コンパイラ

第10回 最適化

http://www.info.kindai.ac.jp/compiler
E館3階E-331 内線5459
takasi-i@info.kindai.ac.jp

1

コンパイラの構造

- 字句解析系
- 構文解析系
- 制約検査系
- 中間コード生成系
- 最適化系
- 目的コード生成系

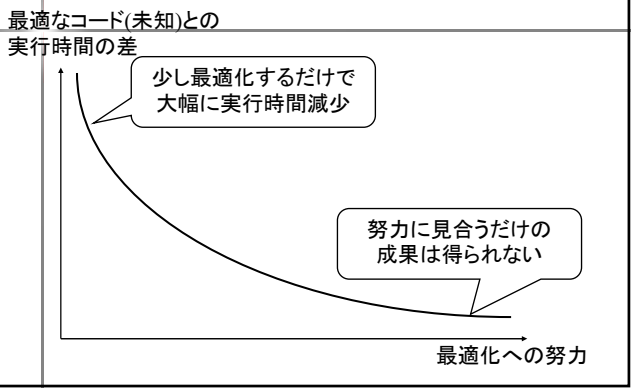
2

最適化(optimization)

- 時間最適化
  - 実行時間を短くする
    - どんな計算機でも重要
- 空間最適化
  - プログラムサイズを小さくする
    - 容量の小さい計算機(組込マイコン等)では重要

3

最適化の効率



4

最適化の分類

- 最適化の範囲
  - 局所的最適化 (local optimization)
  - 大域的最適化 (global optimization)
- ハードウェアとの関係
  - 機械独立最適化 (machine independent)
  - 機械依存最適化 (machine dependent)

5

時間最適化

- 命令の実行回数を減らす　機械独立最適化
  - より少ない命令に置き換える
  - 冗長な命令の削除
  - ループ内命令をループ外に移動
- より速い記憶装置を使う　機械依存最適化
  - 記憶位置をメモリからレジスタに
- ハードウェアの機能を利用する
  - ベクトル化・並列化を行う

6

命令実行回数削減

- 命令の実行回数を減らす
  - 可能なことはコンパイル時に計算
  - 式の性質を利用して計算を簡略化
  - 冗長な命令を取り除く
  - 実行頻度の少ない場所に移動
  - 一度求めた結果を再利用
  - ループ回数を減らす
  - 手続き呼び出しの展開

7

覗き穴最適化 (peephole optimization)

- 覗き穴最適化
  - 局所的最適化

一定範囲(覗き穴)内の命令をチェック

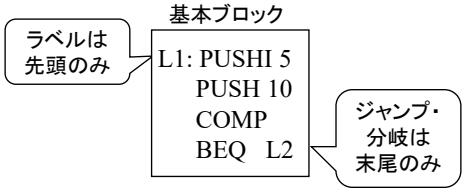
より効率的な命令に置換え

覗き穴の範囲 = 基本ブロック

8

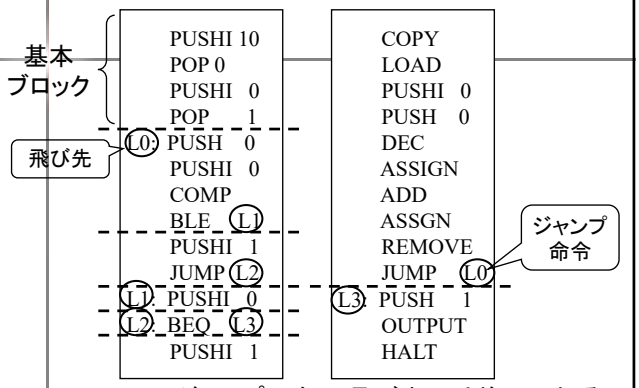
基本ブロック(basic block)

