プログラミング 12－1
（1997．3．28）

手続きの完全な入出力について

大場克彦† 貴島寿郎‡ 湯浅太一‡‡

†京都医療技術短期大学
‡豊橋技術科学大学情報工学系
‡‡京都大学工学部情報工学教室

概要 手続きの入力と出力を明確に記述しようとすると、しばしば困難が生じる。本稿では、その問題を明確にし、その解決法について議論する。まず、プログラムの中で、手続き及び手続き呼び出し文の入力と出力を記述する。次に、この手続きをフローグラフで表し、手続きの入力と出力が満たすべき条件を、データフロー方程式として表現する。このデータフロー方程式を基に、手続きの入力と出力を定義する。この定義は、手続きを使用するために必要十分な情報を表す完全な入出力を与える。本稿で得られた結果は、手続きのインタフェース設計、プログラムのコメント文、プログラミング言語、プログラミング方法論に利用できる。

On the complete input and output of a procedure

Katsuhiko Ohba† , Toshiro Kijima‡ and Taiichi Yuasa‡‡

† Kyoto College of Medical Technology
‡ Department of Information and Computer Sciences,Toyohashi University of Technology
‡‡ Department of Information Science Faculty of Engineering, Kyoto University

Abstract When we try to describe clearly an input and output of a procedure, we often encounter difficulties. This paper makes them clear and discusses how to solve them. First, we describe the input and output of the procedure and that of the call statements in it. Next, we represent the procedure by a flow graph. And as some data flow equations, we express the conditions which the input and output of it should satisfy. Being based on them, we define the input and output of it. This definition give us the complete input and output of it which shows the necessary and sufficient information to use it. The result of our study can be applied to interface design of procedures, comments of programs, programming language, programming methodology.

1　はじめに

プログラムのライフサイクルの中で、開発に要する労力よりも保守に要する労力の方が大きいことが指摘されている［1］。保守に要する労力の中で、プログラムテキストを解読しプログラムを理解するのに要する労力が大きな比率を占める。プログラムの解読に労力を要する原因の1つとして、プログラムの中のデータ依存関係の把握が困難なことがあげられる。

データ依存関係を把握するためには、プログラムの入出力及びプログラムの中で使用されている全ての文の入力と出力を知ることが必要である。通常使用されているプログラム言語では、手続き呼び出し文の入力と出力が明示されていない。そのため、手続き呼び出し文が現れると、その入出力を知るために、呼び出された手続きの中を解読しなければならない。解読している手続きの中に手続き呼び出し文が現れると、さらに、呼び出される手続きを解読しなければならない。このような過程が繰り返されるので、下位のレベルからボトムアップに積み上げていかないと、上位の手続きの入力と出力が明確にならない。これがプログラムのデータ依存関係の把握を困難にし、その解読に多くの労力を費やす原因の1つとなっている。

プログラミング言語において、手続きの入力と出力が明記されないことにより生じる問題点については、［2］でも指摘している。しかし、単に手続きの入力と出力を明示するだけでは、上に述べた問題は解決しない。このことを例を用いて示す。

```
ex1()
  { float u;
    if(a > 5){x = b + c ; y = b + a;}
    else {y = b + 5 ; u = b - c ; z = b * u;}
   }
```

図1　　手続き　ｅｘ１

図1に示す手続き、ｅｘ１の入力と出力について考える。ａ、ｂ、ｃは局所変数でなく、かつ、値を設定せずに使用しているので、ａ，ｂ，ｃをｅｘ１の入力とする。次に出力について考える。ｘ、ｙ、ｚは局所変数ではなく、かつ、手続きの中で値を設定する可能性があり、その値をｅｘ１の外部で使用することができる。一方、ｅｘ１の中で値が設定される可能性のある非局所変数は、ｘ，ｙ，ｚ以外にはない。従って、ｘ，ｙ，ｚをｅｘ１の出力とする。ところが、ｅｘ１の入力と出力をこのように決めると問題が生じる。

ｅｘ１を呼び出す手続きｅｘ２を考える。ｅｘ２の中でａ、ｂ、ｃに値を設定してｅｘ１を呼ぶと、ｘかｚのいずれかが未設定になる。従って、ｘとｚを出力の中に含めるのは問題である。そこで、ｅｘ１の中で必ず値が設定されるｙだけを、ｅｘ１の出力としてみる。すると、ｘとｚがｅｘ１の中で副作用を受ける可能性のあることが、明確にならない。故にｙだけを出力とするのも問題である。以上に示したように、手続きの入力と出力を明示するためには、出力をどのように決めるかという問題を解決しなければならない。

