プログラミング言語処理系論 (10)
Design and Implementation of Programming Language Processors

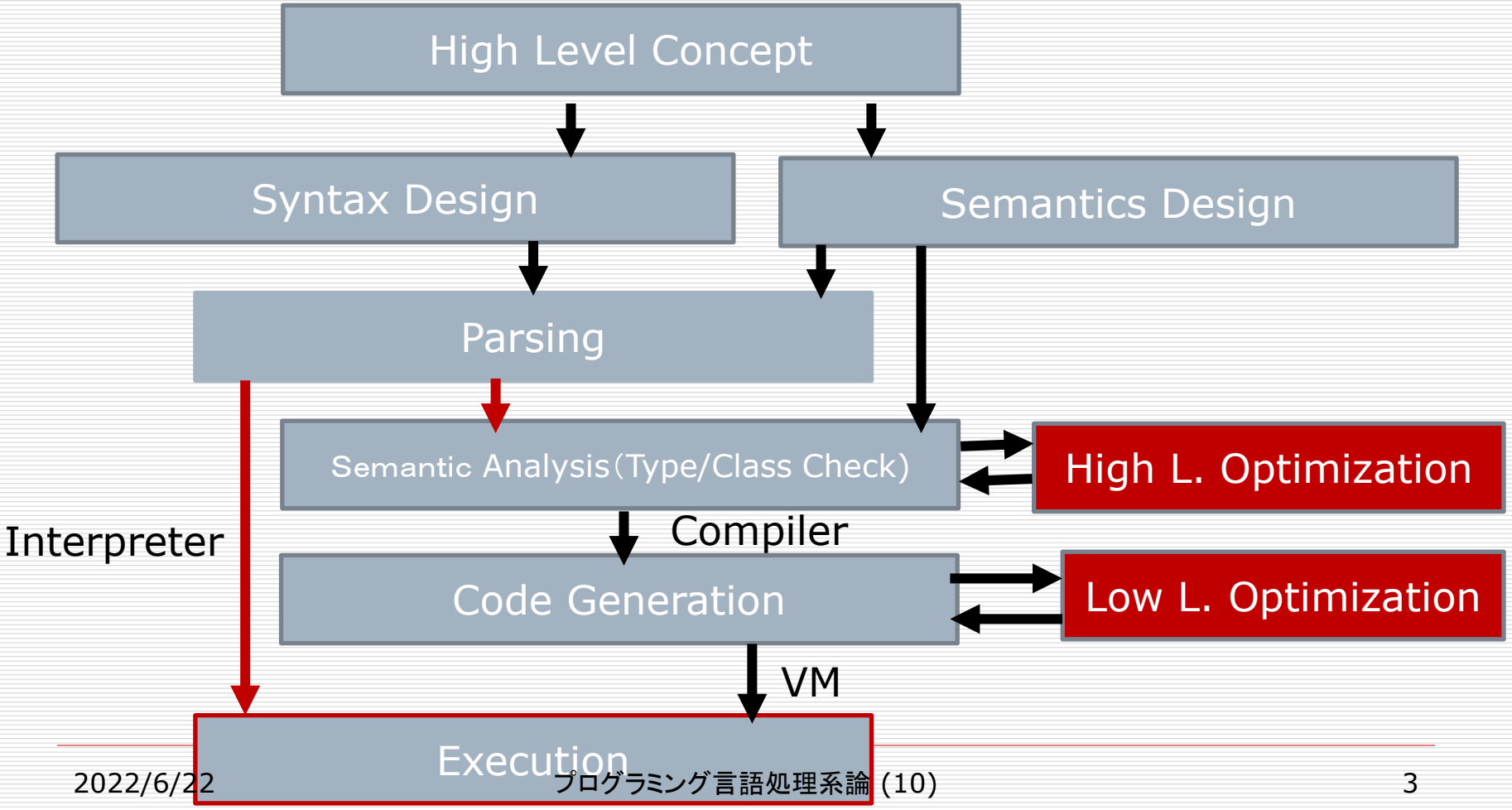
佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

