プログラミング言語処理系論 (12)
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）

今日の予定

- データフロー方程式を利用したグローバル最適化
- SSA
- SSAを利用した最適化

具体例

- Reaching defintion
  gen[n] – definitions at n
  kill[n] -- definitions invalidated at n
  (variables reassigned at n)

- Liveness analysis
  gen[n] – uses at n
  kill[n] – definitions at n

Equations

- Reaching Definitions

$$\mathrm{In}[L] = \bigcup_{p \in L\text{のpredecessor}} \mathrm{Out}[p]$$

$$\mathrm{Out}[L] = \mathrm{gen}[L] \cup (\mathrm{in}[L] - \mathrm{kill}[L])$$

- Liveness Analysis

$$\mathrm{In}[L] = \mathrm{Gen}[L] \cup (\mathrm{Out}[L] - \mathrm{Kill}[L])$$

$$\mathrm{Out}[L] = \bigcup_{s \in L\text{のsuccessor}} \mathrm{In}[S]$$

様々なデータフロー方程式

- Available expressions
- Def. Available expressions
  (x op y)がポイントpでavailableであるとは、pに至るすべてのパスにおいて、(x op y)が計算されていて、かつその計算後にx, yに代入がないことを言う。
- gen[n] = (x op y)の計算
  kill[n] = n中のdefinitionsを含む式

- 方程式

$$\mathrm{in}[n] = \bigcap_{p \in n\text{の}\mathrm{predecessor}} \mathrm{out}[p]$$

$$\mathrm{out}[n] = \mathrm{gen}[n] \cup (\mathrm{in}[n] - \mathrm{kill}[n])$$

データフロー方程式の応用

- 方程式を解いて得られた情報を用いて行なう最適化のいくつか。

1. Dead Code Elimination

   もし、代入 a=...において、aがliveでなければ、この代入は不要 （liveness analysis）

データフロー方程式の応用(続き)

2. Constant Propagation
   t=c (c, constant)だけがreachしている
   s=…t…中のtはcに置き換えることができる。

3. Copy Propagation
   t=v だけがreachしている
   s=…t…中のtはvに置き換えることができる。

4. Common Subexpression Elimination

t=x op y の出現pointにおいて(x op y)がavailableであれば、この計算は削除できる（必要ならばtに計算済みの値を代入しておく）。

5. Global Value Numbering

各変数の参照 & quantityの割り当てにおいてReaching Definitionに対応するquantityが1つのみであれば、そのDefのquantityをそのまま使うことができる

Otherwise　新たにquantityを割り当てる

（1-4は、global value numberingを適用した最適化になっていることに注意する）

ここまで復習

課題17

- 以下のコードについてglobal value numberingを行え。以下の手続きを書けば、プログラムを書く必要はない。(プログラムを書けばなおよい)
  - 各quantityについてreaching definitionを明らかにすること
  - redundant code eliminationを行うこと。ただし、aからtは、このコードの後でも使用されうるとする
  - SSA変換をする必要はないが、しても構わない。

```
[a, b, c, d,i, m, s, t are given]

while (x < 100) {
 if (x == i) {
   i = c × b;
   m = i + 4;
   a = c;
 } else {
   d = c;
   i = d × b;
   s = a × b;
   t = s + 1;
 }
 x = a × b;
 y = x + 1;
}
print i;
```

課題17についてのNote

- 各変数のUseに対して、quantityを計算する
  - 当該変数のUseに対し、reaching definitionの集合Rを計算する
  - If (|R| == 1) { /* unique definition */
　そのdefのquantityをそのまま使う
else
　新たなquantityをそのUseに割り当てる

実際のコンパイラでは

- gcc/df-{core|problems|scan}.cc
  - データフロー方程式の一般的なソルバーの実装
  - 基本のルーチンはUD-chain, Reaching Definition, Liveness Analysis
  - その結果を利用して各種最適化ルーチンを書く

df-problems.ccの104行目以降

```
/*----------------------------------------------------------------------------
   REACHING DEFINITIONS

   Find the locations in the function where each definition site for a
   pseudo reaches.  In and out bitvectors are built for each basic
   block.  The id field in the ref is used to index into these sets.
   See df.h for details.

   If the DF_RD_PRUNE_DEAD_DEFS changeable flag is set, only DEFs reaching
   existing uses are included in the global reaching DEFs set, or in other
   words only DEFs that are still live.  This is a kind of pruned version
   of the traditional reaching definitions problem that is much less
   complex to compute and produces enough information to compute UD-chains.
   In this context, live must be interpreted in the DF_LR sense: Uses that
   are upward exposed but maybe not initialized on all paths through the
   CFG.  For a USE that is not reached by a DEF on all paths, we still want
   to make those DEFs that do reach the USE visible, and pruning based on
   DF_LIVE would make that impossible.
   ----------------------------------------------------------------------------*/
```

課題18

- GCCの最適化ルーチンではdf-core.c, df-problems.c, df-scan.cで一般的なソルバーを提供している。このルーチン群を解析し、それらを利用して実際にどのような問題が解かれているか列挙せよ。そのうち、reaching definitionを例にとり、gen, kill, out, inがどのように表現されているか記述せよ。

実際、gcc 12.1.0では

- 以下のルーチンが登録されている
  - df_add_problem (&problem_RD);
  - df_add_problem (&problem_LR);
  - df_add_problem (&problem_LIVE);
  - df_add_problem (&problem_MIR);
  - df_add_problem (&problem_CHAIN);
  - df_add_problem (&problem_WORD_LR);
  - df_add_problem (&problem_NOTE);
  - df_add_problem (&problem_MD);
  - df_add_problem (&problem_SCAN);

