



# コンパイラ構成

## 最適化の初步

情報工学系  
権藤克彦



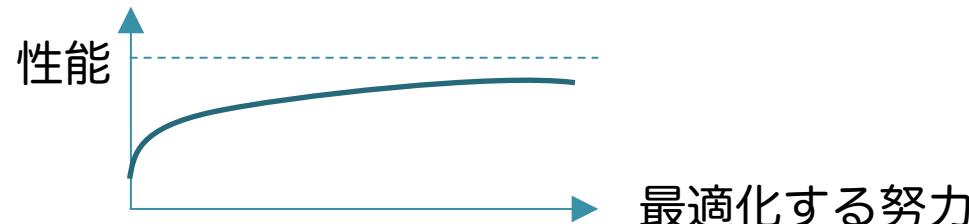
# 最適化とは

- コード最適化 (code optimization)
  - より効率良い目的コードの生成
  - プログラムの意味（計算結果）を変えずにコードを変換して、性能を向上させる
  - 言葉では「最適」だが実際は最適（ベスト）ではなくベター
  - 通常は空間効率より時間効率を重視 (cf. gccの-Osオプション)
  - 人間にしかできない最適化もある
- コード最適化器=コードを最適化するプログラム
  - 通常、中間コードやアセンブリコード（機械語）を変換
  - 説明の都合上、最適化をソースコードレベルで説明することも



# 最適化の努力と効果の関係（1）

- 未最適化コードは少しの努力で大きな効果
- 最適化が進むとコストが割に合わなくなる
  - コスト = 最適化に必要な計算時間やメモリ



- 頻繁に実行するコードに最適化のコストをかけるべき
  - 例：3重ループ中のコード
  - 80/20の法則（パレートの法則）
    - プログラムコードの20%が、実行時間の80%を占める
  - プロファイラ（例：gprof）で頻繁に実行するコードを調べる



# gccの最適化オプション

- 一般的な最適化オプション（通常はこちらを指定）

オプション	意味
-O, -O1	最適化する
-O2	もっと最適化する
-O3	もっともっと最適化する
-Os	コードサイズを小さく最適化
-O0	(ほぼ) 最適化しない

- 特定の最適化アイテムの指定（一部、形式は -f名前）

オプション	意味
-fgcse	大域的な共通部分式削除(CSE)を行う
-funroll-loops	ループ展開を行う
-fdelay-branch	遅延スロットに命令を移動する
-fno-peephole	のぞき穴最適化をしない
-frerun-loop-opt	ループ最適化を2回実行する



# 最適化の努力と効果の関係（2）

- gcc最適化オプションの効果（例）
  - バブルソートするプログラム（約60行）の実行時間
  - 3倍弱の高速化

データ数	-00	-01	-02	-03
5万個	8.8秒	4.2秒	3.4秒	3.4秒
10万個	35.4秒	17.3秒	13.7秒	13.9秒

- gcc最適化オプションの時間コスト（例）
  - bison-3.0とflex-2.6.0のコンパイル時間
  - O3は-O0より2倍弱遅い

ソースコード	-00	-01	-02	-03	
bison-3.0	10.2秒	13.7秒	15.9秒	16.7秒	6.2万行
flex-2.6.0	2.7秒	3.7秒	5.0秒	5.2秒	2.4万行



# 最適化の欠点

- 最適化はデバッグを困難にする（ことがある）
  - 最適化器がコードを移動・消去→プログラマの意図とずれる
  - 最適化をやめると症状（例：競合状態）が消えることも
  - かつては -g と -O を同時に指定できなかった
- 最適化器はアルゴリズムを改善しない
  - 例：バブルソートをクイックソートに変更しない
  - 例：バッファリングを自動的に導入しない
- 最適化にはまれにバグがある
  - でも、ほとんどは[あなたのバグ](#)

# 誰のバグ？（1/4）

- ^Cを押すとループを脱出するコード
  - busy-waitで、無駄にCPUを消費するので悪いコード
- 最適化オプション無しだとうまくいく

```
#include <stdio.h>
#include <signal.h>
int flag = 1;
void handler (int signo)
{
    flag = 0;
}
int main (void)
{
    signal (SIGINT, handler);
    printf ("ループ突入\n");
    while (flag) {}
    printf ("ループ脱出\n");
}
```

```
% gcc loop.c
% ./a.out
ループ突入
^Cループ脱出
%
```

