ソフトウェア工学6－2
（1978. 5. 31）

SIMPL プログラムのデータ・フロー解析とその応用

原田 賢一（慶応義塾大学・情報科学研究所）

1. はじめに

データ・フロー解析は，プログラムにおける制御の流れに沿って，データの定義と参照の関係を明らかにすることを目的としている．その結果は，プログラムの文書化，最適コードの生成，プログラムのデバッグと改良に利用されている．本稿では，SIMPLと呼ばれる単純なGoto-lessプログラミング言語を対象としたデータ・フロー解析について述べ，最適コード生成への利用を考える．

データ・フロー解析によって収集される情報は次のように分けることができる．

(1) 変数の参照があった場合，その変数の値はプログラムのどこで定義されたものが影響しているかを調べる．

(2) 制御の流れの各点において，どの変数があとの実行に影響を与えるかを調べる．

前者は reach analysis（到達可能な変数の解析），後者は live variable analysis（live変数の解析）と呼ばれている．これらの解析には，通常次のような方法がとられている．

(1) フロー・グラフの作成

与えられたプログラムを基本ブロック（basic block）という単位に分割し，制御の流れを基本ブロック間の関係で表わす．基本ブロックは逐次的に実行される命令列で，制御は必ず先頭の命令に与えられ，最後に置かれた命令群の実行によって，制御は他のブロックに移る．基本ブロックを節（N）とし，制御の流れを辺（E）で表わしたものをフロー・グラフ G=(N,E) という．

(2) インターバル法

フロー・グラフをインターバルと呼ばれる単位で分割し，各インターバル内で目的とする情報を収集する．与えられたグラフが単一の節になるまで，この操作を繰返す．次に，一点に収約された情報をインターバルごとに分配していく[1-4]．

解析法としては，インターバル法の他に繰返し法（Iterative method）がある[5]．繰返し法は，フロー・グラフを制御の流れに沿って巡回しながら，各節に情報を収集していき，各節のもつ情報が収束するまで繰返すという方法である．

ここで述べる方法は[6]，Goto-lessプログラムでは良形の制御構造をもつ性質を利用して，ソース・プログラムのレベル（フロー・グラフは作らないで）で名文を走査しながら解析をしていこうというものである．プログラムを制御の流れに沿って，順方向と逆方向とで2回走査することによって結果を得ることができる．以下，live変数の解析について述べる．

2. Live 変数の解析

1つの手続きに対する流れ図を考えたとき，制御がその中のある点Pを過ぎて出口に到達するまでの間に，変数vが定義されることなしに参照されることがある場合，vはPにおいてlive（busy）であるという．また，Pから出口に達する経路上で，vが全く参照されないか，あるいはすべての経路上でvが定義（vへの代入がある）されるとき，vはPにおいてdead（free）であるという．Live変数の解析の目的は，手続きの各文，あるいは各基本ブロック（以下，単にブロックという）の入口と出口における live変数の集合を求めることである．

ブロックBの入口と出口それぞれにおけるlive変数の集合をIN(B)，OUT(B)とする（図1）．このとき，次の関係が成立つ．

$$OUT(B) = IN(S_1) \cup IN(S_2) \cup \cdots \cup IN(S_m) = \bigcup_{i=1}^{m} IN(S_i) \quad (1)$$

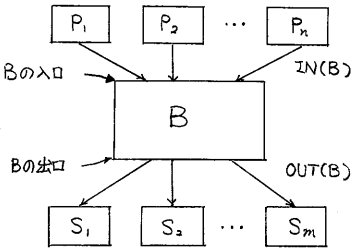


図 1

ブロックBを実行したときに，変数vの値が定値されないか，あるいはvの参照の方が定義よりも先きに行なわれるとき，vはBにおいてdefinition free（D-free）であるという．そのような変数の集合をDFR(B)で表わす．また，Bの実行によって参照されることのある変数の集合をREF(B)で表わす．ただし，B内で定義された値だけを参照するような変数はREF(B)に含めない．Bの入口でbusyな変数は，REF(B)の要素，あるいはBの出口でbusyでかつBではD-freeなものである．

$$IN(B) = REF(B) \cup [OUT(B) \cap DFR(B)] = REF(B) \cup [[\bigcup_{i=1}^{m} IN(S_i)] \cap DFR(B)] \quad (2)$$