- 命令の基本ブロック
  - ラベルからジャンプ・分岐まで
    - 先頭以外ラベル(ジャンプの飛び先)命令無し
    - 末尾以外ジャンプ・分岐無し



9

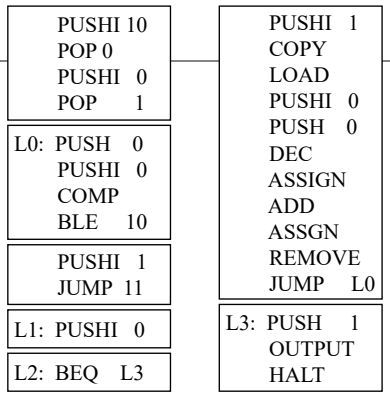
基本ブロック



ジャンプの次 or 飛び先の手前 で区切る

10

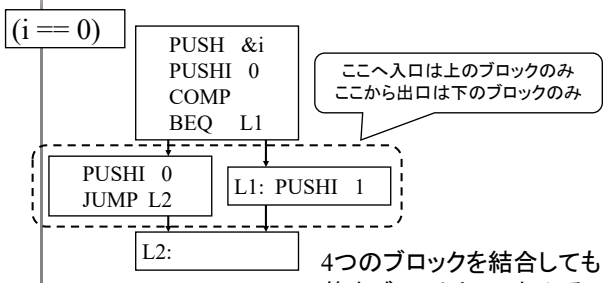
基本ブロック



11

拡張基本ブロック

- 比較演算部分のブロックの組み合わせ



4つのブロックを結合しても基本ブロックとして扱える

12

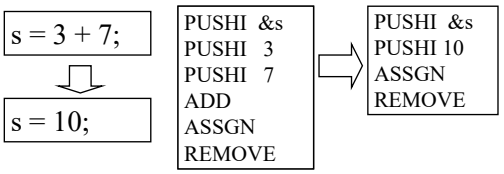
覗き穴最適化

- 基本ブロックごとに最適化
  - コンパイル時定数計算
  - 代数的簡約化
  - 強さ軽減
  - 冗長命令削除

13

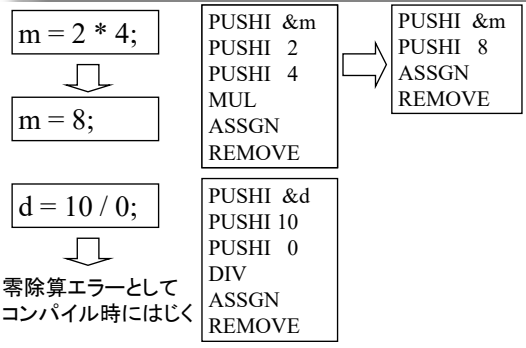
コンパイル時定数計算

- コンパイル時定数計算
  - コンパイル時に定数を計算しておく



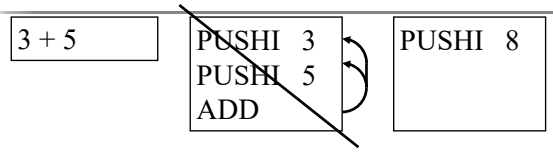
14

コンパイル時定数計算



15

コンパイル時定数計算



ADD挿入時に
1. 前2個の命令をチェック
2. 両方とも PUSHI なら命令を削除
3. 計算後の命令を積む

16

PseudoIseg クラス

	Kc		命令表格納部
	pIseg	: ArrayList <Instruction>	# 命令表
	pIsegPtr	: int	# カウンタ
	PseudoIseg ()		# コンストラクタ
-	setI (opcode : Operator, flag : int, addr : int)	: int	# 命令を格納
	appendCode (opcode : Operator, addr : int)	: int	# 命令を格納
	appendCode (opcode : Operator)	: int	# 命令を格納
	getLastCodeAddress()	: int	# 命令末尾位置
	dump()	: void	# 命令表表示
	dump2file ()	: void	# 命令表出力
	dump2file (outputFileName : String)	: void	# 命令表出力
	replaceCode (ptr : int, op : Operator)	: void	# 命令を変更
	replaceCode (ptr : int, addr : int)	: void	# 命令を変更
	checkOperator (ptr : int, op : Operator)	: boolean	# 命令の一致判定
	getOperand (ptr : int)	: int	# オペランドを得る
	removeLastCode ()	: void	# 末尾の命令を削除

17

命令の一致判定

- Isegの命令の一致判定は
  PseudoIseg.checkOperator (int, Operator)
  を利用

boolean checkOperator (int ptr, Operator op)

例 : 10番地の命令が PUSHI か？

iseg.checkOperator (10, Operator.PUSHI)

18

オペランドの習得，命令の削除

- isegの命令からオペランドを習得

```
int getOperand (int ptr)
```

- isegの末尾の命令を削除

```
void removeLastCode ()
```

19

<Term> “+” <Term> の最適化

```
void parseExp () {
  parseTerm();
  while (token == “+” ) {
    token = nextToken ();
    parseTerm();

    /* 直前の2つの命令が PUSHI か判定する */
    int addr = iseg.getLastCodeAddress();
    if (iseg.checkCode (addr-1, Operator.PUSHI)
       && iseg.checkCode (addr, Operator.PUSHI)) {
                                  :
```

20

```
                                  :
  int addr = getLastCodeAddress();
  if (checkCode (addr-1, PUSHI)
     && checkCode (addr, PUSHI)) {
     /* 定数+定数の場合 */
     int val1, val2, val3;
     val1 = getOperand (addr-1); // 2つ前のオペランド
     val2 = getOperand (addr) ;   // 1つ前のオペランド
     val3 = val1 + val2;        // 定数計算
     removeLastCode ();
     removeLastCode (); // 命令を2個削除
     appendCode (PUSHI, val3 ); // 計算後の値を積む
  } else appendCode (ADD);
```

21

コンパイル時定数計算

a[1] = 10;

```
PUSHI  10
PUSHI   1
ADD
PUSHI  10
ASSGN
REMOVE
```

⇒

```
PUSHI  11
PUSHI  10
ASSGN
REMOVE
```

x = b[20];

```
PUSHI   0
PUSHI  30
PUSHI  20
ADD
LOAD
ASSGN
REMOVE
```

⇒

```
PUSHI   0
PUSHI  50
LOAD
ASSGN
REMOVE
```

⇒

```
PUSHI   0
PUSH   50
ASSGN
REMOVE
```

※ &x = 0
&a = 10
&b = 30
の場合

22

- <Unsigned> ::= NAME [“[” <Exp> “]”] の場合

```
if (token == NAME) {
  String name = // tokenから変数名を得る
    int address = // 変数表を参照してname の番地を得る
  token = nextToken();
  appendCode (PUSHI, address);     // 左辺値右辺値共通
  if (token == “[”) { // 配列の場合
     token = nextToken();
     if (token ∈ First (<Exp>))
        parseExp(); else syntaxError();
     if (token == “]” )
        token = nextToken(); else syntaxError();
     int addr = getLastCodeAddress();
     if (checkCode (addr, PUSHI)) { // 配列の添え字が定数
                                  :
```