# 誰のバグ？ (2/4)

- O3をつけると、ループ脱出できなくなる。  
最適化器のバグ！？

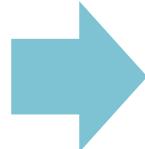
```
% gcc -O3 loop.c  
% ./a.out  
ループ突入  
^C^C^C^C^C
```

この場合、^C (SIGQUIT) や killコマンドで  
このプロセスを終了させる

# 誰のバグ？ (3/4)

- loop.s を読むと,
  - gcc -S -O3 loop.c

```
LBB0_1:  
    cmpl    $0, _flag  
    je     LBB0_3  
    jmp    LBB0_1  
LBB0_3:
```



```
        cmpl    $0, _flag  
        je     LBB0_2  
LBB0_1:  
        jmp    LBB0_1  
LBB0_2:
```

- ループ中で変数flagの値が不变, と最適化器が判断
- その結果, 条件ジャンプをループ前に移動
- でも, これは最適化器のバグでは無い

# 誰のバグ？（4/4）

- この場合、flag変数をvolatile修飾する必要がある
  - コンパイラが知り得ない方法でflagの値を変更しているから
  - volatileは最適化を抑制する効果がある

```
#include <stdio.h>
#include <signal.h>
volatile int flag = 1;
void handler (int signo)
{
    flag = 0;
}
int main (void)
{
    signal (SIGINT, handler);
    printf ("ループ突入\n");
    while (flag) {}
    printf ("ループ脱出\n");
}
```

```
% gcc -O3 loop.c
% ./a.out
ループ突入
^Cループ脱出
%
```

# volatile修飾が必要な場合（例）

- 変数の値をキャッシュしてはいけない場合、つまり変数の値が知らない間に変化する場合はvolatile必要
- 例：別スレッドやシグナルハンドラが非同期的に共有変数を書き替える場合
- 例：変数が memory-mapped I/O の領域に割り当てられている場合
  - 例えば、タイマーがメモリ領域に割り当てられている場合、読むたびに値が変わる



# 最適化オプションの使い方

- 十分なデバッグやテストが終わってから最適化する
  - デバッグ中やテスト中はコンパイル時間を節約する
- 怪しげなバグに出会ったら、念のため最適化をやめる
  - 自分のバグか、最適化器のバグか、まず問題を切り分ける
- 最適化オプションを使っても遅すぎるなら、アルゴリズムやデータ構造を手動で変更してチューニングする
  - ただし、プロファイラで遅い部分を見極めてから
  - 修正・機能追加・移植が難しくならないように、変更部分はなるべくカプセル化する（一般的に難しい）
  - 保守性向上のためには「手動チューニングしない」が正しい



# 最適化の手法

- 最適化で考慮する範囲. 下の方が広範囲を解析
  - のぞき穴 (peephole) : 数命令
  - 局所的 (local) : 基本ブロック内
  - 大域的 (global) : 手続き内
  - 手続き間 (inter-procedural) : 複数の手続きをまたがって解析
- 最適化を行うフェーズ
  - ソースレベル
  - 中間言語レベル : 多くの最適化はここで行う
  - 機械語レベル : のぞき穴最適化, レジスタ割り付け, 命令スケジューリング



# のぞき穴最適化（概要）

- ごく短いアセンブリコード断片の局所的な最適化
- (基本的に) 文脈を考慮せずに可能な最適化
- 文や式単位でナイーブにコード生成  
→ 無駄なコードができやすい
- 他の最適化の結果、のぞき穴最適化が可能になることも
- 例：0の引き算は削除可能

```
_main:  
    pushq   %rbp  
    movq   %rsp, %rbp  
    subq   $8, %rsp  
    subq   $0, %rsp  
    subq   $4, %rsp  
    pushq   $20  
    pushq   $10
```

# のぞき穴最適化（代数的単純化）

- 代数的単純化 (algebraic simplification)
  - 「0の足し算」 「1のかけ算」 「0ビットシフト」などはレジスタやメモリの値を変更しない→その命令を削除可能
- 注：副作用がある場合は削除できない
  - 例：フラグレジスタの結果を使っている場合

```
_main:  
    pushq  %rbp  
    movq  %rsp, %rbp  
    subq  $8, %rsp  
    subq  $0, %rsp  
    jo    foo  
    subq  $4, %rsp  
    pushq $20  
    pushq $10
```

