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）

残業:型システム

- 定義の整理
- 型:
  - 式term e と型term Tに関して 関係 e:T
- 型推論: ei:Ti (i≧-1) からe:Tを導出する

$$\frac{e0:T0 \ \ldots \ en:Tn}{e:T}$$

- 型の意味:
  - （集合論的）e:Tに対して〚e〛∈ 〚T〛
  - (圏論的) e:Tに対して〚e 〛: 1→ 〚T〛
- Syntaxと意味の区別をしましょう
- この授業では、意味の話を展開しません
  - 〚T〛が集合(圏のオブジェクト)として成立するかどうかが議論になる…

- 準同型：e:Tとする。計算 e → e′ に対して e′:T （ 意味的には ⟦e⟧ ＝ ⟦e′⟧ ）
- e:TとなるTがなければ、 ⟦e⟧ は定められない

- 型チェック：e:Tが与えられたときに、それが型推論の結果として導出できるかを確認（チェック）すること

型チェックと言っても

- 型推論の構成と対になる
  - 以下のようなプログラムを書くわけにはいかない。
    a:number = 1:number
    c:number = a:number+2.0:number
- 型推論は、容易？
  - 型構成子に対応する分解をすればよいだけなら…
  - 型階層（subtype, supertype）の反映
  - 型パラメタの処理
  - TypeScriptには全部あります

TypeScriptの型システム

- 基本型
  "string" | "number" | "bigint" | "boolean" | "symbol" | "undefined" | "object" | "function“
- any, unknown, undefined, null
- literal
  - 式term(の一部)を取り込む
    var g: 7 = 3+4;

- union, intersection
  type TwoD = {x:number, y:number}
  type ThreeD = {x:number, y:number, z:number}
  type TwoOrThreeD = TwoD|ThreeD
  type TwoAndThreeD = TwoD & Three
- 典型例
  type nodetype = 'a' | 'ol' | 'ul'
  type strOrnum = string | number

- typeof, instanceof
  - これも、式termと型termの混合
  - 昔はこれ(を含むもの)をreflectionと言っていた
- 条件型、型ガード
  type ISString<T> = T extends string? true: false
  - これは、型termへの計算の導入

関数型

- JavaScript functionに対してcall signatureを対応させる

  function sum(a:number, b:number) { return a+b; } : (a:number, b:number) => number

- 引数の形状
  - Optional (?), Rest (…)

- Polymorphism as generics function

```
filter<T>(arr:T[],
          f:(item:T)=>boolean):T[]
{ let result = []
  for (let i=0; i < arr:length; i++) {
        …
  }
```

Class

- Classは、メンバーとメソッドの集合
- abstract class, implementsによる抽象化具体化
- extendsによる型階層

型階層

表 1: TypeScript の型階層

				any			
↓	↓	↓	↓	↓	↓	↓	↓
string	number	bigint	boolean	symbol	undefined	object	function
↓	↓			↓		↓	
string enum	number enum			unique symbol		array	
						↓	
						tuple	

表 2: クラスの階層と array, function 内の階層

S	S	`array<S>`	S1	S2	`S1 => S2`
↓	↓	↓	↑	↓	↓
T (such that T extends S)	T	`array<T>`	T1	T2	`T1 => T2`

表 3: literal の階層

string	number	···
↓	↓	↓
string literal	number literal	

- 型階層と型推論

$$\frac{e:T \quad T < T'}{e:T'}$$

- Refinement
  - Discriminated union 型に対し、条件文(if, switch)によって型の条件を追加していく方法

残業終わり

今日やること

- 関数コールの実現と関係するさまざまな話題
  - Frameのデザイン
  - Calling Convention
- Compiler
  - VMの定義
  - コード生成

環境と変数のバインディング

定義として

- 環境＝変数と値の結合の世界
- 特に、関数呼び出しにおける仮引数（parameter）と実引数（argument）の結合
- 変数が大域、局所とスコープが切られている場合変数の参照が何を指しているかを正しく表現することが必要

→　この部分はSemantic Analysisにおいて、名前解決をしました

```
def fac(n) {
   my(r)
   r = 1;
   while (n>0) {
      r=r*n;
      n=n-1;
   }
```

X X

```
r = 10; n = 3;
fac(5); r; n;
```

関数の出現

- 関数の出現につれて考えなければならない問題
  - コードセグメントの管理
  - フレームの管理（関数呼び出しごとに、環境設定をする必要がある）
    - ローカル変数、グローバル変数
    - スコープの管理
    - 引数の渡し方

変数のバインディング

- 関数コールについて考えるべきこと

  (a) 関数には引数をどう渡すのか？

  (b) 引数が渡されると、変数名とデータの対応がどのように変更されるか。また、関数からリターンするとき、どのような処理が必要か？

フレーム

- 関数実行の時に必要な局所的な情報を格納しておく場所を一般にフレーム (frame) といいます
- フレームを一つ作ると、環境が変化します
- 関数呼び出しが終了したら、環境を復帰させなければなりません
- Frameに含まれる情報には以下があります
  - Return address
  - 引数のバインディング情報（特に実引数と仮引数の対応関係）
  - 関数内のローカル変数のための領域

- 局所的な環境は、任意の場所に現れるというのが現代的なプログラミング言語の常識
  - ブロック
  - 関数
- 考え方はフレームと同じ
  - 局所的な環境を定義するときに、変数と値の結合の仕方を改めて定義する
  - 局所的な環境が「終了」したら、変数と値の結合を元にもどす

バインディングの余談：Call by **

- Call by Value
  - 値を実引数として渡す (C, Python)
  - ほとんどの現代的言語で実装される
  - JavaのreferenceやCのポインタをわたすのも実はCall by Value
- Call by Reference
  - 変数のアドレスを引数として渡す（Fortran, C++の一部機能）
- Call by Name
  - 変数が含まれる式を、呼び出しごとに評価する（現在ではごく少数）
- Call by Keyword
  - パラメタに「名前」が付いている (Fortran, Python)

- 実は、変数の評価についても同じことが言える
  - 特にスクリプト言語では、Referenceが実行時にしか決まらないことがある（by Nameの必要性）
  - 局所変数等、Referenceが静的にわかる場合はby Referenceを採用すべきである

