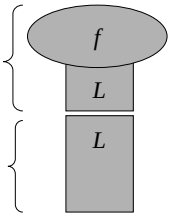
# 仮想機械とプログラミング言語

プログラミング言語特論講義資料
八杉 昌宏

## プログラムの作成と実行

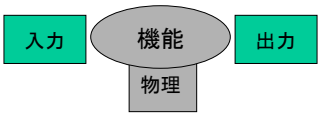
- プログラミング言語$L$で
  プログラムを書く。
  - 実行したい計算(仕事)を、
    機能 $f$ を持つプログラム
    として記述。
- $L$のプログラムを仮想機械
  上で実行する。






## 機械と利用者: 入出力

- 利用者は機械に入力を与え、出力を得る
- 入出力が情報のとき入出力を表現するための言語が必要
  - どのような形の表現が許されるのか
  - 表現は何を意味するのか





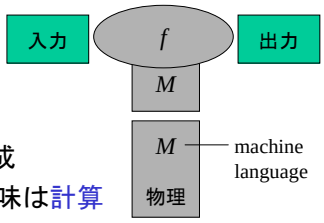

## 機能例・入出力例

- 電卓機能
  - 入力: ５＋１１=
  - 出力: １６
- 検索機能
  - 入力: ｌｌ,ｈｅｌｌｏ
  - 出力: ３



## プログラム可能な機械

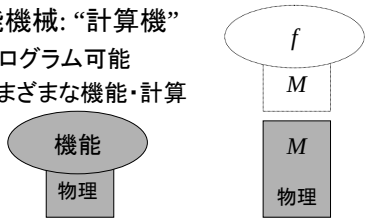
- 入力の一部としてプログラム
- プログラムの指示に従う機械
- プログラムは機械語 $M$ で作成
- プログラムの意味は計算






## 専用機械と万能機械

- 専用機械: 決まった計算のみ・高速
- 万能機械: “計算機”
  - プログラム可能
  - さまざまな機能・計算



## 大規模計算

- 最適化問題
- 探索問題
- 多体シミュレーション
  - 重力、分子間力
  - タンパク質の折りたたみ: どんな機能か
  - 遺伝子 アミノ酸
  - 専用機械が有利なことも



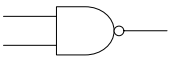
## 機械の動作原理

- 物理法則
  - 電子回路(しばしば論理回路)の原理:
    - ディジタル計算機を含む多くの電子機器
    - 基礎として: 電磁気学、量子力学、流体力学
    - 参考: アナログ計算機
  - 他に考えられるのは:
    - 力、光、熱、量子、超電導、磁気、分子、単電子
    - 正しく動くならどんな動作原理でもよい

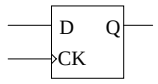


## 論理回路(順序回路)

- NANDゲート
  - 入力 $I_1$, $I_2$, 出力 $O$
    $O = \neg (I_1 \wedge I_2)$
  - 論理関数



- D-フリップフロップ
  - 入力 $D$, 出力 $Q$
    $Q$ = (最近クロックが与えられたときの$D$)
  - 状態遷移






## ソフトウェアとハードウェア

- ソフトウェア
  - プログラム(データとして)の物理的表現に電圧の高低などのすぐに変えられる性質のものを利用
- ハードウェアは固い
- ファームウェア
  - すぐには変えられない (参考: FPGA)



## 電卓プログラム例 (一部)

- 実はアセンブリ言語なので、本当はさらにアセンブラやリンカを通したものが機械語プログラム

```
L25:
        testl %ebx,%ebx
        je L27
        cmpl $1,%ebx
        je L28
        jmp L12
        .align 4
L27:
        leal (%edi,%edi,4),%eax
        leal -48(%edx,%eax,2),%edi
        jmp L12
        .align 4
L28:
        leal (%esi,%esi,4),%eax
        leal -48(%edx,%eax,2),%esi
        jmp L12
        .align 4
```



## 高水準プログラミング言語

- 機械語(アセンブリ言語)である程度以上大規模なプログラムを書くのは困難
  - 機械語は機械が効率よく実行するための言語
- 高水準プログラミング言語の必要性
  - 特定の機械語からは離れて、人が扱い易い(読み書きし易い)形で、計算を表現
  - 明確な言語仕様

