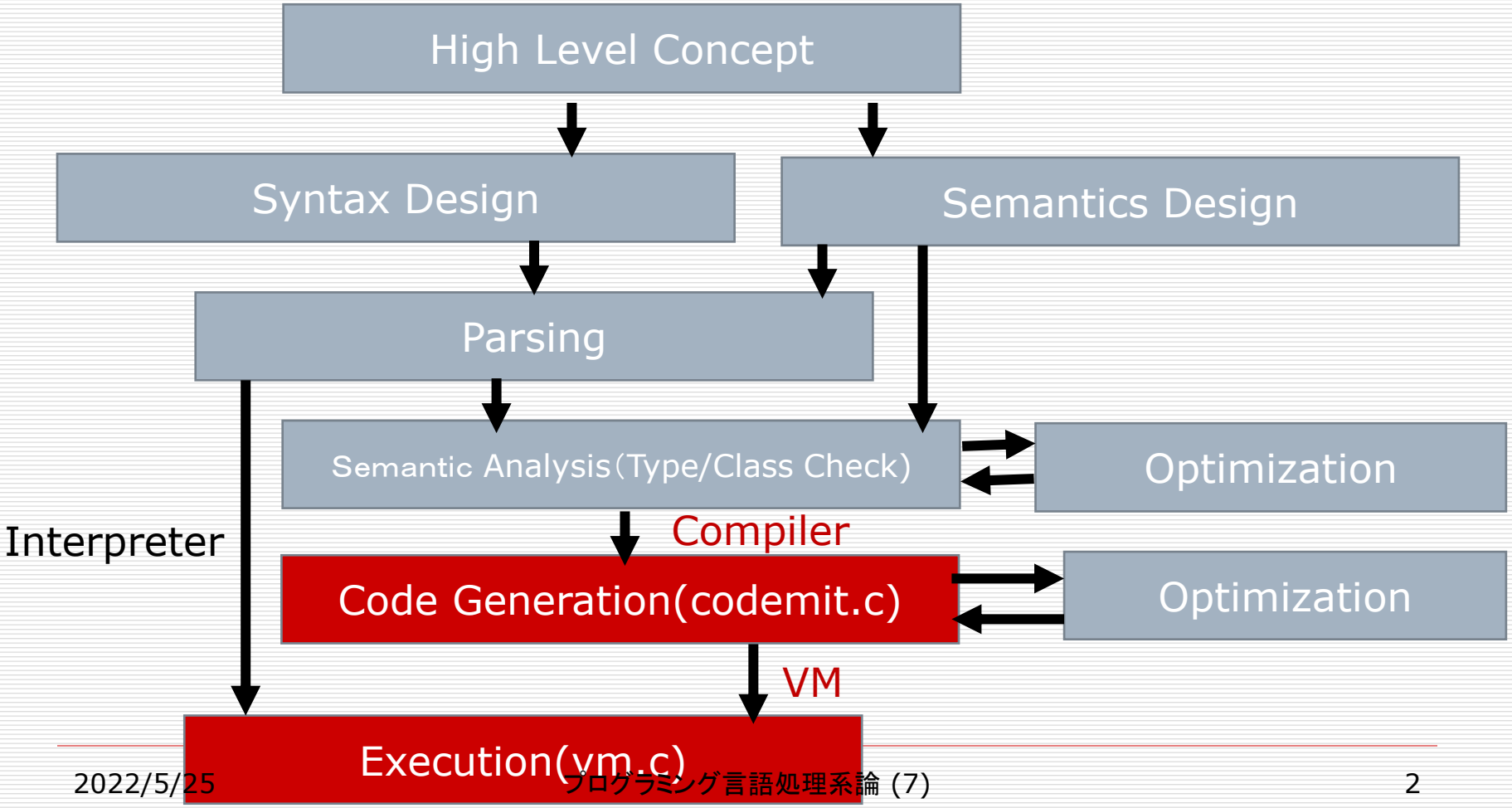
プログラミング言語処理系論 (7)
Design and Implementation of Programming Language Processors

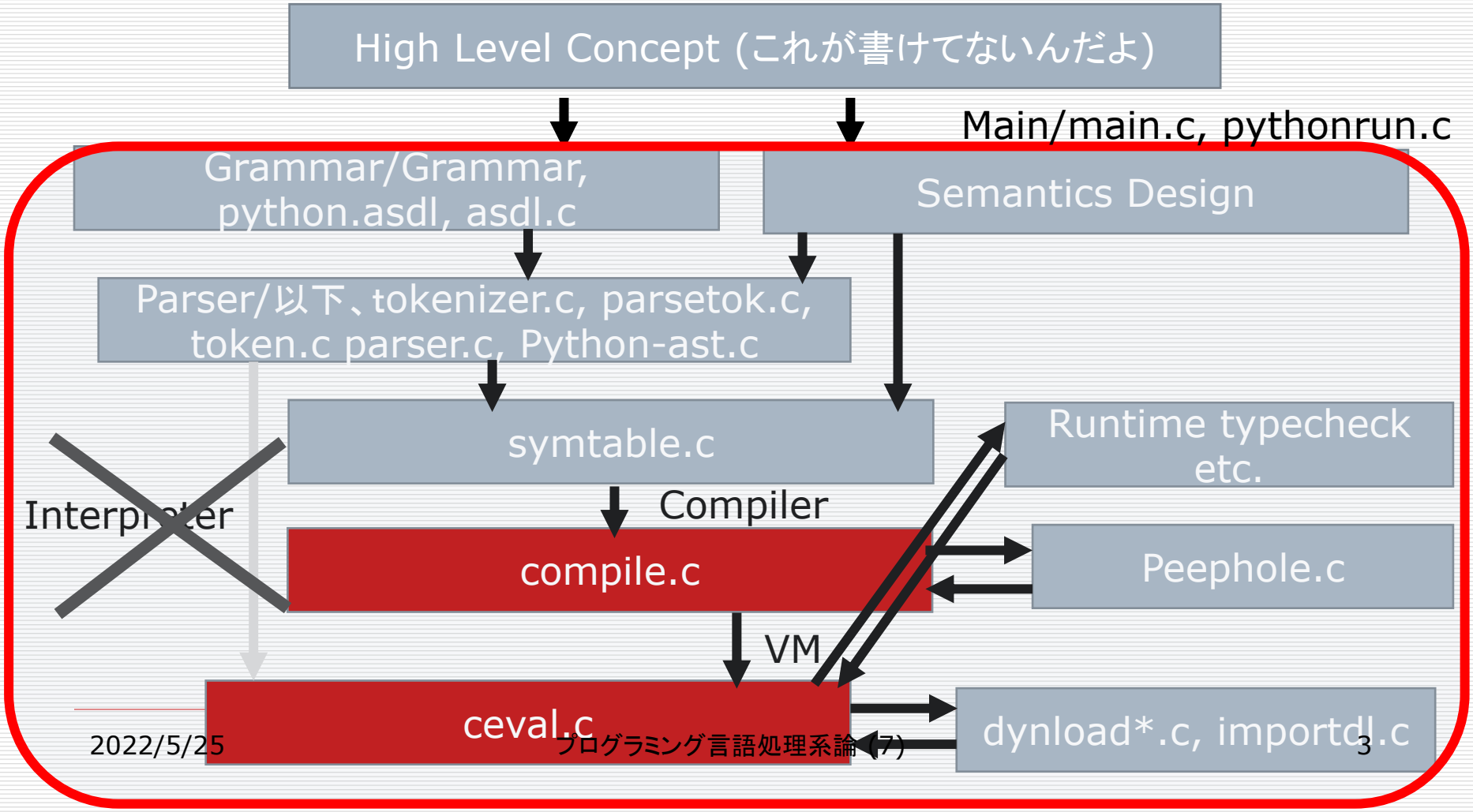
佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

次のステップ（教材）


High Level Concept
Syntax Design
Semantics Design
Parsing
Semantic Analysis（Type/Class Check）
Optimization
Interpreter
Compiler
Code Generation(codemit.c)
Optimization
VM
Execution(vm.c)