関数コールについて

- 呼び出し側と呼び出される側がどのように引数(実引数と仮引数)の対応を取るかはデザイン上の問題の一つ
- この「約束」をcalling conventionという
  - HW, VMのサポートによる制約
  - 言語実装上の「取り決め」(自由にデザインできる)
- HW, VMで様々な機能がcallの実行時にサポートされている場合がある
  - Frameの自動構築
  - レジスタの自動rename
  - x86は、このサポート(制約)がほとんどない
- Code生成をする場合、このサポート(制約)を満たさなければならない

後述しますが…

- JVMの場合、Invoke***を実行すると
  - そのメソッド用のframeが構築される。
  - callerは、呼び出し側のオブジェクト（this）とメソッドの引数を指定しておく
  - calleeは、指定されたデータを0から順にアクセスできるように、frameの初期化を行う
  - Return addressの操作は（原則）implicitに行う

CPythonの場合

- Pythonの関数呼び出しは
  - CALL_FUNCTION
  - CALL_METHOD

  - JAVAの通常のコール、staticコールに対応

  - CALL_FUNCTIONは、スタックに関数アドレス、引数を積み、引数の数とともに呼び出す
    - 実際のコードをdisassembleしてみる
    - これに対応してコンパイラはコードを生成する

CPythonの場合

```
>>> def f(a,b,c):
...     return a+b+c
...
>>> import dis
>>> dis.dis(f)
  2           0 LOAD_FAST                0 (a)
              2 LOAD_FAST                1 (b)
              4 BINARY_ADD
              6 LOAD_FAST                2 (c)
              8 BINARY_ADD
             10 RETURN_VALUE
>>> def g(x):
...     return f(x,x+1,x+2)
...
>>> dis.dis(g)
  2           0 LOAD_GLOBAL              0 (f)
              2 LOAD_FAST                0 (x)
              4 LOAD_FAST                0 (x)
              6 LOAD_CONST               1 (1)
              8 BINARY_ADD
             10 LOAD_FAST                0 (x)
             12 LOAD_CONST               2 (2)
             14 BINARY_ADD
             16 CALL_FUNCTION            3
             18 RETURN_VALUE
>>>
```

- load_fast <n>
- load_global <n>

- 対応するDictionary＝フレームからn番目の値を「計算用のスタック」にロードする
- フレームをスタック上に作るかどうかはデザインの問題

HWの制約(or support)も存在する

- Native codeを出す場合は、HWのリソースにマップするのが普通
- EG. SPARCの場合call命令を実行すると
  - callerのレジスタ%o0-%o7を%i0-%i7にマップする
  - return時には、Return addressがcallerによって%i7に指定されていると考える
- EG. x86の場合call命令を実行すると
  - Stack Segmentに、return addressがpushされる
- Security視点でのアタックの場合、これらが悪用される場合があって…

Calling conventionのデザイン

- perlでは…

```
sub perr
{
    my ($a,$b) = @_;
…
```

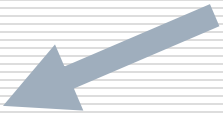
  - スタック上に$a, $bの場所を確保し、大域変数"_"の値（可変長）を「流し込む」
  - perlは、原則全部が「大域変数」（frameを作らない）発想だったが、そういうわけにはいかなくて…

Perlでの引数の渡し方の観察

```
sub factorial {

    my ($i, $j) = @_;

    my($r);

    $r = 1;

    while ($i>0) {
      $r = $r * $i;
      $i = $i-1;
    }
    return $r;
}

print factorial(7, 8);
```

呼ぶ方

```
$ perl -MO=Concise,-src fact2.pl
a  <@> leave[1 ref] vKP/REFC ->(end)
1     <0> enter ->2
# 17: print factorial(7, 8);
2     <;> nextstate(main 6 fact2.pl:17) v:{ ->3
9     <@> print vK ->a
3        <0> pushmark s ->4
8        <1> entersub[t3] lKS/TARG,1 ->9
-           <1> ex-list lK ->8
4              <0> pushmark s ->5
5              <$> const[IV 7] sM ->6
6              <$> const[IV 8] sM ->7
-              <1> ex-rv2cv sK ->-
7                 <#> gv[*factorial] s ->8
```



この実行により、
実行結果がLabel
引数として
参照可能になる

呼ばれる方

Return addressはここ

```
$ perl -MO=Concise,factorial,-src
      fact2.pl
main::factorial:
z  <1> leavesub[1 ref] K/REFC,1 -
      >(end)
-     <@> lineseq KP ->z
# 3:      my ($i, $j) = @_;
1        <;> nextstate(main 1
      fact2.pl:3) v ->2
8        <2> aassign[t5] vKS ->9
-          <1> ex-list lK ->5
2             <0> pushmark s ->3
4             <1> rv2av[t4] lK/1 ->5
3                <#> gv[*_] s ->4
```

引数は_で渡ってくる

```
-          <1> ex-list lKPRM*/128 ->8
5             <0> pushmark sRM*/128 ->6
6             <0> padsv[$i:1,5]
      lRM*/LVINTRO ->7
7             <0> padsv[$j:1,5]
      lRM*/LVINTRO ->8
# 5:      my($r);
9        <;> nextstate(main 2 fact2.pl:5) v -
      >a
a        <0> padsv[$r:2,5] vPM/LVINTRO ->b
# 7:      $r = 1;
b        <;> nextstate(main 3 fact2.pl:7) v:{
      ->c
e        <2> sassign vKS/2 ->f
c           <$> const[IV 1] s ->d
d           <0> padsv[$r:2,5] sRM* ->e
```

リストとして
frameを構

名前のresolution
済んでいる

関数呼び出しをデザインする

- Perlにおいても、いろいろな情報が貯められている（ASTの処理）
  - 引数情報（実引数@_の取り込み）
  - ローカル変数のリスト（pad）
  - リターンアドレス（end）
  - 戻り値()

- data segment vs. stack segment
- 一般的に、frameは関数コールごとに作成されるので、stackとして実現するとよいかもしれない
  - 並列実行等、関数実行のオプションが増えると、単一のstack segmentでframeを実現するのは時代遅れかもしれない
  - 関数コールごとに概念上独立のframe構成を許すJavaのチームは、頭が良いんでしょうね

- frameを設計する
  - 環境ローカルにどのような情報を格納するか
    - 局所変数の場所の確保

```
t_vardat frames[128];
int frametop;
int savedframetop;

int framestack[EVALSTACKLEN];
```