手続きの出口Eでは，すべての変数はfree，$IN(E) = \phi$を仮定する．ここで述べるlive変数の解析法は，与えられたプログラムを，recursive descentな構文解析法と同じ順序で文を走査しながら，まず各文におけるD-free変数を求める．つぎに，プログラムの出口から入口に向って，各文におけるlive変数を求めていく．SIMPL言語の概要を図2に示す[7]．

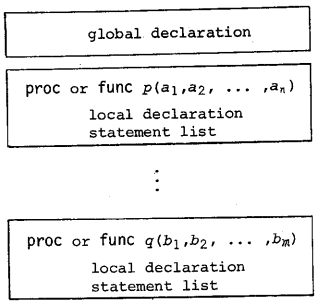
2.1 D-free変数の計算

(1) 基本文

与えられたブロックBが read，write，代入文のようなときには，単に文を走査することによって，REF(B)が求まっているものとする．また，Bの実行によって代入が行なわれる変数の集合DEF(B)も求まっているものとする．

$$DFR(B) = REF(B) \cup [V_P - DEF(B)]$$

V_Pは，解析の対象である手続きPで使用される局所的変数，および非局所的変数の集合である．



(a) general form of a SIMPL program

Assignment statement: v:=e

If statement: if e then S_1 [else S_2] end

Case statement:
case e of
\a_1\ S_1
\a_2\ S_2
⋮
\a_n\ S_n
[else S_{n+1}]
end

While statement: [\l\] while e do S end

Repeat statement: [\l\] repeat S until e end

For statement: [\l\] for v:=e_1 [step e_2] until e_3
do S end

Call statement: call p[(b_1,b_2,... ,b_m)]

Exit statement: exit [(l)]

Return statement: return [(e)]

v: identifier or array element, e, e_1, e_2, e_3: expression, S_1,S_2,... : statement list, a_1,a_2, ... : integer constant or character constant, l: label, p: procedure name, b_1,b_2, ... : actual parameter [ ] means optional syntax.

(b) SIMPL statements

図2 SIMPL 言語

(2) 文の並び

図3のような構造を考える。

$$REF_i(B) = REF_{i-1}(B) \cup REF(B_i),\ i=1,2,\cdots,n$$

とする。ただし、 $REF_0(B) = \phi$

$$DFR(B) = \bigcap_{i=1}^{n} [REF_{i-1}(B) \cup DFR(B_i)],$$

$$REF(B) = REF_n(B) \cap DFR(B).$$

B内の各ブロック B_i, $i=1,2,\cdots,n$ について、まず $DFR(B_i)$ と $REF(B_i)$ を求め、それから、この計算を行なう。