構造化プログラミングでよく使用される設計法に、複合設計［3］、ワーニェ法［4］、Ｊａｃｋｓｏｎ法［5］がある。手続きの入力と出力を明記するドキュメント法としてＨＩＰＯのＩＰＯダイアグラム［6］がある。また、行動ダイアグラムは、手続きの入力と出力を明記することが可能である［7］。手続きの入力と出力を明記するプログラム論理の記述法として、手続きの論理を木構造で表現する試み［8］がある。また、［9］は、プログラムの理解を容易にする上で、プログラミング言語に望まれる機能の1つとして、手続きの入力と出力を明示することをあげている。しかし、上記のいずれの場合においても、上に述べた問題に関する検討がなされていない。ＨＯＳ［10］では、基本的な制御構造を2進木で表すと共に、制御の流れに対応して許されるデータの流れを決めることにより、上に述べた問題が生じないようにしている。しかし、ＨＯＳでは構造化プログラミングで使用する制御構造を使用することができない。この点がプログラミングの際の負担を大きくしている。以上に見たように、構造化プログラミングで推奨されている制御構造を使ってプログラムの論理を記述する時に、手続きの入力と出力をどのように決めるべきかに関する検討が、十分に行われていない。本稿では、手続きの入力と出力として必要十分な情報を明確にし、手続きの入力と出力を明記するプログラミング法を有効にすることを試みる。これ以降、2節で用語の説明をする。3節では手続きの必要十分な入出力（これを完全な入出力と呼ぶ）の定義を与える。そのために、手続きの処理を記述した文を命令と呼ぶ基本単位に分解し、命令をノードとする、手続きのフローグラフを考える。このフローグラフを用いてデータフロー解析を行い、手続きの入力と出力が満たすべき条件をデータフロー方程式で表す。その結果を基に完全な入出力を定義する。4節では、まず、本節で指摘した問題点が解決されることを例を用いて示す。次に、本稿で得られた結果の利用法について述べる。本稿で得られた結果は、手続きのインタフェース設計、プログラムのコメント文、プログラミング言語、プログラミング方法論などへの利用が可能である。5節でまとめを行う。

2　用語の説明

本稿で使用する用語を説明する。2．1節で本稿で使用するプログラミング言語、2．2節で変数の有効範囲、2．3節で命令、2．4節でフローグラフについて説明する。

2．1　プログラミング言語

本稿で手続きの記述に使用するプログラミング言語

の書式を示す。言語機能は本稿の議論に必要なものに絞るので、データの型に関する記述は省略してある。また、データは構造を持たないデータだけを扱う。

手続き::＝手続き頭部　本体
本体::＝｛｛局所変数宣言｝　文｝
手続き頭部::＝ｎａｍｅ（入力仮引数並び）→
（出力仮引数並び）；
入力仮引数並び::＝［ｖ｛，ｖ｝］
出力仮引数並び::＝［ｖ｛，ｖ｝］

ここでｖは変数を表す。出力仮引数並びの変数は参照渡しである。出力仮引数と同名の入力仮引数は参照渡しである。それ以外の入力仮引数は値渡しである。

局所変数宣言は、手続き内部で局所的に値を設定して使用する変数を宣言する。

局所変数宣言::＝ｌｏｃａｌ　ｖ｛，ｖ｝；

文は、代入文、手続き呼び出し文、分岐文、反復文、複文よりなるものとする。

代入文::＝ｖ＝Ｅ；
手続き呼び出し文::＝ｎａｍｅ（ｉｎ_ｌｉｓｔ）
→（ｏｕｔ_ｌｉｓｔ）；
分岐文::＝ｉｆ（Ｃ）ｓ　ｅｌｓｅ　ｓ
反復文::＝ｗｈｉｌｅ（Ｌ）ｓ
複文::＝｛ｓ…ｓ｝

ここで、Ｅは式、ｉｎ_ｌｉｓｔは入力パラメータの並び、ｏｕｔ_ｌｉｓｔは出力パラメータの並び、Ｃ及びＬは条件式、ｓは文を表す。出力パラメータは変数である。入力パラメータは、式か変数である。