- 環境が新たに作られる
  - {…}の始まり

```
 case LENV:
{ int r;
  lvarstack[lvarp++] = frametop;
 r = eval(cstack[top].val1);
  frametop = lvarstack[--lvarp];
  return r;
}
```

関数呼び出しでは

- 関数コールでは、加えて、frameに、引数との対応を格納する＋return address

```
funprolog(v->param, cstack[top].val2,
v->paramlen);
```

frameを作って、parameter名で
引数の値を参照可能にする

```
 r = eval(v->val);
calledtop = savedcalledtop;
funepilog();
```

frameの破壊（stackと同じく、topのpoint
を巻き戻す）

CPython

- CpythonのCALL_FUNCTIONの実装も本質的にはこれ

- 選択肢:
  - 実装言語(！＝実装対象言語)のCALLを使う
  - VMのフレーム(または見えないところ)にreturn address情報を格納する
  - HWのreturn命令を使う(フレームにreturn address情報を格納する)

CPythonでの関数コールの実現

- 観察すべきは以下
  - Python/{compiler.c, ceval.c}
  - Objects/call.c

- 呼び出し側でやられていること:
  - 呼び出す側は、スタックに引数を積んで、引数の数とともに関数処理のルーチンを呼び出す
  - 呼び出しが終了したら、積まれた引数をキャンセルする

- 関数処理のルーチンでやられていること
  - 引数の並びを受け入れる領域を確保する
    - _PyStack_UnpackDict → バインディング
    - 関数コールは、この領域（のポインタ）をフレームとして用いる（Load_FAST）
    - 結果が返されたら、この領域をFreeする（_PyStack_UnpackDict_Free）
  - 一般のフレームの処理はPyFrame_*

Perlの名前解決

- frame内の変数か？大域変数か？

- 参照の仕方が異なることが通常
  - 関数ボディ内の変数参照をあらかじめ解析しておく。Perlではgvとpadの二つのアクセス方法を提供する
  - Perlでは、実行前に解析して解決しておく(gv or pad)
  - ODCIでは実行時に、変数参照を解析する。

Native Codeの場合：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 (9.4.0)
- 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VMのところで触れる

Calling Convention

- 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」
- もちろん、その前のHWの制約を理解することが必要
- スタックに積むのはx86で一般的だった
- 現在の、一般的なアーキテクチャでは、レジスタの一定の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
  - X86-64

Calling Conventionを調べるには

- コンパイラの内部ドキュメントをみましょう
  - 他のコンパイラとの連携するためにはCalling Conventionは公開情報でなければなりません
  - なければ探しましょう。
  - cdecl　かregcall
- 間違っても最初からアセンブリコードを解析してはいけません。
- で、GCCだ
  - あなたはWikipediaを信用するか？（結果的に正しくても）

Intelの定めるCalling Convention

- まずはnative環境から見る
- cdecl（古いアーキテクチャ。スタックの利用。最近は観察することは普通では不可能）
  - 引数はスタックに積む
  - EAX, ECX, EDXはcaller save
  - そのほか(ESI, EDI)はcallee save
  - Return value はEAXに返る
  - FloatならST0に返る
  - スタックをクリーンアップするのはcaller
- Stack machineの素朴な実装
  - Stack machineは、VMの設計において有力なアーキテクチャです
  - 理由はいろいろある（すぐ後で言及する）

x86-64の場合

- SysV AMD64 ABIがcdeclに加えて定められている
  - Caller save: RAX, RCX, RDX, RDI, RSI, R8, R9 (引数を渡すのに使われる)
  - Callee save: R10 - R15, RBX, RBP, RSP
  - 引数はRDX, RCX, RDI, RSI, R8, R9 (整数、ポインタ), XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6, XMM7 (floating)
  - 返値はRAX

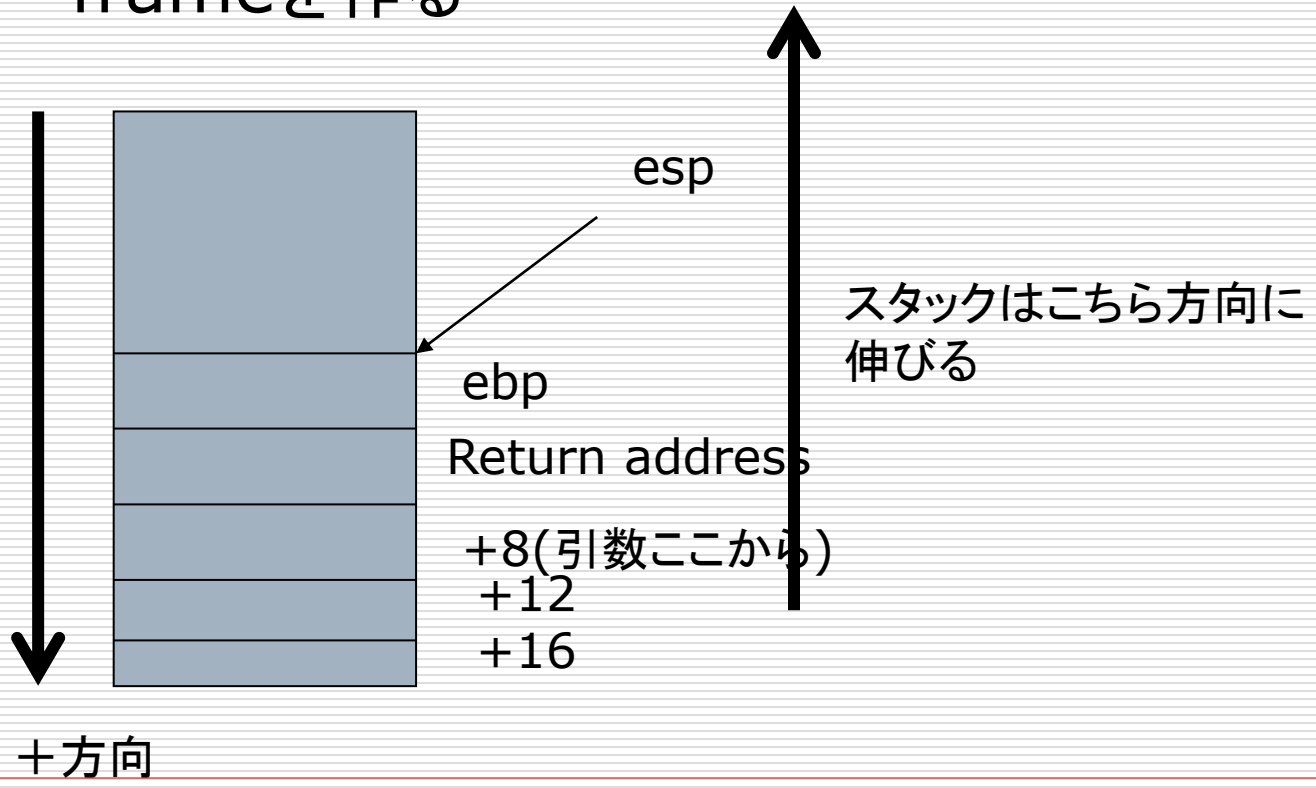
簡単なプログラムで実験(cdecl x86: 考古学)

```
Int a(int x0, int x1, int x2)
{

  return x0+x1*2+x2*3;
}

Int b()
{
  return a(1,2,3);
}
```