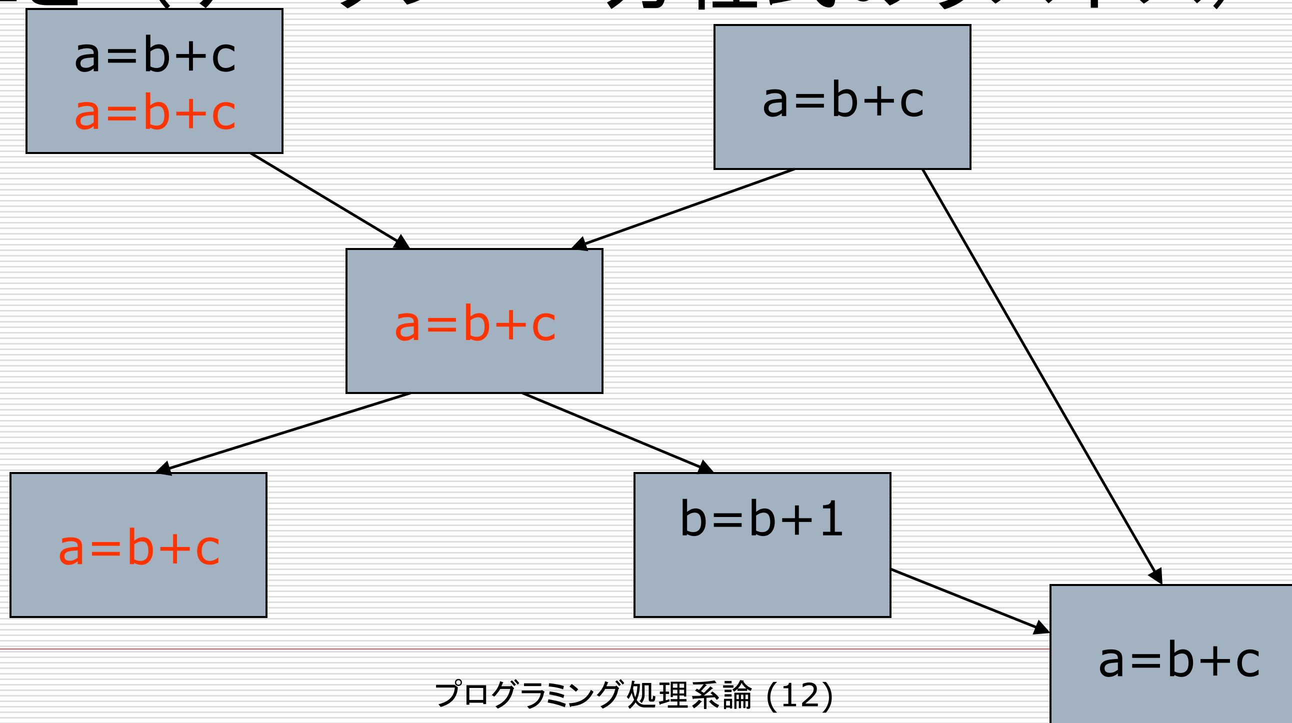
説明は以下の通り(df.h)

```
enum df_problem_id
  {
    DF_SCAN,
    DF_LR,                    /* Live Registers backward. */
    DF_LIVE,                  /* Live Registers & Uninitialized Registers */
    DF_RD,                    /* Reaching Defs. */
    DF_CHAIN,                 /* Def-Use and/or Use-Def Chains. */
    DF_WORD_LR,               /* Subreg tracking lr.  */
    DF_NOTE,                  /* REG_DEAD and REG_UNUSED notes.  */
    DF_MD,                    /* Multiple Definitions. */
    DF_MIR,                   /* Must-initialized Registers.  */

    DF_LAST_PROBLEM_PLUS1
  };
```

(Partial) Redundancy Elimination

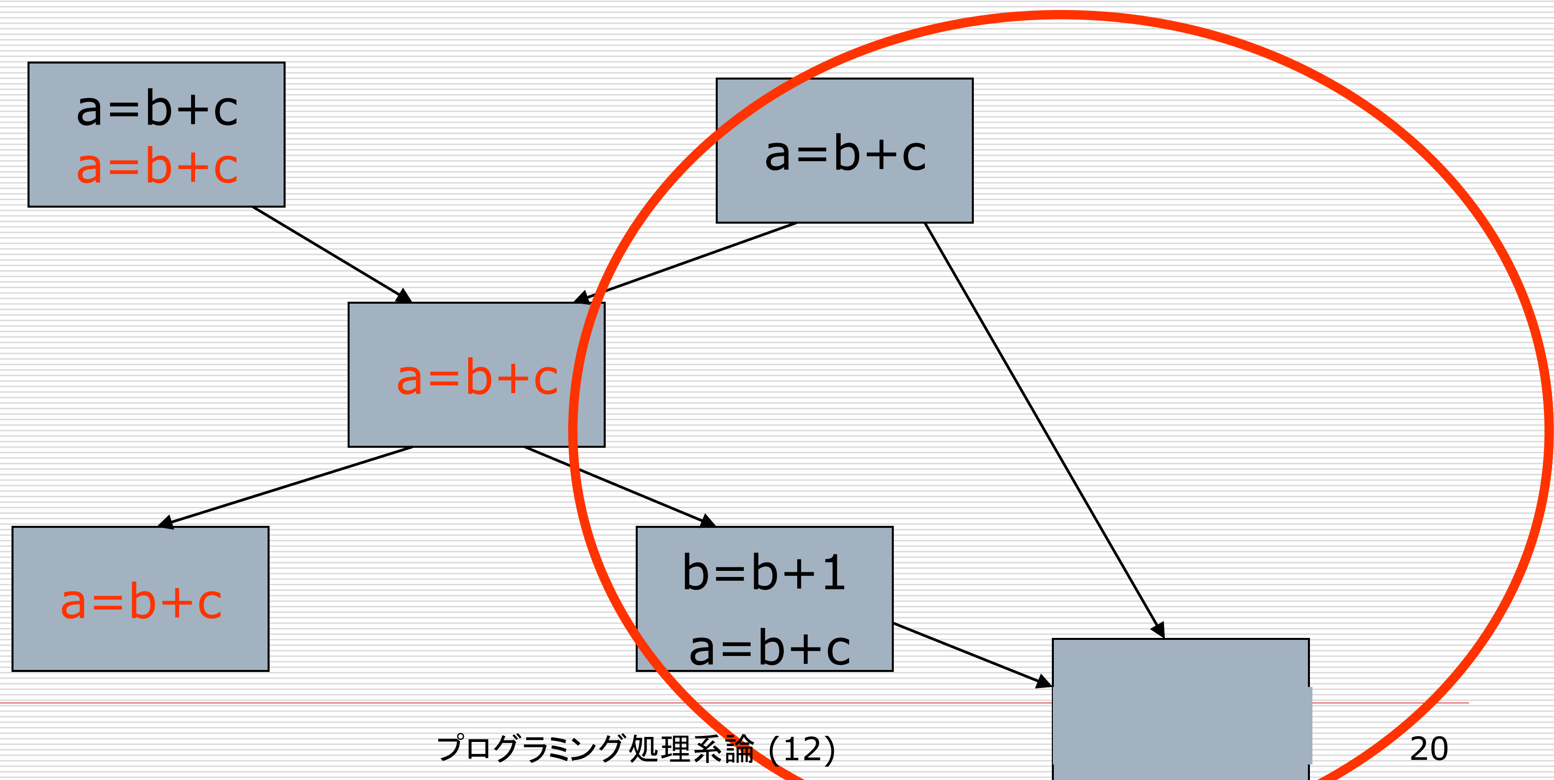
- Def. Redundancy Elimination
  計算のパスにおいて、重複する計算を取り除くこと（データフロー方程式のラスボス）