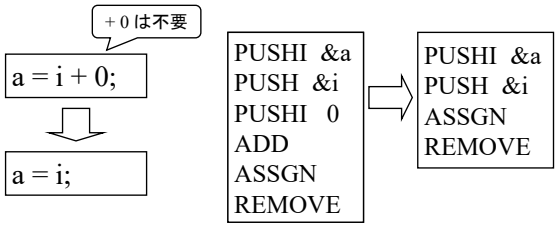
23

```
                                  :
     if (token == “]” )
       token = nextToken(); else syntaxError();
     int addr = getLastCodeAddress()
     if (checkCode (addr, PUSHI)) { // 配列の添え字が定数
       int val = getOperand (addr); // 添え字の値を得る
       address += val;          // 先頭の番地 + 添え字 を計算
       removeLastCode ();
       removeLastCode ();   // 命令を2個削除
       if (token == “=” ) {    // 次のトークンが代入の場合
          appendCode (PUSHI, address);       // 左辺値を積む
       else  appendCode (PUSH, address);    // 右辺値を積む
     } else {
       appendCode (ADD);
                                  :
```

24

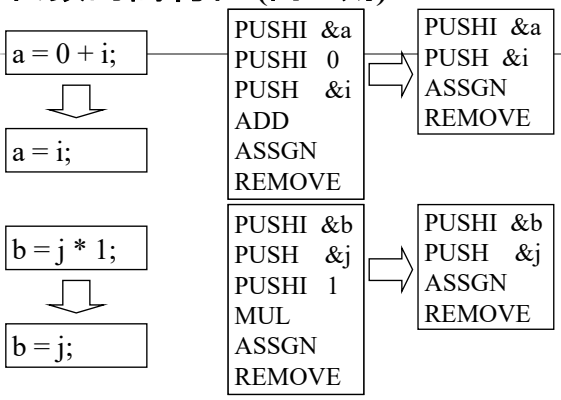
代数的簡約化 (同一則)

- 代数的簡約化
  - 簡略できる演算を簡略化する



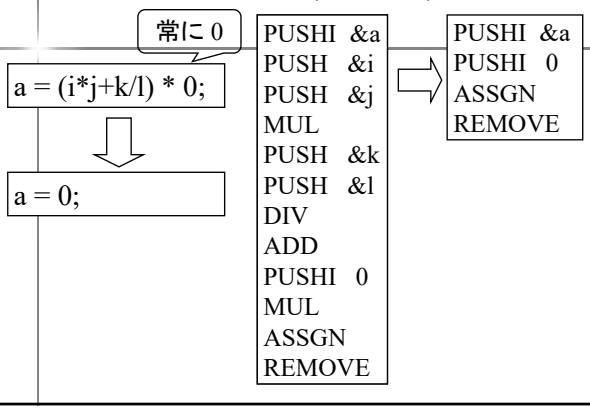
25

代数的簡約化 (同一則)



26

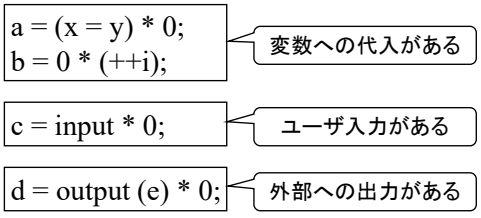
代数的簡約化 (有界則)



27

代数的簡約化

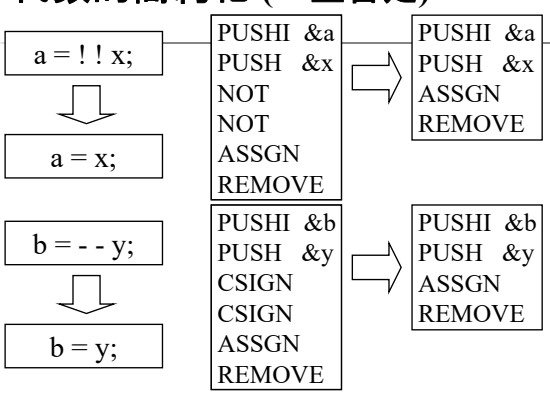
- *0 でも削除できない(削除に注意が必要な)例



代入, 入力, 出力がある場合は注意が必要

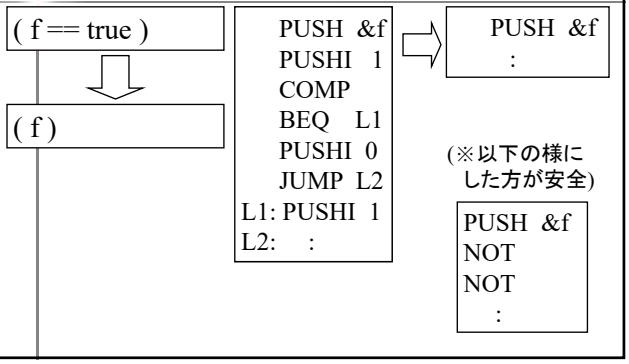
28

代数的簡約化 (二重否定)



29

代数的簡約化 (零判定)



30

代数的簡約化 (零判定)

(f == false)

⇩

(! f)

```
    PUSH  &f
    PUSHI 0
    COMP
    BEQ   L1
    PUSHI 0
    JUMP  L2
L1: PUSHI 1
L2:   :
```

⇨

```
    PUSH  &f
    NOT
      :
```

31

代数的簡約化 (零判定)

(n == 0)

⇩

(! n)

```
    PUSH  &n
    PUSHI 0
    COMP
    BEQ   L1
    PUSHI 0
    JUMP  L2
L1: PUSHI 1
L2:   :
```

⇨

```
    PUSH  &n
    NOT
      :
```

32

代数的簡約化 (零判定)

(n != 0)