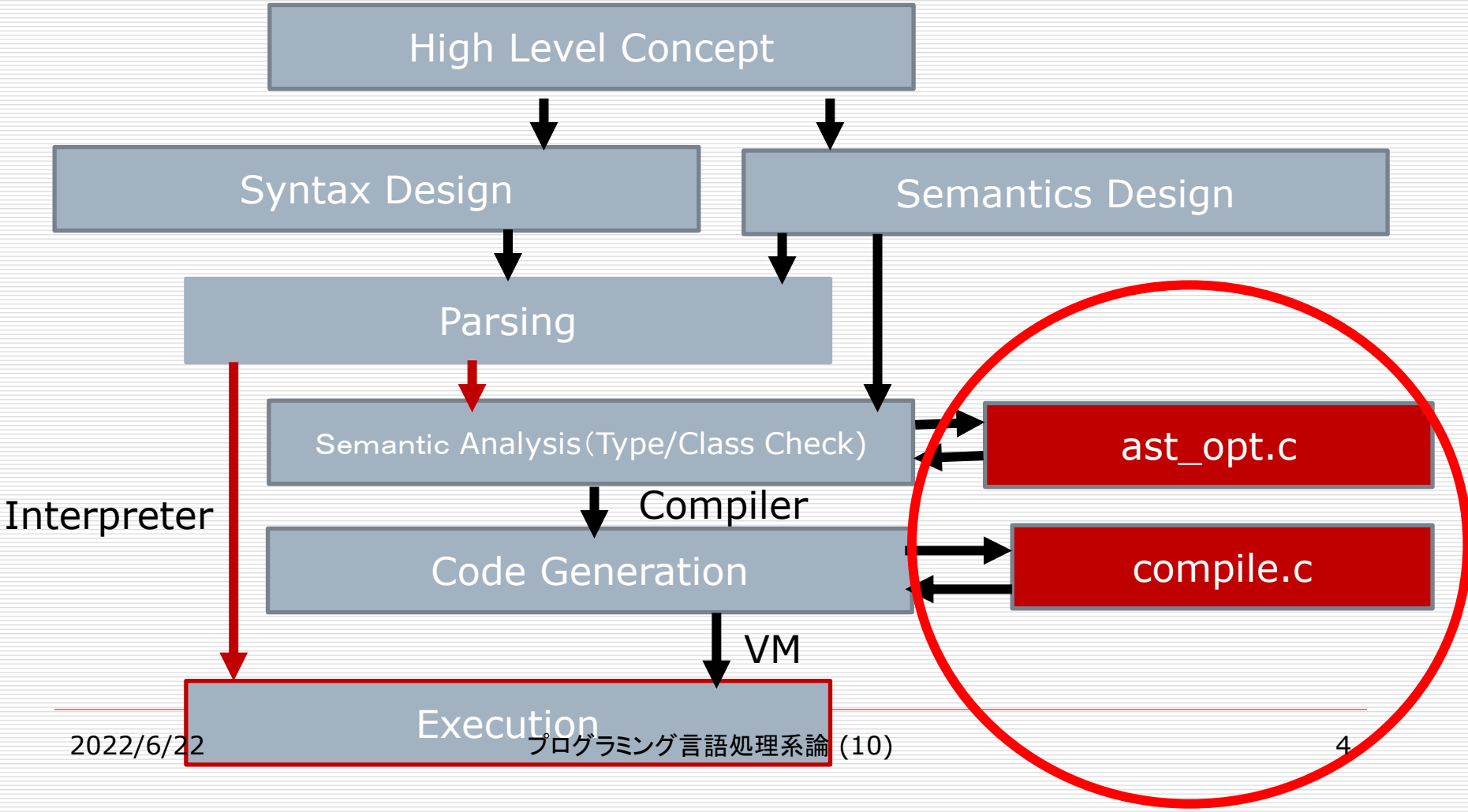
今回の予定

- コンパイラ最適化の解説 (I)
  - 最適化の役割
  - 最適化の分類
  - 最適化のための中間言語の設計
  - Peephole最適化その他
  - Pythonでの最適化の実際
- 次回以降の予定
  - Value Numbering
  - データフロー解析をもとにした最適化
  - Global Value NumberingとSSA

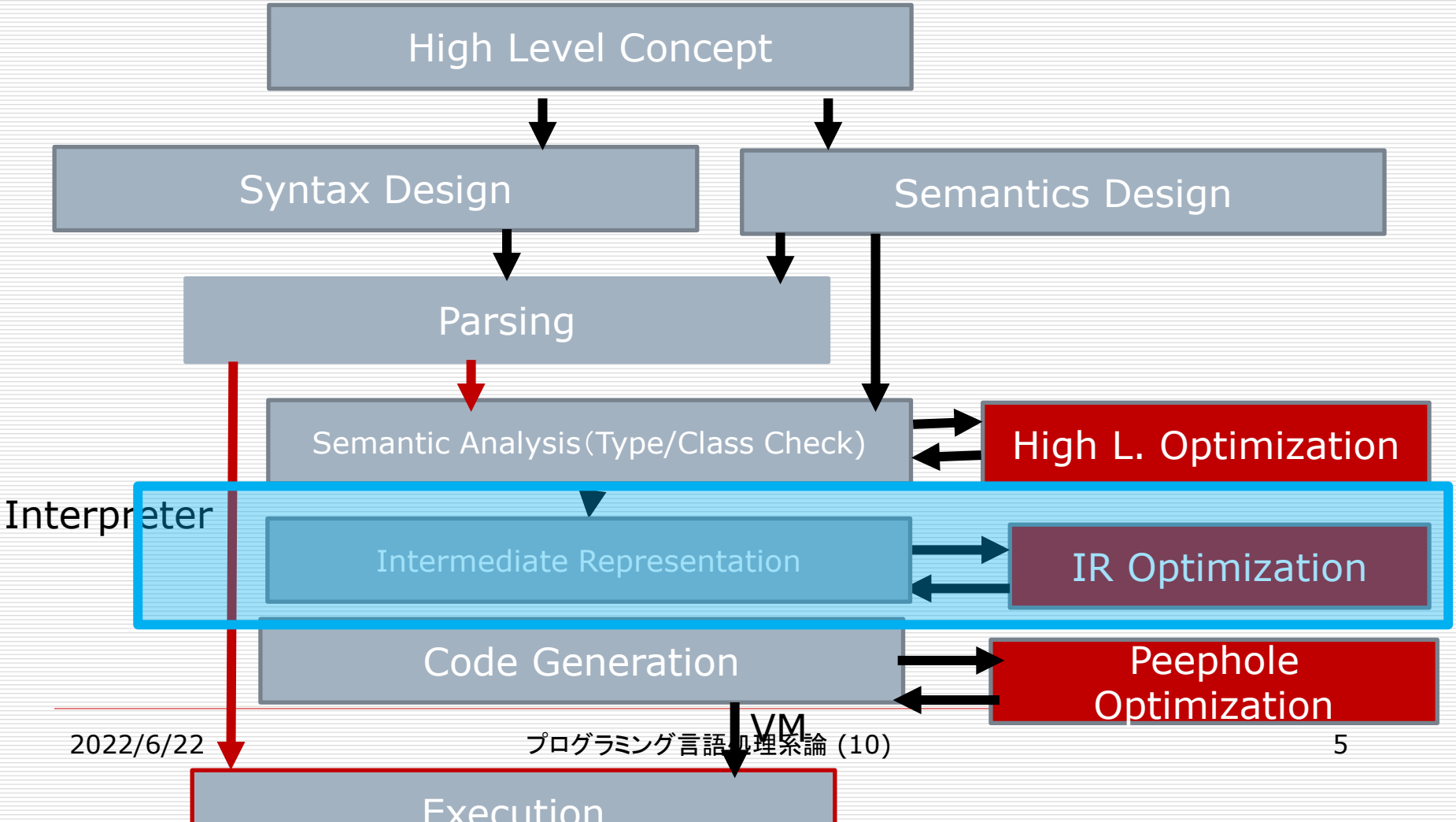
Where are we?


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

Where are we in Python?



Where are we in General Optimizing Compilers?



最適化の意味と役割

- What is “Optimization”?

Def. Optimization
同じ出力を得られることを保証しながら、プログラムをよりよいものに変換すること。通常はある「基準」（命令数等）に関して「最適」であることを求める

- What is「同じ出力」?
- What is「よりよい」?

- これを、数理的最適化理論の人が「全然最適ではないではないか」としたり顔でいじめるケースが昔から多発（今はいなくなったかな？）

同じ出力

Def. Same Output
プログラムAとBが同じ出力を持つとは、すべての入力について「結果」が同一であることをいう。
「結果」をビット列で考えるか、より抽象的な数学的なドメインで考えるかによって話は違う。

例：ビット列で考えると　$\Sigma 1/i \neq \Sigma 1/(n-i)$
　　数学的なドメインで考えると
$$\Sigma 1/i = \Sigma 1/(n-i)$$

- 一般的なものとしては、「同じ出力を持つ」ことを以下で代替する

  「各変数それぞれについて、代入される値とその順番が変更されないこと」

  「同じ値を続けて2度代入するときは、変数の状態は変化しないとみなす」

  （もちろんこれを超える定義を使うこともある。その時はいちいち明示して使う）

同じ出力　＝　同じ計算順序

- 以下のものを考える

```
a = 1
b = 2
c = 3
b = 2
a = b+1
b = a+1
 c = 3
c = c+1
```

- 以下は左と「同じ」

```
b=2
a=1
c=3
c=c+1
a=b+1
b=a+1
```

最適化の意義

- このような(きついと考えられる)制約を課しても、最適化をする価値がある。
- 「制約がきつい」かというと、そうでもない
  - 特にHPC

同じ計算順序（進んだ話題）

- 配列の計算で、この概念は重要

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
      for(k=0;k<N;k++)
         c[i][j] += a[i][k]*b[k][j];
```

- 各c[i][j]について、値の計算と代入の順序は変更されていない