これは人工的な例。  
0の引き算では絶対に  
オーバーフローは起きない。  
0の引き算を消すと、オーバー  
フローが起きる可能性がある  
→ 動作が変更

# のぞき穴最適化（冗長なロードストア）

- 冗長なロード・ストア (redundant load/store)
  - 削除しても結果が変わらないロード命令・ストア命令は削除して良い
- 例：
  - 2行目のmovq命令は削除可
  - 2行目のmovq命令実行直前に %rax と -8(%rbp) の値は常に等しいから
  - 注：2行目のmovq命令にラベルがある場合は削除不可
    - 2行目のmovq命令の実行直前に、1行目のmovq命令が実行されたとは限らないから

```
movq -8(%rbp), %rax # ロード  
movq %rax, -8(%rbp) # ストア
```

```
movq -8(%rbp), %rax # ロード  
foo:  
    movq %rax, -8(%rbp) # ストア
```

# のぞき穴最適化（強さの軽減）

- 強さの軽減 (reduction in strength)
  - ある操作（演算, 命令）を等価で効率の良いもので置換
- 例：

imulq \$8,  
\_x, %rax

→

movq \_x, %rax  
shlq \$3, %rax
- 「8の乗算」を「3ビット左シフト」で置換
- 他の例（ただし効率はアーキテクチャ依存）
  - $x * 2 \rightarrow x + x$
  - $x^2 \rightarrow x * x$
  - $x/2.0 \rightarrow x * 0.5$

# のぞき穴最適化（強さの軽減）

- 命令長が短くなる方を選ぶ場合もある
  - 実行時間は大差なくても
- 例：0の代入

```
movq $0, %rax
```

7バイト

```
xorq %rax, %rax
```

3バイト

- 例：0と比較

```
cmpq $0, %rax
```

4バイト

```
testq %rax, %rax
```

3バイト

test命令はandした場合のフラグ計算をするので、ZFが立つのは %rax が0と等しい場合だけ

# のぞき穴最適化（制御フロー）

- 制御フローの最適化 (flow-control optimization)
  - 不要なジャンプ命令や、冗長なジャンプ命令を削除
- 例 1：不要なジャンプ命令の削除

```
jmp foo    # このジャンプ命令は削除可
foo:
```

- 例 2：不要なジャンプ命令の削除

```
jne L1    # 無条件ジャンプ命令を飛び越すジャンプ
jmp L2
L1:
```

```
je L2      # 条件を反転して、無条件ジャンプを削除
L1:
```



# のぞき穴最適化（制御フロー）

- 例3：冗長なジャンプ（ジャンプにジャンプ）を削除

```
je L1    # L1にジャンプ後，すぐにL2にジャンプ  
...  
L1:  
    jmp L2
```

```
je L2    # 最初からL2にジャンプ  
...  
L1:  
    jmp L2
```



# のぞき穴最適化（デッドコード削除）

- 実行されない命令の削除 (dead code elimination)
  - 実行が到達しない命令は不要→削除可
  - のぞき穴最適化では、制御フロー解析せずに削除可能なデッドコードを探す
- 例

```
jmp L
movq %rax, %rbx # この命令は削除可能
L:
```

- 制御フロー解析やデータフロー解析をすると、デッドコードをもっと削除可能

# 基本ブロック

- 基本ブロック (basic block)
  - 命令列。ただし途中にジャンプが無く、ジャンプで途中から実行されることもないもの
  - 基本ブロック内の全命令の実行は必ず逐次的に行われる
- 制御フローグラフ等のノードは基本ブロック

```
pushq %rbp  
movq %rsp, %rbp  
subq $8, %rsp
```