C言語

```
int r1 = 0, r2 = 0, r3 = 0;
int state = 0;
while((c = getchar()) != EOF){
   switch (c){
   case '0': case '1': case '2': case '3': case '4':
   case '5': case '6': case '7': case '8': case '9':
     switch (state){
     case 0: r1 = r1 * 10 + c - '0'; break;
     case 1: r2 = r2 * 10 + c - '0'; break;
     } break;
   case '+': case '-':
     switch (state){
     case 0: op = c; state = 1; break;
     case 1: error(); break;
     } break;
       ...
  }
 }
```



## プログラミング言語の仕様

- シンタックス(構文論)
- セマンティクス(意味論)



## シンタックス(構文)

- プログラムの正しい形
- 通常、文法により与える。
- BNF記法がよく使われる。

```
<exp>    ::= <digits> <op> <digits>
<digits> ::= <digit> | <digits><digit>
<op>     ::= + | -
<digit>  ::= 0 | 1 | 2 | ...
```

- 言語について記述するためにも言語が必要: メタ言語



## セマンティクス(意味)

- プログラムの意味: 計算
- よくあるのは日本語など自然言語をメタ言語に使って記述 (BNFのような決定版なし)
- ある言語のプログラムも、メタ言語としてはデータである。
- メタ言語に人工言語を使うことも可能だが、メタメタ言語が必要になる。
- どこかで自然言語が必要。



## セマンティクス(意味)

- 操作的意味論
  - プログラムの実行過程を与える
- 表示的意味論
  - 数学的オブジェクトの表示と考える
- 公理的意味論
  - プログラムの満たす性質の証明論



## 操作的意味論

- *L*言語のプログラムの実行過程を与える
- 仮想的な計算機による実行ともいえる
- *ML*言語と、「*ML*プログラムを実行する仮想機械」を使って意味を与えることも:
  - 解釈: *L*プログラムを解釈実行する*ML*言語で書かれた解釈器
  - 翻訳: *L* 言語から *ML*言語への翻訳

## 仮想機械(Virtual Machine)

- 仮想計算機、抽象機械(abstract machine)などともいう
- (単純な)仮想機械の振る舞いを状態遷移規則などの形で与えることで、*ML*言語のプログラムの意味も決まる
- *L*言語のプログラムの実行過程もこれから分かる
- 物理的には動作する必要はない



## 言語仕様と処理系実装

- 高水準言語のプログラムを実行するためのシステム: 言語処理系
- 言語の仕様(シンタックスとセマンティクス)により、その言語のプログラムを実行する処理系の満たすべき仕様も定まる。
- 処理系の実装方式は、仕様を満たす限り自由 (腕の見せ所)



## 仕様と実装

- (一般論)
- システムはある機能を持つ
- 持つべき機能を仕様として与える
- 仕様を満たすようにシステムを実装(実現)
- 実装方式は仕様を満たす限り自由

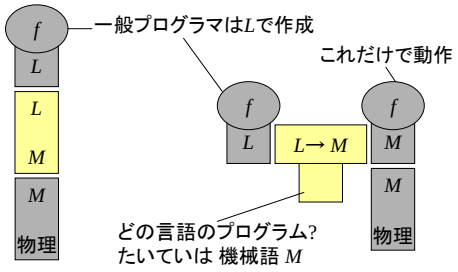


## 言語処理系の実装方式

- 解釈器
  - インタプリタ、評価器、シミュレータ、エミュレータなどともいう
  - *L*言語のプログラムを(入力の一部として)解釈実行する機械語プログラム
- 翻訳系
  - コンパイラ、トランスレータ、変換系などとも
  - *L*言語のプログラムを機械語プログラムに翻訳する (機械語プログラムだけで実行可)