- Def. Partially Redundant
式がプログラムポイントでpartially redundantであるとは、そこにいたるパスの中でredundant expressionな式であるものが存在すること(すべてのパスである必要はない)
- たとえば、ループ不変式はpartially redundantである

Partial Redundancy

- （赤字の除去、一部コードの移動）



- Def. Lazy Code Motion
  Partially Redundant Eliminationのためにコードを移動すること

- Knoop, et.al., Lazy Code Motion, Programming Language Design and Implementation ’92, 1992.

- Def. Anticipable Expressions
  式(x op y)がanticipableとは、その計算が後続のパスのどこかでなされ、しかもそのパスの中にx, yの定義が入らないこと（計算を先に実行してもよい）。
- $\mathrm{ANTout}[n] = \bigcap_{s \in n\text{のsuccessor}} \mathrm{ANTin}[s]$

  $\mathrm{ANTin}[n] = \mathrm{Defed}[n] \cup (\mathrm{ANTout}[n] - \mathrm{kill}[n])$

Lazy Code Motion

- Def. Earliest(i,j)　(i,j – ブロック)
  (1)ブロックjの先頭まで移動でき、
  (2)ブロックiの終端ではavailableでなく(よって、挿入してもredundantにならない)、
  (3)ブロックiの先頭には移動できない
  式の全体 → エッジ(i,j)に挿入できる

- Earliest(i,j) = ANTin(j) – AVAILout(i)
  ∩ (Kill(i) – ANTout(i))

- Def. e ∈LaterIn(k)
  kに到達するパスすべてについて、そのどこかでeがEarliestであり、そこからkまで、eを評価していない。(eの計算はkから前に移すことができる)
- Def. Later(i,j)
  Earliestであるか、またはiから後ろに移動でき、かつiの先頭に移動するとiではanticipableでない式の全体
- Later(i,j) = Earliest(i,j)∪(LaterIn(i)–Defed(i))
- LaterIn(j) = $\bigcap_{i \in predecessor(j)}$ Later(i,j)

- INSERT(i,j) = Later(i,j)– LaterIn(j)
- Delete(k) = Defed(k) – LaterIn(k)

ここで反省

- データフロー方程式で集める情報の基本はDefinitionとUseに関するもの。
- Def Def/Use Chain
- 似たような事を何回もやっている。
- Def/Use Chainを効率的に表現できる中間表現があれば、この種の最適化も効率的にできるはずである。
- → SSA

データフロー解析で困ったこと…

- Reaching definitionってそもそも方程式をたてなければわからないものか？
- 同じ変数にとっかえひっかえ値を代入すると、冗長性の解析でも無駄に複雑度が増す

- 中間言語ならば、変数の数は任意に取れるから、使い回しを気にする必要はない

Motivating Example: Reaching Definition （再掲）

- Def. Definition of t
  t への値の代入のこと
- Def. Use of t
  tの値を使うこと
- Def. Reaching Definition
  Definition *d* (of *t*) reaches a statement *u* if there is a path from d to u that does not contain any definition of *t*

- IBMが中間言語としてSSA（Static Single Assignment)を提唱
  - Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, Oct 1991.
  - 開発は1980年代といわれている
  - 「SSAはIBMが作った関数型言語だ！」
A. Appel, “SSA is functional programming,” ACM SIGPLAN Notices 33(4):17—20, April 1998.

定義

- Def.　Static Single Assignment Form
プログラムの文面上、各変数に対して定義がひとつしか存在しないもの
- 注意：プログラムの文面上の話だから、同じ文が何回も実行されて、定義が複数回行なわれることは禁止しない

例

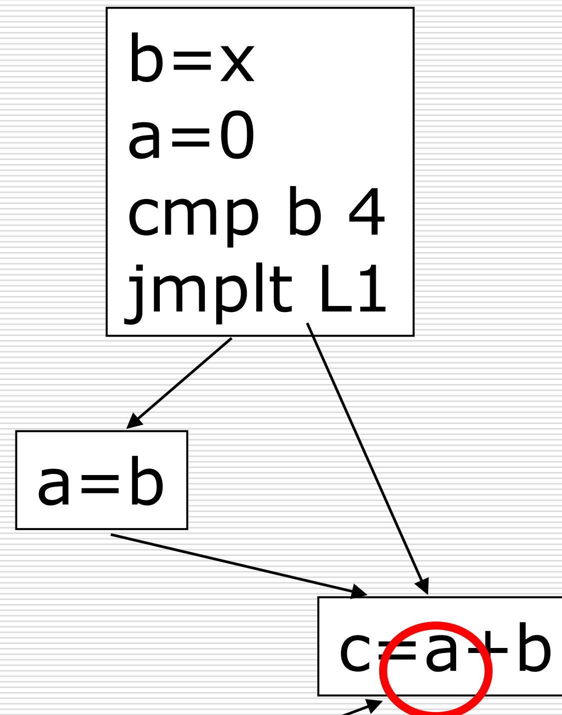
```
a=x+y
b=a-1
a=y+b
b=x*4
a=a+b
```

```
a1=x+y
b1=a1-1
a2=y+b1
b2=x*4
a3=a2+b2
```

(変数の名前の書き換え)