⇩

(n)

```
    PUSH  &n
    PUSHI 0
    COMP
    BNE   L1
    PUSHI 0
    JUMP  L2
L1: PUSHI 1
L2:   :
```

⇨

```
    PUSH  &n
      :
```

(※以下の様にした方が安全)

```
PUSH &n
NOT
NOT
  :
```

33

簡約化可能な演算命令の例

- 0	0							- - x	x
x * 0	0	0 * x	0	x * 1	x	1 * x	x		
x / 0	err	0 / x	0	x / 1	x			x / x	1
x % 0	err	0 % x	0	x % 1	0			x % x	0
x + 0	x	0 + x	x						
x - 0	x	0 - x	- x					x - x	0
x == 0	! x	0 == x	! x	x != 0	x	0 != x	x		
! F	T			! T	F			! ! x	x
x && F	F	F && x	F	x && T	x	T && x	x	x && x	x
x || F	x	F || x	x	x || T	T	T || x	T	x || x	x
x == F	! x	F == x	! x	x == T	x	T == x	x	x == x	T
x != F	x	F != x	x	x != T	! x	T != x	! x	x != x	F

34

比較命令

t : スタックトップ　s: スタックの2番目

比較命令	s < t	s == t	s > t
COMP	-1	0	1
EQ	0	1	0
NE	1	0	1
LE	1	1	0
LT	1	0	0
GE	0	1	1
GT	0	0	1

情報システムプロジェクトI の
VSMアセンブラでは COMP のみ使用可

35

比較演算のアセンブラコード

(<Exp>$_1$ == <Exp>$_2$)

```
    <Exp>1のコード
    <Exp>2のコード
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2:
```

演算子	分岐コード
==	BEQ
!=	BNE
<=	BLE
<	BLT
>=	BGE
>	BGT

36

比較演算のアセンブラコード (EQ 命令がある場合)

(<Exp>$_1$ == <Exp>$_2$)

<Exp>$_1$のコード
<Exp>$_2$のコード
EQ

演算子	比較命令
==	EQ
!=	NE
<=	LE
<	LT
>=	GE
>	GT

37

COMP と EQ

比較命令	s < t	s == t	s > t
COMP	-1 (true)	0 (false)	1 (true)
EQ	0 (false)	1 (true)	0 (false)

COMP と EQ は 真偽が逆
⇒ COMP の値を否定すれば EQ と等価

COMP
NOT
⇔ 等価 EQ

38

代数的簡約化(比較演算)

(<Exp>$_1$ == <Exp>$_2$)

<Exp>$_1$のコード
<Exp>$_2$のコード
COMP
BEQ L1
PUSHI 0
JUMP L2
L1: PUSHI 1
L2:

⇒

<Exp>$_1$のコード
<Exp>$_2$のコード
COMP
NOT

39

COMP と NE

比較命令	s < t	s == t	s > t
COMP	-1 (true)	0 (false)	1 (true)
NE	1 (true)	0 (false)	1 (true)

COMP と NE は 真偽が同じ
⇒ COMP の値はそのままで NE と等価

COMP
⇔ 等価 NE

40

代数的簡約化(比較演算)

(<Exp>$_1$!= <Exp>$_2$)

<Exp>$_1$のコード
<Exp>$_2$のコード
COMP
BNE L1
PUSHI 0
JUMP L2
L1: PUSHI 1
L2:

⇒

<Exp>$_1$のコード
<Exp>$_2$のコード
COMP

(※以下の様にした方が安全)

COMP
NOT
NOT

41

COMP と LE

比較命令	s < t	s == t	s > t
COMP	-1 (true)	0 (false)	1 (true)
LE	1 (true)	1 (true)	0 (false)
COMP DEC	-2 (true)	-1 (true)	0 (false)

COMP の値から 1 を引くと LE は 真偽が同じ
⇒ COMP の値から1を引けば LE と等価

COMP
DEC
⇔ 等価 LE

42

代数的簡約化 (比較演算)

(<Exp>$_1$ <= <Exp>$_2$)

```
    <Exp>1のコード
    <Exp>2のコード
    COMP
    BLE    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2:
```

⇒

```
    <Exp>1のコード
    <Exp>2のコード
    COMP
    DEC
```

(※以下の様にした方が安全)

```
COMP
DEC
NOT
NOT
```

43

比較演算のアセンブラコード

(<Exp>$_1$ == <Exp>$_2$)

```
    <Exp>1のコード
    <Exp>2のコード
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2:
```

⇒

演算子	コード
==	COMP NOT
!=	COMP
<=	COMP DEC
<	COMP INC NOT
>=	COMP INC
>	COMP DEC NOT

44

代数的簡約化 (条件分岐)

if (<Exp>$_1$ == <Exp>$_2$) ...

```
    <Exp>1
    <Exp>2
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2: BEQ    L3
```

代数的簡約化 ⇒

```
<Exp>1
<Exp>2
COMP
NOT
BEQ   L3
```

代数的簡約化 ⇒

```
<Exp>1
<Exp>2
COMP
BNE   L3
```

否定した結果
0 のとき分岐 ⇔ 0以外のとき分岐

45

代数的簡約化 (条件分岐)

if (<Exp>$_1$ <= <Exp>$_2$) ...

```
    <Exp>1
    <Exp>2
    COMP
    BLE    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2: BEQ    L3
```

代数的簡約化 ⇒

```
<Exp>1
<Exp>2
COMP
DEC
BEQ   L3
```

代数的簡約化 ⇒

```
<Exp>1
<Exp>2
COMP
BGT   L3
```

1 を引いた結果
0 のとき分岐 ⇔ 1 のとき分岐

46

条件分岐のアセンブラコード

if (<Exp>$_1$ == <Exp>$_2$)

```
    <Exp>1のコード
    <Exp>2のコード
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2: BEQ L3
```