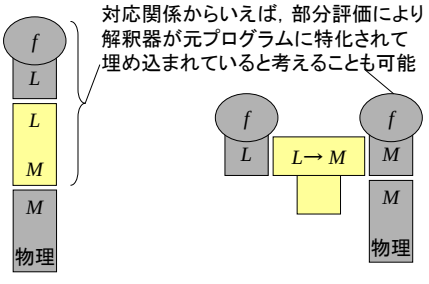


## 解釈器 (I)　　翻訳系(T)






## 解釈器 (I)　　翻訳系(T)



## 高水準機能の低水準での実装 (翻訳の場合)

- たとえば手続き呼出し
  x = f(a, b);
- 機械語の例としては:

```
load  [a], r0
load  [b], r1
mov   pc, r7
jmp   [f]
store [x], r0
```

  - 単なるデータと制御の移動
  - 同じ呼出し慣例に従わないと、正確に呼び出せない



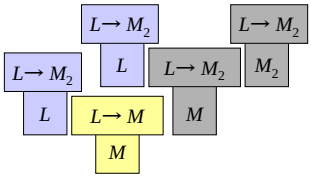
## 最適化

- 機械語のプログラムは、もとのプログラムの意味を保存しさえすれば、(気づかれなければ) 何をやってもよい。
  - 使われていない変数の値を計算しない
  - 同じ計算を2回しない
  - 絶対に実行しない部分は翻訳しない
  - あらかじめ分かることは計算しておく



## クロスコンパイラ

- 高水準言語 $L$ で $L$から $M_2$への翻訳系
- 機械語 $M$ 上の開発済み翻訳系



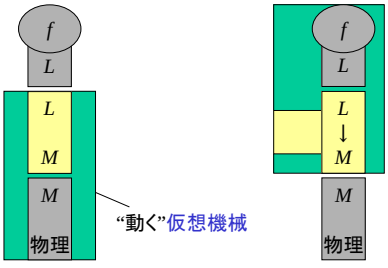



## 対話的プログラミング

- ML処理系、Lisp処理系などで多い
- read-eval-print ループ
- インタプリタと呼ばれるが、実は、read-compile-execute-print しても構わない
- インタプリタが内部で翻訳しても構わない
  - 翻訳結果は取り出さないが

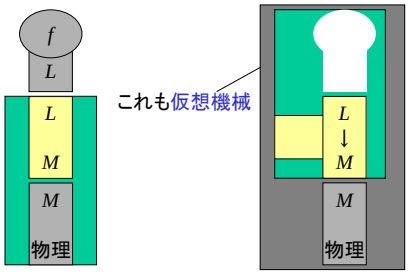


## 解釈器 (I)　翻訳系(T)






## 解釈器 (I)　翻訳系(T)



## 仮想機械

- ここでは、意味を与えるため(抽象機械)というより、実際にプログラムを実行するためのもの (言語処理系を作成するため)
- 利用者としては、実装の詳細が不明でも機械の機能(プログラムの意味)は分かる。
- 物理的機械と対比: 見た目は変わらない
- 物理的機械上でエミュレートするか
- 物理的機械上のプログラムに翻訳するか



## ハードウェアとの協調による仮想化

- 仮想記憶、仮想機械、プロセス
  - プログラミング言語というよりは オペレーティングシステムの話
- ハードウェアのサポートにより機械語プログラムが仮想的な環境で動く
- 仮想記憶: 物理メモリより多くのメモリ
- 仮想機械: OSの上で 別のOS の実行
- 時分割: 複数の計算機、マルチタスク



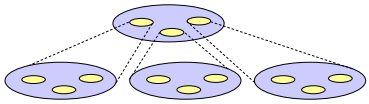
## メタ言語的抽象以外の実装方式

- 手続き抽象やデータ抽象
  - 名前をつけ、内部の詳細を隠す
  - モジュール化、オブジェクト指向
  - ある機能を実現するのに部品を組み合わせる
    - システムを部品から構成
    - 構成されたシステム自身が部品にもなる。
- 隠さない例も: 文法構造

  <exp> :: = if <exp> then <exp> else <exp>



## モジュールの階層構造



- 同じ言語内での話: 解釈器・翻訳系と違う
  - ただし組み込みプリミティブはメタ言語で実現される
- 似たような階層構造: 論理回路でも
  - 計算機ハードウェア: プロセッサ+メモリ+…
  - プロセッサ: ALU+レジスタ+…



