PASCAL プログラムにおける変数定義・使用に関するデータフロー解析†

宮本 衛市††

本論文は，PASCAL プログラムを対象として開発したデータフロー解析用ソフトウェアツールで用いられている解析手法を述べたものである．PASCAL はその言語仕様の簡潔さと強力なデータ構造記述能力により普及のめざましい言語であるが，その反面，抽象データタイプの概念がないこと，ポインタ型のデータタイプを陽に使用できること，変数パラメータあるいは大域的変数の使用に関して何ら制限がないことなどにより，誤ちを犯しやすい言語でもある．今回開発したツールは，PASCAL の弱点をデータフロー解析に基づいたプログラム診断により補おうとするものであって，その主な診断項目としては，動的変数の領域割付け，変数の定義・使用，手続き呼出し時のデータ受渡しなどに関する問題であり，データフロー上の問題点を指摘するが，最終的にはプログラマの判断に委ねることになる．

本手法では，プログラムの1回の走査に基づいて診断する．そのための基本的な考え方は，プログラムの任意の個所で，診断するためのフローが未解析の場合，その個所でフローに異常をきたさないための要求事項を生成し，診断を後出のフロー解析まで保留しておくことである．この考えをパラメータあるいは大域的変数の振舞い，さらにはこれらによって参照される動的変数，および **goto** 文による分岐・合流に対して適用して解析している．

1. はじめに

文法的な誤りを除去したプログラムに潜んでいる誤りのなかに，未定義の変数を使用したり，変数を未使用のまま終らせてしまう，いわゆる変数の定義・使用に関する問題 (def-use relationships) がある．これに加えて，PASCAL のように動的なデータ構造を構築できるような言語で書かれたプログラムの場合には，前述の問題に先立って，動的変数に領域が配分ずみであるかどうかの問題が生じる．

前者の問題の典型的な例として，変数の初期値設定の書き忘れがある．これに対しては，Dijkstra が提唱する言語[1]や CLU[2]のように，言語の構文規則で初期値設定を義務づけることが完全な解決策であろうが，大部分の言語ではプログラマの責任に委ねている．そこで初期値問題を含めて，変数の振舞いをデータフロー解析により調べることが考えられる．データフロー解析には，インターバルと称するプログラム・セグメントを見つけフローグラフを集約してゆく方法[3),4)]や，フローグラフを繰返し走査する方法[5),6)]などがあり，通常は中間生成コードの段階で解析し，大域的に最適な命令コードを生成するのが主目的である．これらに対し，ALGOL 形式のような制御構造を有する高水準言語で書かれたプログラムを，ソースイメージのままでプログラム構造に従って解析する方法も提案されている[7)~9)]．これらの主目的も最適コード生成のための情報収集であり，プログラム上の任意の点で変数が live[7),8)] であるか，dead[9)] であるかを解析するものであるが，動的なデータ構造までは扱っていない．

本論文は，PASCAL プログラムを対象として開発したデータフロー解析用ソフトウェアツールで用いている解析手法を述べたものである．PASCAL はその言語仕様の簡潔さと，強力なデータ構造記述能力により普及のめざましいプログラミング言語であるが，その反面，抽象データタイプ (abstract data type) の概念がないこと，ポインタ型のデータタイプを陽に使用できること，レコード型の可変フィールド (variant field) の使用がプログラマに委ねられていること，また変数に関しては初期値設定の構文規則がないこと，大域的変数 (global variable) が自由に使用できること，変数パラメータ (variable parameter) の使用に関して何ら制限がないことなどにより，特に変数の取扱いやデータ構造構築に際して誤りを犯しやすい言語でもある．今回開発したツールは，PASCAL の言語仕様の簡潔さの代償でもある言語の弱点を，プログラム診断によって補おうとするものであって，主な診断項目をあげると次のようになる．

† Data Flow Analysis for Def-Use Relationships of Variables in PASCAL Program by EIICHI MIYAMOTO (Division of Information Engineering, Faculty of Engineering, Hokkaido University).

†† 北海道大学工学部情報工学専攻

（1） 参照された動的変数に領域の割当てが行われているか.

（2） レコード型の変数の可変フィールドが重複して使われていないか.

（3） 変数の参照に先立って，値の定義が行われているか.

（4） 変数の再定義の前，あるいは変数が消滅する前に値の参照が行われたか.

（5） 手続き* の呼出し時に，変数パラメータおよび大域的変数のデータの受渡しで上記の項目に問題はないか.