$ cc –S –m32 a.c

GCCの旧calling convention (x86)

- Stack segmentに frameを作る



考古学は時に役立つことがあって

- Stack machineを設計するときに、この方法は一般的
- Stack machineは、VMの標準的なアーキテクチャ
  - Registerについて余分なケアをする必要がない
  - コードサイズが小さくなる
  - ポータビリティ

- （課題6）
  手近にあるコンパイラ on HW CPU 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

X86-64ではどうか

コンパイラのオプションは効いているか？

```
extern int gv;
a(int x0, int x1, int x2)
{

  return x0+x1*2+x2*3;
}

b()
{
  return gv+a(1,2,3);
}
```

cc –O0 -S

```
 .file   "x.c"
        .text
        .globl  a
        .type   a, @function
a:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    %edx, -12(%rbp)
        movl    -8(%rbp), %eax
        leal    (%rax,%rax), %edx
        movl    -4(%rbp), %eax
        leal    (%rdx,%rax), %ecx
        movl    -12(%rbp), %edx
        movl    %edx, %eax
        addl    %eax, %eax
        addl    %edx, %eax
        addl    %ecx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   a, .-a
        .globl  b
        .type   b, @function
```

```
b:
.LFB1:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    a
        movl    %eax, %edx
        movl    gv(%rip), %eax
        addl    %edx, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   b, .-b
```

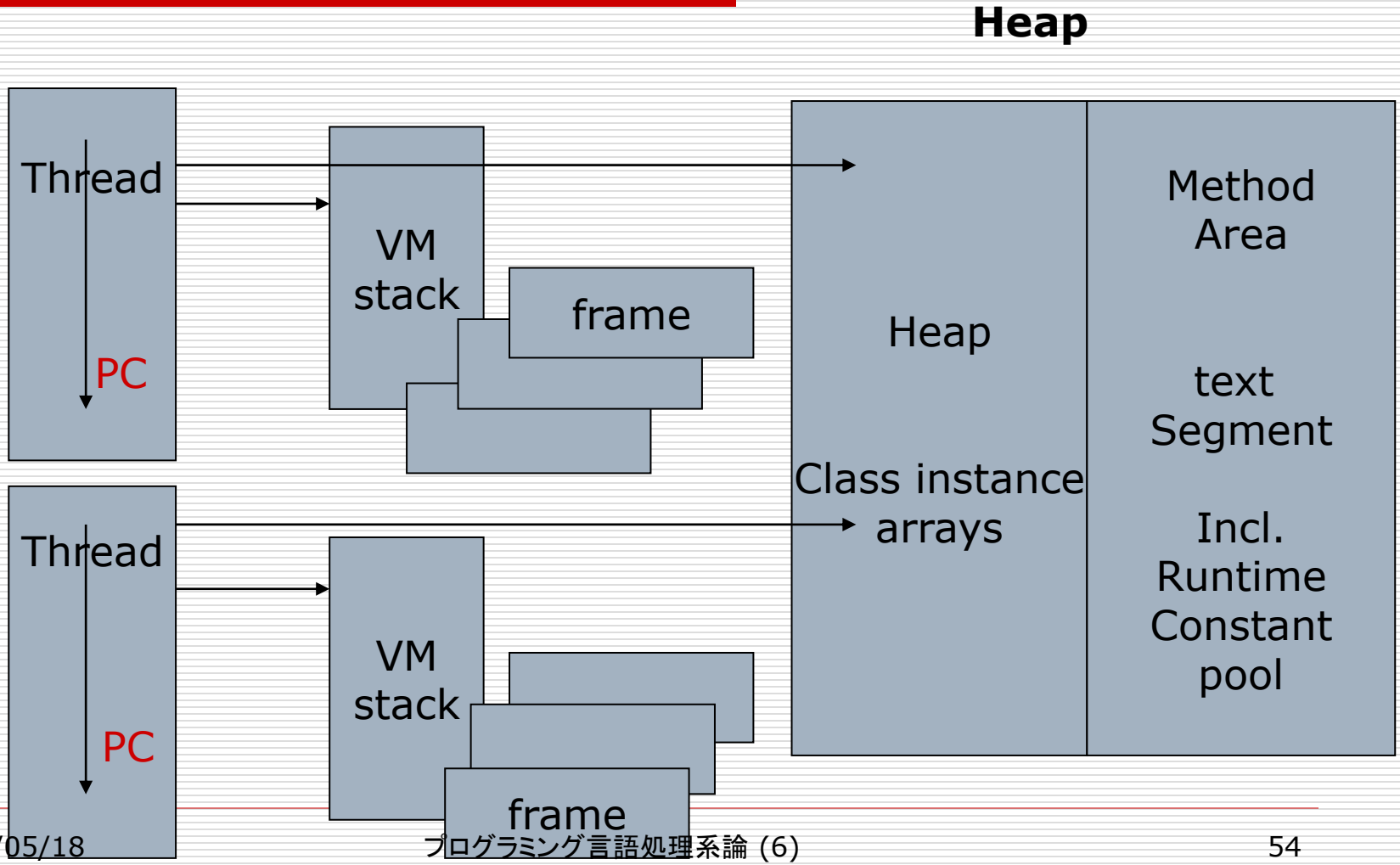
cc –O3 -S

```
 .text
        .p2align 4,,15
        .globl  a
        .type   a, @function
a:
.LFB0:
        .cfi_startproc
        leal    (%rdi,%rsi,2), %eax
        leal    (%rdx,%rdx,2), %edx
        addl    %edx, %eax
        ret
        .cfi_endproc
.LFE0:
        .size   a, .-a
        .p2align 4,,15
        .globl  b
 .type   b, @function
b:
.LFB1:
        .cfi_startproc
        movl    gv(%rip), %eax
        addl    $14, %eax
        ret
        .cfi_endproc
.LFE1:
        .size   b, .-b
        .ident  "GCC:
(Ubuntu7.3.0-16ubuntu3)
7.3.0"  .section        .note.GNU-
stack,"",@progbits
```