２．２　変数の有効範囲

局所変数は、手続きの中で局所的に宣言された変数である。値渡しの入力仮引数と局所変数を合わせたものを内部変数と呼ぶことにする。内部変数は手続き内に存在する変数である。手続きの外に存在する変数を外部変数と呼ぶことにする。内部変数と外部変数の関係は相対的である。ある手続きの内部変数は、別の手続きから見れば外部変数である。内部変数は、その変数が存在する手続きの中と、その子孫の手続きの中で有効である。親手続きの中の変数は、手続き命令の参照渡しのパラメータとして指定された変数だけが、子手続きの中で操作可能である。子手続きから見れば、参照渡しの変数は外部変数である。外部変数も手続き命令の参照渡しのパラメータとすることができるので、親手続きの内部変数を、子手続きを通じて、孫手続きに渡すことができる。同様にして、さらに下位の手続きにも、親手続きの内部変数を渡すことができる。しかし、親手続きから、子孫の手続きの内部変数を操作の対象とすることはできない。

２．３　命令

命令は文を構成する基本要素である。命令の種類は、代入文を表す代入命令、手続き呼び出し文を表す手続き命令、分岐文の条件式の部分を表す分岐命令、反復文の条件式の部分を表す反復命令の４つである。命令の種類は［１１］に準じている。

２．４　フローグラフ

問題の解析はフローグラフを用いて行う。データフロー解析については［１２］を参考にした。以下に示すフローグラフの定義は、［１１］を参考にした。

フローグラフとは、プログラムの制御の流れる方向を表した有向グラフである。手続きＦのフローグラフとは、以下の条件を満たす有向グラフ

Ｇ＝（Ｎ，Ａ，ｉ，ｏ）である。

①Ｎはノードの集合で、各ノードは手続きＦ内の命令を表す。

②Ａはアークの集合で、アーク（ｓ，ｔ）は、ノードｓを実行した後、制御が直ちにノードｔに移る可能性があることを示す。

③ｉは、手続きＦの入口を表すために便宜上導入された仮想的な手続き命令に対　応するノードである。この仮想的な手続き命令は入力パラメータを持たず、出力パラメータ並びはＦの入力パラメータ並びと一致するものとする。

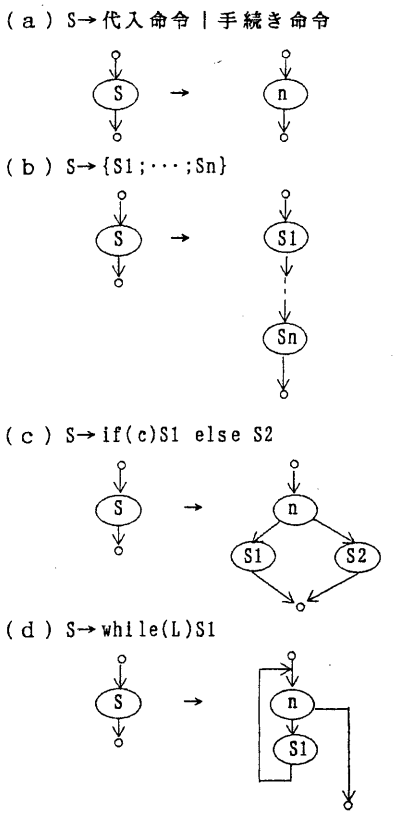

図２　文とフローグラフの関係

④oは、手続きFの出口を表すために便宜上導入された仮想的な手続き命令に対応するノードである。この仮想的な手続き命令は出力パラメータを持たず、入力パラメータ並びはFの出力パラメータ並びと一致するものとする。
⑤図2に示すパターンを用いて、フローグラフの全てのノードが命令に置き換えられるまで、再帰的に文をフローグラフに変換する。S、S1、…、Snは文、nは命令を表す。図1の手続きを、フローグラフで表すと図3のようになる。