これらの点などに関し，さらに細かい診断項目を用意している.

一方，ツールで使われている解析手法の特長をあげると次のようになる.

（1） プログラム診断は，**goto** 文あるいは手続きの再帰的呼出しの有無にかかわらず，ソースプログラムの1回の走査で行う．そのため，診断に必要なデータフローが未解析の場合には，先の解析個所への要求事項としてまとめておき，診断を一応保留しておく.

（2） 動的変数の構築するデータ構造は参照関係をもとにして解析する.

（3） 診断個所に到るまでの変数の履歴に基づいて診断を行う点が，解析個所以降に着目する最適コード生成を目的とするデータフロー解析と異なる点である.

以下2章では，ソースプログラムの1回の走査で解析するために，問題を手続きごとに分割する考え方について，3章ではそれに基づく解析手法について述べる．4章では解析に基づく診断項目をあげると共に，プログラムの診断例を掲げる．最後の5章で問題点と今後の課題について要約する.

2. 解析時におけるプログラムの分割

一般に，PASCAL プログラムはプログラム本体 (main body) を含めて，1つ以上の手続きの入れ子構造 (nesting structure) から成り立っており，手続きの呼出し以前に手続きの宣言が先行するが，再帰的呼出し (recursive call) の場合には手続きの頭書き (heading) のみが宣言されており，その本体は目下解析中ないしは後で出現する．これを手続き本体から眺めると，その手続きを非再帰的に呼出している側の手続きは未解析であるのに対し，**図1**に示すような再帰

* 以後，手続きというときには関数も含めて考えるものとする.

```
procedure   A1(……);
    procedure A2(……);
      procedure A3(……);
      begin…… A2(……); …… end;
    begin……A3(……); …… end;
  begin……A2(……); …… end;
```

図 1 再帰的呼出しを含んだ手続きの呼出し例

Fig. 1 Example of the invocation of procedures including a recursive call.

的呼出しの場合には，手続き A_2 の解析に際して，手続き A_3 は解析ずみであるが，手続き A_1 は未解析である．すなわち，手続き本体にとっては，それが再帰的に呼ばれていると否とにかかわらず，その手続きの頭書きに宣言されている変数パラメータおよび大域的変数に関して，さらに値パラメータを含めてこれらがポインタ型のときには，呼出し時に抱えているデータ構造を含めて未知である.

そこで，手続きのデータフローを解析するときに次のような方策を採用する.

〔**方策 1**〕 手続き本体内で，見かけ上変数パラメータあるいは大域的変数を未定義のままで使用したり，さらに値パラメータを含め，これらがポインタ型のとき，手続き内で領域割当てを受けることなく動的変数への参照を行った場合，この手続きを呼出す手続きに，対応する実パラメータあるいは大域的変数が呼出し時に定義，さらには領域割当てが行われていることを要求する．後で手続きの呼出しがあったとき，呼出す側の手続きが呼出される手続きの要求を満足していることを確認する. □

しかし，手続きを再帰的に呼出したときには，呼出された手続きの解析は終っておらず，方策1を適用することができない．そこで手続きの再帰的呼出しがあったときには，次のような方策を適用する.

〔**方策 2**〕 再帰的呼出しのパラメータとして渡された変数および呼出される手続きにとっても大域的である変数が，手続きから戻ってきた後で，その変数が抱えるデータ構造内の動的変数を含めて，未定義のままで使われたり，領域割当てのない動的変数への参照を行った場合，再帰的に呼出された手続きに，対応するパラメータおよび大域的変数の定義，さらには領域割当てを要求する．ただし，この要求は再帰的呼出しを行ったフローブランチが他のブランチと合流するまでとする．後で手続きの宣言が解析されたとき，再帰的に呼出した側の手続きの要求を満足していることを確認する. □

3. データフローの解析手法

前章で述べた基本的な方策に基づいて，本章では具体的な解析手法を展開する．

3.1 変数の状態の把握

解析の対象の単位となる変数は，単純変数，配列変数，セット変数，ポインタ変数，レコード変数およびレコード内の各フィールド変数であり，さらにポインタ変数を介して生成される動的な変数である．このうち配列変数に関しては，個々の要素のデータフローを追求することは一般的には不可能であるので，解析上は配列変数として一まとめにして考える．