```
for (i=0;i<N;i++)
   for (k=0;k<N;k++)
      for(j=0;j<N;j++)
         c[i][j] += a[i][k]*b[k][j];
```

- iとjを固定したとき、

```
c[i][j] += a[i][0]*b[0][j];
c[i][j] += a[i][1]*b[1][j];
              :
c[i][j] += a[i][N-1]*b[N-1][j];
```

- この理論は1980年代にCrayのスーパーコンピュータを対象にして大きく発展しました

考えるポイント

- コンパイラの最適化オプションでどれだけ性能が変化するか？
  - 冗長性の除去、計算の工夫、データフローの解析等
- GCCができない最適化の中で、かつHPC向けコンパイラができるもので、性能がどれだけ変化するか？
  - コンピュータアーキテクチャ、特にメモリアーキテクチャの利用等
  - コンピュータアーキテクチャやメモリアーキテクチャは、特定の最適化オプションを有効にしたり、無効にしたりすることがあります

「よりよい」

Def. よりよい
ある基準を定めた上でよりよい。たいていの場合は「最適」
基準の例：
- スピード
- コードサイズ
- メモリ使用量
- 消費電力
- …

- 誰のために？
  - 通常のプロセッサ
  - 組み込みプロセッサ
  - 大量に電力を消費するシステム
  - …
- スピードや消費電力などの物理世界の基準は、予測しづらいことが多いのだが…
- 例えば

P. Yuan, Y. Guo, X. Chen and H. Mei, "Device-Specific Linux Kernel Optimization for Android Smartphones," *2018 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, Bamberg, 2018, pp. 65-72.

Source of Performance Improvement

- 実行モデルの抽象度に依存する
- 抽象度の高いもの
  - ソースコードの行数
  - 実行される命令数
- より具体的なもの
  - CPU内の演算器の有効利用
  - アクセラレータの利用度
  - メモリの利用の仕方

最適化の占める位置

- 最適化がないとどうにもならん
- GCCでは、最適化のオプションが厖大なものに
  - man gcc で見てみましょう
    （コマンドたたくのに技術はいらない）
  - ここではgcc 9.4.0を使います
    - Ubuntuのstableな版だから、少し古い
    - 最新版は12.1.0なんだそうですが(2022-05時点)

Optimization options (1/7)

Optimization Options

-faggressive-loop-optimizations -falign-functions[=n]
-falign-jumps[=n] -falign-labels[=n] -falign-loops[=n]
-fassociative-math -fauto-profile -fauto-profile[=path]
-fauto-inc-dec -fbranch-probabilities
-fbranch-target-load-optimize -fbranch-target-load-optimize2
-fbtr-bb-exclusive -fcaller-saves
-fcombine-stack-adjustments
-fconserve-stack -fcompare-elim
-fcprop-registers -fcrossjumping
-fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules
-fcx-limited-range -fdata-sections -fdce -fdelayed-branch
-fdelete-null-pointer-checks -fdevirtualize
-fdevirtualize-speculatively -fdevirtualize-at-ltrans -fdse

Opt options (2/7)

-fearly-inlining -fipa-sra -fexpensive-optimizations
-ffat-lto-objects -ffast-math -ffinite-math-only -ffloat-store
-fexcess-precision=style -fforward-propagate -ffp-contract=style
-ffunction-sections -fgcse -fgcse-after-reload -fgcse-las
-fgcse-lm -fgraphite-identity -fgcse-sm -fhoist-adjacent-loads
-fif-conversion -fif-conversion2 -findirect-inlining
-finline-functions -finline-functions-called-once
-finline-limit=n -finline-small-functions -fipa-cp -fipa-cp-clone
-fipa-bit-cp -fipa-vrp -fipa-pta -fipa-profile -fipa-pure-const
-fipa-reference -fipa-icf -fira-algorithm=algorithm
-fira-region=region -fira-hoist-pressure -fira-loop-pressure

Opt options (3/7)

-fno-ira-share-save-slots -fno-ira-share-spill-slots
-fisolate-erroneous-paths-dereference
-fisolate-erroneous-paths-attribute -fivopts
-fkeep-inline-functions -fkeep-static-functions
-fkeep-static-consts -flimit-function-alignment
-flive-range-shrinkage -floop-block -floop-interchange
-floop-strip-mine -floop-unroll-and-jam -floop-nest-optimize
-floop-parallelize-all -flra-remat -flto -flto-compression-level
-flto-partition=alg -fmerge-all-constants -fmerge-constants
-fmodulo-sched -fmodulo-sched-allow-regmoves
-fmove-loop-invariants -fno-branch-count-reg -fno-defer-pop

Opt options (4/7)

-fno-fp-int-builtin-inexact -fno-function-cse
-fno-guess-branch-probability -fno-inline -fno-math-errno
-fno-peephole -fno-peephole2 -fno-printf-return-value
-fno-sched-interblock -fno-sched-spec -fno-signed-zeros
-fno-toplevel-reorder -fno-trapping-math
-fno-zero-initialized-in-bss -fomit-frame-pointer
-foptimize-sibling-calls -fpartial-inlining -fpeel-loops
-fpredictive-commoning -fprefetch-loop-arrays
-fprofile-correction
-fprofile-use -fprofile-use=path -fprofile-values
-fprofile-reorder-functions -freciprocal-math -free
-frename-registers -freorder-blocks
-freorder-blocks-algorithm=algorithm
-freorder-blocks-and-partition
-freorder-functions -frerun-cse-after-loop

Opt options (5/7)

-freschedule-modulo-scheduled-loops -frounding-math