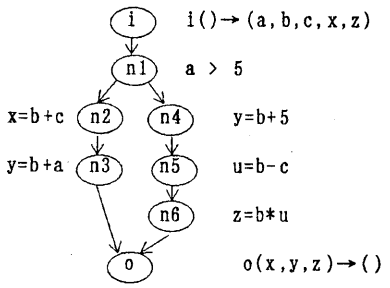

図3　ex1のフローグラフ

3　入出力の定義

3．1　出力に関する検討

手続きの出力は、親手続きの命令が利用するために、手続き内部で値を設定する変数である。故に、出力は親手続きの命令が参照できなければならない。外部変数は親手続きの命令が参照することができる。従って、手続き本体で値を設定する外部変数は、手続きの出力の候補である。一方、内部変数は親手続きの命令が参照することはできない。また、変数は内部変数か外部変数かのいずれかである。従って、手続き本体で値を設定する外部変数以外に、手続きの出力の候補はない。Out（F）を、手続き本体の中で値を設定する外部変数の集合とする。今、手続きの入口ノードiから出口ノードoに到る1つのパスpを考える。pの中の本体部分で値が設定される外部変数の集合を Out。（F）で表すと、Out（F）は、全てのパスに対するOut。(F）の和集合となる。

$$Out(F) = \bigcup_{p \in Path(F)} Out_{。}(F) \qquad (1)$$

ここで、Path（F）は、手続きFの入口ノードiから出口ノードoに到るパスの集合である。パスpの中で手続きの本体部分に属するノードの命令が値を設定する変数の集合をDef。(F）、内部変数の集合をInternal（F）とする。Out。(F）はパスpの本体部で値が設定される外部変数の集合であるので、（2）式で表される。

$$Out_{。}(F) = Def_{。}(F) - Internal(F) \qquad (2)$$

局所変数として宣言されている変数の集合をLocal（F）、値渡しの入力仮引数の集合をValIn（F）で表すと、Internal（F）は（3）式で表される。

$$Internal(F) = Local(F) + ValIn(F) \qquad (3)$$

次に、（2）式の中のDef。(F）を定義する。Fの中のノードnの命令が値を設定する変数の集合をDef（n）、パスpの中の本体部のノードの集合をBody。(F）で表すと、Def。(F）は（4）式で表される。

$$Def_{。}(F) = \bigcup_{n \in Body_{。}(F)} Def(n) \qquad (4)$$

但し、

$$Body_{。}(F) = p - \{i, o\} \qquad (5)$$

各命令に対するDef（n）は3．3節で定義する。

Out（F）に含まれる変数は全て出力の候補であるとしたが、実は、これらの変数が全て出力でなければならないことを示す。いま、出力以外の変数xが、Out（F）の中に含まれているものとする。xは出力でないから、親手続きで参照されることはない。故にxは内部変数にするべきである。xを内部変数にすると、Out（F）に含まれる変数は外部変数であるという条件に反する。故に、Out（F）に含まれる変数は全て出力にするべきである。

未確定の値を持つ変数を演算に使用すると演算の結果が保証されないので、手続きの出力は全て確定した値を持たなければならない。次にこの点に関して検討する。（1）式は、手続きの出力の中に、パスによって、手続き本体の中で値が設定されない変数がある可能性を表している。手続き本体の中で必ず値の設定される外部変数の集合をSOut（F）、パスによって値が設定されない場合がある外部変数の集合をUsOut（F）とすると、Out（F）は（6）式で表される。

$$Out(F) = SOut(F) \cup UsOut(F) \qquad (6)$$

ここで、SOut（F）、UsOut（F）は、各々

（7）（8）で表される。

$$SOut(F) = \bigcap_{p \in Path(F)} Out_p(F) \quad (7)$$

$$UsOut(F) = Out(F) - SOut(F) \quad (8)$$

SOut（F）に属する変数は、手続き本体の中で必ず値が設定されるが、UsOut（F）に属する変数は、値を設定しないパスがある。従って、手続き本体の中だけを考えると、出力が確定した値を持つことは保証されない。この問題の解決法を次に示す。

x∈UsOut（F）なるxを考える。本体の中でxに値を設定するパスを通ると、xは手続きの出口で、パスの中で設定された値をとる。本体の中でxに値を設定しないパスを通ると、手続きの入口のxの値が、そのまま手続きの出口のxの値になる。故に、UsOut（F）に含まれる全ての変数の値が、手続きの入口で確定した値を持てば、出力の値は確定する。一方、手続きの入力には、手続きの実行時に値が設定される。故に、UsOut（F）に含まれる変数を手続きの入力に含めれば、手続きの出力の値は確定する。