⇒

演算子	コード
==	COMP BNE L3
!=	COMP BEQ L3
<=	COMP BGT L3
<	COMP BGE L3
>=	COMP BLT L3
>	COMP BLE L3

47

代数的簡約化 (条件分岐)

if (<Exp> == 0) ...

```
    <Exp>
    PUSHI  0
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2: BEQ    L3
```

代数的簡約化 ⇒

```
<Exp>
NOT
BEQ   L3
```

代数的簡約化 ⇒

```
<Exp>
BNE   L3
```

48

条件分岐のアセンブラコード

if (<Exp> == 0) …

```
    <Exp>のコード
    PUSHI  0
    COMP
    BEQ    L1
    PUSHI  0
    JUMP   L2
L1: PUSHI  1
L2: BEQ L3
```

演算子	コード
==	BNE L3
!=	BEQ L3
<=	BGT L3
<	BGE L3
>=	BLT L3
>	BLE L3

49

冗長命令削除 (自己代入)

- 冗長命令削除
  - 無用な命令を取り除く

自分自身への代入

a = a;

削除

```
PUSHI  &a
PUSH   &a
ASSGN
REMOVE
```

50

冗長命令削除 (自己代入)

```
a = a + (2 - 2);
b = b * (3 / 3);
c = c && (5 == 5);
d = (d == (1 > 0));
```

定数計算

```
a = a + 0;
b = b * 1;
c = c && true;
d = (d == true);
```

代数的簡約化

```
a = a;
b = b;
c = c;
d = d;
```

削除

51

冗長命令削除 (ASSGN と REMOVE)

```
s = i + j;
output (s);
```

```
PUSHI  &s
PUSH   &i
PUSH   &j
ADD
ASSGN
REMOVE
PUSH   &s
OUTPUT
```

```
PUSHI  &s
PUSH   &i
PUSH   &j
ADD
ASSGN
OUTPUT
```

スタックに s の値が残る

s の値を削除

s の値を積む

output (s = i + j);

52

冗長命令削除 (ASSGN と REMOVE)

```
i = j*k;
```

```
PUSHI  &i
PUSH   &j
PUSH   &k
MUL
ASSGN
REMOVE
```

```
PUSH   &j
PUSH   &k
MUL
POP    &i
```

スタックに i の値が残る

即座に値を削除

53

冗長命令削除 (COPY と REMOVE)

```
i++;
```

```
PUSH  &i
COPY
INC
POP   &i
REMOVE
```

```
PUSH  &i
INC
POP   &i
```

スタックに値を残すため

即座に値を削除

```
++j;
```

```
PUSH  &j
INC
COPY
POP   &j
REMOVE
```

```
PUSH  &j
INC
POP   &j
```

54

冗長命令削除 (不要な式文)

```
a * b + c / d;
```

代入, 入力, 出力の無い式文

⇒ 削除可能

代入, 入力, 出力がある場合は注意が必要

```
x * (y=1) + (++z);   ⇨  y=1; ++z;
x * y + input / z;   ⇨  input;
x * (output (y)) + z; ⇨  output (y);
```

55

強さの軽減 (strength reduction)

- 強さの軽減
  - より速い演算に置き換える

```
a = x * 2;   ⇨   a = x + x;
```

多くの計算機では掛け算より足し算の方が速い

機械依存最適化

```
a = x << 1;
```
の方が速いかも？

56

強さの軽減

```
a = x ** 2;   ⇨   a = x * x;
```
** : べき乗

```
b = y * 4;    ⇨   b = y << 2;
```
<< : 左シフト

```
c = z / 8;    ⇨   c = z >> 3;
```
>> : 右シフト

```
d = u / 5.0;  ⇨   d = u * 0.2;
```
有効桁数に注意

```
e += 1;       ⇨   ++e;
(f < g / h)   ⇨   (f * h < g)
```

57

冗長命令削除 (不要な代入)

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

以降のプログラムで z が不使用ならば削除可能

⇒ 以降のプログラムの解析が必要

制御フロー解析
データフロー解析

大域的最適化

58

制御フローグラフ (control flow graph)

- 制御フローグラフ
  - ノード : 基本ブロック
  - ノード間の矢印 : ノードA→ノードB が存在
    ⇔ ノードAの最後の命令から
    ノードBの最初の命令に行く可能性あり

```
10  PUSH   5
11  PUSHI 100
12  COMP
13  BGE    20
```

```
14  PUSHI  5
15  PUSH   5
    :
```

```
20  PUSHI  1
21  OUTPUT
```

13 BGE 20
からは
14 と 20 へ

59

制御フローグラフ

```
0  PUSHI 10
1  POP 0
2  PUSHI  0
3  POP    1
```

```
4  PUSH   0
5  PUSHI  0
6  COMP
7  BLE    10
```

```
8  PUSHI  1
9  JUMP  11
```

```
10  PUSHI  0
```

```
11  BEQ   23
```

```
12  PUSHI  1
13  COPY
14  LOAD
15  PUSHI  0
16  PUSH   0
17  DEC
18  ASSIGN
19  ADD
20  ASSGN
21  REMOVE
22  JUMP   4
```

```
23  PUSH   1
24  OUTPUT
25  HALT
```

60

制御フロー解析 (control flow analysis)

- 制御フロー解析
  - 大域的最適化
  - 制御フローグラフを用いて解析する

基本ブロック

```
10  PUSH   5
11  PUSHI 100
12  COMP
13  BGE    20
```

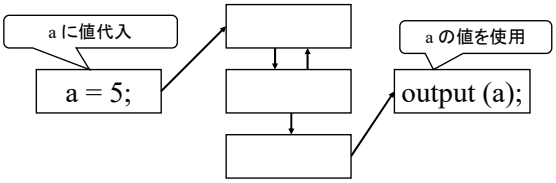
```
14  PUSHI  5
15  PUSH   5
      :
```

```
20  PUSHI  1
21  OUTPUT
```