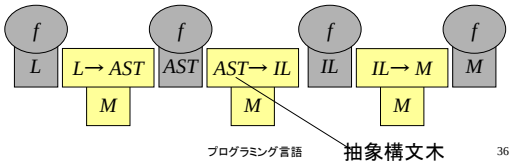
## 仮想化(抽象化)の階層

- 高水準言語をより低い水準の言語へ翻訳 (より低い水準の言語: 中間言語 *IL*)
- 中間言語*IL*から機械語へ翻訳
- 何段も中間言語があることも
- 翻訳だけでなく解釈も
  - 計算機の高性能化が解釈も実用的にした
- 中間言語インタプリタを仮想機械と呼ぶことも多い (最近では Java Virtual Machine)



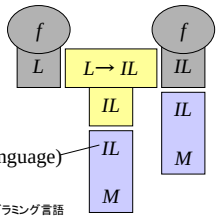
## コンパイラ(翻訳系)

- 翻訳だけでなく
  - 構文の正しさの確認
  - 型チェック
- *IL*中間表現上で最適化も
  - 意味を保存して変形



## 解釈器・翻訳系による実装

- 機械語 *M* 上の *IL* 言語解釈器
- *L*言語から *IL*言語へのコンパイラを*IL* 言語で記述
- コンパイラは使いまわせる

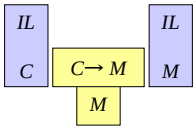



中間言語(intermediate language)



## 解釈器・翻訳系による実装

- 高水準言語 *C*の機械語 *M* 上の開発済み翻訳系があれば、
  *IL* 言語解釈器は*C*で記述可能






## インタプリタ(解釈器)

- 中間言語*IL*の実行コードとしては:
  - 実行しやく、コンパクトなのがよい
  - バイトコードが使われることが多い: JVM等
- 内部的に、高速に実行できるプログラム(機械語コード) に変換(翻訳)してよい
  - Just-In-Timeコンパイラ, Hotspot



## 仮想化

- 21世紀になっても宇宙時代はこなかったがCGなどでの宇宙旅行の仮想体験も
- 何が物理的で何が仮想的かはどんどん曖昧になっていっている。



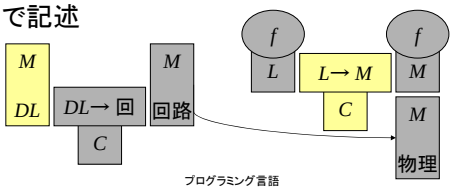
## 仮想機械

- 物理的機械に相当する仮想機械も
- エミュレータ
  - 古い計算機用のプログラムが実行できる
  - 処理速度が課題



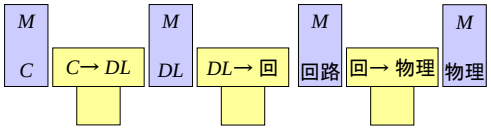

## ハードウェア・ソフトウェア協調設計

- システムの機能 *f* を *L* 言語で記述
- 機械語 *M* の計算機を*DL*言語で記述
- *L*言語から*M*言語へのコンパイラを*C*言語で記述



## プログラミング言語による ハードウェア設計

- 高水準言語 *C* でハードウェア設計
- 設計言語 *DL*へと翻訳






## シミュレータ

- 回路シミュレータ
- 半導体デバイスシミュレータ
- 実際にハードウェアを作ると高い
- 試験




## ハードウェア仮想化技術

- 気づかれなければ何やってよい
  - レジスタのrenaming: 物理レジスタを増やす
  - 予測、投機的実行: 間違ったらキャンセル
  - Out-of-Order 実行: できることからやる
  - キャッシュ: メモリアクセスをさぼる
- CISC vs RISC 論争、今は…
  - なぜか、i386系が速い
  - 内部で命令を μOP という命令に翻訳



## 通信での仮想化の階層

- ISOのOSI (Open System Interconnection)
  - アプリケーション層 アプリケーション間
  - プレゼンテーション層 データ表現方法
  - セッション層 開始と終了 connect, close
  - トランスポート層 プロセス間通信 TCP, UDP
  - ネットワーク層 ノード間通信(中継) IP
  - データリンク層 隣接ノード間 ethernet, PPP
  - 物理層 伝送路の物理特性 UTP
- 近年: 仮想プライベートネットワークなど