3．2　入力に関する検討

本稿では、手続きの入力を、手続きの実行に際して、親手続きにより確定した値が与えられるべき変数であると定義する。このように定義すると、前節の検討結果より、UsOut（F）を入力の一部とする必要がある。次に、UsOut（F）以外で入力とするべき変数を求める。

入力の定義より、入力変数は、手続き本体の中で値を設定せずに参照することができる。逆に考えると、手続き本体の中で値を設定せずに参照する変数は、入力変数にしなければならない。もし入力変数でないと、未確定値を持つ変数が演算に使用されるので、演算の結果が保証されなくなるからである。

入力変数の集合をIn（F）、手続きFの本体で値を設定せずに参照している変数の集合をUdUse（F）とすると、In（F）は（9）式で与えられる。

$$In(F) = UdUse(F) \cup UsOut(F) \quad (9)$$

手続きFのパスの中の本体部分に属するノードの命令が参照する変数の内、値が手続きの中で未設定のまま参照される変数を、未設定参照変数と呼ぶことにする。手続きFのパスpの中の未設定参照変数の集合を、$UdUse_p(F)$で表すと、UdUse（F）は全てのパスに対する$UdUse_p(F)$の和集合である。

$$UdUse(F) = \bigcup_{p \in Path(F)} UdUse_p(F) \quad (10)$$

次に、$UdUse_p(F)$を定義する。$Body_p(F)$を、手続きFのパスpの中で本体部分に属するノードの集合とする。$UdUse_p(n)$を、$n \in Body_p(F)$なるnの命令が参照する、未設定参照変数の集合とする。$UdUse_p(F)$は、$n \in Body_p(F)$なるnに対応する$UdUse_p(n)$の和集合である。

$$UdUse_p(F) = \bigcup_{n \in Body_p(F)} UdUse_p(n) \quad (11)$$

$UdUse_p(n)$は、パスpのノードnの命令が使用する変数の集合Use（n）から、パスpの中でnに先行する本体部分のノードが値を設定する変数の集合を引いた差集合である。これは（12）式で表される。

$$UdUse_p(n) = Use(n) - \left(\bigcup_{k \in PredB_p(n)} Def(k) \right) \quad (12)$$

ここで，$PredB_p(n)$は、パスpの中のnに先行する本体部分のノードの集合である。各命令に対するUseとDefの定義は次節で行う。

In（F）には入力以外の変数が含まれていないことは、In（F）を求める過程から明かである。次に、入力として必要な変数は、In（F）の中に全て含まれていることを示す。今、In（F）の中に含まれない任意の変数をxとする。xは、次のいずれかである。①手続きFの本体の中で値が参照される。②手続きFの本体の中で値が参照されず、かつ、Fの出力となっている。③それ以外。①の場合は、xはIn（F）に含まれないので、Fの本体の中でxに値が設定されてから、xが参照される。故に、xを入力にする必要はない。②の場合は、xはIn（F）に含まれないので、UsOut（F）にも含まれない。故にxを入力にする必要はない。③の場合は、xはFの中で値が設定されるだけで参照されないか、Fの本体の中に現れないかのいずれかである。いずれの場合も入力にする必要はない。以上に見たように①②③いずれの場合も入力にする必要はない。故に、入力として必要な変数は、In（F）の中に全て含まれている。

3．3　UseとDef

各命令に対するUse（n）とDef（n）の定義を示す。ここでεは空集合を表す。

（1）代入命令：　n　::=　v=E；

Use（n）＝

{x | x∈式Eの中に現れる変数}　（13）

Def（n）＝{v}　（14）

（2）手続き命令：

n　::=　name（入力並び）→（出力並び）

Use（n）＝

{x | x∈入力並びに現れる変数}　（15）

Def（n）＝

{x | x∈出力並びに現れる変数}　（16）

(3) 分岐命令： n ::= C
Use (n) =
{x | x∈分岐命令Cの中に現れる変数} (17)
Def (n) = {ε} (18)
(4) 反復命令： n ::= L
Use (n) =
{x | x∈反復命令Lの中に現れる変数} (19)
Def (n) = {ε} (20)
(5) 入力命令：
n ::= i () → (入力仮引数並び)
Use (n) = {ε} (21)
Def (n) =
{x | x∈入力仮引数並びに現れる変数} (22)
(6) 出力命令：
n ::= o (出力仮引数並び) → ()
Use (n) =
{x | x∈出力仮引数並びに現れる変数} (23)
Def (n) = {ε} (24)