一般に変数は，まず値が定義されたあと何回か使用され，再び値が再定義されて定義・使用のパターンが繰返される．そこで，このパターンに対応して変数の定義・使用に関する状態を表わすため，次の4つの状態を考える．

$$state=(undef, fuz, def, use) \qquad (3.1)$$

ここで，

undef：変数が宣言あるいは生成されたばかりで値が未定義な状態．

fuz：未定義のフローと定義ずみのフローが合流した場合のように，変数の状態を確定できない場合．

def：代入文，*read* および *new* の呼出しなどで値の定義があり，かつまだ使用されていない状態．

use：変数が1回以上使用された状態．

一方ポインタ変数 p は標準手続き *new*(p) を呼出すことにより，p を介して間接的に参照する動的変数 $p\uparrow$ を生成する．そこで，ポインタ変数に対して動的変数の有無に関する4つの状態を考える．

$$alloc=(empty, null, vague, exist) \qquad (3.2)$$

ここで，

empty：ポインタ変数が未定義で，動的変数も存在しない状態．

null：ポインタ変数が **nil** で動的変数が存在しない状態．

vague：ポインタ変数は定義ずみであるが，その値が **nil** かまたは動的変数を指しているか判別しがたい状態．

exist：ポインタ変数が動的変数を指している状態．

new(p) の出現により，動的変数 $p\uparrow$ が宣言されたものと見なし，以後の解析対象に加える．ただし，$p\uparrow$ の解析用テーブルは p の解析用テーブルから参照するようにして，実行時の動的なデータ構造を解析用テーブルの参照関係でシミュレートしてゆく．**図2**に，フローに応じた各変数の状態遷移の例を示す．ただし，$S(p)$ および $A(p)$ はそれぞれ変数 p の定義・使用の状態および動的変数 $p\uparrow$ に関する領域割当ての状態を示す．

```
type rp=↑rec;
     rec=record key: integer; next: rp end;
var p, q: rp;
begin ·····S(p)=undef, A(p)=empty,
           S(q)=undef, A(q)=empty
q :=nil; ·····S(q)=def, A(q)=null
new(p); ·····S(p)=def, A(p)=exist,
           S(p↑.key)=undef, S(p↑.next)=undef
 p↑.key :=10;·····S(p↑.key)=def, S(p=use
 p↑.next:=q;·····S(p↑.next)=def, A(p↑.next)=A(q)
```

図2 変数の状態遷移の例（以後の図で，タイプ *rp* は上図のように定義ずみとする．）

Fig. 2 Example of the state transition of variables.

いま，q をポインタ変数，p をポインタ変数または **nil** とすると，代入文

$$q:=p \qquad (3.3)$$

により，q は p のもっている動的変数を引継ぐことになる．そこで，q に対し

$$S(q)=def,\ A(q)=A(p) \qquad (3.4)$$

の設定と，p が抱えている動的変数の解析用テーブルへの参照関係を設ける．

一方，変数 p が変数パラメータあるいは大域的変数か，さらに値パラメータを加えてこれらがポインタ型のとき，これらにより参照される動的変数とすると，**方策1**に従い，呼出し時に要求する状態 $S^c(p)$，p がポインタ型のときには $A^c(p)$ を考える．ここで，$S^c(p)$ は (*undef*, *def*) のいずれかの値をとり，それぞれ次のような要求を意味するものとする．

undef：手続きの中では，まず定義してから使用しているので，呼出し時に対応する変数は未定義であってもよい．

def：手続き内では定義しないまま使用しているので，呼出し時には定義ずみであることを要求する．

$A^c(p)$ は (*empty*, *vague*, *exist*) のいずれかの値をとり，それぞれ次のような意味をもつ．

empty：同時に $S^c(p)=undef$ であり，手続きの中で動的変数の領域割当てを受けているので，呼出し時には領域割当てを受けていなくて

もよい.

vague: 手続き内で $p\uparrow$ を参照するとき, p が **nil** でないことを確認しているので, 手続き呼出し時には p に対応する変数は定義されていればよくて, 動的変数の割当ての有無は問わない.

exist: 手続き内では **nil** でないことの確認をしないで動的変数の参照を行っているので, 呼出し時には対応する変数に *exist* の状態を要求する.

図3 に手続き内のフローに応じてパラメータおよび大域的変数の要求が形成される例を示す.