どうしても対応するSSAがない例

```
  b=x
  a=0
  cmp b 4
  jmplt L1
  a=b
L1:
  c=a+b
```



このaはどこから
きたのか？

言語の拡張

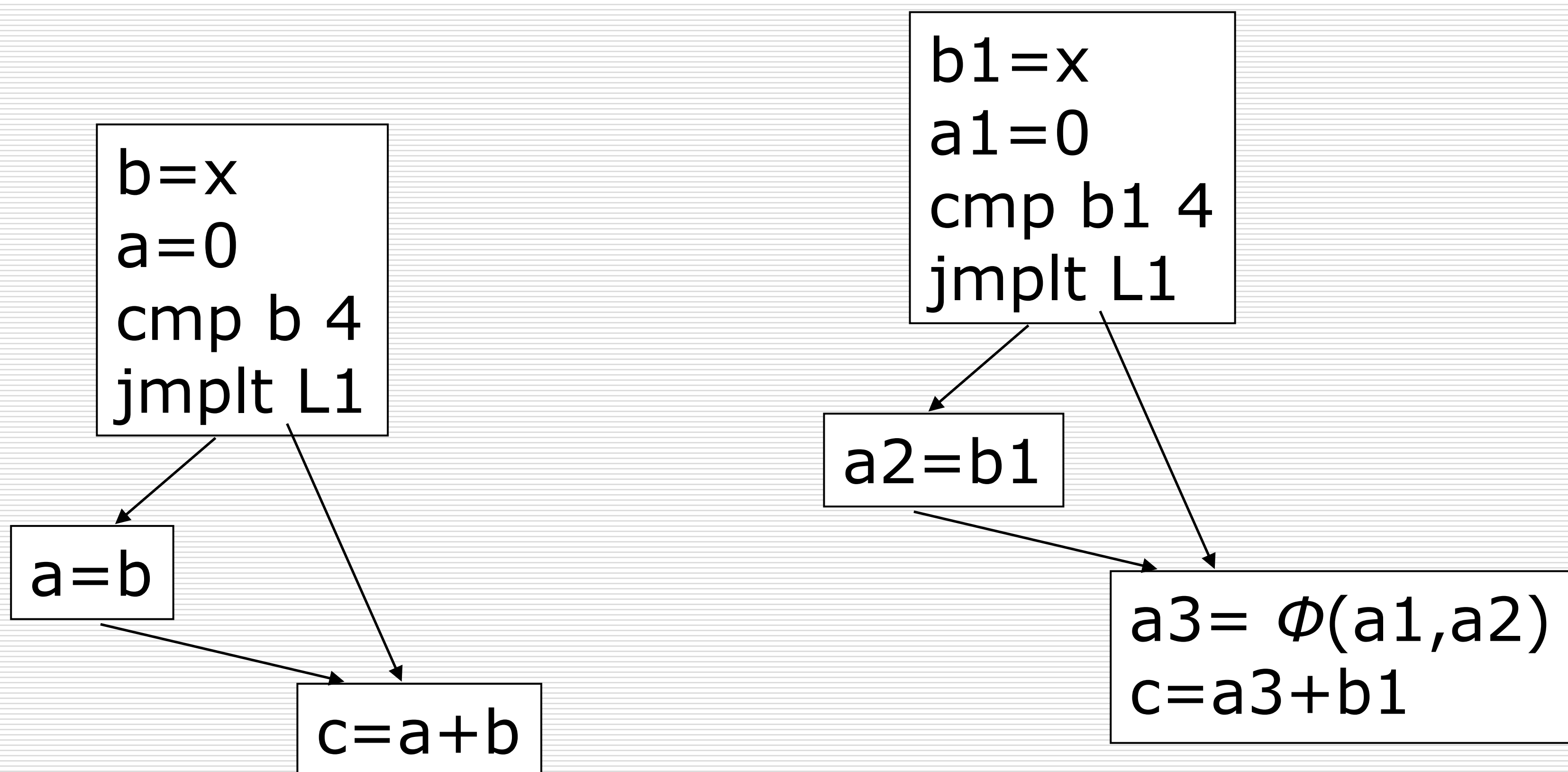
- 今までの言語に以下を追加する

INSN ::= …
　　　　　Φ(var ‘,’ var+)

Φ（pseudo）関数は、以下の意味をもつとする。
 x = Φ(x1,x2)