Javaの実行環境

- Cと同じことをJavaで調べられるだろうか？（nativeでもVMでも基本は同じ）
- Java on JVMのCalling Conventionは…
- JNI (Java Native Interface)が決められてはいます

JAVA VM

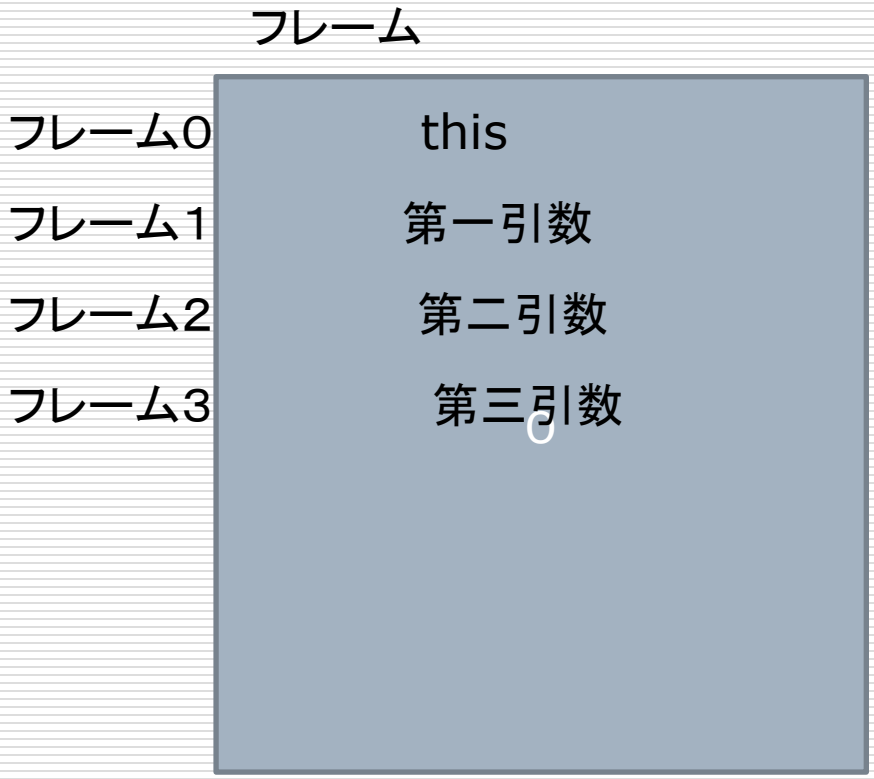

Heap
Thread
PC
VM
stack
frame
Thread
PC
VM
stack
frame
Heap
Class instance
arrays
Method
Area
text
Segment
Incl.
Runtime
Constant
pool

フレームに格納されるデータ

- リターンアドレス (*returnで使用)
- local variable (*load/*storeで使用)
- Exception Table

コンパイルしてみる

```
class x {
    static int content;

    x(int y) {
        content = y;
    }

    public int get_content() {
        return content;
    }

    public void addxx(x x1) {
        int y;

        y = x1.get_content();
        content += y;
    }
}
```



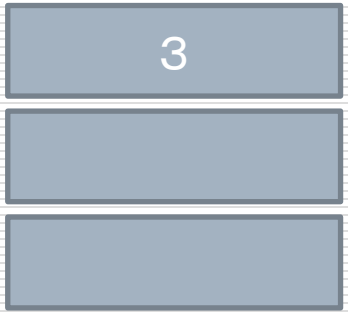
iload_n, aload_n, istore_n, aload_n等の命令が
フレーム内のデータにアクセスするために用意されている

コンパイル結果

```
class x {
  static int content;

  x(int);
    Code:
       0: aload_0
       1: invokespecial #1
      // Method
      java/lang/Object."<init>":()V
       4: iload_1
       5: putstatic     #2
       8: return

  public int get_content();
    Code:
       0: getstatic     #2
       3: ireturn

  public void addxx(x);
    Code:
       0: aload_1
       1: invokevirtual #3
      // Method get_content:()I
       4: istore_2
       5: getstatic     #2
      // Field content:I
       8: iload_2
       9: iadd
      10: putstatic     #2
      // Field content:I
      13: return
}
```