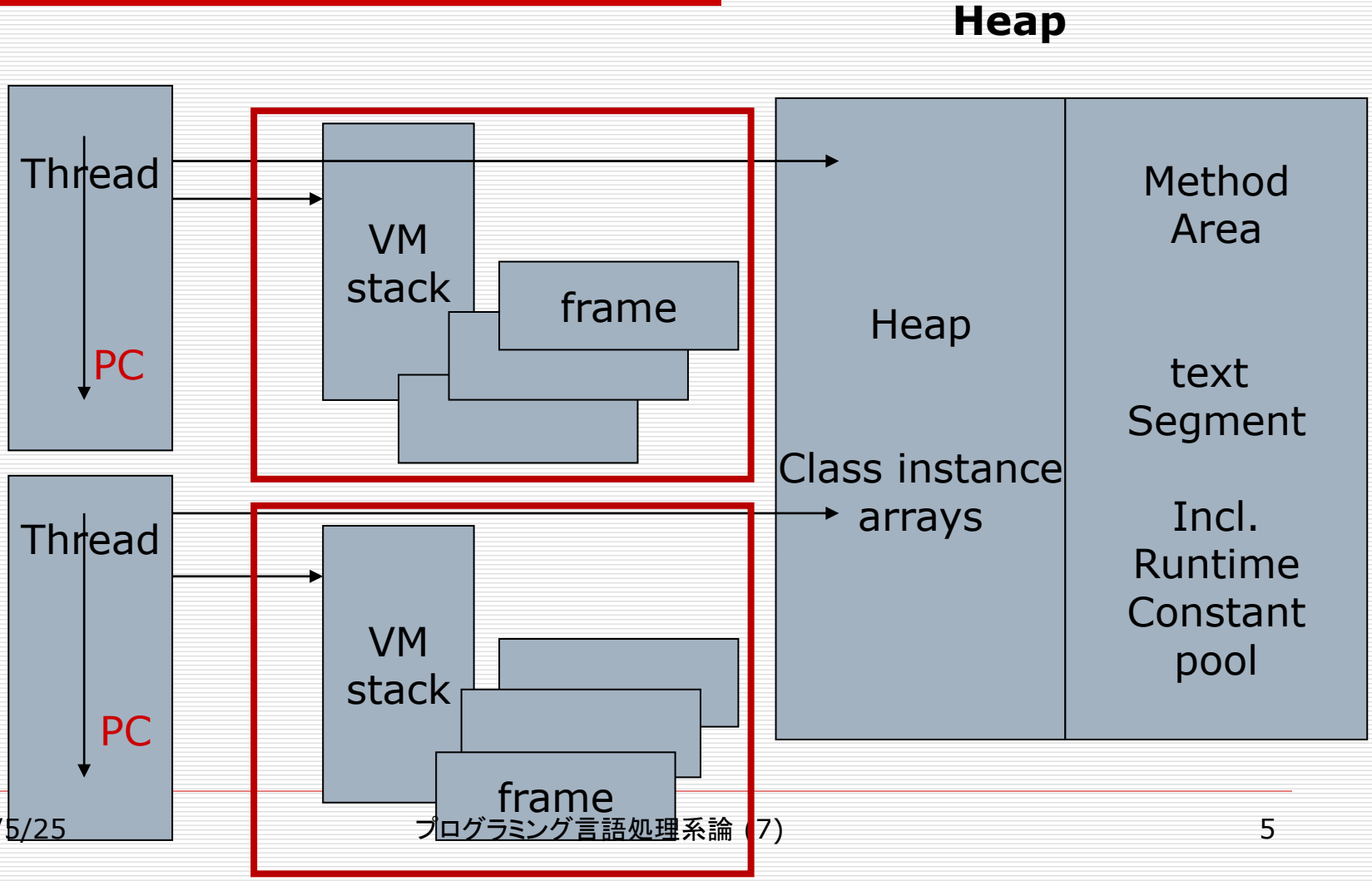
CPython


High Level Concept (これが書けてないんだよ)
Main/main.c, pythonrun.c
Grammar/Grammar, python.asdl, asdl.c
Semantics Design
Parser/以下、tokenizer.c, parsetok.c, token.c parser.c, Python-ast.c
symtable.c
Runtime typecheck etc.
Interpreter
Compiler
compile.c
Peephole.c
VM
ceval.c
dynload*.c, importdl.c

stack machineの定義：ターゲットの特定

- ISAを定める
  - Stack
  - Code segment
  - Data segment
  - Special registers/Data Structure
- 実行エンジンのプログラムを書く
  - 教材ではvm.c，CPythonではceval.c
- ASTから命令列を生成するプログラムを書く (教材ではcodemit.c/compile.c) ← evalとの相似に注目する（部品をコンパイルした結果を積み上げてコードを生成するという意味ではhomomorphism）

JAVA VM (本格的)


Heap
Thread
PC
VM
stack
frame
Thread
PC
VM
stack
frame
Heap
Class instance
arrays
Method
Area
text
Segment
Incl.
Runtime
Constant
pool

CPythonで必要なセグメント

- Code(コード領域)
- Stack (計算をする)

ここから下はGCの対象

- Frame (関数実行のためのローカルな環境)
  - ここにあるデータはLOAD_FASTでstackに移動
- Heap (グローバルなデータ)
  - ここにあるデータはLOAD_GLOBAL, STORE_GLOBALでstackとのデータ移動

Compile

- VMを定義して、その上のコード生成を行う
  - 教材：stack machineの設計

```
t_codeseg  codeseg[CODESEGSIZE];
t_stackseg stackseg[STACKLEN];
t_dataseg dataseg[DATASEGSIZE];

t_symtable  objectab[128];

int framestack[STACKLEN];
```

Mnemonic （最小限）

```
#define OP_NOP 1
#define OP_POP 2
#define OP_DUP 3
#define OP_DUP2 4
#define OP_U_NOT  5
#define OP_B_ADD 6
#define OP_B_SUB 7
#define OP_B_MUL 8
#define OP_B_DIV 9
#define OP_B_GREATEREQ 10
#define OP_B_GREATER 11
#define OP_B_LESS 12
#define OP_B_LESSEQ 13
#define OP_B_EQUAL 14
#define OP_B_NOTEQUAL 15
```

```
#define OP_JMP 16
#define OP_JMPPOS 17
#define OP_JMPZ 18
#define OP_JMPNEG 19
#define OP_PUSH_CONST 20
#define OP_PUSH_GVAR  21
#define OP_PUSH_LVAR  22
#define OP_PUSH_ADDR  23
#define OP_PUSH_FIELD 24 /* */
#define OP_STORE_GVAR 25
#define OP_STORE_LVAR 26
#define OP_STORE_ADDR 27
#define OP_STORE_FIELD 28 /* */
#define OP_CALL 29
#define OP_CALL2 30
#define OP_RETURN 31
#define OP_PRINT_TOP  32
```

CPython (Include/opcode.h)