3．4 入出力の定義

以上の検討結果を基に、手続きの完全な入出力を定義する。

定義1：手続きの出力とは、手続きの中で値が設定される可能性のある外部変数である。

定義2：手続きの入力とは、その手続きを実行する前に、外部で値を設定しなければならない変数である。手続きの入力は2種類の変数の集合より成る。

(1) 手続きの中で値が設定されずに使用される変数の集合、UdUse (F)。

(2) 手続きの中で値が設定されるパスと設定されないパスがある外部変数の集合 UsOut (F)。

4 検討・考察

4．1 検討

前節の定義を具体例に適用し入力と出力を求める。例として図1のex1を用いる。ex1の出力は定義1よりx、y、zである。入力は定義2よりa、b、c、x、zである。図1のex1の入出力を明記したものが図4である。ex1本体の中で副作用を受ける可能性にある外部変数は、全て出力となっている。次に出力値について検討する。a＞5の場合は、xとyの出力値は、ex1本体の中で設定される値であり、zは入力値がそのまま出力値となる。また、a≦5のときは、yとzの出力値は、ex1本体の中で設定される値であり、xは入力値がそのまま出力値となる。従って、x、y、zは、親手続きで参照するとき必ず確定した値を持つ。即ち、1節で指摘した問題は解消している。

```
ex1(a,b,c,x,z)→(x,y,z);
 { local u;
   if(a > 5){ x = b + c ; y = b + a ;}
   else {y = b + 5 ; u = b - c ; z = b * u ;}
 }
```

図4 入出力を明記した手続き ex1

```
ex2()→(w);
 { local a,b,c,x,y,z;
   a = 1 ; b = 2 ; c = 3 ;
   ex1(a,b,c,x,z)→(x,y,z);
   w = x * y * z
 }
```

図5 手続き ex2

次に、手続きex1を呼び出す手続き、ex2を図5に示す。ex2は、xとzに値を設定しないで手続き命令ex1を実行するので、誤りを含んでいる。この誤りを、ex2のテキストの中だけで発見できる。これは、完全な入力と出力を明示したことによる効果である。

4．2 利用法

本稿の結果は、手続きのインタフェース設計、プログラムのコメント文、プログラミング言語、プログラミング方法論などに利用できる。

4．1．1 手続きのインタフェース設計

手続きの中で、ある条件が満足されたとき値を設定し、そうでないとき値を設定しない変数を外部で使用するとき、これを出力として扱うにはどうしたらよいかという問題に対して、本稿では明確な答えを与えた。本稿の提案に従えば、次に示すような性質を持つ手続きのインタフェースが得られる。①手続きの中で副作用を受ける可能性のある外部変数が出力である。②出力が確定した値を持つために、手続きの実行に際して、親手続きが値を設定しなければならない変数が入力である。

上に述べた性質を有する入出力は、入出力として必要十分な情報を表す、完全な入出力である。

4．1．2 プログラムのコメント文

本稿の提案に基づいて手続きのインタフェースを設計し、その結果を、手続きや手続き呼び出し文の入出力を表すコメント文としてプログラムテキストに記述する。すると、関連する手続きの内容を解読しなくても、手続きや手続き呼び出し文の入出力が明確になるので、プログラムの理解が容易になる。

4．1．3 プログラミング言語

本稿の結果に基づきプログラミング言語に下記の機能を付け加えると、コメント文をつけ加えなくても、手続き及び手続き呼び出し文の入力と出力が明確になる。また、手続きの入出力が完全な入出力の条件を満足することを、コンパイラで自動的に検査することが

できる。

① 手続き及び手続き呼び出し命令の入力と出力を明記する。

② 外部変数、内部変数、局所変数を明確にする。

③ 手続きの入出力が、完全な入出力の条件を満足することを、コンパイラで検査する。

4．1．4 プログラミング方法論

構造化プログラミング［13］［14］が提唱されて以来、プログラムの機能を段階的に詳細化することが、プログラミングの基本手法の一つとなっている。段階的詳細化の結果を手続きの階層構造で表現すると、プログラムの機能を階層的に表現できるので、プログラムが理解しやすくなる。しかし、これだけではデータ依存関係が明確にならない。機能とデータ依存関係を関連させて階層的に表現することにより、大幅にプログラムが理解しやすくなるものと考える。データ依存関係を階層的に表現するためには、手続きの完全な入力と出力を知ることが必要である。本稿では、手続きの完全な入出力を明確にした。この結果は、データ依存関係を階層的に表現するプログラミング方法論の理論的基礎の1つとなるものである。