61

データフロー解析 (data flow analysis)

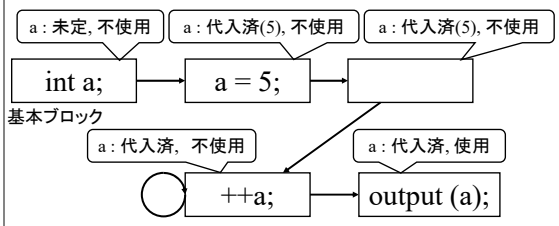
- データフロー解析
  - 式で求めた値が何処で利用されているか
  - 制御フロー解析と共に使用



62

データフロー解析

ブロック内で各変数の
左辺値(代入), 右辺値(値の使用)の
有無をチェック



63

制御フロー・データフロー解析

- 制御フロー・データフローを用いて解析
  - 冗長命令削除
  - 計算結果の再利用
  - 定数伝播
  - 複写伝播
  - 到達不能命令削除
  - 実行頻度の少ない場所に移動
  - ループ回数を減らす
  - 手続き呼び出しの展開

64

冗長命令削除 (恒真の条件分岐)

```
while (true) {
  <st>
}
```

```
L1: PUSHI  1
    BEQ    L2
    <st>
    JUMP   L1
L2:
```

```
L1: <st>
    JUMP   L1
L2:
```

```
if (true) {
  <st>
}
```

```
    PUSHI  1
    BEQ    L1
    <st>
L1:
```

```
    <st>
L1:
```

65

冗長命令削除 (連続ジャンプ)

```
while (<exp>1) {
  while (<exp>2) {
    <st>
  }
}
```

```
L1: <exp>1
    BEQ    L4
L2: <exp>2
    BEQ    L3
    <st>
    JUMP L2
L3: JUMP L1
L4:
```

```
L1: <exp>1
    BEQ    L4
L2: <exp>2
    BEQ    L1
    <st>
    JUMP L2
L3: JUMP L1
L4:
```

66

冗長命令削除 (次の行へのジャンプ)

```
if (<exp>) {
   <st>
} else {}
```

```
    <exp>
    BEQ   L1
    <st>
    JUMP  L2
L1:L2:
```

⇒

```
    <exp>
    BEQ   L1
    <st>
L1:
```

```
if (<exp>) {}
else {
   <st>
}
```

```
    <exp>
    BEQ   L1
    JUMP  L2
L1: <st>
L2:
```

⇒

```
    <exp>
    BNE   L2
    <st>
L2:
```

2行下へ分岐
かつ1行下がジャンプ

67

到達不能命令の削除

- 到達不能命令の削除
  - 決して実行されない命令を削除

```
while (<exp>) {
   :
   break;
   <st>
}
```

⇒

```
while (<exp>) {
   :
   break;
}
```

68

到達不能命令の削除

```
func () {
   :
   return x;
   <st>
}
```

⇒

```
func() {
   :
   return x;
}
```

```
while (<exp>) {
   :
   continue;
   <st>
}
```

⇒

```
while (<exp>) {
   :
   continue;
}
```

69

到達不能命令の削除

```
if (false) {
   <st>
}
```

⇒ if 文全体を削除

```
if (false) {
   <st1>
} else {
   <st2>
}
```

⇒ $<st_2>$

恒真・恒偽の条件分岐は
デバグ等で使用される

```
final boolean DEBUG = false;
    :
if (DEBUG) (デバグ情報出力)
```

70

到達不能命令の削除

```
while (<exp>1)
  while (<exp>2)
    <st>
```

```
L1: <exp>1
    BEQ   L4
L2: <exp>2
    BEQ   L3
    <st>
    JUMP L2
L3: JUMP L1
L4:
```

⇒

```
L1: <exp>1
    BEQ   L4
L2: <exp>2
    BEQ   L1
    <st>
    JUMP L2
L3: JUMP L1
L4:
```

⇒

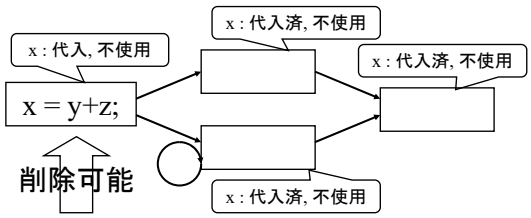
```
L1: <exp>1
    BEQ   L4
L2: <exp>2
    BEQ   L1
    <st>
    JUMP L2
L4:
```

JUMP 命令, RET命令の直後に
到達不能命令が発生し易い

71

冗長命令削除 (不要な代入)

- 不要な代入
  - それ以降使用されない代入は削除可能



以降の全ブロックで x の値は使用されない

72

結果の再利用

■ 一度求めた結果を再利用

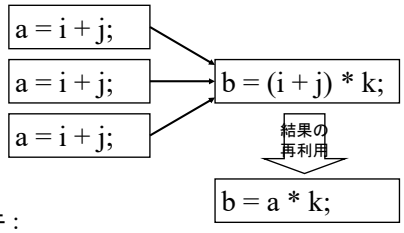
```
a = i + j;          a = i + j;
b = (i + j) * k;    b = a * k;
```