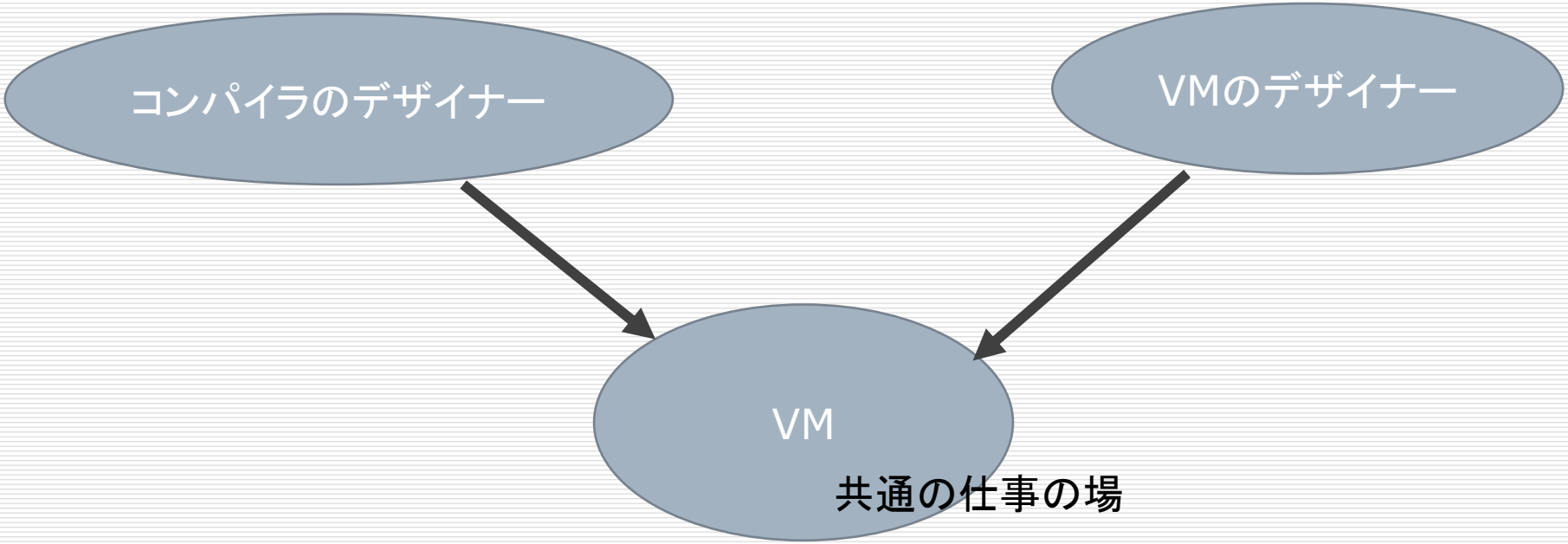
```
#define POP_TOP                   1
#define ROT_TWO                   2
#define ROT_THREE                 3
#define DUP_TOP                   4
#define DUP_TOP_TWO               5
#define ROT_FOUR                  6
#define NOP                       9
#define UNARY_POSITIVE           10
#define UNARY_NEGATIVE           11
#define UNARY_NOT                12
#define UNARY_INVERT             15
#define BINARY_MATRIX_MULTIPLY   16
#define INPLACE_MATRIX_MULTIPLY  17
#define BINARY_POWER             19
#define BINARY_MULTIPLY          20
#define BINARY_MODULO            22
#define BINARY_ADD               23
#define BINARY_SUBTRACT          24
#define BINARY_SUBSCR            25
#define BINARY_FLOOR_DIVIDE      26
#define BINARY_TRUE_DIVIDE       27
#define INPLACE_FLOOR_DIVIDE     28
#define INPLACE_TRUE_DIVIDE      29
#define GET_AITER                50
#define GET_ANEXT                51
#define BEFORE_ASYNC_WITH        52
#define BEGIN_FINALLY            53
#define END_ASYNC_FOR            54
#define INPLACE_ADD              55
#define INPLACE_SUBTRACT         56
#define INPLACE_MULTIPLY         57
#define INPLACE_MODULO           59
```

```
#define STORE_SUBSCR            60
#define DELETE_SUBSCR           61
#define BINARY_LSHIFT           62
#define BINARY_RSHIFT           63
#define BINARY_AND              64
#define BINARY_XOR              65
#define BINARY_OR               66
#define INPLACE_POWER           67
#define GET_ITER                68
#define GET_YIELD_FROM_ITER     69
#define PRINT_EXPR              70
#define LOAD_BUILD_CLASS        71
#define YIELD_FROM              72
#define GET_AWAITABLE           73
#define INPLACE_LSHIFT          75
#define INPLACE_RSHIFT          76
#define INPLACE_AND             77
#define INPLACE_XOR             78
#define INPLACE_OR              79
#define WITH_CLEANUP_START      81
#define WITH_CLEANUP_FINISH     82
#define RETURN_VALUE            83
#define IMPORT_STAR             84
#define SETUP_ANNOTATIONS       85
#define YIELD_VALUE             86
#define POP_BLOCK               87
#define END_FINALLY             88
#define POP_EXCEPT              89
#define HAVE_ARGUMENT           90
#define STORE_NAME              90
#define DELETE_NAME             91
#define UNPACK_SEQUENCE         92
#define FOR_ITER                93
#define UNPACK_EX               94
#define STORE_ATTR              95
#define DELETE_ATTR             96
#define STORE_GLOBAL            97
#define DELETE_GLOBAL           98
#define LOAD_CONST              100
#define LOAD_NAME               101
#define BUILD_TUPLE             102
#define BUILD_LIST              103
#define BUILD_SET               104
#define BUILD_MAP               105
#define LOAD_ATTR               106
#define COMPARE_OP              107
```

```
#define IMPORT_NAME             108
#define IMPORT_FROM             109
#define JUMP_FORWARD            110
#define JUMP_IF_FALSE_OR_POP    111
#define JUMP_IF_TRUE_OR_POP     112
#define JUMP_ABSOLUTE           113
#define POP_JUMP_IF_FALSE       114
#define POP_JUMP_IF_TRUE        115
#define LOAD_GLOBAL             116
#define SETUP_FINALLY           122
#define LOAD_FAST               124
#define STORE_FAST              125
#define DELETE_FAST             126
#define RAISE_VARARGS           130
#define CALL_FUNCTION           131
#define MAKE_FUNCTION           132
#define BUILD_SLICE             133
#define LOAD_CLOSURE            135
#define LOAD_DEREF              136
#define STORE_DEREF             137
#define DELETE_DEREF            138
#define CALL_FUNCTION_KW        141
#define CALL_FUNCTION_EX        142
#define SETUP_WITH              143
#define EXTENDED_ARG            144
#define LIST_APPEND             145
#define SET_ADD                 146
#define MAP_ADD                 147
#define LOAD_CLASSDEREF         148
#define BUILD_LIST_UNPACK       149
#define BUILD_MAP_UNPACK        150
#define BUILD_MAP_UNPACK_WITH_CALL 151
#define BUILD_TUPLE_UNPACK      152
#define BUILD_SET_UNPACK        153
#define SETUP_ASYNC_WITH        154
#define FORMAT_VALUE            155
#define BUILD_CONST_KEY_MAP     156
#define BUILD_STRING            157
#define BUILD_TUPLE_UNPACK_WITH_CALL 158
#define LOAD_METHOD             160
#define CALL_METHOD             161
#define CALL_FINALLY            162
#define POP_FINALLY             163
```

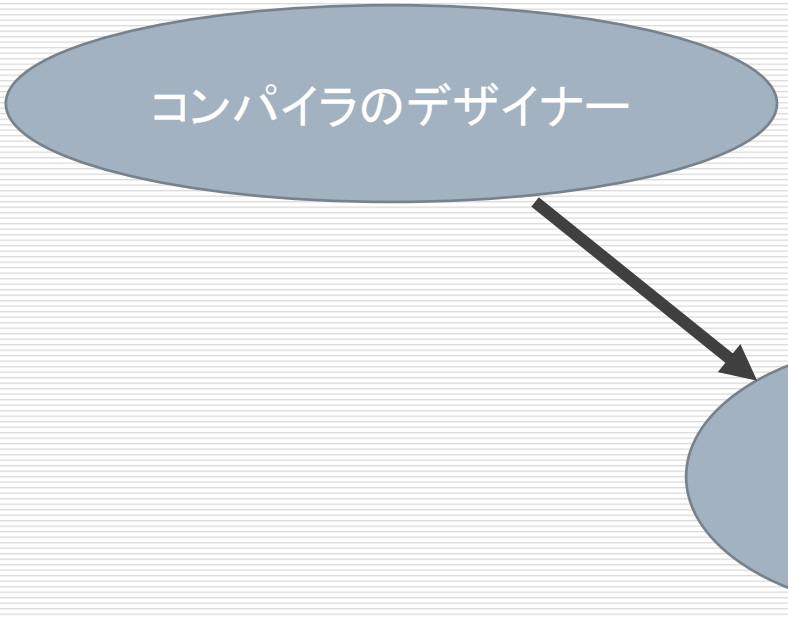
Compile

- VMを設計した上で、その上で動くコードを生成（完結したチーム等で全部やる場合）



Compile for Native Codes

- ターゲットとなるVM（or CPU）が与えられている場合



- o CPUをターゲット
- o すでにあるVMをターゲット
  （Java VMその他）
  e.g. Jython
  → calling convention を意識する
  （call instruction）

Interpreterの構成との類似性

- どちらもAST (Abstract Syntax Tree)からの変換を考える
- InterpreterのEval → Compilerのcompile
  - 類似性はCPythonのコードを解析することで「実証」します

- プログラムPの評価プログラム evalを考える。この部分評価PEを考えて
  PE(P) -> P* such that eval(P, D) = P*(D)
  - これ自身は、部分評価の標準的な式

部分評価

- 部分評価（Partial Evaluation）
  - たとえば、if (true) 1 else 2 → 1の評価は…
  - 計算する前にわかる
  - 一般に、プログラムPへの入力を実行前にわかるもの（S）と実行時にしかわからないもの(D)にわけ、出力を得ることを P: S × D → O
  - それから同じ出力を得る P*: D → O を生成する（P*をresidualと言うことがある）
- 部分評価は今もCSのメインストリームにあります

Futamura Projection

- PE(eval, P) → eval(P, -)*
  - プログラムPのコンパイル結果
- PE(PE, eval) → PE(eval, -)*
  - コンパイラ？
- もちろん、一般的にはこんな風にはうまくいかなくて、さまざまな仮定が必要（当時はLISPという幸運なプラットフォームがあって…）
  - evalの対象言語と実装言語が同一
  - 評価のための環境の定義
  - Frame, calling conventionの”→”が定義されている

準同型ではない

- しかし、同様の仕組みが…
  - 【e1+e2】=【e1】+【e2】(interpreter)
  - [e1+e2] = [e1] ;[e2]; BINARY_ADD; (compiler)

  - 【while e s】= while【e】【s】(interpreter)
  - [while e s] =
    L: [e]; jpz M; [s]; jmp L; M:
    (compiler)
- 部分要素の処理を合わせて全体を構築する

VAR (変数参照)に対応するコード生成

```
case VAR:
   {int varpos = staticvarsearch(cstack[top].val1);

    if (varpos == -1) {
      codepush2(OP_PUSH_GVAR, cstack[top].val1);
    } else {
      codepush2(OP_PUSH_LVAR, varpos);
    }
    break;
  }
```

対応するevalの部分

```
case VAR:  /* 2018 */
   {int varpos;
    if ((varpos = varsearch(cstack[top].val1)) == -1) {
          return vars[globalvarsearch(cstack[top].val1)].val;
    } else {
          return frames[varpos].val;
    }
  }
```

- staticvarsearch()は、静的にフレームを解析し、名前解決を行う
- globalvarsearch()は、実行時に用いる名前テーブル中の名前を参照する

- ブロックの情報としてさらに必要なものは、ブロックのstart, end
  - これが制御できると、loopの中でのbreakやredoが書ける

CALLに対応するコード

```
case FUNCALL:
  { int alen;

    alen = codemitargs(cstack[top].val2);
    codepush2(OP_PUSH_CONST, alen);
    codepush2(OP_CALL, cstack[top].val1);
    break;
  }
```

VMでの実行

```
switch (objectab[gv].typ) {
    case FUN:
      framestack[savedframetop++] = pc+1;
      framestack[savedframetop++] = stacktop-objectab[gv].paramlen+1;
      pc = objectab[gv].val;
      break;
```

CPython (compile.c)

```
static int compiler_nameop(struct compiler *, identifier,
expr_context_ty);

static PyCodeObject *compiler_mod(struct compiler *,
mod_ty);
static int compiler_visit_stmt(struct compiler *, stmt_ty);
static int compiler_visit_keyword(struct compiler *,
keyword_ty);
static int compiler_visit_expr(struct compiler *, expr_ty);
static int compiler_augassign(struct compiler *, stmt_ty);
static int compiler_annassign(struct compiler *, stmt_ty);
static int compiler_visit_slice(struct compiler *, slice_ty,
                                expr_context_ty);
```