-fsched2-use-superblocks -fsched-pressure -fsched-spec-load
-fsched-spec-load-dangerous -fsched-stalled-insns-dep[=n]
-fsched-stalled-insns[=n] -fsched-group-heuristic
-fsched-critical-path-heuristic -fsched-spec-insn-heuristic
-fsched-rank-heuristic -fsched-last-insn-heuristic
-fsched-dep-count-heuristic -fschedule-fusion -fschedule-insns
-fschedule-insns2 -fsection-anchors -fselective-scheduling
-fselective-scheduling2 -fsel-sched-pipelining
-fsel-sched-pipelining-outer-loops -fsemantic-interposition

Opt options (6/7)

-fsingle-precision-constant -fsplit-ivs-in-unroller -fsplit-loops
-fsplit-paths -fsplit-wide-types -fssa-backprop -fssa-phiopt
-fstdarg-opt -fstore-merging -fstrict-aliasing
-fstrict-overflow
-fthread-jumps -ftracer -ftree-bit-ccp -ftree-builtin-call-dce
-ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop
-ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop
-ftree-fre -fcode-hoisting -ftree-loop-if-convert
-ftree-loop-im
-ftree-phiprop -ftree-loop-distribution
-ftree-loop-distribute-patterns -ftree-loop-ivcanon
-ftree-loop-linear -ftree-loop-optimize -ftree-loop-vectorize

Opt options (7/7)

-ftree-parallelize-loops=n -ftree-pre -ftree-partial-pre
-ftree-pta -ftree-reassoc -ftree-sink -ftree-slsr -ftree-sra
-ftree-switch-conversion -ftree-tail-merge -ftree-ter
-ftree-vectorize -ftree-vrp -funconstrained-commons
-funit-at-a-time -funroll-all-loops -funroll-loops
-funsafe-math-optimizations -funswitch-loops -fipa-ra
-fvariable-expansion-in-unroller -fvect-cost-model
-fvpt -fweb
-fwhole-program -fwpa -fuse-linker-plugin
--param name=value -O
-O0 -O1 -O2 -O3 -Os -Ofast -Og

余談

- この厖大なフラグの選択肢から、最適なフラグのセットを選び出すのは大変
  - そうはいっても-O3でたいていは事足りる
  - それでも、特にHPC関係では、アーキテクチャを反映した微妙なものがある（loop unrollingの段数とか）
- 「最適なフラグのセットの探索」を自動化するツールは、実用的な意味があるだろう
  - 言語処理系の研究とは別
  - 理論の外にある（言語処理系の理論の裏付けがない）
  - 「自動チューニング」と呼ばれる没理論的な分野の一分野として結構人気がある

- 全探索 -- ATLAS

R. C. Whaley and J. J. Dongarra, “Automatically tuned linear algebra software,” in Supercomputing, Washington, DC, 1998.

- 機械学習 – OpenTuner

J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U. O'Reilly, S. Amarasinghe,

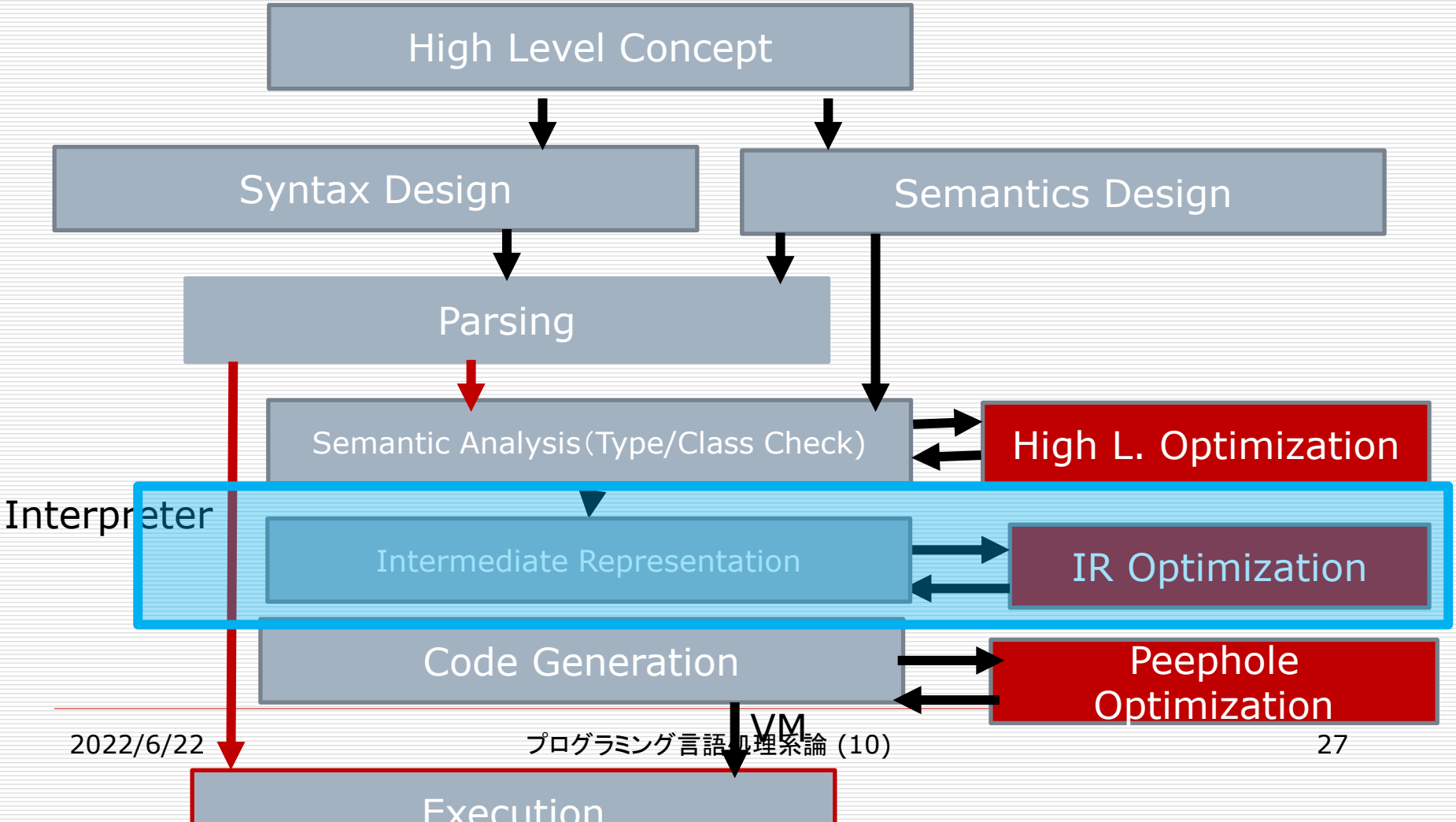
“OpenTuner: An Extensible Framework for Program Autotuning,” in International Conference on Parallel Architectures and Compilation Techniques. Edmonton, 2014.

- 最近は、このあまりよくない風潮もようなく下火になったようで…

Tuning?

- 数値計算ならば、アルゴリズム選択の効果がより大きい
- 最適化オプションが、互いに邪魔することがある
  - 例）ループアンローリングの段数を指定しても、命令パイプライニングをするときには、とりあえずfoldしてから
- 最適化オプションが、生成されるコードにどう影響するかは、実際のコードを観察しないとわからない
  - 実際のコードを観察することなしに、最適化オプションのリストと性能の数値を観察だけでは、それはサイエンスでもエンジニアリングでもないだろう
- 生成されたコードの特徴づけをして、最適化オプションの選択をするならまだわかる
  - 細々とこの手の研究が続いている

Where are we in General Optimizing Compilers?



実際の最適化ルーチンの構成 (12.1)

- gcc/passes.def, passes.c
  - gccは、GNUCCになる前後で構成が大整理
  - init_optimization_passes()で設定
    - 最適化ルーチンの数は、バージョンを追うごとに増える
    - passes.defを見てみましょう
      - NEXT_PASSを数えるだけで358個
    - これだけの最適化のパスが走る
      - 一個のルーチンを複数呼んでいる例がある（358個のうち異なりで数えると291個）

スクリプト言語の最適化はまた別で...

- 前回話したこととして
- 実行時のオーバーヘッドをどう削減できるか？
  - 動的型チェック
  - 動的型チェックは、厳密にいえばコード最適化ではない
- 動的な型変換が性能にどれだけのマイナス要因になっているかを理解する
- 静的にでも動的にでも型変換のルーチンをスキップできれば、性能が上がる
- TypeScriptの重要さの半分
- JVM上でPythonを実行する場合
  - 引数の動的型チェックを行う
  - 引数の型チェックをできるものは静的にすませてしまう

(静的)最適化の分類

1. ローカル最適化
    1. 基本ブロック内外で、一定のパターンに当てはまるものを変換(ピープホール最適化)
    2. 基本ブロック内で、変数の挙動を解析して変換
2. データフロー解析をもとにした大域最適化
3. データ依存解析をもとにしたループ最適化
4. 手続き間解析をもとにした手続き間最適化
    1. LTO (Link Time Optimization)

最適化の分類(その他)

- CPUアーキテクチャを意識した最適化
  - レジスタアロケーション
  - 命令スケジューリング
    - 実は、ループアンローリングはこれに分類される(unroll and jam)
    - ループアンローリングが現在のCPUアーキテクチャで効果があるかというと…
      - Speculationの前では、あまり…かもしれない

  (ここでは時間の関係で扱わないけど、少し実験してみましょうか → 問題 11)
- キャッシュアーキテクチャを意識した最適化
  - ループ関係

面倒なものから（簡単に触れます）

- 手続き間解析をもとにした手続き間最適化
  - Interprocedural Optimization
  - Link Time Optimization（LTO）

- 関数をまたいだ最適化を効率よく行うためにリンク時に最終的に最適化を行うことが普通になりつつある
- コンパイラシステムがサポートすることも普通になってきている

代表的な手続き間解析

- 部分評価（おぼえていますか？）
  - 手続きに渡される引数の性質を利用する