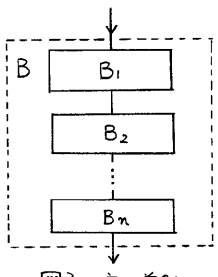


図3 文の並び

(3) if文

図4のような構造を考える。

$$DFR(B) = REF(e) \cup DFR(B_1) \cup DFR(B_2),$$

$$REF(B) = REF(e) \cup REF(B_1) \cup REF(B_2)$$

B_1, B_2 のそれぞれについて、DFR, REF を求めてから、上の計算をする。

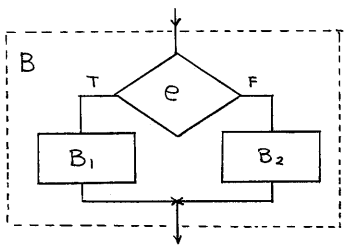


図4 if文

(4) while文

B: while (e) Bw end

としたとき、Bw を1度も実行しなかったとすると、すべての変数は D-free となるから、

$$DFR(B) = V_P,$$

$$REF(B) = REF(e) \cup REF(B_W)$$

(5) repeat文

B: repeat B_R until e

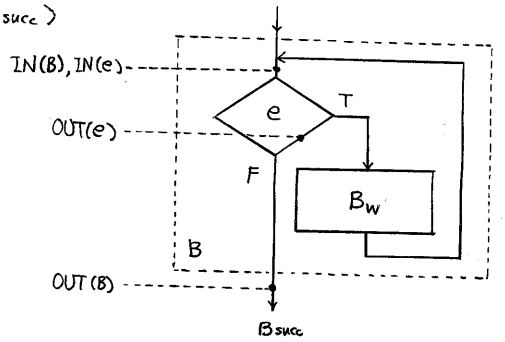
$$DFR(B) = DFR(B_R),$$

$$REF(B) = REF(B_R) \cup [REF(e) \cap DFR(B_R)]$$

2.2 Live変数の計算

手続きを構成しているすべての文について，DFRとREFが求まったら，関係式(2)を用いてlive変数を計算することができる．そのために，次のような手続き LVIO(B, B_{succ}) を考える．Bは文の並び，または文で，B_{succ} はBの直接後続ブロックとする．この手続きは，B_{succ} における IN(B_{succ}) が求まっていたときに，それを用いて，IN(B) および OUT(B) を計算するものとする．

```
proc  LVIO(B, Bsucc)
  if B is return block  then  IN(B) = φ  OUT(B) = φ  return end
  OUT(B) = IN(Bsucc)
  IN(B) = REF(B) ∪ [OUT(B) ∩ DFR(B)]
  case type of B of
    statement list →   for i := n, n-1, ···, 2, 1 do call LVIO(Bi, Bi+1) end
                       where  Bn+1 = Bsucc
    if - then - end →  IN(e) = IN(B)        ref. Fig. 4
                       call LVIO(B1, Bsucc)
                       call LVIO(B2, Bsucc)
    while →            ref. Fig. 5
                       IN(e) = IN(B)
                       call LVIO(Bw, e)
                       OUT(e) = IN(Bsucc) ∪ IN(Bw)
    repeat →           ref. Fig. 6
                       OUT(e) = IN(B) ∪ OUT(B)
                       IN(e) = REF(e) ∪ OUT(e)
                       call LVIO(BR, e)
  end
```



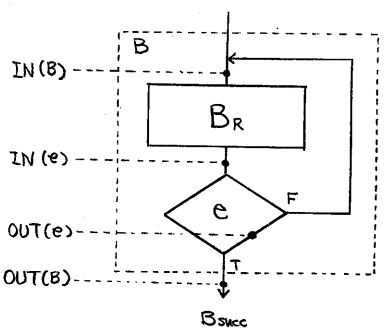
while e do Bw end

図5 while 文

手続き内の各文における live 変数の解析は，手続き本体を構成する文または文の並びをPとし，仮りの文S，ただし IN(S)=φ があったとして，

call LVIO(P, S)