CPythonでも構造は同じ

- CALLに対応するコード
  - compiler_call() → call_helper

スケルトンは

```
VISIT(c, e->v.Call.func);
 for (i = 0; i < nelts; i++) {
     …
   VISIT(c, expr, elt);
 }
 ADDOP_I(c, CALL_FUNCTION, n + nelts);
```

```
case Assign_kind:
        n = asdl_seq_LEN(s->v.Assign.targets);
        VISIT(c, expr, s->v.Assign.value);
        for (i = 0; i < n; i++) {
            if (i < n - 1)
                ADDOP(c, DUP_TOP);
            VISIT(c, expr,
                  (expr_ty)asdl_seq_GET(s-
>v.Assign.targets, i));
        }
```

- からのcompiler_visit_expr →… → compiler_nameop

- compiler_nameop()でsymbol tableの情報によりFASTとGLOBALを選択

Iteratorに対応するコードのコンパイル

```
compiler_for(struct compiler *c, stmt_ty
s)
{
    basicblock *start, *cleanup, *end;

    start = compiler_new_block(c);
    cleanup = compiler_new_block(c);
    end = compiler_new_block(c);
    if (start == NULL || end == NULL ||
cleanup == NULL)
        return 0;

    if (!compiler_push_fblock(c,
FOR_LOOP, start, end))
        return 0;
    VISIT(c, expr, s->v.For.iter);
    ADDOP(c, GET_ITER);
    compiler_use_next_block(c, start);
    ADDOP_JREL(c, FOR_ITER, cleanup);
    VISIT(c, expr, s->v.For.target);
    VISIT_SEQ(c, stmt, s->v.For.body);
    ADDOP_JABS(c, JUMP_ABSOLUTE,
start);
    compiler_use_next_block(c, cleanup);

    compiler_pop_fblock(c, FOR_LOOP,
start);

    VISIT_SEQ(c, stmt, s->v.For.orelse);
    compiler_use_next_block(c, end);
    return 1;
}
```

- Iterator(に付随するbodyのブロック内の情報)で、ループ終了、break, redoその他を管理
- Break_loopというmnemonicはなくて…

```
>>> def g(i):
...     for j in range(1,i):
...             if (j > 10):
...                     break;
...             r=r+j
...
>>> dis.dis(g)
 2           0 SETUP_LOOP              36 (to 38)
             2 LOAD_GLOBAL              0 (range)
             4 LOAD_CONST               1 (1)
             6 LOAD_FAST                0 (i)
             8 CALL_FUNCTION            2
            10 GET_ITER
        >>  12 FOR_ITER                22 (to 36)
            14 STORE_FAST               1 (j)

 3          16 LOAD_FAST                1 (j)
            18 LOAD_CONST               2 (10)
            20 COMPARE_OP               4 (>)
            22 POP_JUMP_IF_FALSE       26

 4          24 BREAK_LOOP

 5     >>   26 LOAD_FAST                2 (r)
            28 LOAD_FAST                1 (j)
            30 BINARY_ADD
            32 STORE_FAST               2 (r)
            34 JUMP_ABSOLUTE           12
        >>  36 POP_BLOCK
        >>  38 LOAD_CONST               0 (None)
            40 RETURN_VALUE
>>>
```

```
static int
compiler_break(struct compiler *c)
{
    for (int depth = c->u->u_nfblocks; depth--;) {
        struct fblockinfo *info = &c->u->u_fblock[depth];

        if (!compiler_unwind_fblock(c, info, 0))
            return 0;
        if (info->fb_type == WHILE_LOOP || info->fb_type ==
FOR_LOOP) {
            ADDOP_JABS(c, JUMP_ABSOLUTE, info->fb_exit);
            return 1;
        }
    }
    return compiler_error(c, "'break' outside loop");
}
```

ここで気づくのが…

- 前に、名前解決でASTをトラバースしたことをおぼえていますか？
- Compileも、ASTのトラバースをしてコードを生成します

Native Codeへのコンパイル

- Native Codeへのコンパイルも、その本質は変わりません
  - Assembly codeの出力ができるようにするにはプラス少しの文法の勉強で足りる

- Native Codeを相手にするときは、アセンブラに使うファイル形式（セグメントの指定方式）, load, linkについて、少し勉強する必要があります
  - 今回の後半、おおまかなところについてふれます

WrapUp

- プログラミング言語を定義したら、仕様にまとめてみる。
  - これができない人が…
- 概念の説明を最初につける
- プログラミング言語を定義するときに必要な（デリケートな）要素は大体以下の通り
  - プログラム全体の構造
  - 制御構造
  - データオブジェクト
  - 式
  - 関数（手続き）の扱い
  - 環境、変数のバインディング
- ＋　多言語とのインターフェイス、ライブラリ関数

課題7

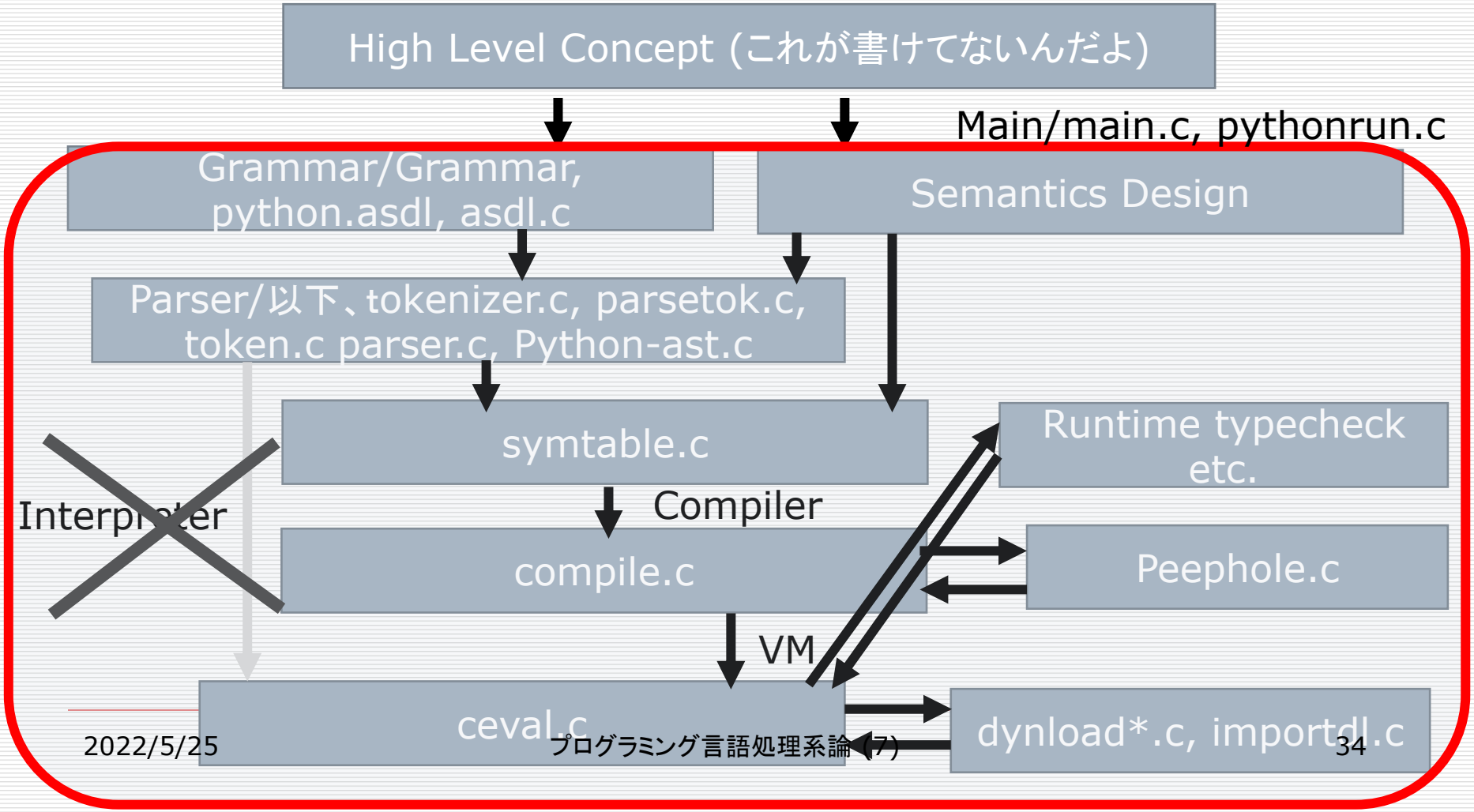
- 自分で言語を定義してみよ
  - 仕様書を書け(これが一番大事)
  - High Level Conceptは「新しい概念のない電卓である」でもかまわない
  - ただし、変数は扱えるようにすること
  - 関数コールをデザインすること
  - Lexical analysisに関係するコードは自分で書くこと
  - Parse したら、ASTは生成できるようにすること
  - どちらか
    - 評価系 (eval)を書いてください
    - 自分で適当にVMを設計して、それをターゲットにしたコンパイラを作ってもよい
  - スクラッチからやるのが面倒であるならば、教材その他を適当に拡張してかまいません。
    - 教材は、最低限のことはできる(最低限のことしかできない)ようにしています
    - 大きい(本格的な)言語は、読み込むのに一苦労ということはよくわかります

