# プログラミング言語処理系論 (6)
# Design and Implementation of Programming Language Processors

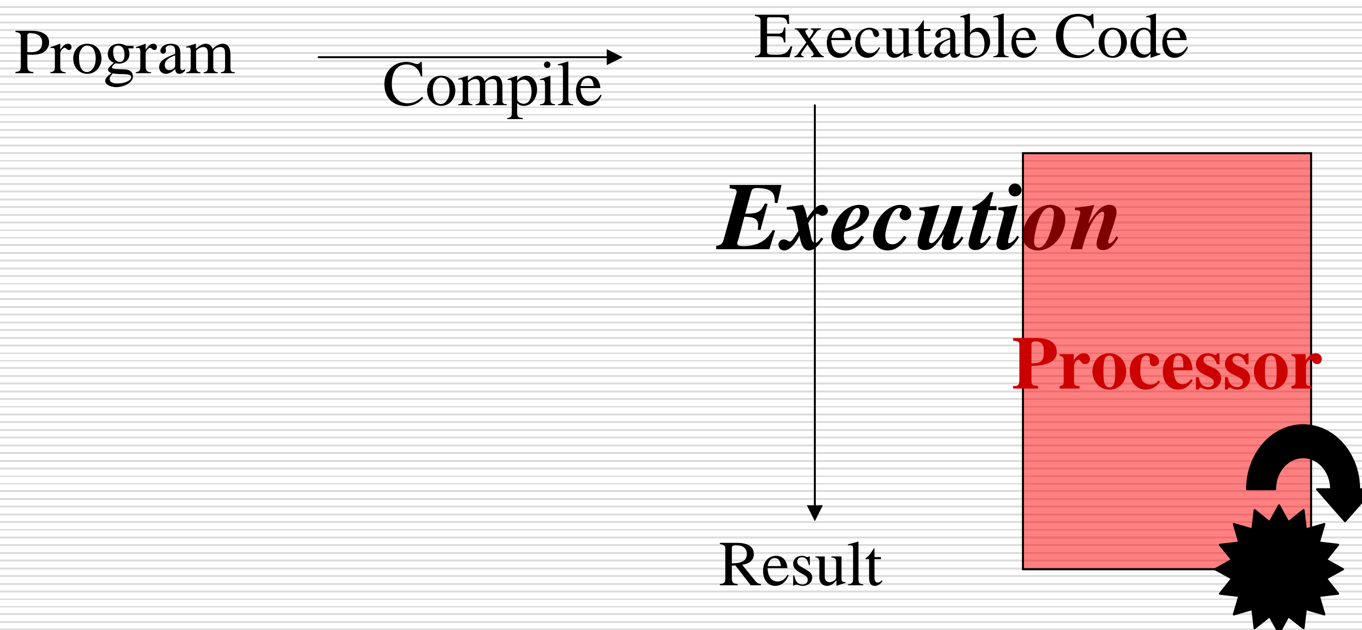
佐藤周行
（情報基盤センター/電気系専攻融合情報学コース）

# 今回の予定

- プログラムを実行するために
  - プログラムを実行するためのいろいろなこと
  - Engine
  - 実行環境
  - 実行

# 一般的な説明

- Compile and Go



# 例：Windows上での実行

% gcc a.c

% ./a.exe

ソースコードとしてa.cをとり、実行ファイルとしてa.exeが作られる

a.exeを実行すると結果が出てくる

それだけ？

% javac SecTest.java

% java SecTest

ソースコードとしてSecTest.javaをとり、クラスファイルとしてSecTest.classが出てくる。

SecTest.classを引数にしてjavaを実行すると結果が出てくる

それだけ？

# 実行コードとEngine

- 実行コードって何?