によって，行なわれる．



repeat BR until e

図6 repeat 文

DFR, REF と live 変数の例を次ページ，図7と図8に示す．

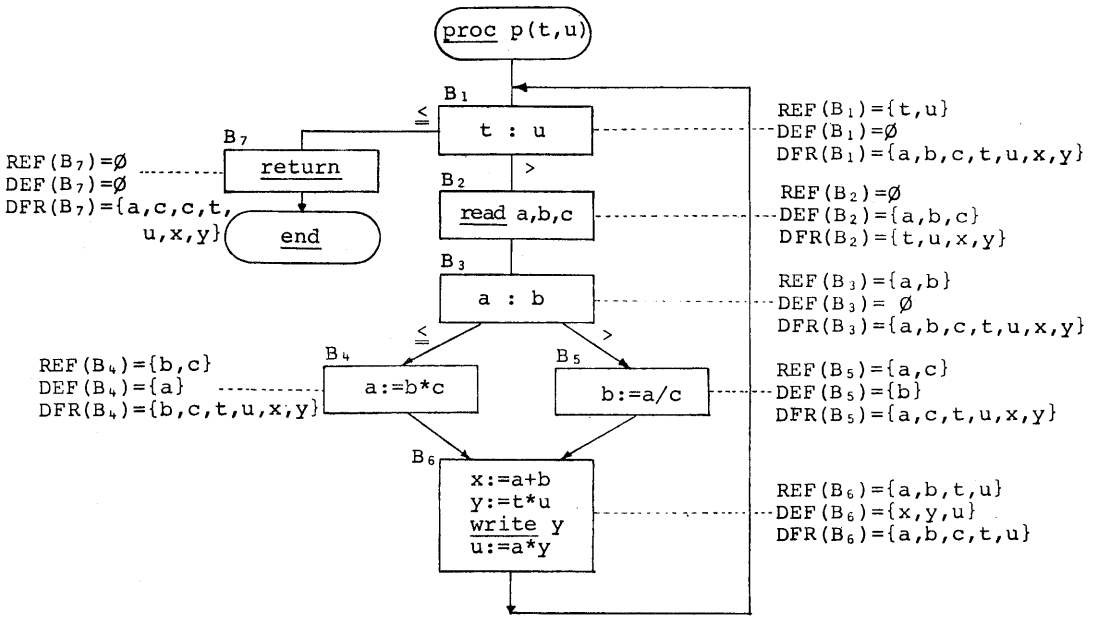


図7　D-free 変数

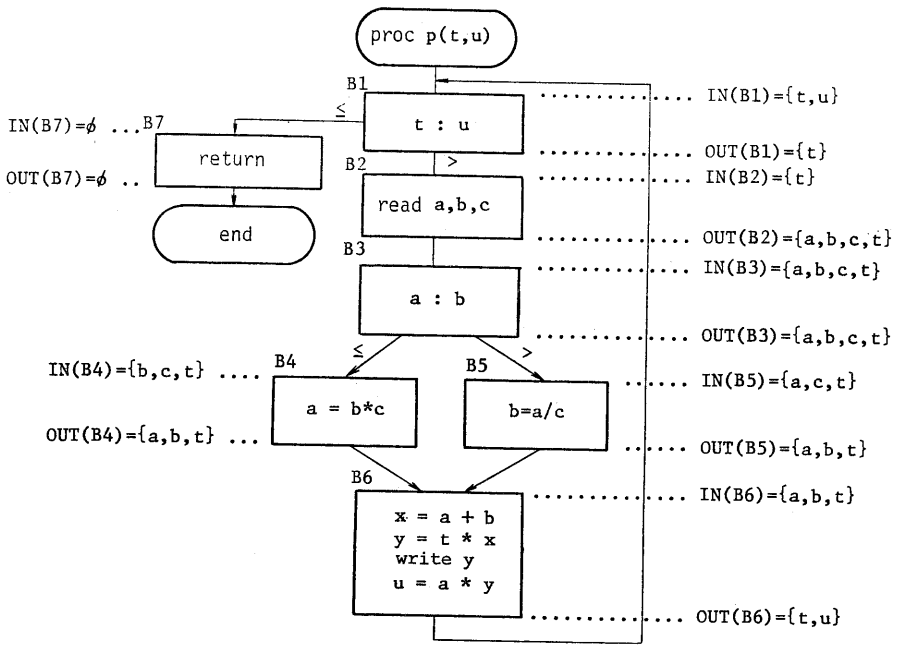
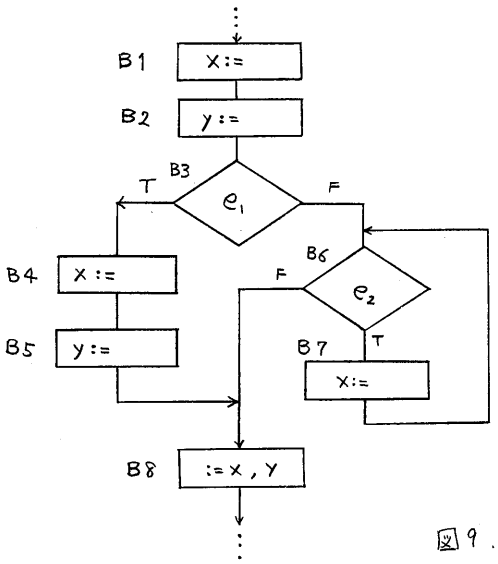


図8　live 変数

3. 到達可能な変数の解析

変数vの値がブロックBで再定義されるとき，Bはvの定義点（definition point）であるという．Bで定義されたvの値を，ブロックB'で参照することができれば，定義点Bのvは B'に到達可能であるという．到達可能な変数の解析は，各ブロックについて，そこに到達可能な変数とそれらの定義点を求めることである．例を図9に示す．



各ブロックの到達可能変数

B1 = $\{\phi\}$
B2 = $\{x_1\}$
B3 = $\{x_1, y_2\}$
B4 = $\{x_1, y_2\}$
B5 = $\{x_4, y_2\}$
B6 = $\{x_1, x_7, y_2\}$
B7 = $\{x_1, x_7, y_2\}$
B8 = $\{x_1, x_4, x_7, y_2, y_5\}$

X_L は，XがブロックLで定義されることを表わす．

図9．到達可能な変数の例

このような解析も，制御の流れに沿って順方向に2回走査することによって可能となる．収集する情報を最少にするためには，liveな変数だけを解析の対象とすればよい．

変数の参照に関して，その参照の種類を次のように分け，結果を各定義点に集めることも，データ・フロー解析の1つとして重要である．

(i) 算術演算の被演算子
(ii) 論理演算，関係演算の被演算子
(iii) 式におけるその他の被演算子
(iv) スイッチ変数（単に，流れを制御する目的のために使用される）
(v) 繰返しの制御変数
(vi) 手続き呼出しのパラメタ
(vii) 配列要素の添字