PyPy (少し余談)

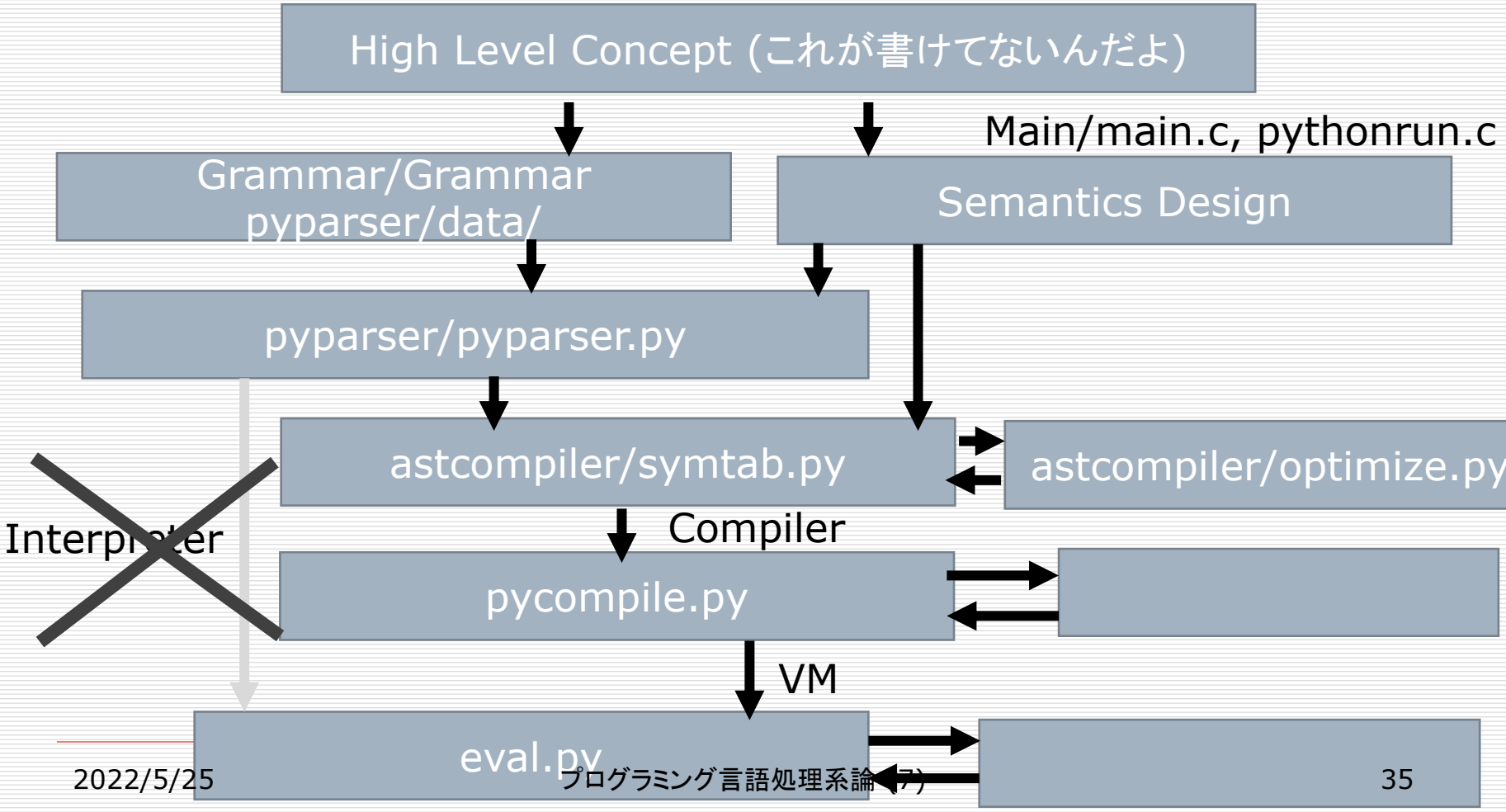
- Pythonの処理系が提供された(Cを実装言語とするCPython)
- では、CPythonの実装言語をPythonに変えることはできるか？
  - この場合、CPythonをbootstrapとして使うことになる
  - 今まで、CPythonのソース(C)で説明してきたことを全部Pythonで置き換えることはできるか？
  - 実装言語をPythonにできたら、いろいろなことが抽象的にかけて、すっきりするんだろうなぁ...
- 結果としてPyPy

- PyPyは、いろいろな目的を持った人たちが10年くらい前に一応の完成をみせたもので
  - 目的？
    - Self hosted Python
    - Faster VM (RPython)
  - 「完成」とは？
    - 論文を書く？
    - パッケージが世界を制覇する？

CPython


High Level Concept (これが書けてないんだよ)
Main/main.c, pythonrun.c
Grammar/Grammar, python.asdl, asdl.c
Semantics Design
Parser/以下、tokenizer.c, parsetok.c, token.c parser.c, Python-ast.c
symtable.c
Runtime typecheck etc.
Interpreter
Compiler
compile.c
Peephole.c
VM
ceval.c
dynload*.c, importdl.c

PyPy


High Level Concept (これが書けてないんだよ)
Main/main.c, pythonrun.c
Grammar/Grammar
pyparser/data/
Semantics Design
pyparser/pyparser.py
astcompiler/symtab.py
astcompiler/optimize.py
Interpreter
Compiler
pycompile.py
VM
eval.py

PyPy

- Grammarを確認する
  - parser部分
- ASTCOMPILEを確認する
  - シンプルです
- eval.py, pyframe.pyで実行エンジンを確認する

Virtual Machine

- VMの定義
  - ISA
- プログラム実行に必要なもの
  - Linuxでのプログラム実行
    - Link, Load
  - JVMでのプログラム実行
    - Link, Load

VMを用いた実行モデルの記述

- Virtual Machineを定義する
  - Operational Semanticsの忠実な表現
    - ISAを定義する(大別してstack machineとregister machine)
    - プログラムの意味とは、VM上にコンパイルされたコードのVM上の意味と定義する
    - ハードウェアシステムとは薄皮一枚で隔てられている
  - 「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
- VMで行うのは、計算機械の「実装」だけではない
  - オブジェクトの生成・GCに係るメモリ管理
  - 関数呼び出しに係るフレーム管理、並列性に係るスレッド管理

- VMのISAのデザインは、哲学の発露（少し大げさ）
  - コンパイラとの仕事の責任分界
    - 押し付けあうのか、引っ張り合うのか
- 対象プログラムを高速（または省電力）に実行するためのVMのISAの設計
  - 汎用CPUの設計とは違う
  - ASICではそれがcritical
    - TPU

Native vs. Virtual

- Nativeなマシンの命令セット設計はアーキテクチャ設計と密接に関係する

- VMの設計思想は
  - 一般的な実行の効率性
  - コードのコンパクト性（Nativeなマシンの差を吸収する）
  - 対象となる言語の重要なフィーチャーを効率よく実行する

- VMでは、プログラムの抽象度が設計に影響する
  - JVM
    - なぜ、invokeが複数種類あるのか
    - なぜ、基本型ごとの命令セットに分離されているのか
  - CPython
    - なぜ、ループ命令があるのか
      - GET_ITER, FOR_ITER
    - BINARY_MATRIX_MULTIPLYは、MLの人たちの想像力をどのくらい刺激したか

Java VM

- Java VMもPython VMもスタックマシンです
- JVMの当初の目的は、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）
  - コードの流通性の確保が目的の一つに入っていた
  - HTML5で廃止
- JavaのSemanticsはJava VMの上で定義することができる
- (速度を考えてJITが出現した)
  - 速度はこんな技術で稼ぐんじゃない
  - JITの話はRPythonで少しすることにしましょう

言語designとVMのInterface

- 「値」とは何か？
  - 基本型だけではない
  - Scalar＋Array（Fortran）
  - Tuple, list, (Python)
  - Object (OOP)
- 計算機械の上での実装（表現）の検討の必要性（処理系の検討には必須）
  - Referenceのメカニズムの検討
- オブジェクト管理、メモリ管理 (in VM) へつながる