```
int g(x) {
   if (x==0) return 1;
   else return h(x-1);
}

main() {
   print g(2)+g(0);

}
```

```
int g(x) {
   if (x==0) return 1;
   else return h(x-1);
}

main() {
   print h(1)+1;

}
```

LTO

- 情報量は
- 静的なコンパイル時 ＜ 最終的な実行コードを生成する時点 ＜ 実行時
- 「静的なコンパイル時」に得られる情報をもとにした最適化が、普通に言う「静的解析」をもとにした「コンパイル、リンク時最適化」
- 「最終的な実行コードを生成する時点」での情報をもとにした最適化が部分評価をもとにしたLTO

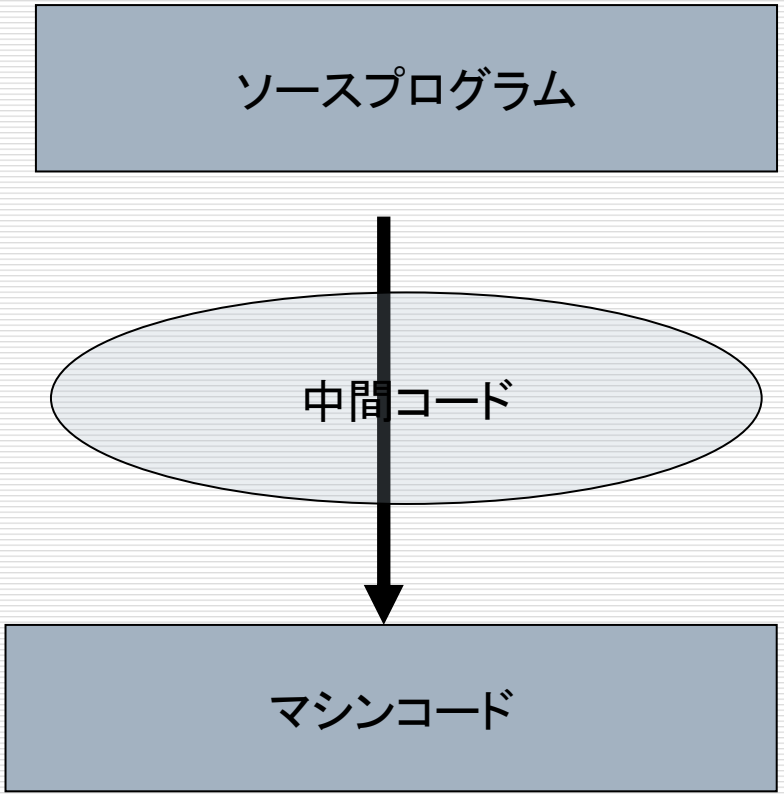
GCCでの実装

- -flto-* 関係
- 中間表現を保存
- リンク時に、部分評価を用いた最適化を行う
  - 一つのファイル内で完結している場合、コンパイル時に手続き間解析を行う
  - ファイルをまたがってコールされるものについてはLTOが必須

中間言語の設計

最適化をするための抽象度のレベルには何種類かある

- Source-to-source conversion は直感的にもわかりやすい(パース木を直接変換することもある)
- Machine codeに落としてから最適化をする(peephole最適化)は、限定的なことしかできない
- 最適化に便利なように中間言語を設計する(情報の追加・削除)
  - High Level IR
  - Low Level IR



中間表現

- ASTから、直接ByteCodeを生成してどうなるのか？
  - 性能最適化はASTを対象に可能か？
  - では、性能最適化をするにはどうしたらよいか？
- 最終的に生成されるコードを想定し”imaginary”な表現を設計する
  - 中間表現（Intermediate Representation）

中間表現版