基本ブロックの例

```
pushq %rbp  
je foo  
subq $8, %rsp
```

基本ブロックではない例

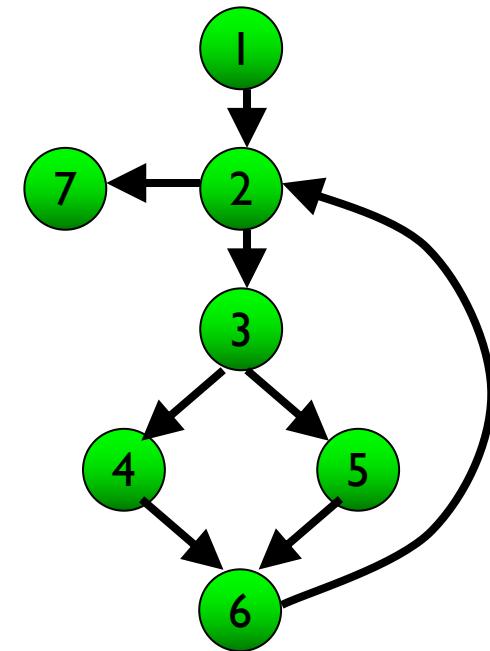
```
pushq %rbp  
foo:  
subq $8, %rsp
```

# 制御フロー解析

間接コールや間接ジャンプがあると  
正しい制御フローの作成は難しい

- 制御フロー解析 (control flow analysis)
  - 実行時に通る可能性がある実行パスを静的に調べること
  - 条件式の値は静的には不明→保守的(conservative)に調査・誤差あり

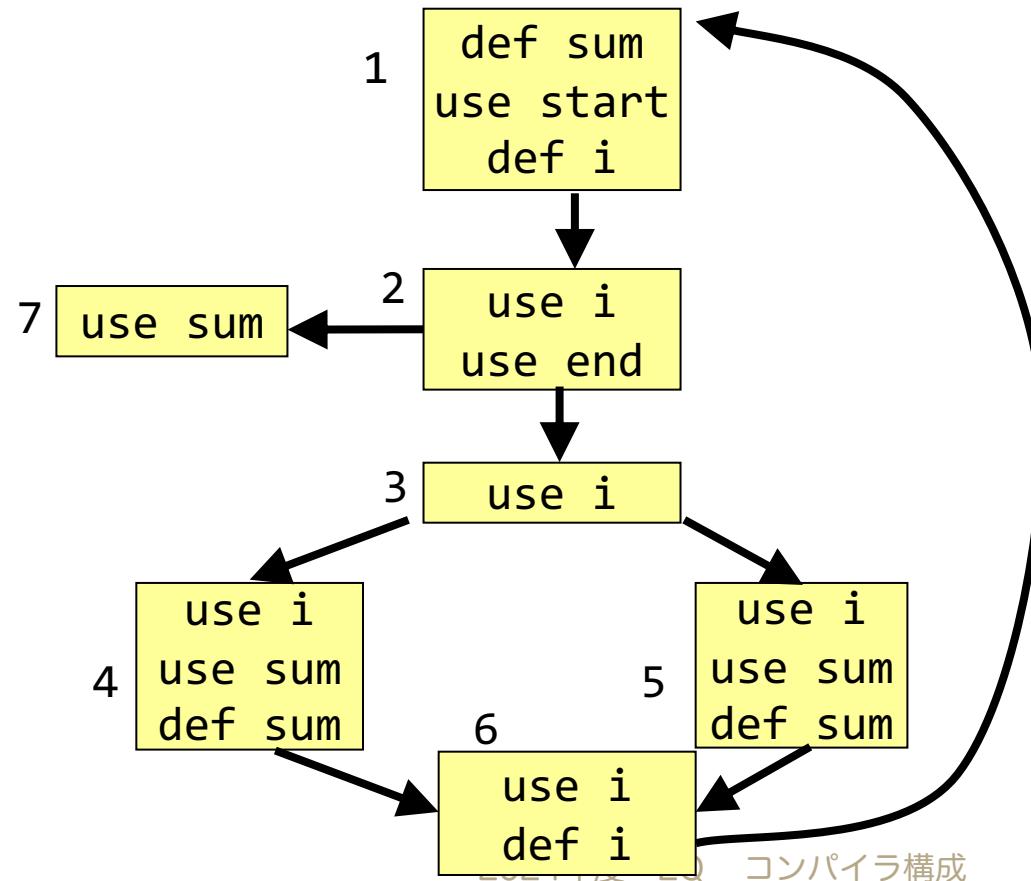
```
int abs_sum (int start, int end)
{
    int sum = 0;          // 1
    int i = start;        // 1
    while (i <= end) {   // 2
        if (i < 0)        // 3
            sum += -i;     // 4
        else
            sum += i;      // 5
        i++;              // 6
    }
    return sum;           // 7
}
```