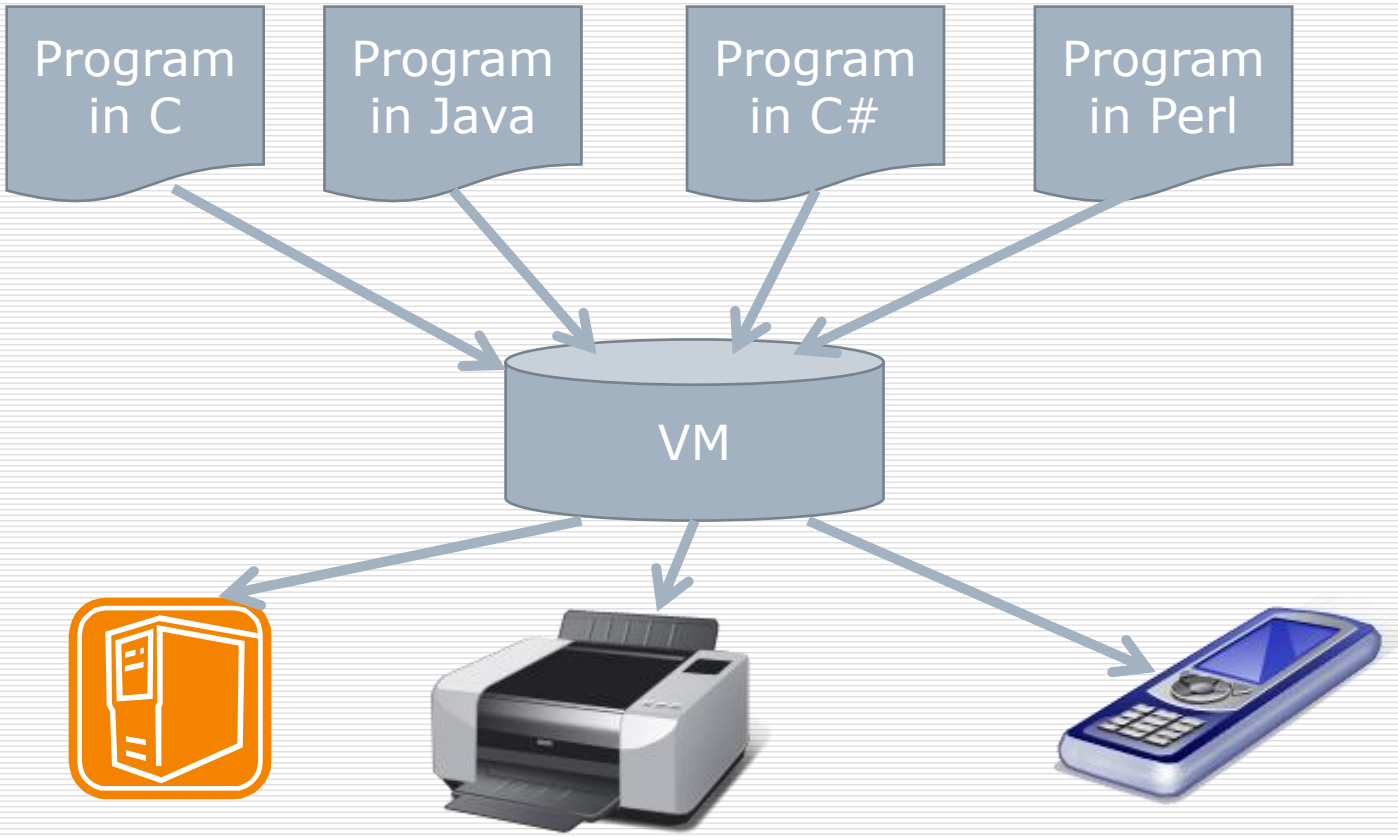
ちょっと見てみる

- Perl
  - stringが基本型のひとつとして入る
  - scalar ($xの形) + array (@xの形)
  - Objectは…
    - 厳密に言えばオブジェクト指向のようなプログラミングスタイルをpackageを導入することで提供
- Python
  - 基本型＋tuple+リスト
  - Objectは最初からデザインの中に入っている