  Depends on Execution Engine.
  - CPU
    - Pentium, Itanium,
    - SPARC, POWER
    - ...
  - Virtual Machine
    - Java VM
    - PostScript

# ＋ライブラリ関数のサポート

- 通常のプログラミングでは、多くのライブラリ関数を付与している
- 組み込みシステム上での実行など、それらを想定しないものもある
- Cの場合、ライブラリ関数は通常の関数のほかに、ヒープの管理、I/Oなど、OSとのインタフェースを取る関数を含む
- ライブラリ関数の定義は言語の定義の一部になっている

- Def.Hosted Environment
  - 規格中のライブラリ関数をすべてサポートする環境
  - 関数はOSとのインタフェース関数を含むので、OSが動いていることを仮定していると考えてよい
  - Mainからはじまることを仮定する

- Def. Freestanding Environment
  - Freestanding implementationが動くのに十分な環境
  - ライブラリはごく限られたもの。OSとのインタフェースを仮定しない
  - Mainからはじまることを仮定しない

# 実行コードとEngine

- Engine ＋ 支援機能
  ＝ **実行時環境**

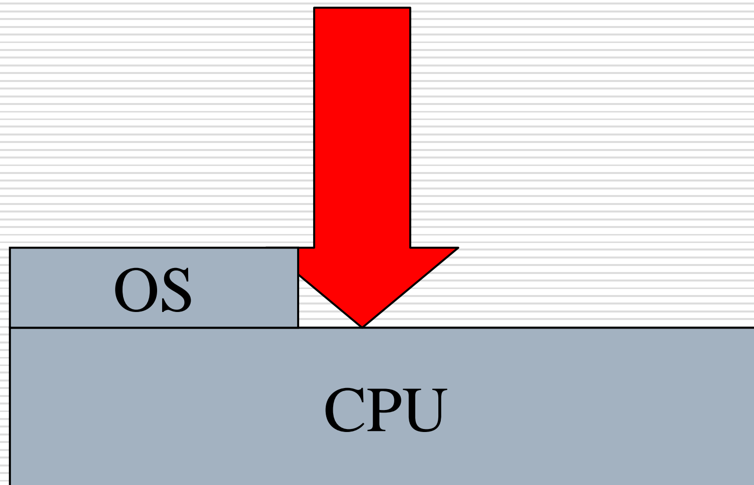
| CPU | Operating System<br>Memory Management<br>Thread Management |
| --- | --- |
| Java VM<br>Engines for Many Functional Languages | Memory Management including Garbage Collection<br>Thread Management including Synchronization. |

# Compiler and Interpreter

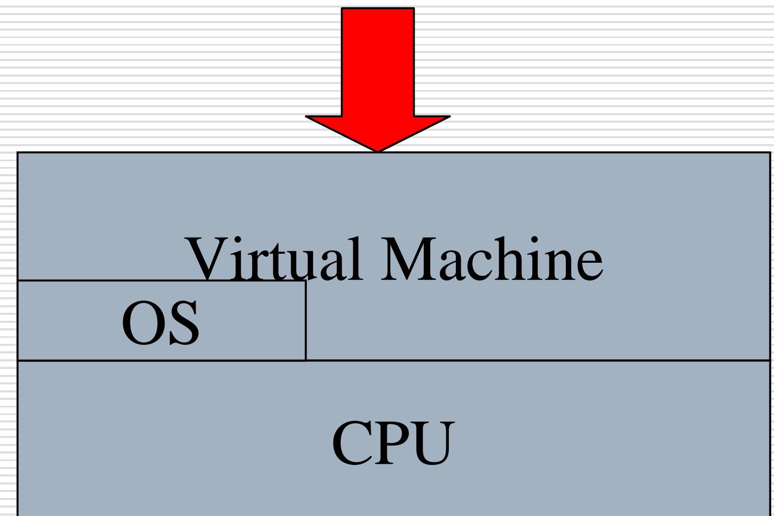
( Note this discussion is Obsolete)

- Conversion to Executable Code on :