利点：演算回数を減らせる

条件：
b を計算する前に必ず a を計算
a の計算から b の計算までの間で i, j の値に変化無し

73

結果の再利用



条件：
全てのブロックで a に i+j を代入
a の計算から b の計算までの間で i, j の値に変化無し

74

共通部分の再利用

■ 共通部分の計算結果を一時変数に記憶

例：

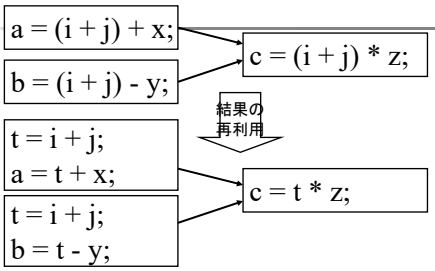
```
a = (i + j) + x;        t = i + j;
b = (i + j) - y;        a = t + x;
c = (i + j) * z;        b = t - y;
                        c = t * z
```

（一時変数）

条件：a, b, c を計算する間 i, j の値に変化無し

75

共通部分の再利用



条件：
全てのブロックで i+j を計算
a, b, c を計算する間 i, j の値に変化無し

76

定数伝播

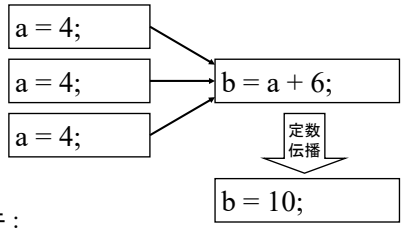
■ 定数を代入した変数は定数として扱う

```
a = 4;          a = 4;
b = a + 6;      b = 10;
```

条件：
b を計算する前に必ず a に定数を代入
a の値に変化無し

77

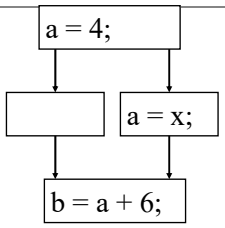
定数伝播



条件：
全てのブロックで a に同一の定数を代入

78

定数伝播

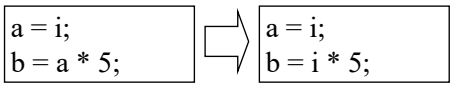


a に定数 4 以外が入るルートがあるので
定数とは見做せない

79

複写伝播

- 値をコピーした変数は同一として扱う

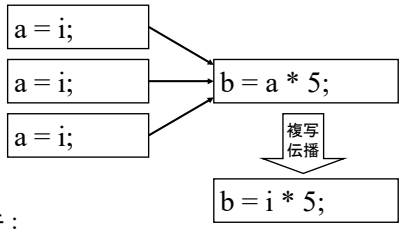


利点 : a への代入が不要になる場合がある

条件 :
b を計算する前に必ず a に i をコピー
i の値に変化無し

80

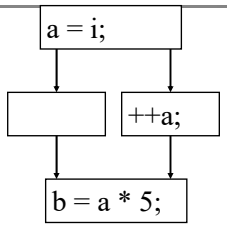
複写伝播



条件 :
全てのブロックで a に i をコピー
i の値に変化無し

81

複写伝播

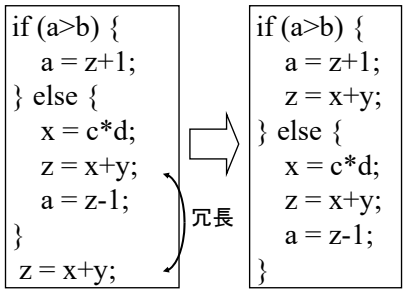


a の値が変更されるルートがあるので
a は i の複写とは見做せない

82

部分冗長性の削除

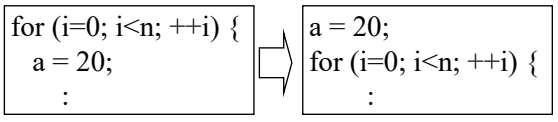
- 部分冗長性の削除



83

実行頻度の少ない場所に移動

- ループ内変数をループ外に



条件 :
a がループ内で不変 = 相対定数

84

実行頻度の少ない場所に移動

■ 制御変数の計算をループ外に

```
while (i < n*n) {
      :
```

⇨

```
t = n*n;
while (i < t) {
      :
```

条件：
n がループ内で不変 = 相対定数

85

実行頻度の少ない場所に移動

■ 誘導変数の強さ軽減

誘導変数：ループ時に定数だけ値が変わる変数

```
for (i=0; i<n; ++i) {
   j = i*10 + 5;
      :
```

⇨

```
j = -5;
for (i=0; i<n; ++i) {
   j += 10;
      :
```

j はループ毎に
値が 10 増える

86

ループ回数を減らす

■ ループ融合

```
for (i=0; i<n; ++i) {
   a[i] = c[i] + x;
}
for (i=0; i<n; ++i) {
   b[i] = c[i] + y;
}
```

⇨

```
for (i=0; i<n; ++i) {
   t = c[i];
   a[i] = t + x;
   b[i] = t + y;
}
```

利点：ループ制御命令の実行回数が半分
c[i] へのアクセスの時間局所性が利用可能

87

ループ回数を減らす

■ ループ展開

```
for (i=0; i<10; ++i) {
   a[i] = b[i] + x;
}
```

⇨

```
a[0] = b[0] + x;
a[1] = b[1] + x;
a[2] = b[2] + x;
a[3] = b[3] + x;
a[4] = b[4] + x;
a[5] = b[5] + x;
a[6] = b[6] + x;
a[7] = b[7] + x;
a[8] = b[8] + x;
a[9] = b[9] + x;
```