制御がx1の定義を通ってこの場所に来た場合はx1を、
　そうでない場合はx2を表す。

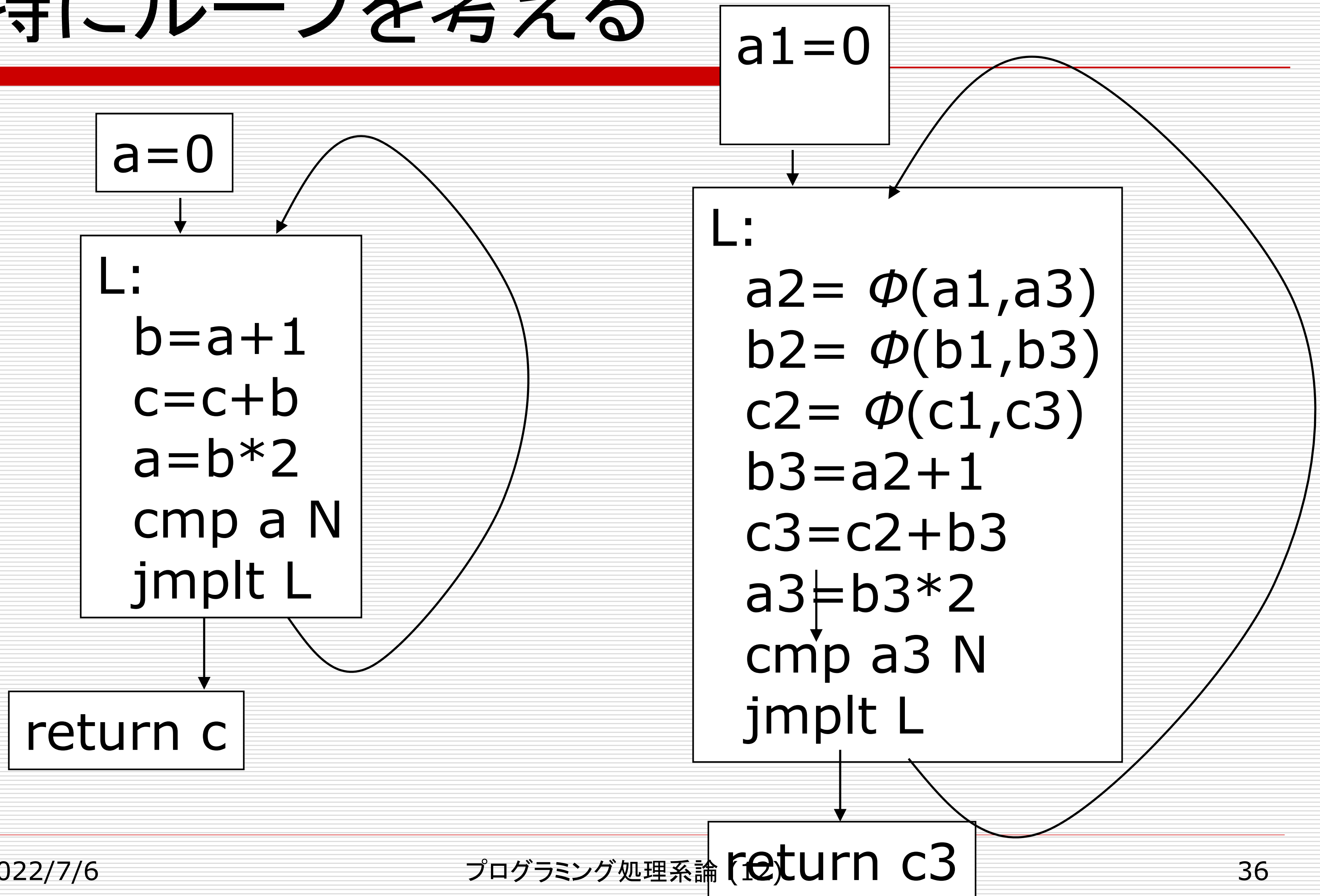
Φを使っての解決



SSAの利点

- Def/Use関係の簡潔な表現　→　さまざまな解析の簡潔化
- データフロー解析の大域化（killされるものがないので、ローカルなものとグローバルなものを分ける必要がなくなる）　→　ローカルな最適化の大域的なところでの適用（e.g. global value numbering）

特にループを考える



- 基本的な要請:
  （1）プログラムのフローは両者で同じでなければならない
  （2）計算の終了後、両者の計算の結果は、変数名を除いて同一でなければならない
  （3）任意のSSA形式は、Φを持たないプログラムで、同一の効果を持つものを持たなければならない

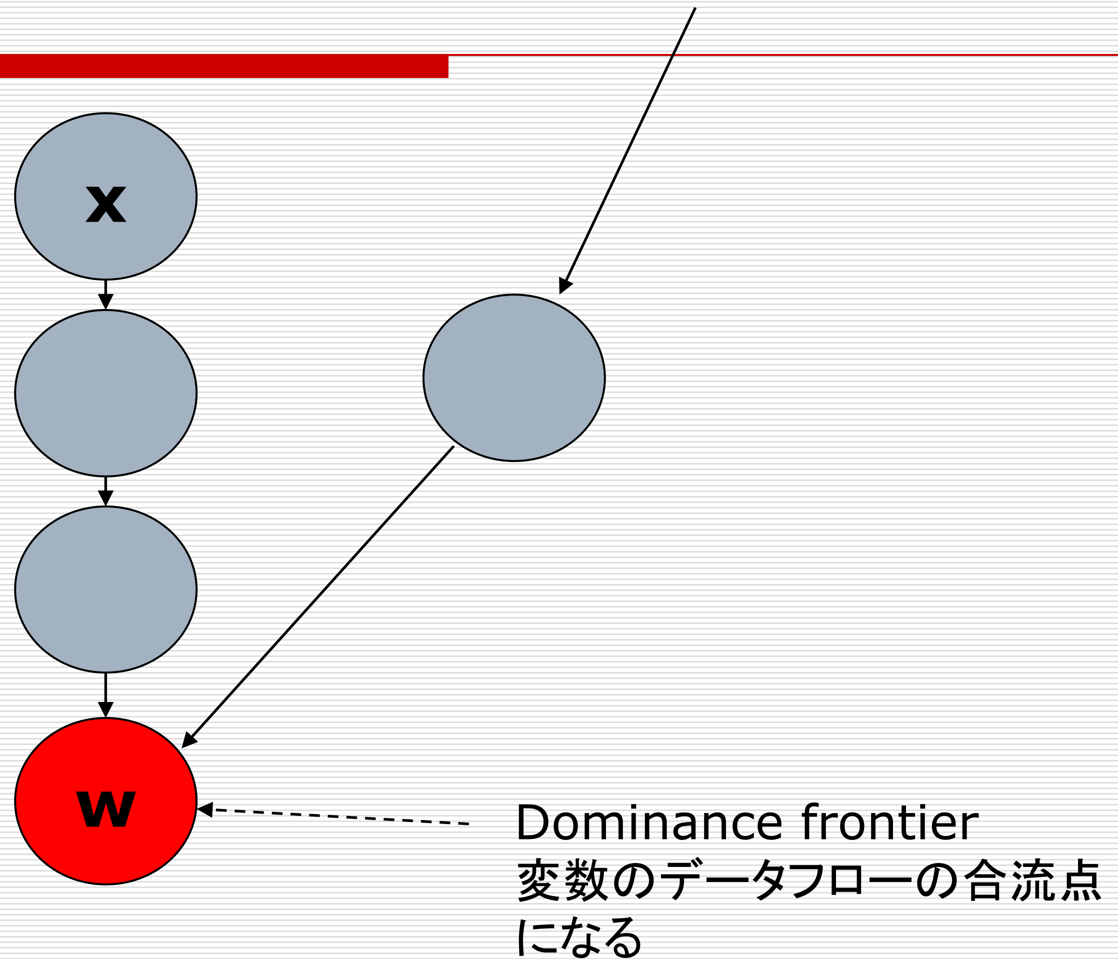
SSA変換・SSA逆変換

- SSA形式に変換するとは、Φをどこに挿入するかに（だいたい）帰着される。
- 変数が（データフロー上）合流する点にΦを挿入すればよい。
- 変数aのΦ関数をブロックzに挿入するための方針:
  （1）zにおいて、変数aのreaching definitionsが複数あり、
  （2）zにおいてそれら複数の定義が初めて合流する