Compiler




Interpreter

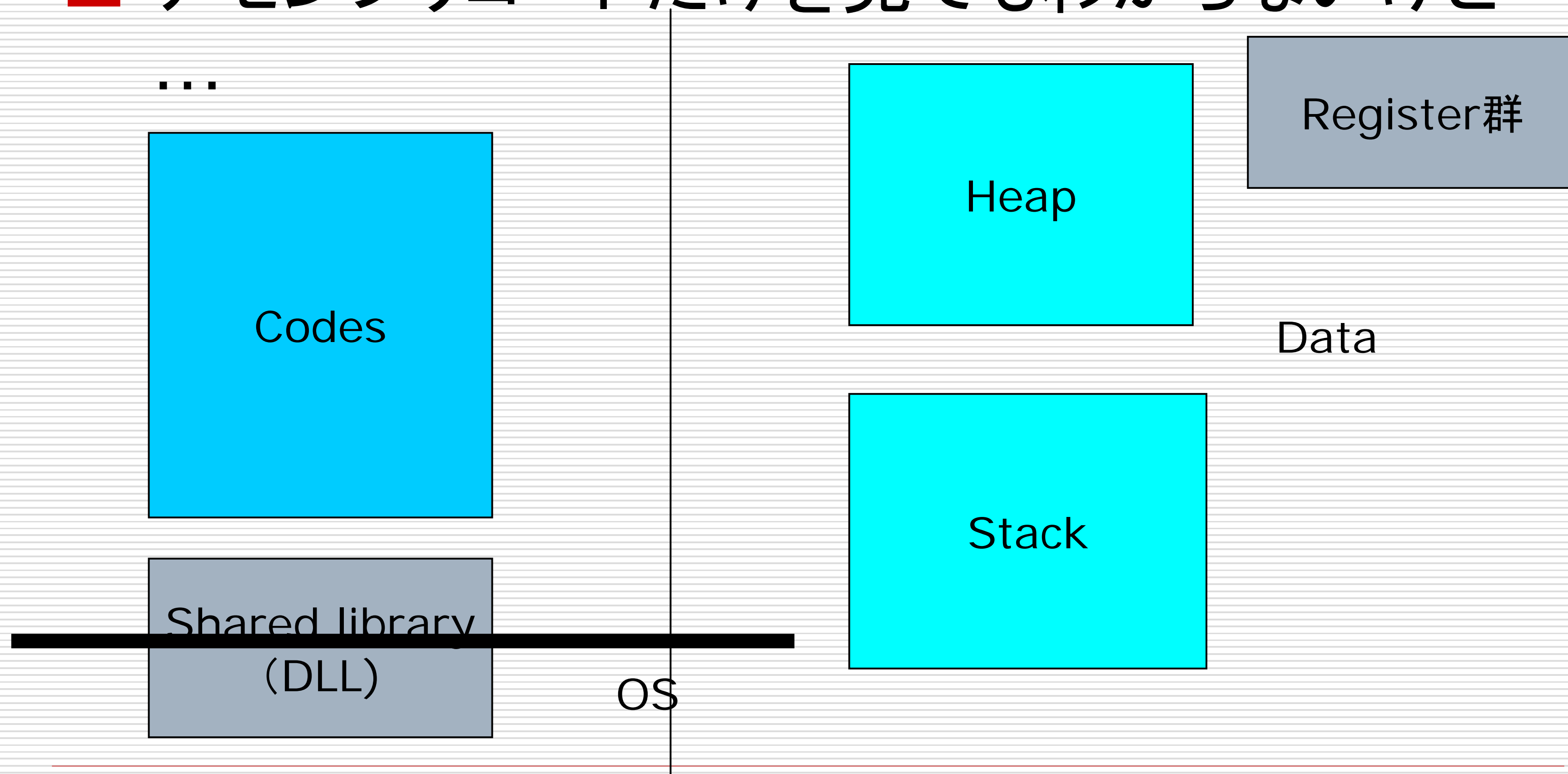


# Cの実行環境

- GCCはCPUを直に実行するコードを生成する
- アセンブリコードを見てみる
- ソースコードa.c
- Gcc –O3 (3.4.4)
- アセンブリコードa.s
- この後に、リンクが入るのだが、これは省略する

# Cの実行環境

- アセンブリコードだけを見てもわからないけど…



- OSの機能として
  - I/O、システムコールの提供
  - Shared libraryの準備
  - ヒープ領域の準備
    - データ領域にヒープ領域を確保
    - メモリ管理(malloc)の初期化
    - メモリ管理の実際
    - より進んだ言語ではGCも
  - スタック領域の準備
    - データ領域にスタック領域を確保