再帰的呼出しの場合には, **方策2** に従い, 再帰的呼出しに現われたパラメータおよびそれによって参照される動的変数, さらに呼出し後に現われる, 呼出される手続きにとっても大域的な変数とその動的変数に対して $S^r(p)$ および $A^r(p)$ を設定する. ここで p は再帰的呼出しに対し要求できる前述の変数群に含まれるものとし, $S^r(p)$ と $A^r(p)$ は非再帰的呼出しにおける $S^c(p)$ と $A^c(p)$ と同様な要求内容を意味するが, 後者は呼出す手続きへの要求であるのに対し, 前者は再帰的に呼出される手続きへの要求である. **図4** に再帰的呼出しに対する要求の設定の例を示す.

```
var glb: integer;
    .....
procedure alloc (p: rp);
 var q: rp;
 begin
  if p<>nil ·····Sc(p)=def, Ac(p)=vague,
                 S(p)=use, A(p)=vague
   then ·····A(p)=exist
    begin
     p↑. key := glb; ·····Sc(glb)=def, S(glb)=use,
                          S(p↑. key)=def
     q :=p↑. next; ·····Scp↑. next)=def,
                        Ac(p↑. next)=vague,
                        S(p↑. next)=use,
                        A(p↑. next)=vague,
                        S(q)=def, A(q)=A(p↑. next)
```

図3 手続き呼出し時への要求の生成例

Fig. 3 Example of setting of required states.

```
procedure recur (var p: rp; k: integer);
 var q :=rp;
begin .....
 if k≠0 then
  begin recur (p, k−1);·····S(p)=def
   new (p↑. next); ·····Ar(p)=exist, A(p)=exist,
                        S(p↑. next)=def,
                        A(p↑. next)=exist
   p :=p↑. next; ·····S(p)=def, A(p)=A(p↑. next)
```

図4 再帰的呼出しにおける要求の生成例

Fig. 4 Setting example of required states to the recursively called procedure.

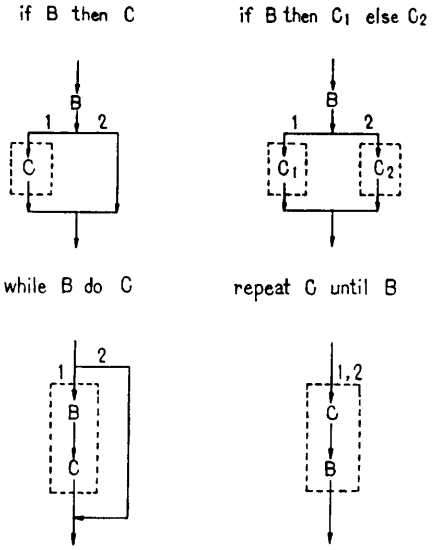

図5 データフロー解析上の構造文に対するフロー表現

Fig. 5 Flows of structured statements for data flow analysis.

3.2 フローの分流と合流

構造文のうち, **case** 文は **if** 文を多分岐にしたもの, **for** 文は繰返しが陽に1回以上の場合には **repeat** 型, そうでない場合には **while** 型とすれば, データフロー上は **if** 型, **while** 型および **repeat** 型の3つに分類して考えることができる. これらに対する解析上のフローの概念図を示すと**図5**のようになる. ここで **while** 型が実際上の実行の流れとは異なっているが, 繰返しのフローを考える上で等価なフローに置き換えてある.

いま, 図5の点線で囲まれた部分を1つの仮想の手続きと考えてみる. ただし, この仮想の手続きではすべての変数は大域的であり, その入口ではそれまでに得られた解析結果を引継ぐことができる.

if 型および **while** 型のフローの分流のうちで, 次のようなポインタ変数を含んだ論理式による分流を考える.

$$(p_1\ \mathrm{op}_1\ \textbf{nil})\ \textbf{and}\cdots\cdots\textbf{and}\ (p_n\ \mathrm{op}_n\ \textbf{nil})\ \textbf{and}\ B \tag{3.5}$$

ここで, $p_1, \cdots, p_n$ はポインタ変数であり, $\mathrm{op}_1, \cdots, \mathrm{op}_n$ は '=' または '≠' の演算子, B は任意の論理式とする. もちろん(3.5)式で B が挿入される位置は任意である. このような形の論理式によるフローの分流の場合, 分流したフローではより明確な状態を設定することができる. すなわち, **if** 文の **then** 節では op_i が

'=' のとき $A(p_i)=null$, **else** 節では $A(p_i)=exist$, op_i が '≠' のときはこの逆の関係が成り立ち, **while** 文では op_i が '≠' のとき $A(p_i)=exist$, op_i が '=' のとき $A(p_i)=null$ と設定することができる. さらに, **while** 文を終了したときには, op_i が '≠' のときには $A(p_i)=null$, op_i が '=' のときには $A(p_i)=exist$ となる.