効率的な方法

- Def. x strictly dominates w if
  x dominates w and x ≠ w
- Def. Dominance Frontier of a node x
  set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.
- DF[x] = {z| ∃y1,y2 ∈predecessor(z).
  x dominates y1, and
  x does not dominate y2.}

Illustrated



- Dominance Frontierの計算
- Def. DF[n]　　　　n– node
- DF[n] = {z|zはnのsuccessor, かつnにdominateされていない}

$$\cup \bigcup_{idom(c) = n} DF[c]$$

- DFは、次のようにして効率的に計算できる

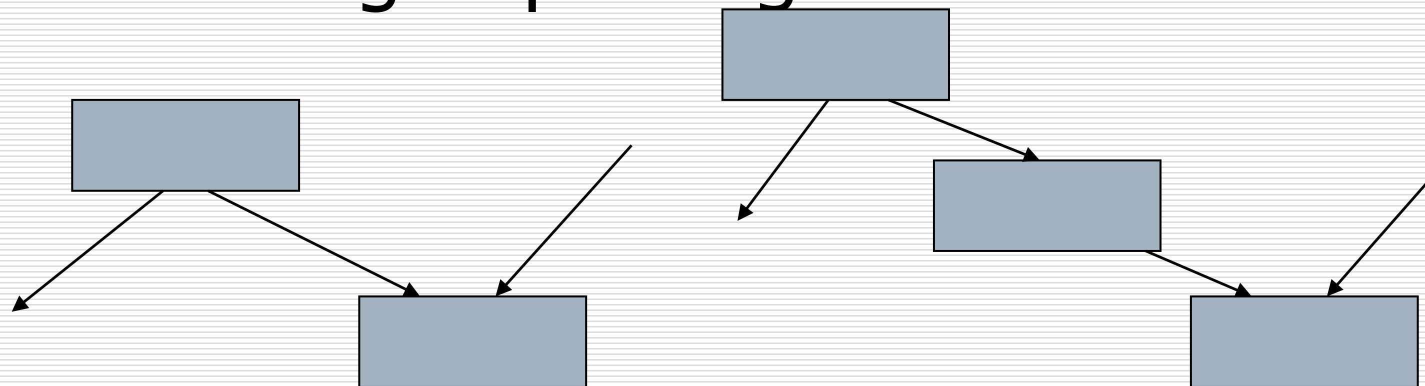
```
S = Φ
for each node y in successor[n]
    if (idom(y) ≠ n)
            S = S ∪{y}
for each child c imm. dominated by n
    compute DF[c]
    for each element w ∈ DF[c]
            if n does not dominate w
                    S = S ∪{w}
DF[n] = S
```

Φの挿入（Last Stage）

```
for each node n
    for each variable a (defined in n)
        defsite(a) = defsite(a) + n
for each variable a s.t. defsite(a) ≠ Φ
    for each n ∈ defsite(a)
        for each Y ∈ DF[n]
            insert a= Φ(a,a,…)
```

残りの仕事

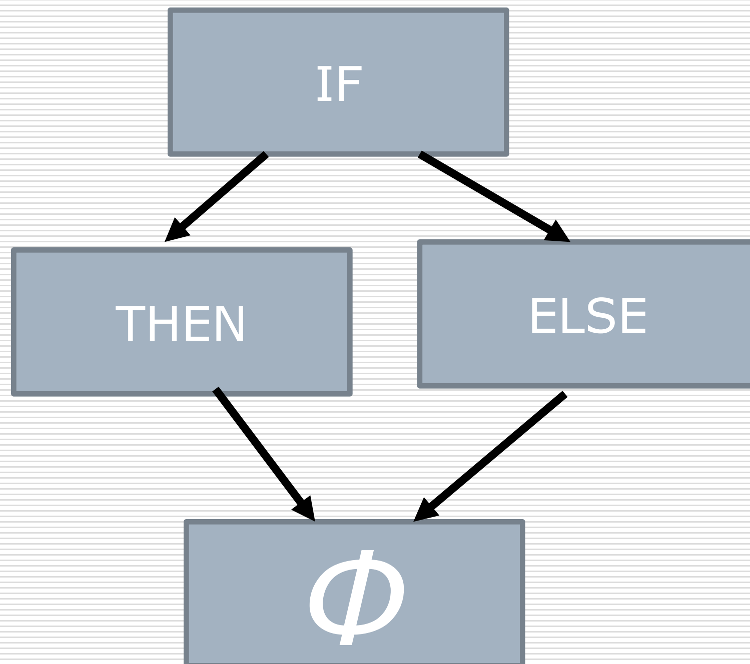
- Renaming variables.
  - Dominator treeを上から下に下りていって、変数名をx → xnの形にしていく(どのdefがreaching definitionであるかは簡単にわかる)。
- Critical Edge splitting.