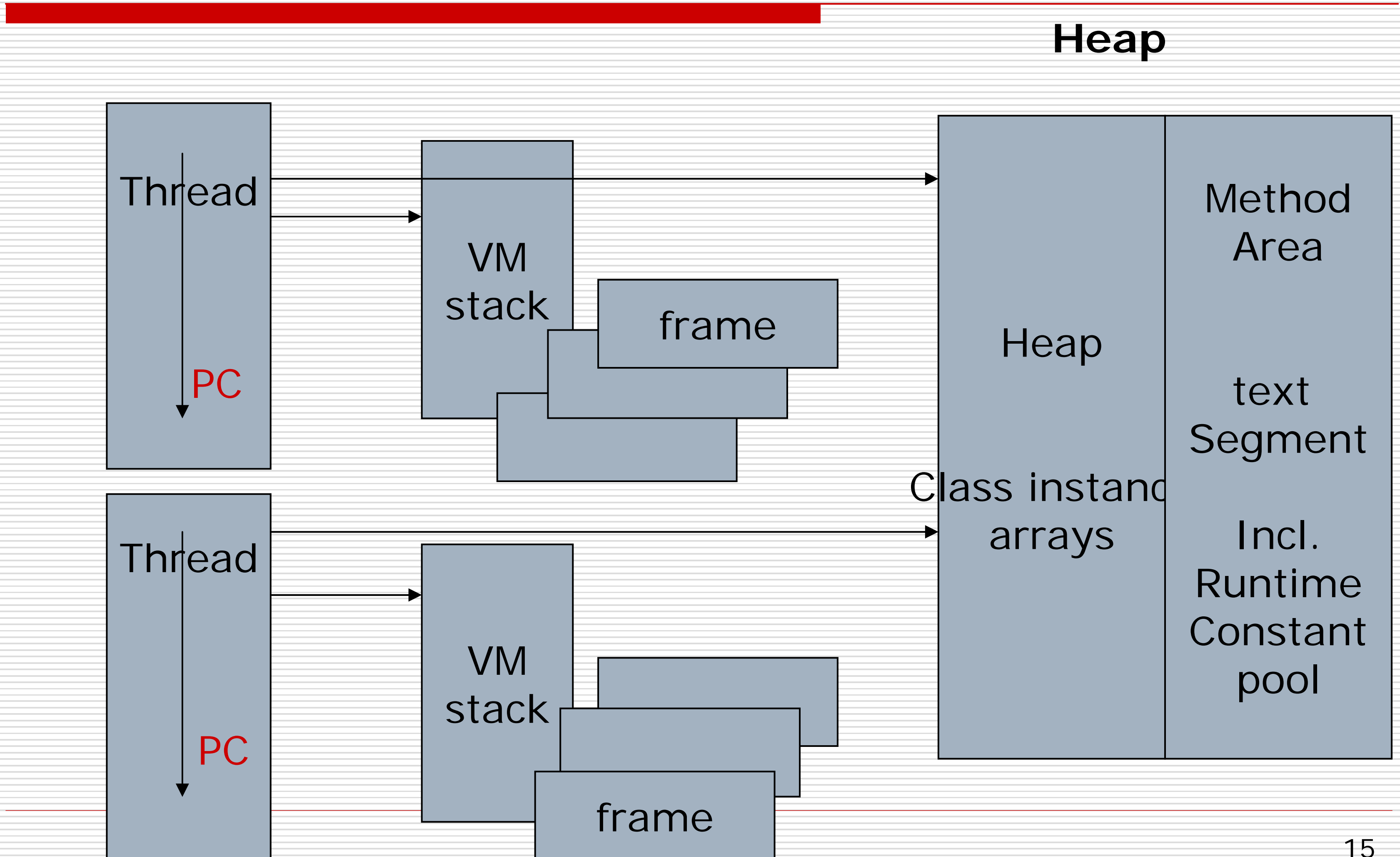
# Javaの実行環境

- Cと同じことがJavaに言えるだろうか？

# Specification of Java VM


- 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



# Java VM

- スタックマシン
  - Code Size Dramatically Small
  - Simple → Mobility, Portability ◎
- 特殊な用途のために少数のレジスタ
- Instructions Operate on Stack
Specification:
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

- （課題6）
  JAVA　VMの仕様を読み、以下の観点から整理して特徴を記せ。
  （１）VMの命令があつかうデータ
  （２）メモリ（スタックとヒープ）管理
  （３）スレッド管理
  （４）オブジェクト管理

- (課題6’)
  Microsoftが提供する.NETの中でのVM環境であるCLR(共通言語ランタイム)について、Java VMと同じ事を調査せよ。

# Cと同じ事をJavaでしてみる

- ソースファイルa.java
- ディスアセンブルしたコード(javap –c)

```
Compiled from "a.java"
class a extends java.lang.Object{
a();
  Code:
   0:  aload_0
   1:  invokespecial #1; //Method
       java/lang/Object."<init>":()V
   4:  return

int add12and13(int,int);
  Code:
   0:  iload_1
   1:  iload_2
   2:  iadd
   3:  ireturn

}
```