# データフロー解析

正確なデータフロー解析には  
エイリアス解析も必要

- データフロー解析 (data flow analysis)
  - 制御フロー + 「変数の代入・参照・破壊の情報」





# エイリアス解析

- エイリアス (別名, alias)
  - 同じメモリ領域を異なる名前の変数や式が参照すること
  - 例 : int i, \*p; p = &i;
- エイリアス解析 (alias analysis)
  - あるメモリ領域のエイリアスとアクセス方法を解析すること
- C言語のポインタ演算 → 正確なエイリアス解析が困難

```
int *pointer;  
  
void foo (int *p, int n) {  
    pointer = p + i;  
}
```

pointerはどこを指しているか  
正確な解析は難しい（静的には不可能）  
(変数の値は実行しないと不明だから)

# 共通部分式の削除 (1/3)

- 共通部分式の削除 (common subexpression elimination)
  - 同じ値となる式が複数ある時、式の評価を1回で済ませて他の式はその評価結果への参照で置換する
- 例

```
x = (a + b) * 10;  
y = (a + b) * 20; // a+b を2回計算している
```

```
z = a + b;  
x = z * 10;  
y = z * 20;
```



# 共通部分式の削除 (2/3)

- 注：2つの式が離れている時

```
x = (a + b) * 10; // (1)  
...  
y = (a + b) * 20; // (2)
```

- 次の確認が必要
  - (2)の計算の前に、必ず(1)の計算が行われていること  
→ 制御フロー解析が必要
  - (1)と(2)の実行の間で、aとbの値が不变であること  
→ データフロー解析が必要
- 交換則や「同じ値を持つ変数」を考慮すると、  
共通部分式の発見は複雑になる

# 共通部分式の削除 (3/3)

- 配列要素アクセスも共通部分式

```
x = a[i] / 10;  
a[i] = x - a[j]; // 配列要素のアドレス (a+i)を2回計算
```

→

```
p = a + i; // アドレス計算を1回に  
x = *p / 10;  
*p = x - a[j];
```

# 定数たたみ込み

- 定数たたみ込み (constant folding)
  - コンパイル時に定数計算を行うこと
- 例

```
size = 4 * 1024 * 1024; // 4MB
```



```
size = 4194304;
```

# 定数伝播

- 定数伝播 (constant propagation)
  - 定数たたみ込みなどで、コンパイル時に値が決まる変数あり
  - 定数伝播=その変数を含む式をコンパイル時に計算する
- 例

The diagram illustrates the process of constant propagation. On the left, a code snippet is shown in a light blue box:  
x = 10;  
...  
y = x \* 4;In the middle, a teal arrow points from left to right, indicating the flow of analysis or compilation.  
On the right, the transformed code is shown in another light blue box:  
x = 10;  
...  
y = 40;

- 注：p.27と同じ注意が必要

# コピー伝播

- コピー伝播 (copy propagation)
  - 単純に値をコピーする代入文をなるべく削除したい
  - そのために,  $x=y;$  の後の代入文中の  $x$  を  $y$  で置換
  - 定数伝播と同じ注意が必要

- 例

```
x = y;  
z = x + 10;
```



```
x = y;  
z = y + 10;
```

- コピー伝播は  $x=y;$  という代入文を削除できる可能性を増す

# 命令のループ外への移動

- 命令のループ外への移動 (loop-invariant code motion)
  - ループ不变式 (loop-invariant) = ループ中で値の変わらない変数や式. これをループ外に移動して無駄な計算を除去
- 例

```
x = 0;  
for (i = 0; i < N; i++) {  
    x += (a + b) * i;  
}
```



```
x = 0;  
tmp = a + b;  
for (i = 0; i < N; i++) {  
    x += tmp * i;  
}
```