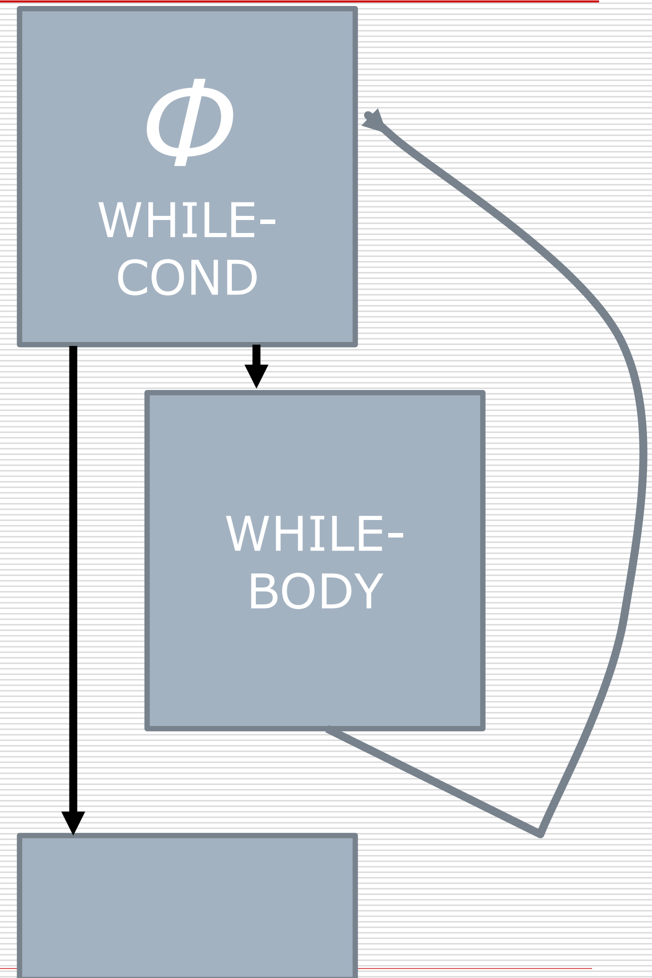


- Critical edge
  - これがあると、SSA逆変換がとたんに難しくなる
  - これがあると、Lazy Code Motionができない

シンプルな制御構造を持っている時

- DFを計算しなくても...





SSAを用いた最適化

- われわれは、SSAを作るときに以下のものを同時につくることができる。
  - DEF/USE chain and USE/DEF chainを、DEFひとつに対するUSEの集合として管理できる。
  - SSAでは、変数の管理表は次のようになるだろう。

```
{…
  var *def;
  var_list use;
…
}
```

- Dead Code Elimination
  - v= x op y において、vのUSEが0であれば、この文は削除できる。
- Constant Propagation
  - v = x において、xのDEFが定数 x = cであればこれをv = cに置き換えることができる。
  - v = Φ(c1,c2,…,cn)で、c1,…cnがすべて同じであればv=c1に置き換えることができる。

- Copy Propagation
- Constant Folding
- Constant Conditions によるjump文の最適化
- Unreachable Codeの除去による unreachable blockの除去

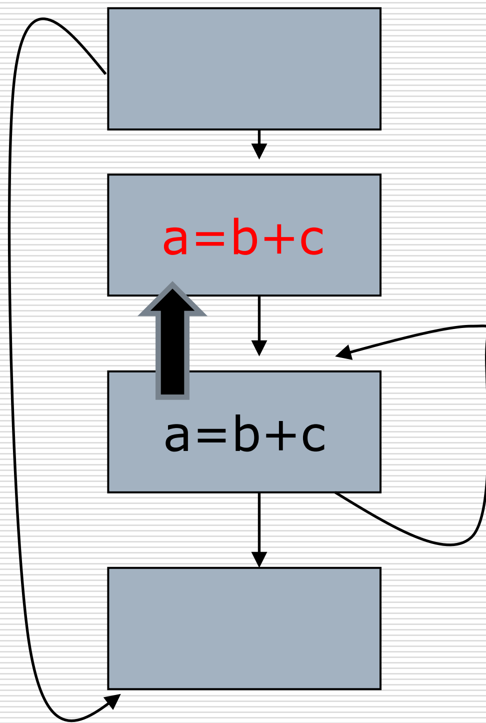
Loop Invariant

- Def Loop Invariant
  当該loop中での式の計算が常に同じ値を生むとき、その計算をloop invariantという。
- 例:
  L1
  cmp x,y
  jmplt L2
  a=1
  x=x+1
  jmp L1

- ループ中に式 t=a1 op a2があるとせよ。次の条件が満たされるとき、この計算はloop invariantである。
(*) aiのDefがループの外側にある。

Loop invariant hoisting

- Loop invariantはそのループの外に出せる



方法論はPREに含まれる。

PREの一環でなくても、Defがループの外にあるかどうかはSSA変換されているなら一瞬でわかる

左のようにしても、peephole最適化をして、ループの形を変えることができる。

注意

- よくやる間違い（b+cが必ず実行される保証はない）
- 演算が例外を起こさない場合適用可

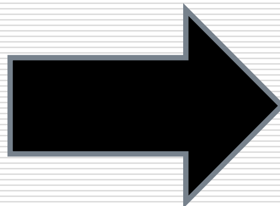
```
L2:
   cmp x, y
   jmplt L3
   …
   a=b+c
   …
   jmp L2
L3:
```

```
  a=b+c
L2:
  cmp x,y
  jmplt L3
…
  jmp L2
L3:
```

一般解(do while文への変換)

```
while (cond) {
  body
}
```

(loop invariant hoistingやinduction variableを用いた最適化に注意が必要)



```
if (!cond) {
} else {
  do {
      body
  } while (cond)
}
```

(else内なら、loop invariant hoistiingを自由に適用化)

Induction variables

- Def induction variables x
  In every loop iteration, x's value is incremented/decremented in constants.
- 例:
  L1:

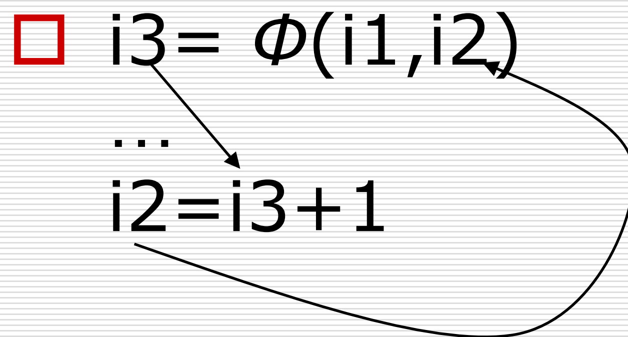
```
   i=i+1
   …
   goto L1
```

Induction variablesの例

- ループのインデックス変数 (basic induction variables)
- Basic induction variablesから計算されるもの (derived induction variables)
  - 配列のアドレス計算
    a[i] → a + i * 4
  - こちらは結構重要

Induction variable detection

- SSAを仮定する。ループ中から変数のDefをたどっていく。

- 

- たどっていって、Φのうち、バックエッジから来るDefが、自分自身に定数を足したもの、もしくはそれと等価なものであればそれはbasic induction variableである。

