org/TR/2006/REC-xml11-20060816/
  - BNFの読み方を理解しましょう
  - 規格独特の言い回しを理解しましょう