Argumentの扱い

Invoke前
メソッドAの
フレーム



Invoke完了後の
メソッドAの
フレーム

Invokeされた時の
AとBのフレーム

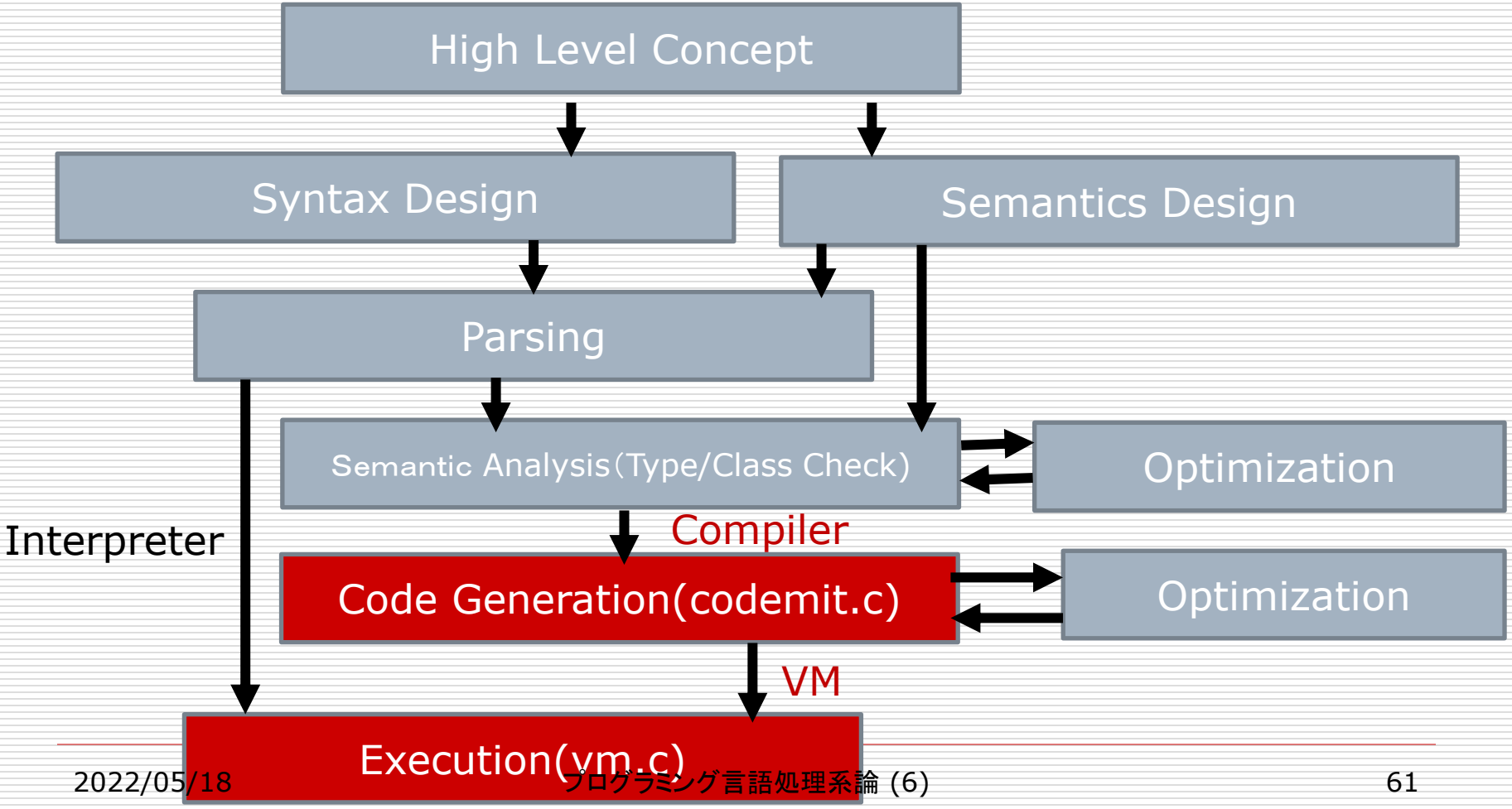
Pythonでも同様な観察ができる

```
>>> def myplus(i, j):
...     return i+j;
...
>>> dis.dis(myplus)
  2           0 LOAD_FAST     0 (i)
              3 LOAD_FAST     1 (j)
              6 BINARY_ADD
              7 RETURN_VALUE
>>> def myplus2(i,j,k):
...     return i+k+myplus(i,j);
...
```

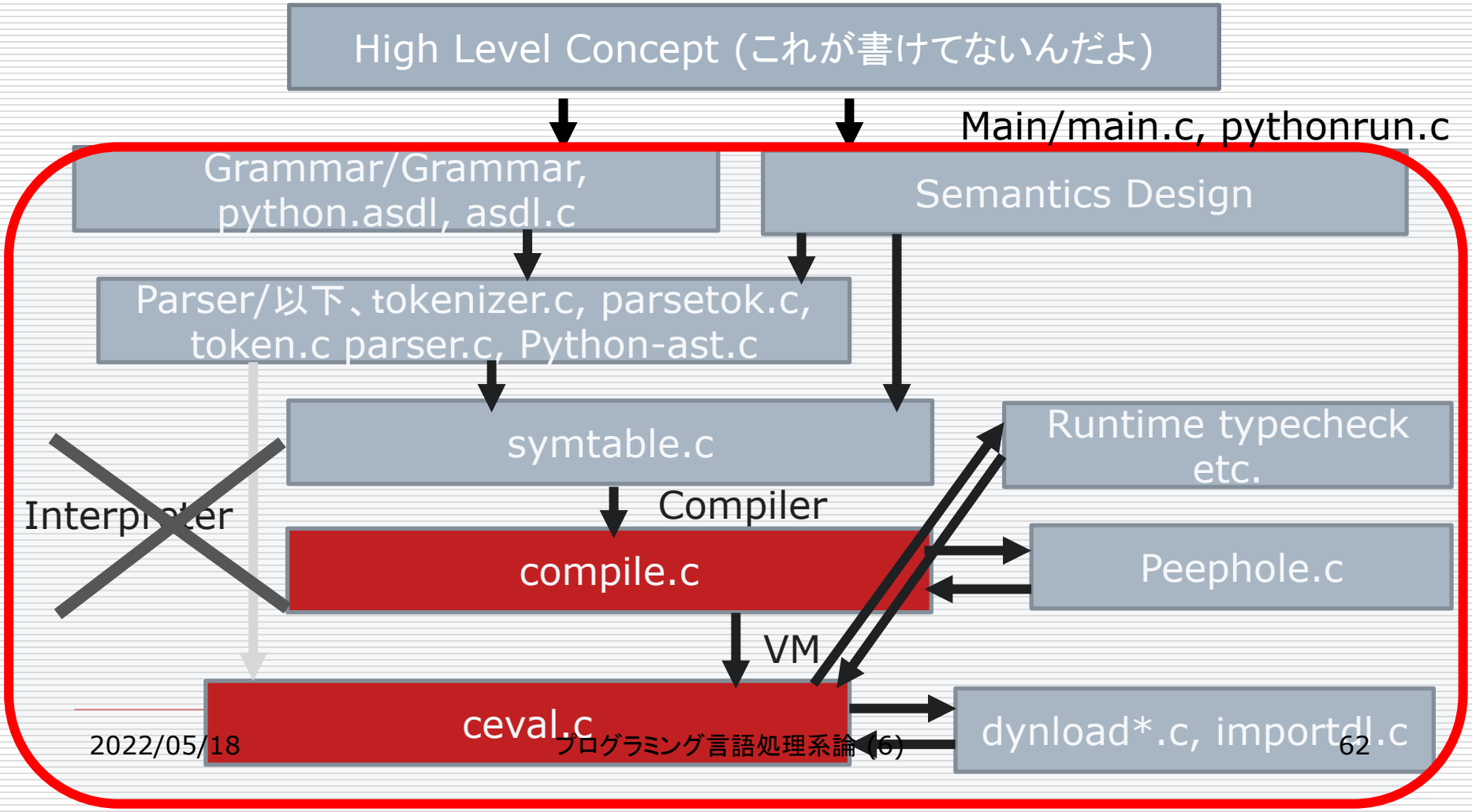
```
>>> dis.dis(myplus2)
  2    0 LOAD_FAST     0 (i)
       3 LOAD_FAST     2 (k)
       6 BINARY_ADD
       7 LOAD_GLOBAL          0 (myplus)
      10 LOAD_FAST     0 (i)
      13 LOAD_FAST     1 (j)
      16 CALL_FUNCTION    2
      19 BINARY_ADD
      20 RETURN_VALUE
```