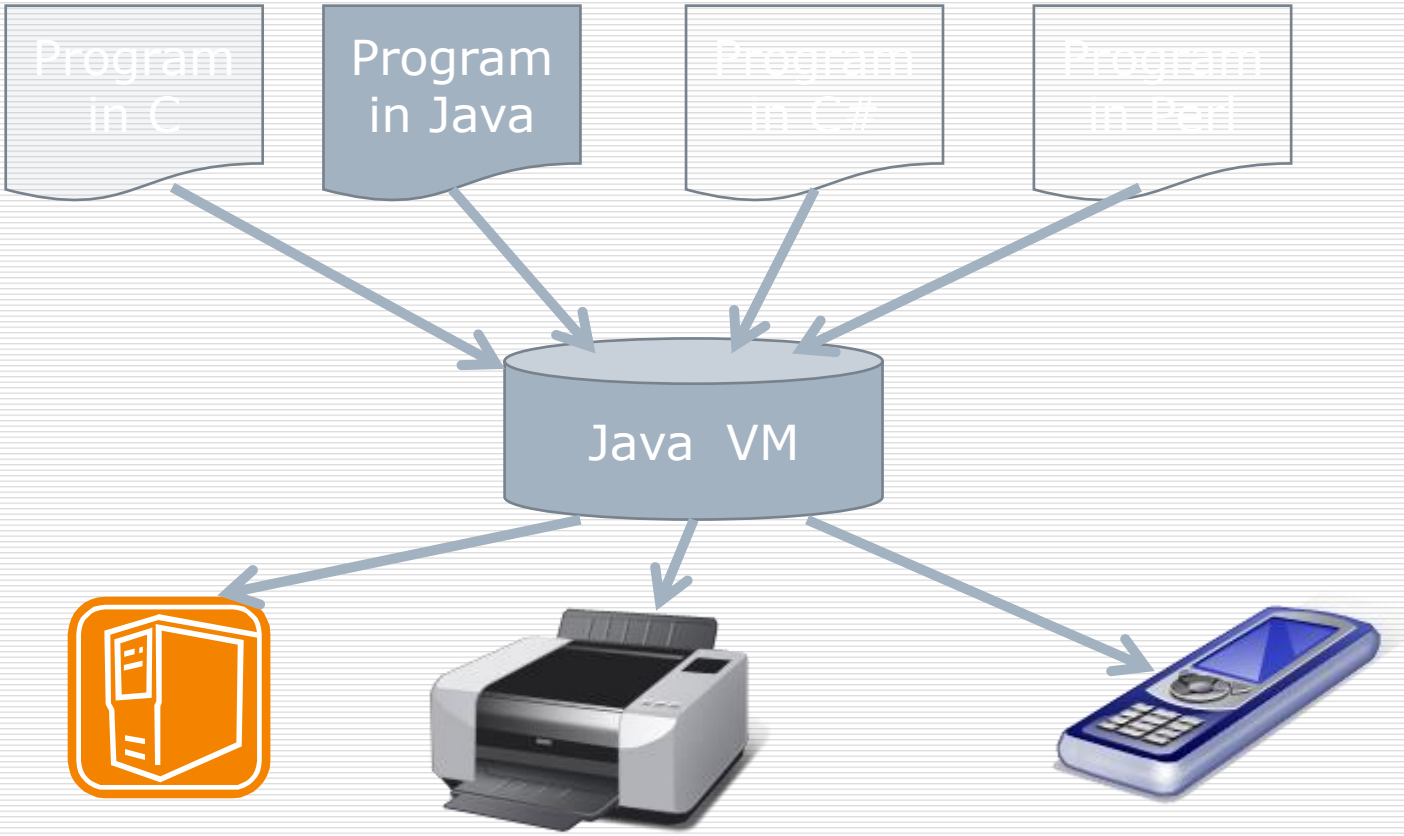
VMのご利益

- OS独立、CPU独立
  - .NET
  - Jython
  - …
- 昔、UNCOLという思想があって…

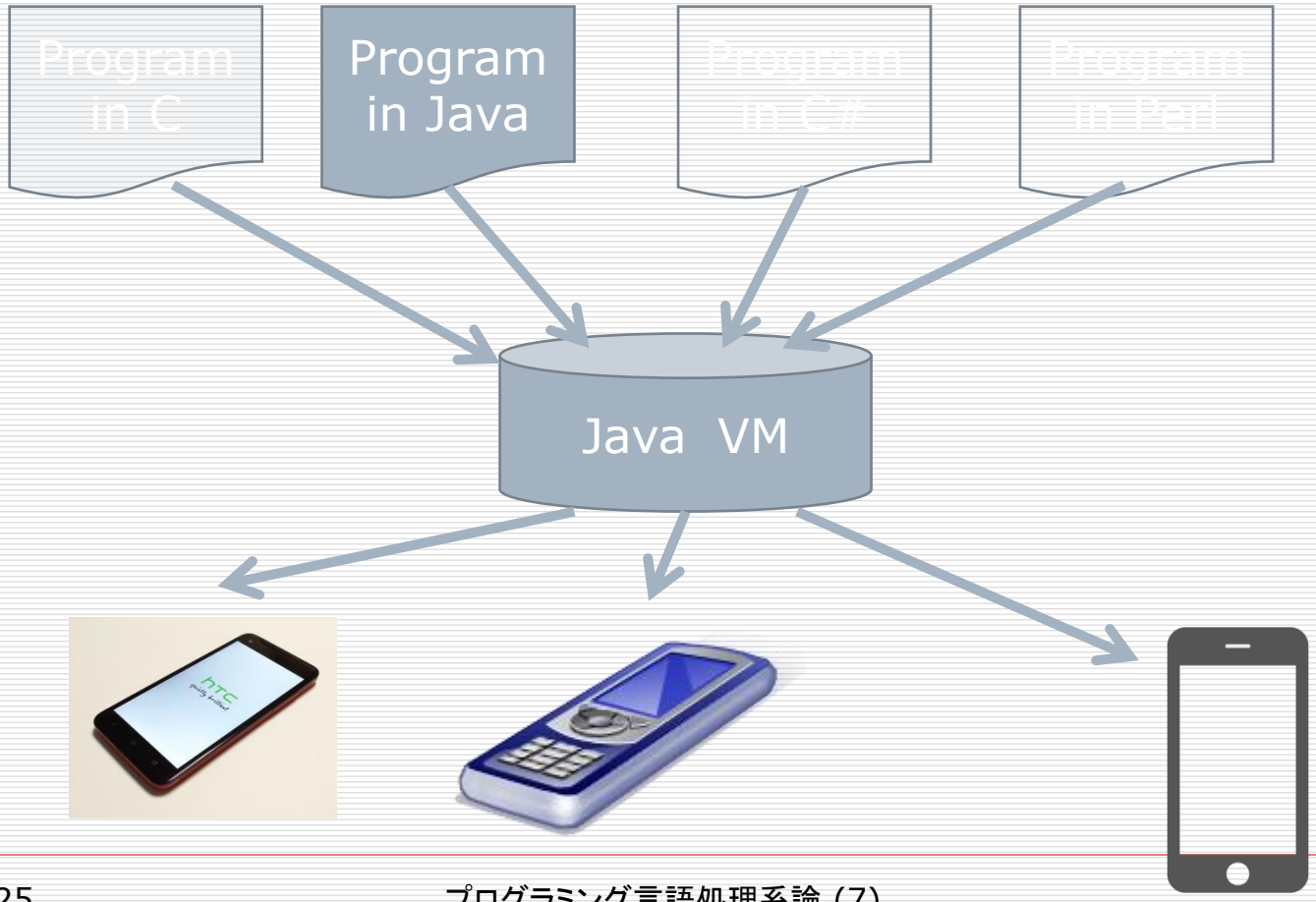
N×M → N+M



JavaのVMも含めた設計思想は実際そうだったわけだが…

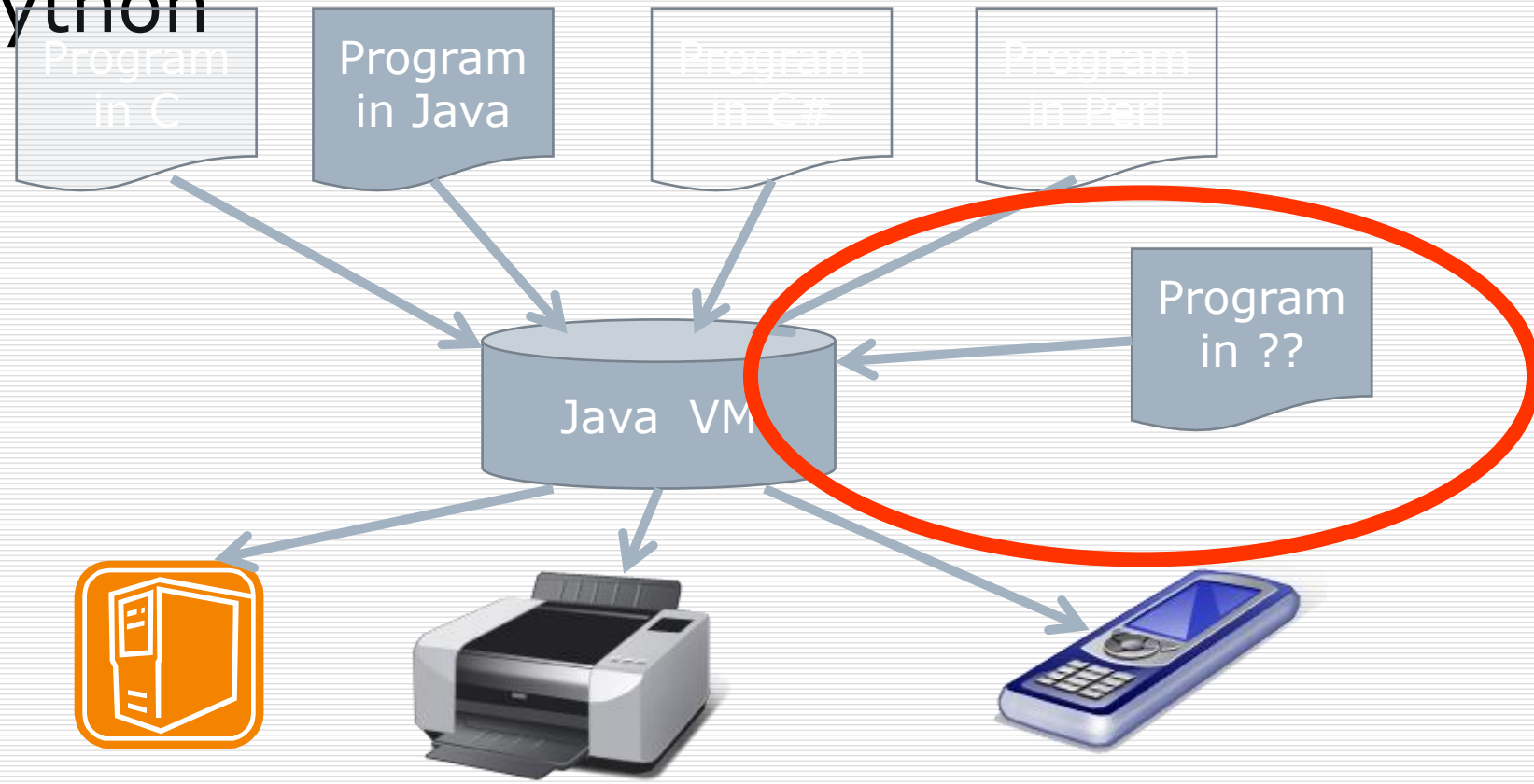


今はSmartphoneのアプリのために



では実際に2番目の言語があるか...

- Jython



- JythonやJRubyは、インタープリタをJavaで実装している
  - Javaクラスのインポート等で大きなご利益
  - でも、これはJVMで動くコードを出力するのではない（Jython）
  - JRubyは、RubyをJVM上で実装したもの
  - Jython: https://github.com/jythontools/jython
  - JRuby: http://www.jruby.org/

Nativeな実行環境

- 実行ファイルは定められた形式に沿って書かれていることを要求します。

- では、手始めにNativeな実行環境で要求される実行ファイルの形式を見てみましょう
  - Windows
  - Linux
- 実行のためには、「実行エンジン」が理解できる形式の「実行ファイル」を用意する必要がある
  - ELF, PE/COFF, Classファイル, …

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

一連の流れの詳細は？

% 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

一連の流れの詳細は？

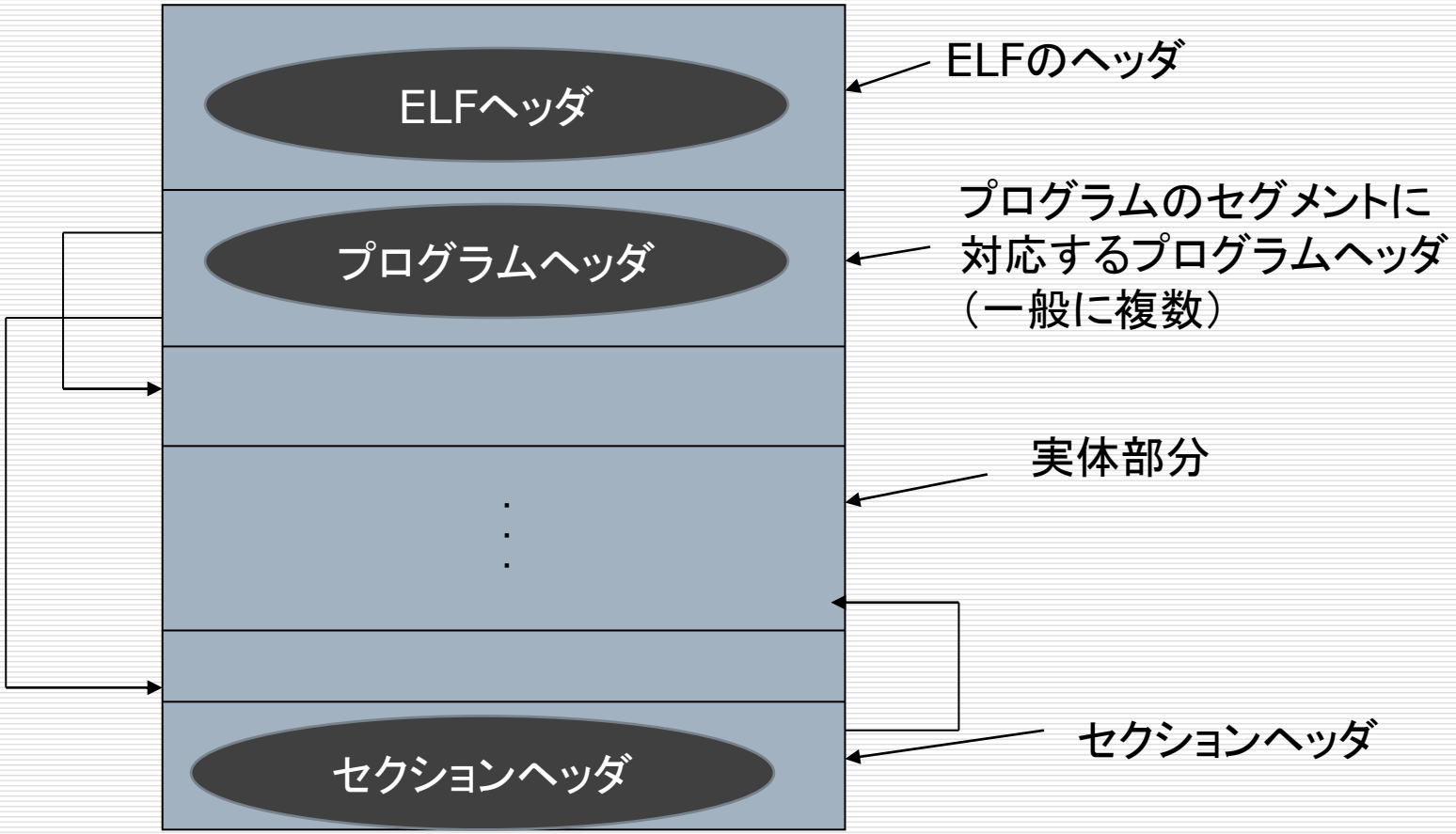
PE/COFF

- Windows上でプログラムを実行するとはどういうことか考えてみる。
- Windowsの実行ファイルの形式はPE/COFFと呼ばれる。
  - 複数のセクションの指定
  - 動的リンク情報
  - シンボルテーブル情報
  - . . .