次にフローの合流について考える. 図5でフロー1およびフロー2の出口における変数 p の状態を $S_1(p)$ および $S_2(p)$, p がポインタ型のとき動的変数の割当てに関する状態を $A_1(p)$ および $A_2(p)$ とすると, 合流後の状態を**図6**および**図7**に示す. ただし, **if** 文で **else** 節が欠除している場合および **while** 文のときにはフロー2はフロー1をスキップするフローに対応し, $S_2(p)$ および $A_2(p)$ は分流する個所における状態をとる. また **repeat** 文は形式的に図のフローをフロー1とフロー2とする. 図6および図7ではいくつかの選択が示されているが, これらはフローの性格と変数のクラスによる解析上の振舞いを考慮したものである.

手続き呼出しに関する要求に対しては

$$S^k(p)=\max(S_1^k(p),\ S_2^k(p)),\quad k=c,r \qquad (3.6)$$

とし, $A^c(p)$ および $A^r(p)$ は図7 の *1 の場合を適用する.

図8にフローの分流および合流による解析例を示す.

$S_1(p)$ \ $S_2()p$	*undef*	*fuz*	*def*	*use*
undef	*undef*	*fuz*	*fuz/def**4	*fuz/use**4
fuz	*fuz*	*fuz*	*fuz/def**4	*fuz/use**4
def	*fuz/def**4	*fuz/def**4	*def**3*/use**1	*use**1,*3*/def**2
use	*fuz/use**4	*fuz/use**4	*use**1,*3*/def**2	*use*

*1: 配列変数, あるいは while または repeat 型で *def* 要求があるとき (第1順位).
*2: いずれかのフローで p の定義があったとき (第2順位).
*3: *1 および *2 以外のとき (第3順位).
*4: p が *def* 要求を出しているパラメータまたは大域的変数, さらにこれらによる動的変数, 可変フィールドの変数等のとき.

図 6 フロー合流による状態 S の合成

Fig. 6 Composition of states S by the join of flows.

$A_1(p)$ \ $A_2(p)$	*empty*	*null*	*vague*	*exist*
empty	*empty*	*empty/null**1	*empty/vague**1	*empty/exist**1
null	*empty/null**1	*null*	*vague*	*vague*
vague	*empty/vague**1	*vague*	*vague*	*vague*
exist	*empty/exist**1	*vague*	*vague*	*exist*

*1: 図6の注釈 *4 に対応した変数のとき.

図 7 フロー合流による状態 A の合成

Fig. 7 Composition of states A by the join of flows.

```
procedure flow (p: rp);
  var k: integer;
  begin k :=0; ·····S(k)=def
    while p≠nil do·····Sc(p)=def, Ac(p)=vague,
      begin            S(p)=use, A(p)=exist
        k :=k+p↑. key; ···Sc(p↑. key)=def,
                          S(k)=def, S(p↑. key)=use
        p :=p↑. next ·····Sc(p↑. next)=def,
                          Ac(p↑. next)=vague,
                          S(p)=def, A(p)=vague
      end; ·····Sc(p)=def, Ac(p)=vague,
                Sc(p↑. key)=def,
                Sc(p↑. next)=def,
                Ac(p↑. next)=vague,
                S(k)=def, S(p)=use,
                A(p)=null
```

図 8 フローの分流と合流による解析例

Fig. 8 Example of decomposition and composition of flows.

3.3 goto 文によるフローの分岐と合流

goto 文による分岐には解析上, 同一手続きの前方への分岐と後方への分岐, および大域的分岐の3通りが考えられるが, いずれも合流点におけるいずれか一方のフローが未解析である. そのため次のような方策を採用する.

〔**方策 3**〕 同一手続きの後方への分岐の場合には, 分岐する個所のデータフローの解析結果の, 大域的分岐の場合には分岐先の手続きのレベル以下の大域的変数の解析結果の複製をとる. また, ラベルが定義された個所における解析結果の複製もとっておく. そこで後方への分岐の場合には分岐先のラベルが定義されたとき, 前方への分岐の場合には前方分岐の **goto** 文が現われたときに, フロー合流上の診断を行う. ただし, **goto** による分岐は必ず他のフローとの合流が伴うものとし, **goto** 文のみによるフローの開始はないものとする. □

図9に **goto** 文がある場合の解析例を示す.