4. 変則的なデータの検出

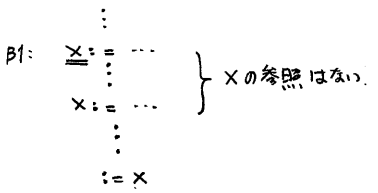
データ・フロー解析をコンパイラなどに組込むことによって，データ（変数）の変則的な使用法を検出することができる．

Dead variable

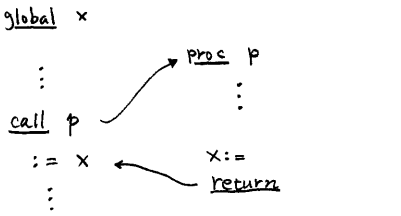
ブロックBで定義される変数vが，vをOUT(B)のとき，vが局所的変数であれば，Bにおけるvへの代入は意味のないものである．非局所的変数の場合は，vに値を代入して戻るというような手続きもあるので，意味がないとは断定

できない．

［例］



X1は意味がない

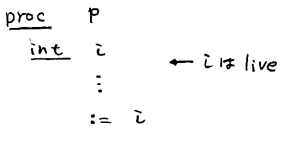


xは dead var. ではない

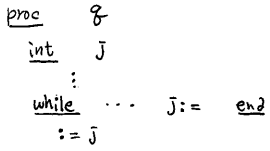
未定義変数の参照

局所的変数 x が手続きの入口で live であれば，その手続きの中で x に値を代入しないで参照することがあるという可能性を意味する．

［例］



未定義変数の参照



何とも言えない場合

手続きのパラメタ X が，手続きの入口で live であるということは，実パラメタによってその値が与えられることを期待しているものと解釈できる．

5. SIMPL 最適化コンパイラにおけるデータ・フロー解析の応用

データ・フロー解析の結果は，SIMPL 最適化コンパイラ[8]において，上述のような規則的な変数の使われ方を検出する以外に，次のような目的に使用されている．

5.1 広域的レジスタ割付け[9]

通常のコンパイラでは，ソース・プログラムで使用された変数に対して，値を保持するために記憶場所を割付ける．その場合には，演算のたびに記憶装置へのアクセスが必要となる．もし，変数をレジスタに割付けることができれば，データの読出し時間は短縮され，書込み操作（付入による）は不要になることがある．しかし，レジスタの個数は限られているために，どの変数にレジスタを割付けたら，もっとも効果的であるかということが問題となる．レジスタを割付けることが望ましい変数（割付け候補）を選ぶのに，次のような方法を使用している．

(1) 選択および繰返しの構造をもつブロックを走査し，それらのブロック内で使用されている変数の中から，割付け候補を選ぶ．

(2) レジスタの割付けは，recursive descent に行なう．

(3) レジスタが不足して，割付け候補にレジスタを与えることができないときには，レジスタの使用に関するコストを評価して，割付け状態を変更するか，

または，新しい割付け候補に対するレジスタの割付けを放棄するか決定する．while文および if文に対するブロックにおける割付け候補の例を図10に示す．

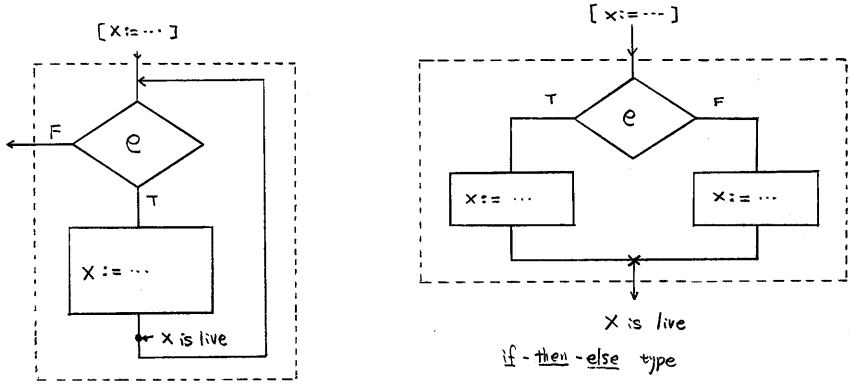


図10　レジスタ割付け候補

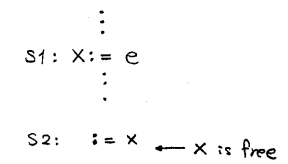
SIMPL 最適化コンパイラは Univac 1106 用に作成しているが，この計算機には A-レジスタ（演算レジスタ）と X-レジスタ（インデックス・レジスタ）の2種類のレジスタがある．レジスタの割付けをするときには，変数の参照目的を調べて，レジスタの種類が決められる．たとえば，割付け候補 x が配列要素の添字として用いられるのであれば，X には，X-レジスタを割付けるようにする．