```
mov a r2
mov b r3
add r2 r3 r4
mov 3 r5
add r4 r5 r6
mov 3 r7
mov a r8
mov b r9
add r8 r9 r10
add r7 r10 r11
mul r6 r11 r12
return r12
```

→

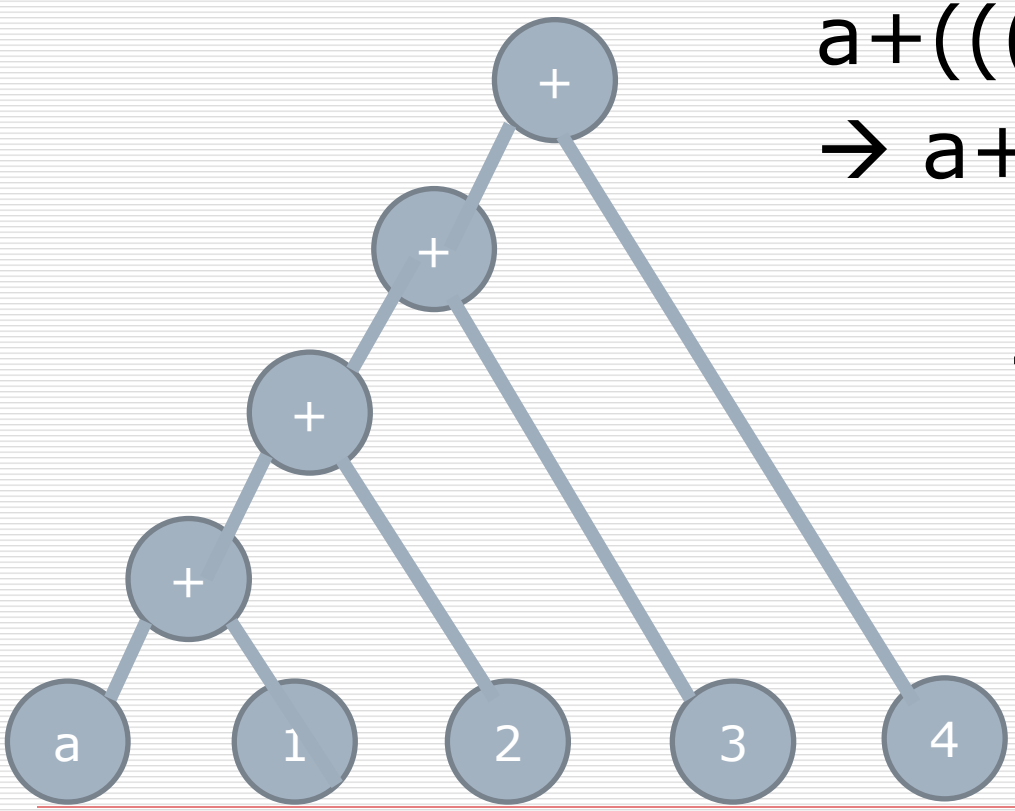
```
mov a r2
mov b r3
add r2 r3 r4
mov 3 r5
add r4 r5 r6
mul r6 r6 r12
return r12
```

中間表現のよしあし
例）Associativity等の利用

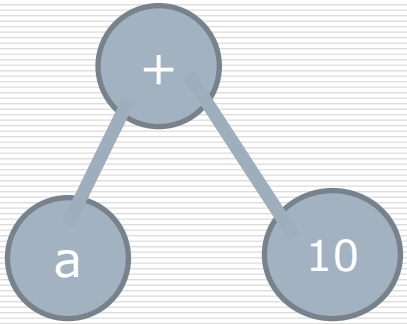
- ((((a+1)+2)+3)+4) →

a+(((1+2)+3)+4)

→ a+10



→



ここまで変換すると、手が出しにくい

```
mov a r2
mov 1 r3
add r2 r3 r4
mov 2 r5
add r4 r5 r6
mov 3 r7
add r6 r7 r8
mov 4 r9
add r8 r9 r10
```

- ここまでくると逆に手出しができる

```
PUSH_GVAR a
PUSH_CONST 1
B_ADD
PUSH_CONST 2
B_ADD
PUSH_CONST 3
B_ADD
PUSH_CONST 4
B_ADD
```

- たとえば、教材やPythonは、ASTから直接Bytecodeを生成するが、その際に最適化を行うことがある
  - ast_opt.c (Python)
- 中間表現を使ってすることは最適化
  - 最適化に適した中間表現の言語とは？
- 教材ではrtltestを提供します
  - 3 address code baseのRTL
  - 興味のある人はどうぞ

有名な中間コード

- Gimple (High/Low used in GCC) -SSA
- RTL (used in GCC)

- SSA（これは2回後で説明）

- 3-address code（op r1 r2 → r3の形式）

- (アセンブリ言語＋変数 = register)くらいで十分有用な中間言語ができます

RTL (Register Transfer Lang.)

- レジスタ(∞個)を使う
- ごく初期からGCCの中間言語
- http://gcc.gnu.org/onlinedocs/gccint/RTL.html に説明があるが、まぁ読まなくても…

- RTLのダンプは-daオプションでできます
  - 興味のある人はどうぞ

ローカル最適化（最初の一歩）

- Peephole Optimization
- では、1977年に初版がでたいわゆるDragon Bookの記述を見ていきましょう
  - ごく短い命令列を走査していき、一定のパターンをみつけて、それをよりよいパターンに置換していく
  - Redundant instruction elimination
  - Flow-of control optimization
  - Algebraic simplification
  - Machine idiom recognition

Redundant instruction elimination

- Redundant load/store

```
r0=a
a=r0
```

(下の命令は不要)

stack machineでは（例）

- push(dup) → popの削除

```
dup  →          push  →
pop             pop
```

- 同一の値をpushするならばdupに

```
store n    →    store n
push  n         dup
```

- 連続したadd/mulの結合則

```
push c0          push (c0+c1)
add         →    add
push c1
add
```

- unreachable code elimination

```
        cmp debug 1
        jmpz L1
        jmp L2
L1: …
        …
L2:
```

(下のように書き換えられる)

```
        cmp debug 1
        jmpnz L2
        …
```

Flow of control optimization

- (厳密に言えば基本ブロック内ではないのだが…)

```
        jmp L1
        …
   L1: jmp L2
```

→

```
        jmp L2
        …
    L1:jmp L2
```

(ラベルL1は最終的になくなるだろう)

Flow of control optimization

```
    jmp L1
    …
L1: jmpnz L2
L3:
→
    jmpnz L2
    jmp L3
    …
```

Algebraic Simplification（ASTでも可能）

- Strength reduction
  x * 2 → x <<1
  x ^ 2 → x * x
  x / 2 → x * 0.5
- 代数的に等価な表現への置き換え
  x + 0 → x
  x * 0 → 0
  x * 1 → x
  x / x → 1
  (浮動小数点ではやらない方が無難)

Associativity, constant folding（ASTでも可能）

- 結合則の利用

  ((a+b)+c) → a+(b+c)

((a+1)+2) → a+(1+2)

- constant folding
  - c1 op c2 → (c1 op c2の結果)
- Branch の最適化ができる
  - if (c1 cmp c2) … → (c1 op c2の結果)に従ってどちらかのブランチを選択する

if (0==0) 1 else 2;

```
PUSH_CONST 0
PUSH_CONST 0
B_EQUAL
JMPZ 3
PUSH_CONST 1
JMP 2
PUSH_CONST 2
```

if (0==0) 1 else 2;