3.4 手続き呼出しと終了時における診断

図 10 は手続き呼出し時における変数の対応関係を

```
procedure branch (p: rp; val: integer);
  label 7;
  begin
    while p≠nil do
      begin
        if p↑. key=val then goto 7;·····A(p)=exist
        p :=p↑. next
      end; ····· A(p)=null
  7: ·····A(p) に exist と null の状態が混在している.
```

図 9 goto 文による分岐に対する診断例

Fig. 9 Example of diagnosis for branching by a **goto** statement.

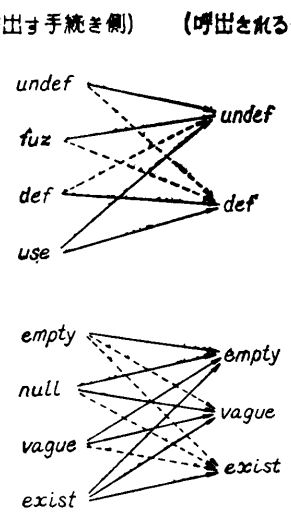

図 10 手続き呼出し時における変数の状態の組合せ
Fig. 10 Combination of states of variables at the invocation of a procedure.

表わしたもので，実線が許される組合せを，点線が許されない組合せを示す．手続きから戻ってきた後の状態は呼出された手続きでの対応する状態を引継ぐ．

再帰的呼出しの場合，呼出された手続きの解析が終了した時点で，図 10 で立場を入れ換えた診断を行う．

```
 1 PROGRAM NUMCOUNT(INPUT,OUTPUT);
 2  TYPE RECP=@REC;
 3       REC=RECORD LLINK,RLINK:RECP;
 4                  NUM,COUNT:INTEGER
 5            END;
 6  VAR ROOTP:RECP; N:INTEGER;
 7
 8  PROCEDURE ENTERNUM(NUMBER:INTEGER);
 9   LABEL 7;
10   VAR P,Q,OLDP:RECP; LEFT:BOOLEAN;
11   BEGIN P:=ROOTP; (* OLDP:=NIL; *)
12    WHILE P<>NIL DO
13     BEGIN OLDP:=P;
14      IF P@.NUM=NUMBER THEN BEGIN (* Q:=P; *) GOTO 7 END
15      ELSE IF P@.NUM<NUMBER THEN
16            BEGIN P:=P@.LLINK; LEFT:=TRUE END
17           ELSE
18            BEGIN P:=P@.RLINK; LEFT:=FALSE END
19     END;
*** 'LEFT' IS NOT DEFINED IN SKIPPED FLOW.
*** 'OLDP' IS NOT DEFINED IN SKIPPED FLOW.
20    NEW(Q); Q@.LLINK:=NIL; Q@.RLINK:=NIL;
21    Q@.NUM:=NUMBER; Q@.COUNT:=0;
22    IF OLDP=NIL THEN ROOTP:=Q
*** 'OLDP' MAY NOT BE DEFINED.
23    ELSE IF LEFT THEN OLDP@.LLINK:=Q
*** 'LEFT' MAY NOT BE DEFINED.
24                  ELSE OLDP@.RLINK:=Q;
*** 'OLDP@.RLINK' IS NOT DEFINED IN EITHER FLOW.
*** 'OLDP@.LLINK' IS NOT DEFINED IN EITHER FLOW.
25    7: Q@.COUNT:=Q@.COUNT+1
*** 'Q' IS NOT DEFINED AT LOCAL-GOTO JOIN.
26   END;
27
28  PROCEDURE PRINTNUM(FP:RECP);
29   BEGIN (* IF FP<>NIL THEN *) WITH FP@ DO BEGIN
30    IF LLINK<>NIL THEN PRINTNUM(LLINK);
31    WRITELN(NUM,COUNT);
32    IF RLINK<>NIL THEN PRINTNUM(RLINK)
33   END;
34
35  BEGIN ROOTP:=NIL;
36   WHILE NOT EOF(INPUT) DO
37    BEGIN READ(N); ENTERNUM(N); END;
38   PRINTNUM(ROOTP)
*** 'ROOTP' MAY NOT BE ALLOCATED DESPITE REQUEST.
39  END.
```

図 11 プログラム診断の例
Fig. 11 Example of diagnosis by data flow analysis.