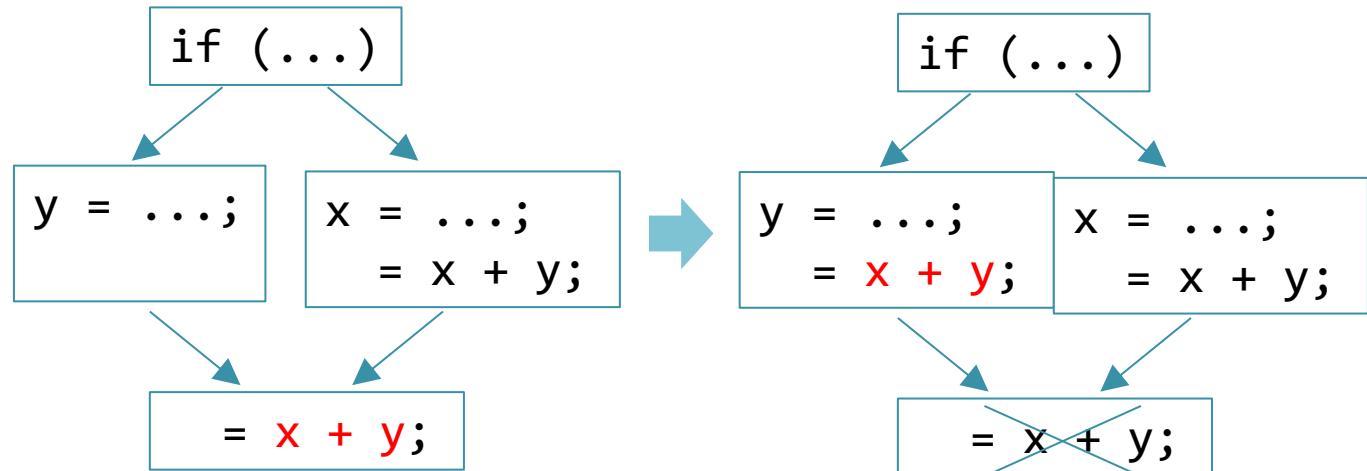
- 注 : p.7などの場合は移動不可



# 部分冗長性の除去

- 部分冗長性の除去 (partial redundancy elimination)
  - 部分冗長性
    - ある実行パスpで冗長で, p以外では冗長では無いこと
  - 部分冗長性の除去
    - 冗長な計算をp以外の全パスにも入れて, 普通の冗長なコードに変換すること. 他の冗長除去手法が可能になる.
    - 「共通部分式の削除」や「命令のループ外移動」は部分冗長性の除去の一種

- 例





# ループ変換

- ループ変換 (loop conversion)
  - ループを別の形に変換する最適化の総称
  - ループ展開 (loop unrolling), ループ融合 (loop fusion), ループ分配 (loop distribution), ループつぶし (loop collapsing)など
- ループ変換の狙い
  - ループ回数の減少
  - メモリアクセスの効率化（局所化）
  - 並列化, ベクトル化

# ループ展開

- ループ展開

```
for (i=0; i<3*N; i++) {  
    z[i]=x[i]+y[i];  
}
```



```
for (i=0; i<3*N; i+=3) {  
    z[i+0]=x[i+0]+y[i+0];  
    z[i+1]=x[i+1]+y[i+1];  
    z[i+2]=x[i+2]+y[i+2];  
}
```

- ループ展開の効果

- ループの終了判定の実行回数が減る（例では1/3）
- 条件分岐が減るので、パイプラインの分岐ハザードが減る
- ループ一回当たりの命令数が増えるので、並列化などが可能に

# ベクトル演算

- ベクトル演算 (vector operation)
  - 1命令でベクトルデータ（配列や行列）に同一演算を施す演算
  - 演算を高速に実行する方法。SIMDの一種

スカラ演算      add 10, 20 = 30

ベクトル演算      addv [1, 2, 3, …], [4, 5, 6, …] = [5, 7, 9, …]

- 配列処理ループを1命令で実行可（うまくいく形なら）

```
for (i=0; i<N; i++) {  
    z[i]=x[i]+y[i];  
}
```

```
addv x, y, z, N
```

- ベクトル型スパコンでは、この変換を（半）自動で行う



# インライン展開

- ・ インライン展開 (inline expansion)
  - callee の関数本体を caller 側に展開すること
- ・ インライン展開の効果
  - 関数呼び出しのオーバーヘッドが無くなる
  - 他の最適化（例：定数伝播）が適用可能になる
  - × コード量が増える
  - × 再帰関数は完全にはインライン展開できない

```
int main (void) {  
    x = abs (-10);  
}  
int abs (int n) {  
    if (n < 0) n = -n;  
    return n;  
}
```



```
int main (void) {  
    n = -10;  
    if (n < 0) n = -n;  
    ret = n;  
    x = ret;  
}
```