5 おわりに

本稿では、手続きの入力と出力を決める際に生じる問題を指摘し、その解決法について議論した。この問題を解決するために、まず、手続き及び手続き呼び出し文の入出力を明示する記述法を提案した。次に、この記述法により記述された手続きを、命令をノードとするフローグラフで表した。このフローグラフを用いてデータフロー解析を行い、手続きの出力と入力が満たすべき条件をデータフロー方程式で表現した。このデータフロー方程式を基に、手続きの入力と出力を定義した。この定義に従えば、次のような性質を持つ手続きのインタフェースが得られる。

（1）手続きの中で副作用を受ける可能性のある外部変数が出力である。

（2）出力が確定した値を持つために、手続きの実行に際して、親手続きが値を設定しなければならない変数が入力である。

本稿で与えた入出力の定義に従って手続きの入出力を決めると、冒頭で指摘した問題が生じないことを例を用いて示した。本稿で得られた結果は、手続きのインタフェース設計、プログラムのコメント文、プログラミング言語、プログラミング方法論などに利用できる。今後の展開として、データ依存関係を階層的に表現するプログラミング方法論を確立したいと考えている。

謝辞 本研究を始める機会を作って下さった、豊橋技術科学大学の北川 孟教授に感謝します。

参 考 文 献

1)情報処理学会 編:情報処理ﾊﾝﾄﾞﾌﾞｯｸ,ｵｰﾑ社(1989)

2)Myers,G.J.:COMPOSITE/STRUCTURED DESIGN, Van Nostrand Reinhold Company, (1978)［邦訳 國友義久、伊藤武夫 訳:ｿﾌﾄｳｴｱの複合/構造化設計、近代科学社、(1979)］

3)Stevens,W.P., Myers,G.J. and Constantine,L.L. : Structured Design、IBM Syst.J., Vol.13, No.2, pp.115-139(1974)

4)Warnier,J.D. and Flanagan,B.M.(鈴木君子訳):構造化ｼｽﾃﾑ設計、日本能率協会 (1980)

5)Jackson,M.A.: PRINCIPLES OF PROGRAM DESIGN, Academic Press(1975)
［邦訳 鳥居宏次 訳:構造的ﾌﾟﾛｸﾞﾗﾑ設計の原理、日本ｺﾝﾋﾟｭｰﾀ協会(1980)］

6)國友義久：効果的プログラム開発技法［第3版］，近代科学社（1991）

7)Martin,J. and McClure,C: ACTION DIAGRAMS Clearly Structured Program Design,Prentice-Hall(1985)［邦訳 國友義久：構造化プログラム設計／行動ダイアグラム、近代科学社（1988）］

8)大場克彦、金戸孝夫:ﾌﾟﾛｸﾞﾗﾑ論理図の形式的表現、ｿﾌﾄｳｴｱ工学80-20、pp.151-158(1991.7.19)

9)大場克彦，湯浅太一：可読性の高いアルゴリズムの記述法，プログラミング3－5，pp．21－28（1995）

10)Hamilton,M. and Zeldin,S.: The Relationship Between Design and Verification, The Journal of Systems and Software 1, pp.29-56, Elsevier North Holland,Inc.,(1979)

11)下村隆夫：プログラミングスライシング技術と応用、共立出版（1995）

12)Aho,A.V., Sethi,R. and Ulman,J.D.: Compilers Principles,Techniques,and Tools, ADDISON-WESLEY PUBLISHING COMPANY(1986)
［邦訳 原田賢一 訳：コンパイラ 原理・技法・ツール II、サイエンス社（1990）］

13)Dijkstra,E.W.: "Notes on Structured Programming" in "Structured Programming", Academic Press(1972)［邦訳 野下浩平 訳:構造化ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ論、構造化ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ、ｻｲｴﾝｽ社(1975)］

14)Wirth,N: Program Development by Stepwise Refinement, CACM, Vol.14, No.4, pp.221-227 (1971)