4. 診断項目と解析例

本ツールで診断する主な項目をあげると次のようになる．

（1） 未定義変数を使用していないか．

（2） 動的変数の領域割当てが確実になされているか．

（3） 可変フィールドの使い方に重複はないか．

（4） 変数を1度も使用しないで再定義していないか．

（5） フローの合流で，ある変数が一方でのみ定義されており，他方では未定義ということはないか．

（6） フローがスキップされるとき，変数が未定義になるようなことはないか．

（7） 未定義変数を抱えた **goto** 文による合流はないか．

（8） 手続き呼出し時に対応する変数間で要求が満足されているか．

（9） すべての局所的変数およびその動的変数は状態 *use* で終っているか．

(10) 値パラメータは状態 *use* で，また変数パラメータは状態 *use* または *def* で終っているか．

(11) 再帰的に呼出された手続きは要求通り宣言されているか．

(12) 関数値が定義されているか．

図 11 に本ツールによる解析例を示す．この例では正常なプログラムの一部をコメントにして，どのような診断が行われるかを示している．

5. おわりに

本論文で述べたプログラム診断用ソフトウェアツールでは，誤りの発生する可能性がある場合を含めて診断を行うので，冗長な診断情報が出されることもあるが，ツールの目的はプログラマにデータフロー上の問題点の警告を発し，最終的にはプログラマの判断に委ねることにある．プログラムの誤りはその原因を想定できないと発見することが甚だ困難であり，そのためのヒントをデータフロー解析から提供しようとするも

のである.

プログラムの実行時に構築されるデータ構造は，主としてポインタ型のレコードフィールドによる参照関係により把握しており，再帰的なデータ構造の個々の要素までは追求できない．しかし，通常は同じ計算のパターンが各要素に適用されるので，十分その能力を発揮する．このような実行時のプログラムの振舞いを静的に解析しようとする試みには，プログラムの記号処理的な実行評価[10]や，変数の値の挙動範囲の評価[11]などがあり，本論文で述べた診断ツールも変数の定義・使用およびデータ構造の動的な割当て問題に着目したプログラム診断を行うものである．今後は変数間の参照関係，あるいはデータ構造の，いわゆる奥行のある把握などから，さらにプログラムを診断する可能性を検討している.

謝辞 北海道大学田川遼三郎教授には，本研究を進めるに当って貴重な御教示をいただいた．ここに記して深謝申し上げます.

参 考 文 献

1) Dijkstra, E. W.: A Discipline of Programming, p. 217, Prentice-Hall (1976).
2) Liskov, B., et al.: Abstraction Mechanisms in CLU, Commun. ACM, Vol. 20, No. 8, pp. 564-576 (1977).
3) Allen, F. E. and Cock, J.: A Program Data Flow Analysis Procedure, Commun. ACM, Vol. 19, No. 3, pp. 137-146 (1976).
4) Kennedy. K.: Use-Definition Chains with Applications, Computer Languages. Vol. 3, No. 3, pp. 163-176 (1978).
5) Kam, J. B. and Ullman, J. D.: Global Data Flow Analysis and Iterative Algorithms, J. ACM, Vol. 23, No. 1, pp. 158-171 (1976).
6) Graham, S. L. and Wegman, M.: A Fast and Usually Linear Algorithm for Global Flow Analysis, J. ACM, Vol. 23, No. 1, pp. 172-202 (1976).
7) 原田賢一: GO TO なしプログラムのデータ・フロー解析，情報処理，Vol. 18, No. 1, pp. 27-35 (1977).
8) Rosen, B. K.: High-Level Data Flow Analysis, Commun. ACM, Vol. 20, No. 10, pp. 712-724 (1977).
9) Babich, W. A. and Jazayeri, M.: The Method of Attributes for Data Flow Analysis, Acta Informatica, Vol. 10, No. 3, pp. 245-272 (1978).
10) Howden, W. E.: Symbolic Testing, and the DISSECT Symbolic Evaluation System, IEEE Trans. Software Eng., Vol. SE-3, No. 4, pp. 266-278 (1977).
11) Harrison, W. H.: Compiler Analysis of the Value Range for Variables, IEEE Trans. Software Eng., Vol. SE-3, No. 3, pp. 243-250 (1977).

(昭和54年3月7日受付)
(昭和54年8月23日採録)