# “Compiling” Java Program

```
int add12and13() {
    return addTwoStatic(12, 13);
    }
```

Method

```
    bipush 12
    bipush 13
    invokestatic #3
    ireturn
```

# Java VM Frameについて

- Frameに入っているもの
  - Local variables
    - Parameterを含む
    - Local var 0 はmethodを呼び出したオブジェクトをさす
  - Operand stack
  - Reference to the runtime constant pool

- Frameの生成・破壊
  - Methodが呼び出されるたびに生成される
  - Methodが「complete」すると破壊される

- Operand stackって？
  - JAVA VMの命令はスタック上にあるオペランドを操作する → これがスタックマシンのいわれ
- Reference to a constant poolって？
  - Dynamic Linkingのサポート

Dynamic linking translates these symbolic method references into concrete method references, loading classes as necessary to resolve as-yet-undefined symbols, and translates variable accesses into appropriate offsets in storage structures associated with the runtime location of these variables.

# Frame Management

- Key Idea in Function Call/ Method Invocation
  - Keeping Information Local to Function/Method
    - Arguments
    - Local Variables
    - Return Address

# Frame Management(2)

- Management Scope and Extent of Variables

  Anyway, something is Necessary.

  Stack for Frame (Access Here)

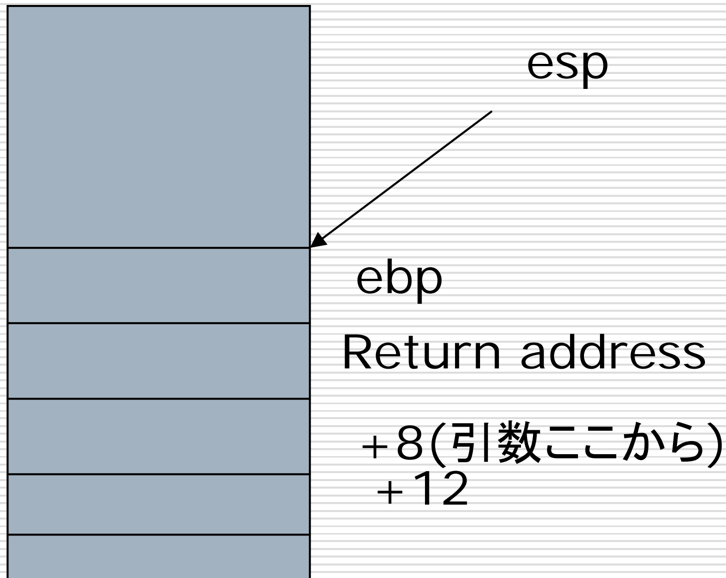
  Stack for Saving Non-Local Values

  Dedicated Area for each Subroutine (Old-fashioned Fortran) ← Recursion ×

# Frame Management(3)

- 関数呼び出しの制御をするには、呼び出し側と呼び出され側で引数をどう扱うかについての合意が必要
  → calling convention
- 例:GCC

# GCCのcalling convention




```
_f2:
    pushl   %ebp
    mov
        %esp, %ebp
    fldl    16(%ebp)
    faddl   8(%ebp)
    fmull   24(%ebp)
    popl    %ebp
    ret
```

- （課題7）
手近にあるコンパイラをひとつ対象にし、
calling conventionとフレームを解析せよ。この時に
以下のことに注意せよ
（1）コンパイラ・OS・CPUを明記すること
（2）解析の手法を明らかにすること
（3）calling conventionは、呼び出し側と呼び出される側の約束であるが、そのときに呼び出される側から呼び出し側へも約束が存在することに注意せよ

- Calling conventionは一般にハードウェアとしてのCPUの機能ではなく、言語処理系での「約束事」
- スタックにつむのはIntel系で一般的
- その他では、レジスタのある部分を引数を割り当てるのに使用する言語処理系が多数
  - IBM　POWER上のコンパイラ
  - SPARC上のコンパイラ

# Need for Optimization

- So far, Simple Interpretation
- For Fast/Efficient Execution, Optimization through Program Analysis

# 次回

- 最適化について！