```
PUSH_CONST 1
```

その他： Machine idioms

- ハードウェア命令があれば
  - GPGPUなどの並列命令を活かすには、並列性の抽出をすることが必須
  - 一般にmachine idiomsとは、特別に作られたハードウェア命令（連続したメモリ領域の移動、sqrt命令、…）を適用可能な部分を見つけることをいう
    - x86のrep movs*, SIMD命令
- 自動認識をあきらめ、高速ライブラリのコールの形で提供されることが多い
  - libcのstring関係ライブラリなど（Macのstrcpyが…）

- Idiom recognitionの順序で性能が変わることがある

Sato Hiroyuki. 2009. Idiom Recognition and Program Scheme Recognition Based Program Transformations for Performance Tuning--Beyond Compiler Optimizations--. In Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '09).

Peephole optimizationの効果

- 結構ばかにできない（丁寧な対応が求められる）
  - コード生成を素直にやり、最後の整形を担当するという役割分担が理想的
- コード生成の直後は、情報がいろいろ失われているので、「高度な」最適化は、peephole optimizationの前にしておくのが筋
  - 前回のJVM コードをregister machineのコードに変換したのも、この点で不利を抱え込んでいる
- パターンの発見と適用がすべて（アドホックな理論しかない）
- そんなことを言ってもですね、
  - Peephole optmizationは現在、完全なパターンマッチとして理解されている
  - 正当性の証明もあるにはあるが…

現在のコンパイラでの実装は？

- Peephole optimizationは完全にパターンマッチの世界。ということで、GCCではgcc/下の
  - フロー関係は主としてcfgcleanup.c
  - 様々なパターンは、各ファイル内に分散しておかれて（だいたい”Convert”を検索すると捕まえられる）
  - マシン特有の命令パターンは.mdファイルでパターンを記述する
    - gcc/config/i386/i386.mdをみてみましょう

Pythonではどうか

- Associativity等、代数的な最適化
  - ast_opt.c
  - Constant foldingだけかなぁ
- Peephole最適化
  - compile.cの最後の部分
  - Stack machine用のBytecodeの最適化
- とりあえず、観察してみましょう

(ふざけてんのか、と)

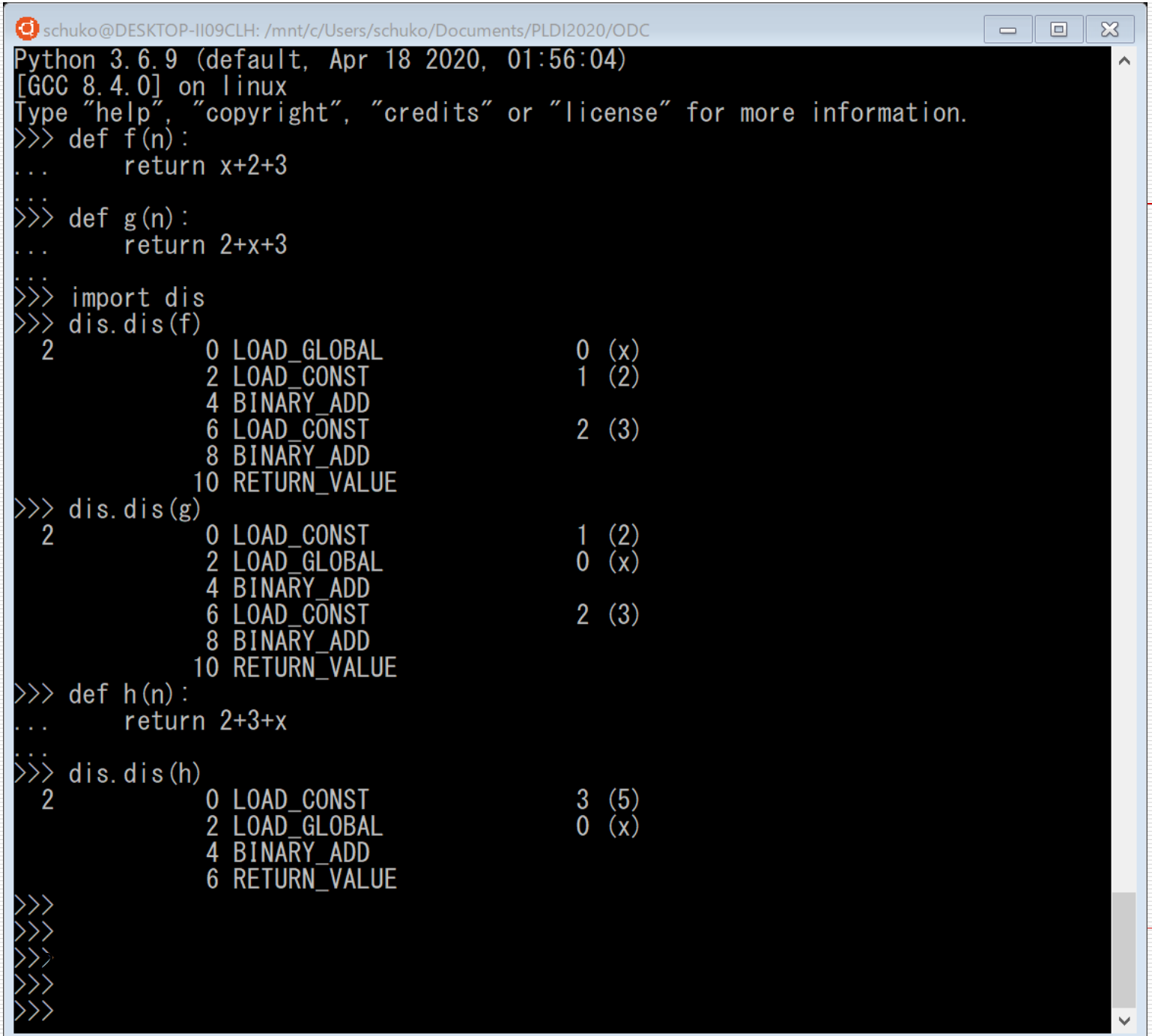
```
  def f(n):
...     if (0==0):
...             return 1
...     else:
...             return 2
...
```

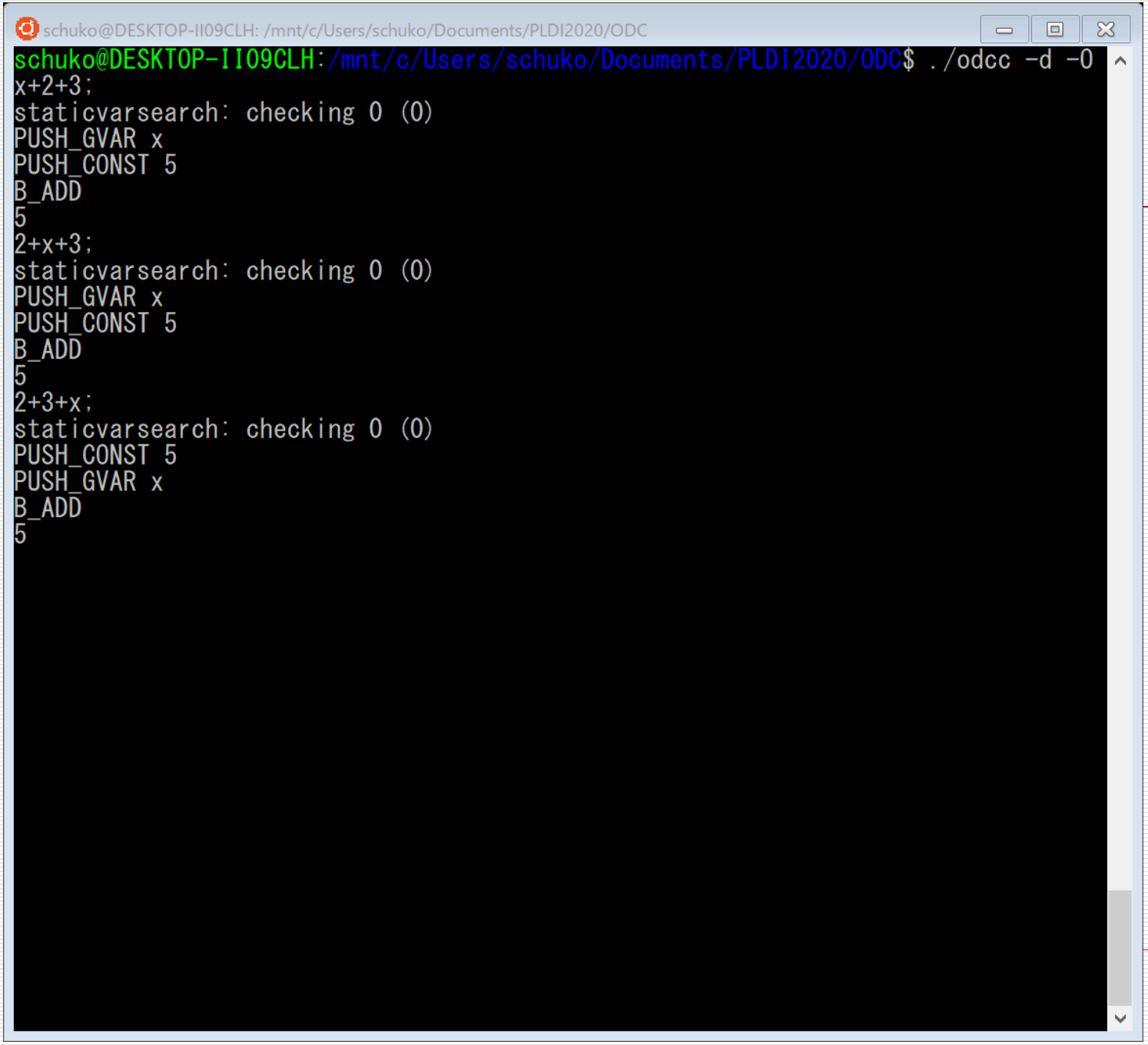
```
>>> dis.dis(f)
  2           0 LOAD_CONST    1(0)
              2 LOAD_CONST    1(0)
              4 COMPARE_OP 2(==)
              6 POP_JUMP_IF_FALSE 12