# 手続きクローニング

- 手続きクローニング (procedure cloning)
  - 手続きの特定の呼び出しに対して、手続きの複製(clone)を作り、callerの情報を使って最適化すること
- 例：foo(10)へのクローン作成例。定数伝播で最適化可

foo(10); foo(x);  
foo(x+10);

foo(10);

foo(x);  
foo(x+10);

foo (int n)  
{ ... }

foo (int n)  
{ ... }

foo (int n)  
{ ... }



- 他の手法との違い

- v.s. インライン展開
  - 手続きクローニングではcaller側コードに変更無し
  - 手続きクローニングは複数の呼び出しを1つのクローンにマップ可能
- v.s. 手続き間データフロー解析
  - 手続き間データフロー解析では全呼び出しが1つの手続きを共有・最適化

# レジスタ割り当て

- レジスタ割り付け (register allocation)
  - プログラムの効率が良くなるように、変数や式にレジスタを割り当てる
  - 通常、レジスタ数は変数の個数よりも少ない
  - レジスタへのアクセスはメモリより高速  
→ 頻繁にアクセスする変数にレジスタを割り当てる
- 最適なレジスタ割り当ては**NP完全問題**
  - 最適解を得る一般的で効率の良い方法は無い
  - なるべく簡単なアルゴリズムで短時間に割り当てる  
→ 通常、手続きやループ単位で解析してレジスタを割り当てる
- 主な手法
  - 簡単な手法、区間グラフを使う手法、干渉グラフを使う手法



# 簡単なレジスタ割り当て手法 (1/2)

- 割り付ける順序
  - 変数のアクセスが多い順に割り当てる（例：ループ内変数）
  - プログラムの先頭から順番に割り当てる
- レジスタ不足時はどうするか
  - 犠牲レジスタを選んでレジスタ値をメモリに退避
  - 一時転送用レジスタを予め確保。演算結果をすぐメモリに書く
- 犠牲レジスタの選び方
  - 次のアクセスが最も遠いレジスタを選ぶ（未来予測なので難）
    - 二度とアクセスしない（=死んでる）変数の値を持つレジスタは最初に選ぶ
  - その場所以降でのアクセス数が最も少ないレジスタを選ぶ
  - 以前のアクセスが最も遠いレジスタを選ぶ。LRUに相当
    - 過去のアクセスパターンが未来にも通用する、という前提



# 簡単なレジスタ割り当て手法 (2/2)

- 複雑なオペランドを先に計算する
  - 例 :  $a + (b + (c + d))$
  - 左オペランドから計算するとレジスタは4つ必要
  - 右オペランドから計算するとレジスタは2つで済む

```
# 左オペランドから計算
movq _a(%rip), %rax
movq _b(%rip), %rbx
movq _c(%rip), %rcx
movq _d(%rip), %rdx
addq %rdx,      %rcx
addq %rcx,      %rbx
addq %rbx,      %rax
```

```
# 右オペランドから計算
movq _c(%rip), %rcx
movq _d(%rip), %rdx
addq %rdx,      %rcx
movq _b(%rip), %rdx
addq %rdx,      %rcx
movq _a(%rip), %rdx
addq %rdx,      %rcx
```

- より一般には、式の計算に必要なレジスタ数をコード生成前に計算して考慮する

# 区間グラフレジスタ割り当て (1/3)

- 変数の生死 (live/dead variables)
  - ある地点  $p$  で変数  $x$  は生きている =  $x$  の値は  $p$  より後で再定義される前に参照される

- 例

```
x = 10;  
...  
x = 20;  
...  
y = x;
```

xは死んでる  
xは生きてる

「…」中に  $x$  は無いと仮定

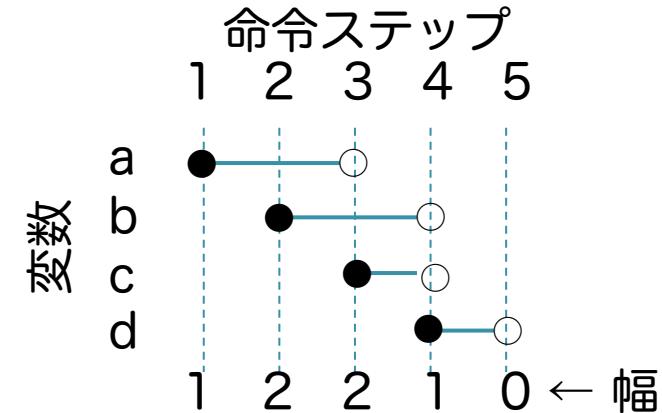
- (静的に正確に変数の生死を正確に計算するのは困難)
  - 一般に変数の値や実行パスは実行しないと分からないから