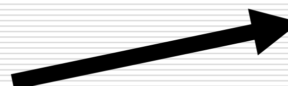
- Derived induction variable
  - Basic induction variableに線形計算を施して得られたものはderived induction variableである。

Induction variableの最適化

- 掛け算は足し算にできる(strength reduction)

```
for (i=0; i<N;i++) {
  a[i] = b[i+1];
}
```

```
for (i=0; i<N;i++) {
  p = a+4*i;
  q = b+4*i+1;
  *p = *q;
}
```



```
for (i=0, p=a, q=b+1; i<N; i++) {
  *p = *q;
   p = p+4;
   q = q+4;
}
```

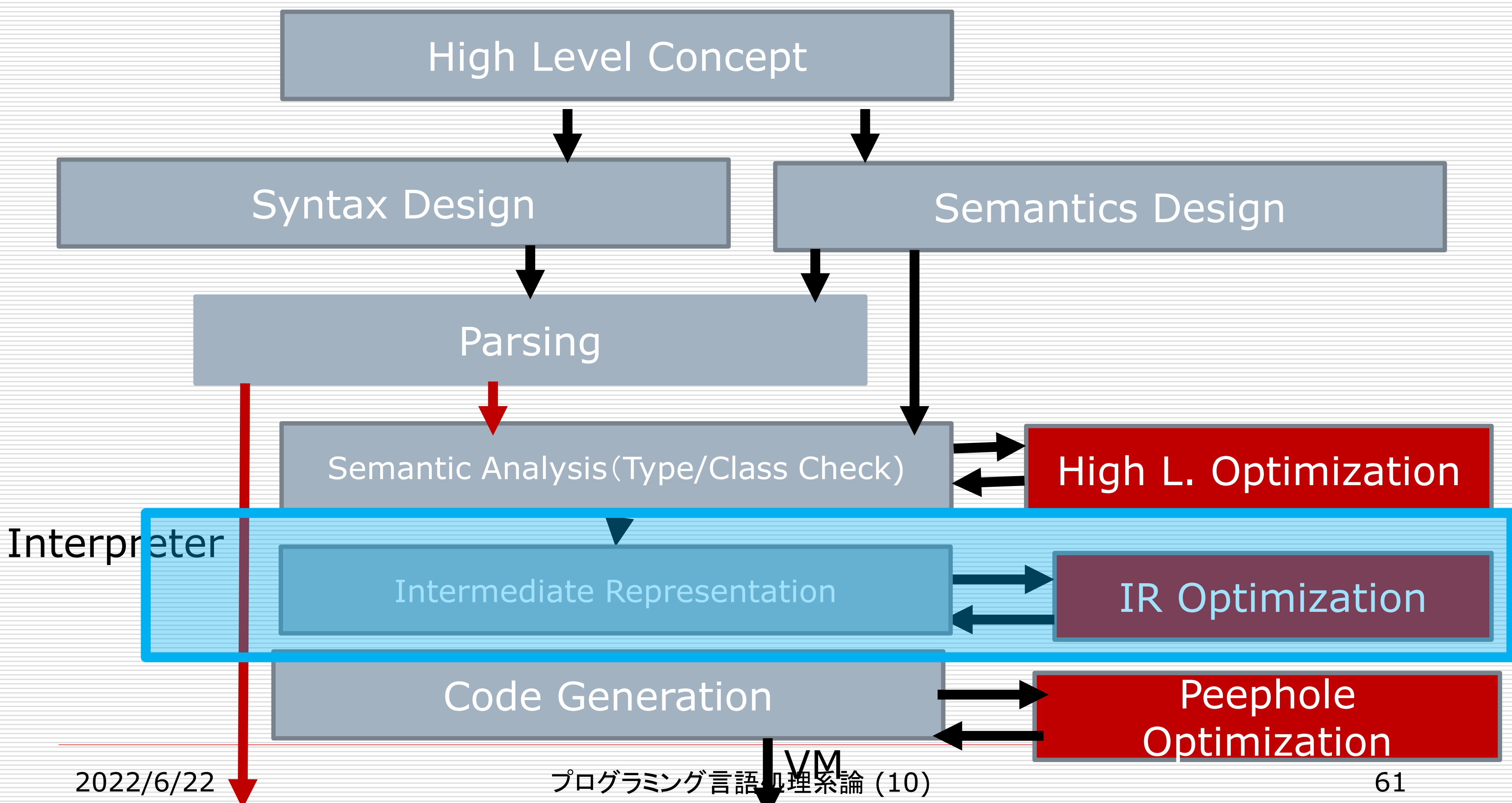
課題19

(1) 説明されたLoop Invariant Hoistingの概要からアルゴリズムを導き出し、SSAを用いて表現せよ

(2) SSAを用いてDerived Induction Variableを見つけるアルゴリズムを考えよ。Strength Reductionを適用できるとなお良い。

(3) (2)に関し、配列のデータを順にアクセスするときの配列のインデックスの計算の方法が手近にあるコンパイラでどのように実装されているか調査せよ。

で、SSAはどこに入るべきなのか？



- 真ん中のIRがきちんと定義されていないと難しい
- GCCでは、GIMPLEがSSAを採用した中間表現として採用されている
  - RTLより上のレベル

SSAの拡張

- Scalar変数からポインタやarrayに
- ポインタの解析と共存させる
  - Extended SSA numbering: introducing SSA properties to languages with multi-level pointers, Christopher Lapkowski, Laurie J. Hendren, CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research
  - Practical Integrated Analysis of Pointers, Dataflow and Control Flow, Stefan Staiger-Stöhr, TOPLAS , 35-1, 2013.

- Arrayの解析と共存させる
  - Region array SSA, Silvius Rus, Guobin He, Christophe Alias, Lawrence Rauchwerger, PACT '06, 2006
  - Array SSA form and its use in parallelization, Kathleen Knobe, Vivek Sarkar, POPL '98, 1998.
  - Embedding Dataflow Information on Arrays into SSA and Extended Optimization Schema for Parallelization, Sato, H., Proc. Parallel and Distributed Computing and Networks 2009.
- ここらへんは、できるものはできる、できないものはできないと割り切るのが吉
  - 少なくともScript言語はターゲットにならないでしょう
  - 環境が激変すると、また新しいアイデアが出てくるかもしれない
  - 量子関係とかね