5.2　式の処理

式またはレジスタが割付けられなかった変数を処理するときのために，A-および X-レジスタの一部を，作業用レジスタとして確保している．式または代入文に対する目的コードを生成するときに，live変数の結果を利用すると，作業用レジスタを有効に用いることができる．

Delayed Store

左図のような例を考える．文 S1 の式 e の値がレジスタ r に求められたとする．もし，S1 から S2 までの間で r を使用することがなければ，S1 において r の値を X に格納する必要はない．その間で他の目的に使用されるときに限って S1 で格納すればよい．代入文をコンパイルするとき，もし左辺の変数にレジスタが割付けられていなかったら，作業用レジスタを用いて右辺の式を評価し，その場では格納命令を出さないでおく．左辺の変数が free になるまでの間に，そのレジスタが使用



されたときに，定義点に印をつけ，次のフェーズで格納命令を挿入するようにする．こうすることによって，無駄な格納命令の生成を省略することができる．

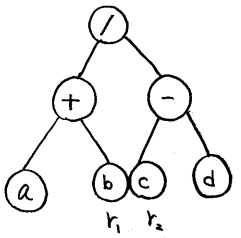
式の処理

式 x+y をコンパイルするとき，x の値がレジスタ r に入っていて，そのあと x が free であれば，次のような加算命令を生成するだけでよい．

add r, y

式は木構造（expression tree）で表現し，コンパイルするときには，まず一度式を走査し，参照後 free となるような値をもつレジスタを使って計算できる部分は最初に処理してしまうようにしている．

(a+b)/(c-d)



r1 と r2 は参照後 free とする．

[add r1, a]
[sub r2, d]
[div r1, r2]

r1 ← (a+b)/(c-d)

目的コード

5.3 制御構造の変形

Goto-less プログラムでは，テスト結果を表示するために，補助変数を導入し，テストしたあとで，補助変数の値を調べて，それぞれ必要な処理をするという形がよく用いられる．次のような例を考える．

```
V := 0
while e1
 do
    ⋮
   if e2 then V := 1 exit end
    ⋮
   if e3 then V := 2 exit end
   if e4 then V := e5 end
end
case V of
 0 → S1
 1 → S2
 2 → S3
```

⇒

```
V := 0
while e1
 do
    ⋮
   if e2 then fgoto L1 end
    ⋮
   if e3 then fgoto L2 end
   if e4 then V := e5 end
end
case V of
 0 → S1
 1 → L1: S2
 2 → L2: S3
```

このとき，case 文の式を評価したあとで，V が free であれば，Go to 文に相当する内部的な表現（fgoto）を用いることによって，上図の左に示すような形に変形し，無駄な代入と，case 文におけるテストを除去することができる．

参考文献

1. Allen, F.E., "Control Flow Analysis" SIGPLAN NOTICES 5, (July 1970), 1-19.
2. Allen, F.E. and Cocke, J., "A Program Data Flow Analysis Procedures," CACM 19, 3(March 1976), 137-147.
3. Kennedy, K., "A Global Flow Analysis Algorithm," Internal J. Computer Mathematics 3, Dec. 1971, 5-15.
4. Allen, F.E., "Interprocedural Analysis and the Information Drived by It," Lecture Notes in Computer Science 23, Programming Methodology, Springer-Verlag, 1974, 291-321.
5. Kam, J.B. and Ullman, J.D., "Global Data Flow Analysis and Iterative Algorithms," JACM 23, 1 (Jan. 1976), 158-171.
6. 原田賢一, Goto なしプログラムのデータ・フロー解析, 処情処理 1-18, 1977年1月, 27-35.
7. 原田賢一, Basili, V.R., コンパイラ作成用構造的プログラミング言語: SIMPL-T, 情報処理 17-3, 1976年3月, 222-228
8. 原田賢一, Global Optimization of Structured Programs, 1977年, 慶応義塾大学.
9. 〃, SIMPL 広域的最適化コンパイラにおけるレジスタ割付け, 1977年, 慶応義塾大学・情報科学研究所.