# 区間グラフレジスタ割り当て (2/3)

- 変数の生存区間 (live range of a variable)
  - 変数が生きている区間
  - 変数  $x$  と  $y$  の生存区間が重ならない → 同じレジスタを使える
- 区間グラフ (interval graph) = 変数の生存区間を列挙したグラフ
  - 例：変数  $a$  の生存区間は 1 から 3。ただし、3 の  $c$  に同じレジスタを割り当て可なので、3 を含まない (○) としている
  - 幅 (width) = ある命令ステップで生きている変数の数

```
a = 1;      // 1
b = a + 3; // 2
c = a * 2; // 3
d = c + b; // 4
```

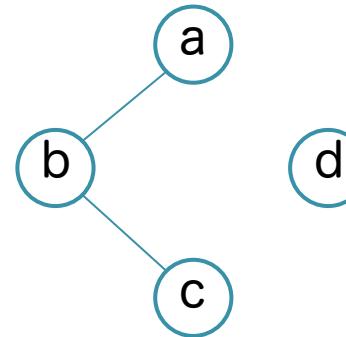
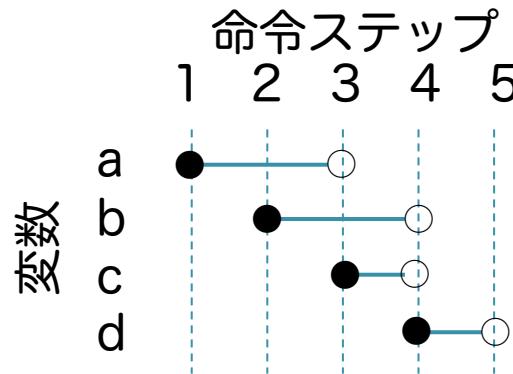


# 区間グラフレジスタ割り当て (3/3)

- 幅が  $n$  の命令ステップではレジスタが  $n$  個必要
- 最大幅 = 必要なレジスタ数
- レジスタ不足ならメモリに一時退避
- 基本ブロックだけでなく、ループや条件文にも拡張可
  - 階層的区間グラフ (hierarchical interval graph)
  - goto があると階層的区間グラフは使えない。  
cf. 干渉グラフでは使える

# 干渉グラフレジスタ割り付け (1/3)

- 干渉グラフ (interference graph)
  - 各生存区間をノードとする
  - 重複する2つの生存区間を無向辺で結ぶ
- 例



- aとbは繋がってる  $\Leftrightarrow$  aとbは同じレジスタを割り当て不可

# 干渉グラフレジスタ割り付け (2/3)

- グラフ彩色問題 (graph coloring problem)
  - 邊で繋がれた 2 つのノードの色がすべて異なるグラフを彩色された (colored) という
  - グラフ彩色問題 = グラフが  $k$  色で彩色可能かを判定する問題
  - グラフ彩色問題は NP 完全 → 近似解を得るアルゴリズムを使う
- 例：前ページの干渉グラフは 2 色で彩色可能
- 干渉グラフが  $k$  色で彩色可能  
↔ レジスタ数  $k$  個でレジスタ割り付け可能
  - 色をレジスタとすると、レジスタ数  $k$  個で、重複する生存区間に別のレジスタを割り付けたことになるから

# 干渉グラフレジスタ割り付け (3/3)

- 準備
  - 隣接ノード数が  $k$ 未満のノード  $x$ を見つける
  - グラフ  $G$ から、 $x$ と  $x$ から出ている辺を取り除き  $G'$ とする
  - $G'$ を  $k$ 色で彩色できれば、 $G$ も  $k$ 色で彩色可能
    - $x$ の隣接ノード数は  $k$ 未満なので、必ず違う色があるから
- アイデア1：まず隣接ノード数が  $k$ 未満のノードを繰り返し削除する
- アイデア2：空にならなければ、あるノードをメモリ退避することにして、生存区間を2つに分割する
  - 注：分割しなくても彩色可能かも。でも、それは調べない