- （課題6′）
  手近にあるコンパイラ on a VM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・VM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
- 注意：CPythonのソースコードを解析して、フレームがどのように構築されるかまで明らかにする必要はない（すればもっと良い）

文法要素以降の作業(VM構築)



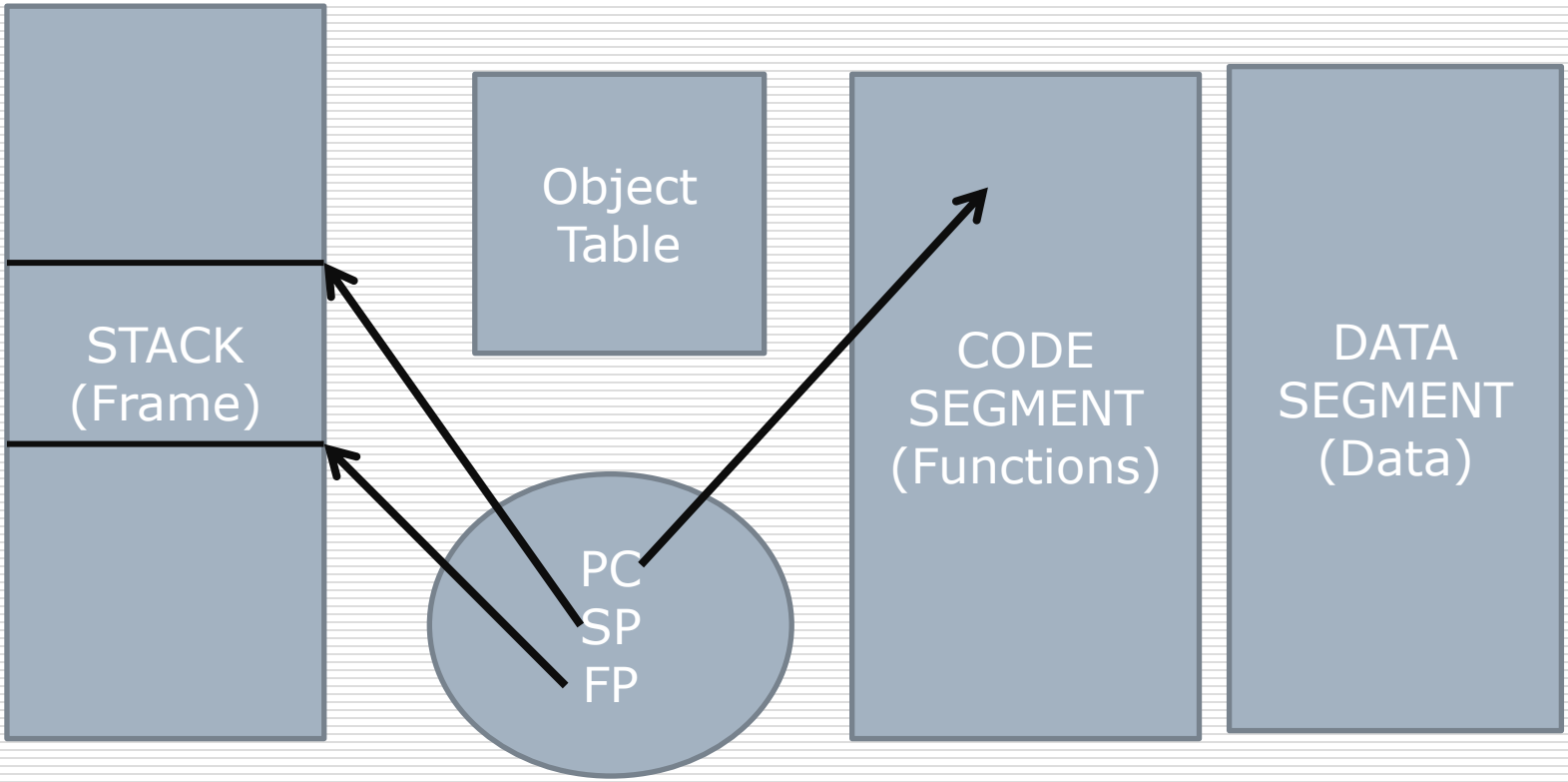
CPython


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

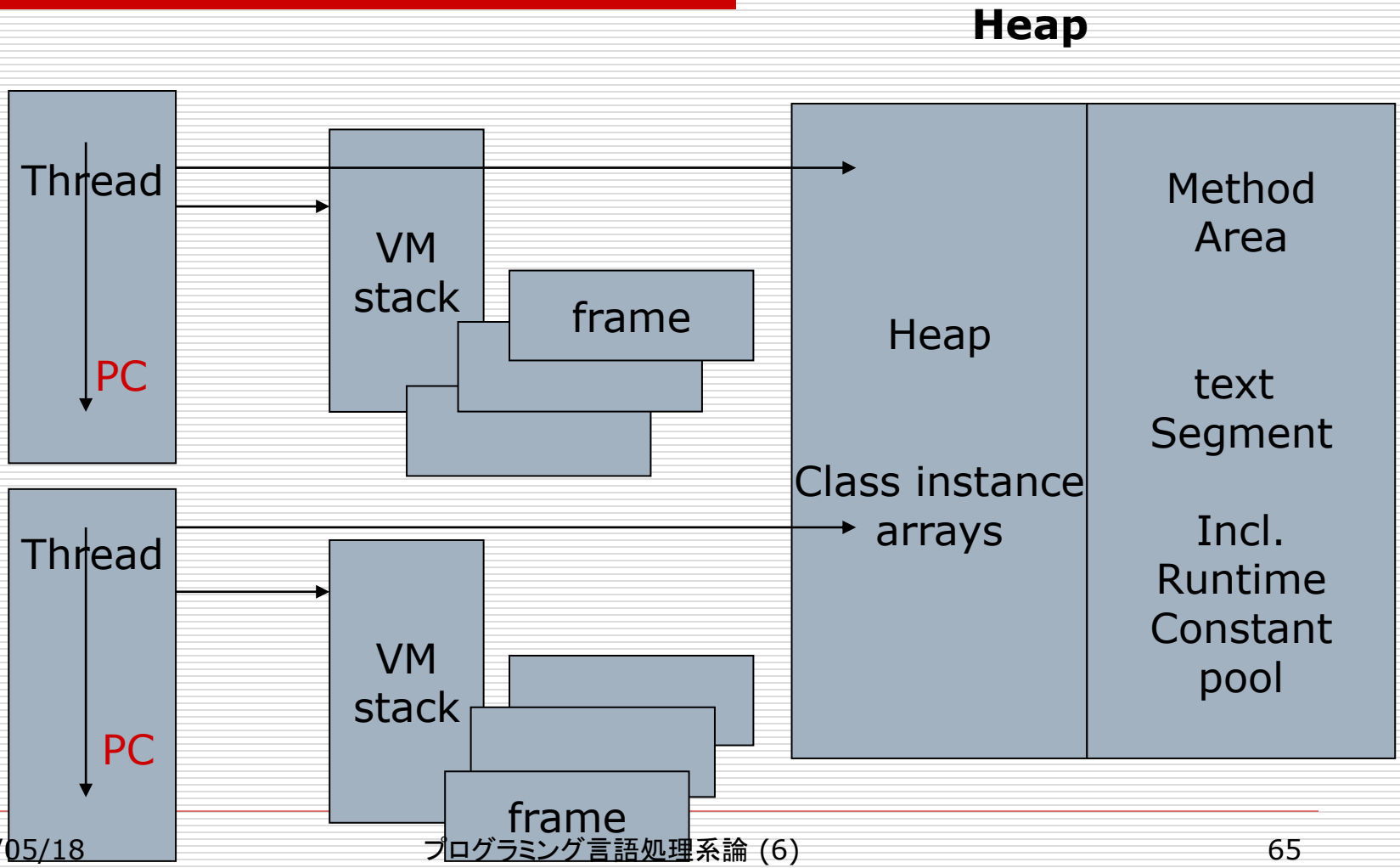
stack machineの定義：ターゲットの特定

- 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と相性がよい）

VM (super simple)



JAVA VMを思い出す



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
```