利点：ループ制御命令が不要
配列のアドレスをコンパイル時に計算可能

88

ループ回数を減らす

■ ループ展開

```
for (i=0; i<n; ++i) {
   a[i] = c[i] + x;
}
```

⇨

```
for (i=0; i<n; i+=2) {
   a[i] = c[i] + x;
   a[i+1] = c[i+1] + x;
}
```

利点：ループ制御命令の実行回数が半分
a[i] と a[i+1] の処理を並列実行可能

89

手続き・関数呼び出しの展開

■ 手続きの展開

```
{
  :
  func (i, j);
  :
}
func (int x, int y) {
  (x, y の処理)
}
```

⇨

```
{
  :
  (i, j の処理)
  :
}
```

利点：手続き呼び出し処理が不要

90

ハードウェア機能の利用

- レジスタの利用
- 局所性の利用
- ベクトル計算の利用
- 並列計算の利用

91

レジスタの利用

- レジスタ
  - CPUが直接演算可能 ⇒ メモリよりも高速
  - 容量はごく僅か

大域的なレジスタの利用
- 使用頻度が高いデータをレジスタに格納

⇒ データの使用頻度の解析が必要

局所的なレジスタの利用
- すぐに使うデータをレジスタに格納

⇒ データの生存区間の解析が必要

92

レジスタの利用

制御変数 i の値を繰り返し参照

制御変数の値をレジスタに格納

```
for (i=0; i<n; ++i) {
  a[i] = b[i] + x*(i+5);
}
```

⇨

```
for (R=0; R<n; ++R) {
  a[R] = b[R] + x*(R+5);
}
```

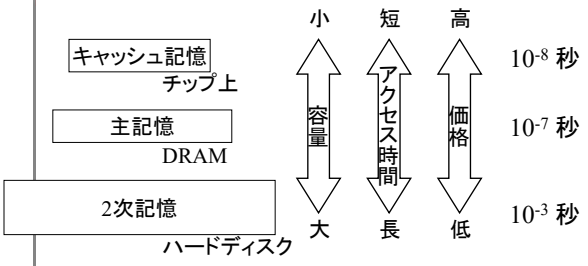
こちらの方が速い可能性も

```
t = x*4;
for (R=0; R<n; ++R) {
  a[R] = b[R] + (t += x);
}
```

93

メモリ

- メモリの記憶階層

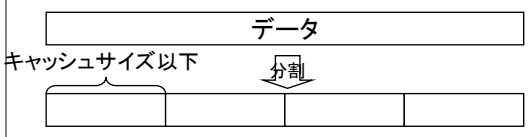


94

局所性の利用

- キャッシュメモリ
  - 高速にアクセス可能
  - 容量は小さい

⇒ データをキャッシュに収まるサイズに分割
分割したデータごとに処理



データのタイル化

95

データのタイル化

```
for (i=0; i<1000; ++i)
  for (j=0; j<1000; ++j)
    for (k=0; k<1000; ++k)
      c[i, j] += a[i,k] * b[k,j];
```

データサイズ
1000*1000*1000

⇒ キャッシュに入らない

```
for (x=0; x<1000; x+=10)
  for (y=0; y<1000; y+=10)
    for (z=0; z<1000; z+=10)
      for (i=x; i<x+10; ++i)
        for (j=y; j<y+10; ++j)
          for (k=z; k<z+10; ++k)
            c[i, j] += a[i,k] * b[k,j];
```

この内部ではデータサイズ 10*10*10

⇒ キャッシュ内で処理可能

96

ベクトル計算の利用

- ベクトル計算
  - 配列(ベクトル)の要素を並列に計算

```
int a[100], b[100], c[100];
a[0:99] = b[0:99] + c[0:99];
```

b[0]	b[1]	b[2]	b[3]	b[4]	...	b[99]
+	+	+	+	+		+
c[0]	c[1]	c[2]	c[3]	c[4]	...	c[99]
↓	↓	↓	↓	↓		↓
a[0]	a[1]	a[2]	a[3]	a[4]	...	a[99]

97

ベクトル計算の利用

```
for (i=0; i<1000; ++i) {
  a[i] = b[i] + c[i];
  e[i] = a[i] + f[i];
}
```

⇩

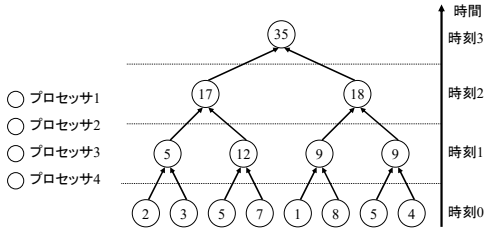
```
a[0:999] = b[0:999] + c[0:999];
e[0:999] = a[0:999] + f[0:999];
```

98

並列計算の利用

- 並列計算
  - 複数のプロセッサで命令を並列計算

```
s = 2+3+5+7+1+8+5+4;
```



99

並列計算の利用

```
a[0] = a[0]+a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7];
```

プロセッサ 0	プロセッサ 1	プロセッサ 2	プロセッサ 3
PUSH 0	PUSH 2	PUSH 4	PUSH 6
PUSH 1	PUSH 3	PUSH 5	PUSH 7
ADD	ADD	ADD	ADD
	POP 1		POP 5
PUSH 1		PUSH 5	
ADD		ADD	
		POP 1	
PUSH 1			
ADD			
POP 0			

100

参考プログラム

情報システムプロジェクトI 配布資料

projI22/material/kc/Kc22Opt.class

```
$ java kc.Kc22Opt bsort.k bsortOpt.asm
$ ./vsm bsortOpt.asm
```

diff でどこを最適化したか確認できる

```
$ java kc.Kc22Opt bsort.k bsortOpt.asm
$ java kc.Kc22 bsort.k bsort.asm
$ diff  bsortOpt.asm bsort.asm
```

101