ELF (Linux etc.)

- Linuxで、「実行ファイル」を実行するとはどういうことか考えてみる。
- Linuxの実行ファイルの形式はELF(Executable and Linking Format)と呼ばれる。
  - ここでのキーワードはExecute, Linking, …

ELFで書かれた実行ファイルのフォーマット



- ELFの種類としては
  - Relocatable File
  - Executable File
  - Shared object File
  - Core
- プログラムセグメントの種類としては
  - Loadable Segment
  - Dynamic Linking Information
  - Program Interpreter Information
  - Auxiliary Information

実行可能ファイルの実行の仕方

- ELFのヘッダを解析する
  - この形式は実行可能か？
  - エントリーポイントはどこか？
  - ロードしなければならないプログラムセグメントは？
- プログラムヘッダを解析する
  - インタープリタは何か？(ld-linux)

ld-linuxのやること

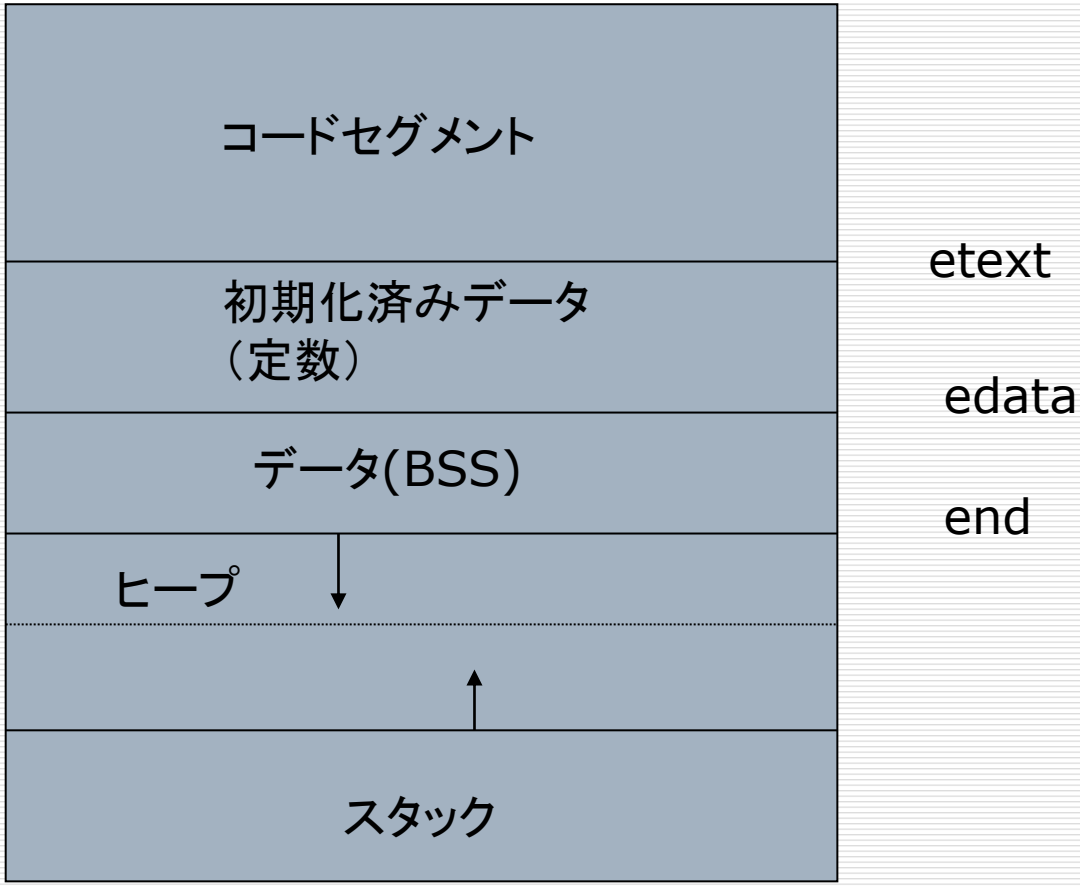
- セグメントのロード
- 動的リンクのためのシンボルテーブル解析
- 動的リンク
- 実行

- OSの重要な機能のひとつ
- 実行ファイルを作る側は、この形式に従って「実行ファイル」を作成する

- テキスト(コード)セグメント
- データセグメント

  は、実行ファイル内に格納されていなければならない
- シンボルテーブルは、リンクのときに使用するので、リンク前のファイルには必須(さて、リンクはいつするか？)

最終的な実行イメージ



実際の実行ファイルの構造

- LinuxならばreadelfでObserveすることができる
  - プログラムのセクション
  - データのセクション
  - 外部ライブラリのリンクのためのセクション（PLT）

＋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分になっている

ライブラリ関数のリンクとロード

- Link
  - Static
    - ルーチンのレファレンスは実行ファイルを作るときに全部resolve
  - Dynamic
    - 動的リンクのために、ルーチンのレファレンステーブルを作っておく（PLT）
    - コールされるときに、動的ライブラリにある実体との対応を取る

- Load
  - Static
  - Dynamic
    - Shared Libraryを動的(プログラム中)に指定して、実行中にロード
    - リンカは、ロードの際のName Resolutionを行う
      - dlopen(), dlsym()
- Interactiveなスクリプト言語では、動的なLoad機能の実装が求められる

- VMに、Dynamic Load機能を実装することは必須
  - ソースコードのloadではなく、コンパイルされたコードのload (当たり前…)
  - importの実装にあたって必要なこと
    - Loadするコード/データのためのスペース
    - コールの時に必要な、Name Resolution (Reference Table)の実装
  - 外部のライブラリのロードなら、OS提供の機能を利用することもできる

CPython

- import.c, importdl.c

- Dynamic linking:
  PyImport_LoadDynamicModuleWithSpec() →
  PyImport_FindSharedFuncptr() →
  dlopen(), dlsym()

Javaの実行環境

- Linuxと同じことがJVMに言えるだろうか？
  - 調べることは何か？
    - 実行対象としてのVM
      - VMのアーキテクチャ
      - VMの命令セット
    - VMで実行できるための「実行ファイル」形式
- JVMはきちんとしています

PythonのVM

- VMに関するまともなドキュメントがないことを前述の通り
- 動的ロード/リンクはimportでサポート
- Cとのインタフェースも記述あり
- Mnemonicを見て、VMの特徴を観察することはできる

Cとのインターフェース

- ctypesなんかはちょっとわきにおいといて

- https://docs.python.org/ja/3/extending/extending.html
  - 決まり事だし…
  - 提供されているということが大事

Javaのクラスファイルの実行

- クラスのロードと実行の定義は以下の https://docs.oracle.com/javase/specs/jvms/se18/jvms18.pdf
  - Chapter 5. Loading, Linking, and Initializing
- やることは
  - クラスのロード
  - リンク(名前解決を含む)
  - 初期化
  - Go

5 Loading, Linking, and Initializing

5.1 The Run-Time Constant Pool
5.2 Java Virtual Machine Startup
5.3 Creation and Loading
  5.3.1 Loading Using the Bootstrap Class Loader
  5.3.2 Loading Using a User-defined Class Loader
  5.3.3 Creating Array Classes
  5.3.4 Loading Constraints
  5.3.5 Deriving a Class from a class File Representation
  5.3.6 Modules and Layers
5.4 Linking
  5.4.1 Verification
  5.4.2 Preparation
  5.4.3 Resolution
    5.4.3.1 Class and Interface Resolution
    5.4.3.2 Field Resolution
    5.4.3.3 Method Resolution
    5.4.3.4 Interface Method Resolution
    5.4.3.5 Method Type and Method Handle Resolution
    5.4.3.6 Dynamically-Computed Constant and Call Site Resolution
  5.4.4 Access Control
  5.4.5 Method Overriding
  5.4.6 Method Selection
5.5 Initialization

「リンク」とは

- Verification
  - 命令がvalid/branchの行き先がvalid/命令が型チェックを通る
- Preparation
  - Staticフィールドの用意
- Resolution of Symbolic References
  - クラス内でクラス名、フィールド名、メソッド名でreferされている箇所を直接referする形に書き換える
- Initialization
  (やることはNativeなものと同じ)