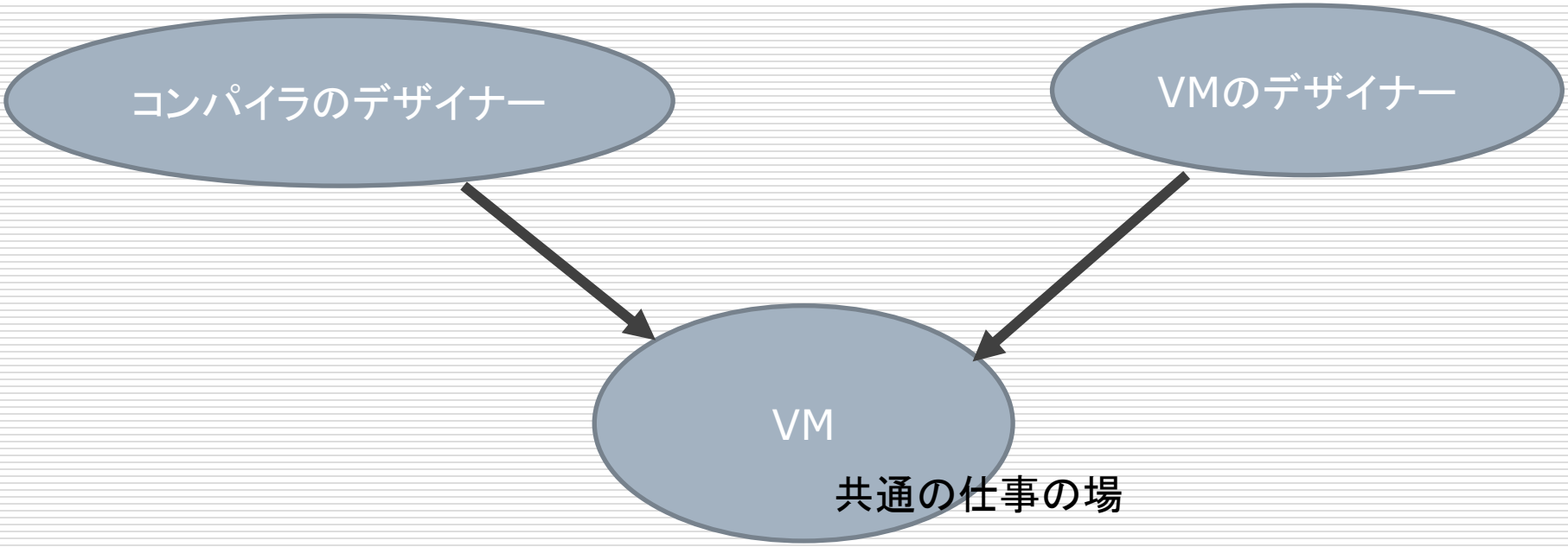
```
#define STORE_SUBSCR            60
#define DELETE_SUBSCR           61
#define BINARY_LSHIFT           62
#define BINARY_RSHIFT           63
#define BINARY_AND              64
#define BINARY_XOR              65
#define BINARY_OR               66
#define INPLACE_POWER           67
#define GET_ITER                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



Interpreterの構成との類似性

- どちらもAST (Abstract Syntax Tree)からの返還を考える
- Interpreterのeval → Compilerのcompile
  - この処理の類似性は、次回、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