  3
              8 LOAD_CONST     2 (1)
             10 RETURN_VALUE

  5   >>     12 LOAD_CONST     3 (2)
             14 RETURN_VALUE
             16 LOAD_CONST  0(None)
             18 RETURN_VALUE
>>>
```

これくらいが標準（のはず）

```
$ ./odcc –d
if (0==0) 1 else 2;
PUSH_CONST 0
PUSH_CONST 0
B_EQUALJMPZ 3
PUSH_CONST 1
JMP 2
PUSH_CONST 2
1
```

```
$./odcc –O –d
if (0==0) 1 else 2;
PUSH_CONST 1
1
```


schuko@DESKTOP-II09CLH: /mnt/c/Users/schuko/Documents/PLDI2020/ODC
Python 3.6.9 (default, Apr 18 2020, 01:56:04)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def f(n):
...   return x+2+3
...
>>> def g(n):
...   return 2+x+3
...
>>> import dis
>>> dis.dis(f)
  2   0 LOAD_GLOBAL   0 (x)
              2 LOAD_CONST   1 (2)
              4 BINARY_ADD
              6 LOAD_CONST   2 (3)
              8 BINARY_ADD
             10 RETURN_VALUE
>>> dis.dis(g)
  2   0 LOAD_CONST   1 (2)
              2 LOAD_GLOBAL   0 (x)
              4 BINARY_ADD
              6 LOAD_CONST   2 (3)
              8 BINARY_ADD
             10 RETURN_VALUE
>>> def h(n):
...   return 2+3+x
...
>>> dis.dis(h)
  2   0 LOAD_CONST   3 (5)
              2 LOAD_GLOBAL   0 (x)
              4 BINARY_ADD
              6 RETURN_VALUE
>>>
>>>
>>>
>>>
>>>


schuko@DESKTOP-II09CLH: /mnt/c/Users/schuko/Documents/PLDI2020/ODC
schuko@DESKTOP-II09CLH:/mnt/c/Users/schuko/Documents/PLDI2020/ODC$ ./odcc -d -O
x+2+3;
staticvarsearch: checking 0 (0)
PUSH_GVAR x
PUSH_CONST 5
B_ADD
5
2+x+3;
staticvarsearch: checking 0 (0)
PUSH_GVAR x
PUSH_CONST 5
B_ADD
5
2+3+x;
staticvarsearch: checking 0 (0)
PUSH_CONST 5
PUSH_GVAR x
B_ADD
5

Pythonのast_opt.c

- fold_unaryop()
  - NOT(NOT(…) → (…)
- fold_binop()
  - ADD c1 c2 → c1+c2
- fold_tuple()
- fold_subscr()
- fold_iter()
- fold_compare()

- 後は、準同型（astfold_*()）→ PyAST_Optimize()

Pythonのcompile.c

- ターゲットはbytecode
- fold_tuple_on_constants()
  - LOAD_CONST c1 LOAD_CONST c2 … MAKE_TUPLE → LOAC_CONST (c1, c2, …)
- fold_rotations()
  - ROT_N ROT_N … ROT_N (n times) → NOP
- optimize_basic_block()
  - 連続するJumpの連結
- extend_block()
  - Exit blockへのジャンプを取り込み
- clean_basic_block()
- optimize_cfg()
- trim_unused_constants()

- optimize_cfg()が元締め

課題12

- Pythonでは、ast_opt.cとcompile.cに最適化が書かれている
  - 講義では、主要な部分にしか触れていないということを前提に、2つのファイル中の最適化を網羅的に説明せよ
  - さらに最適化を進める可能性について論ぜよ。

Ad hocでいいのか？

- 少なくとも、「見つけた」peephole optimizationについては正しさが保証されるべきだ

Lopes, N. et.al.: Provably Correct Peephole Optimizations with Alive, PLDI2015

Mullen E. et.al.: Verified Peephole Optimizations for CompCert, PLDI2016

課題13

- 以下の論文のうち、一つを選択し、その要約をせよ。

  Lopes, N. et.al.: Provably Correct Peephole Optimizations with Alive, PLDI2015

  Mullen E. et.al.: Verified Peephole Optimizations for CompCert, PLDI2016

- (一般論だが)論文の要約は実は難しい。この場合、最低限方法論の詳述をすること。数ページになるはずである