プログラムの正当性証明入門 1

シドニー・L・ハントラー　ジェイムス・C・キング

訳 玉井哲雄

この論文は，プログラムの正しい動作を入力/出力表明 (input/output assertion) を用いて仕様として記述する方法と，これらの表明に関してプログラムが正しいことを示す一つの方法とを，入門風に説明したものである．

初期表明は，プログラムの入口において真であるべき条件を特徴づけるものであり，終了表明は，プログラムの出口において成り立つべき表明を特徴づけるものである．プログラムに分岐がない場合は，記号実行(symbolic execution) と呼ばれる技法が，プログラムの入口で初期表明が正しければ，出口で終了表明が成り立つことを示すのに使える．

一般に，分岐を含むプログラムに対しては，記号実行木 (symbolic execution tree) が定義できる．もしそのようなプログラムの各繰返し (loop) が実行される回数に上限があるならば，正当性の証明はその (有限な) 記号実行木をたどることで簡単に与えることができる．

しかしながら，ほとんどのプログラムでは，各繰返しの実行回数に対して固定的な上限が存在しないから，対応する記号実行木も有限ではなくなる．そのようなプログラムの正当性を証明するには，より一般性のある表明の構造が与えられなければならない．このようなプログラムの記号実行木は，直接にではなく帰納的にたどられる必要がある．このことから，“帰納表明”と呼ばれる新たな表明が，自然に導入されることになる．

Sidney L. Hantler and James C. King, An Introduction to Proving the Correctness of Programs, CACM Vol. 8, No. 3, Sept. 1976.

はじめに

現代の電子計算機の出現以来，計算機プログラムがその意図どおり動くことを証明することへの関心は，常にあった．計算機プログラムの大きさ・複雑さが増すに従い，それらのプログラムが確実に動くことを保証することもその重要性を増してきつつある．

そこで，何が確かな動作であるかを正確に仕様化する問題，およびプログラムが常にこれらの仕様を満たすことを検査する完全な手法の開発が，とくに注目されるようになった．

この論文は，プログラムが仕様 (specification) と一致していることを示す一つの方法について，解説的に述べることを意図している．

“正当表明”を使うという基本的手法およびここで用いる特殊な帰納法の形式は，Floyd [7] による．また，この論文で説明する表明の構成法は，Deutsch [5] や Topor [21] の手法と類似する．書き方は形式的にきちんとはしていない．すなわち，定理を述べ証明するという形をとらない．プログラムを実行するという意味がわかり，簡単な代数や数学的概念を知っている読者には，ここで述べる考え方は理解しやすいはずである．同じような内容を厳密に取り扱ったものは，ほかにある [6, 13, 18].

まず，きわめて簡単なプログラミング言語を定義することから始める．

この言語は簡単ではあるが，通常使われるプログラミング言語の重要な基本的要素はみんな含んでいる．この論文のすべての例題は，この言語で記述してある．

2節では，この言語で書かれたプログラムについての正当性という概念が展開される．プログラムの記号実行が，正当性の証明を構築する基本的な道具として，3節で導入される．

4節*では，証明手法がさらに繰返し構造をもつプログラムに対して，展開される．この技法の汎用性を示すために，5節では，サブルーチンや関数を取り扱えるように，拡張がなされる．最後に6節において，プログラムの検証と，その計算機による機械化技術の現状について，未解決な問題の研究に力点をおきながら，議論を行なう．

1. プログラミング言語と意味論

この節では，正当性の考え方を導入するのに適当な，PL/I 風の簡単なプログラミング言語を，説明する．説明の便と，専門的な細部を極力省くために，基本的な文型と簡単な算術式しか持たない，特別に単純な言語を選んだ．

手続きは，次の形式の文で宣言される．

名前：PROCEDURE $(p_1, p_2, p_3, \cdots, p_n)$;
〈文のリスト〉
END;

ここで名前は手続きの名前であり，$p_1, p_2, p_3, \cdots, p_n$ は手続きのパラメータである．通例のように，手続きの本体は，PROCEDURE とそれに対応する END のあいだにおかれた文のリストより成る．手続きには2種類ある．すなわち，1) 算術式の中で参照される関数，と 2) CALL 文により陽に呼び出されるサブルーチン，である．この区別は，後でより詳しく論じる．

プログラム変数は整数値を取り，DECLARE 文で宣言される．

DECLARE 変数$_1$, 変数$_2, \cdots,$ 変数$_n$ INTEGER; という文により，変数$_1$, 変数$_2, \cdots,$ 変数$_n$ という名前の整数値をとる変数がつくられる．これらの変数は，宣言された手続き内でのみ知られ，手続き呼出しのたびに“新しい”世代がつくられる (cf. PL/I の automatic 変数)．プログラム変数の値に算術演算をほどこすことにより，新しい値が得られる．プログラム変数の値や整数定数のあいだには，たし算 (+)，かけ算 (×)，ひき算 (−) をほどこすことができる．基本的な代入文は，次の形をとる．

変数←〈式〉

ここで変数は，宣言されたプログラム変数であり，〈式〉は，宣言された変数，整数定数，ならびに正しい数の引数に適用された関数名よりなる算術式である．

代入文の右辺に関数名が現われると，その名に結びついた関数手続きが呼び出される．したがってもし，名前というのが1つのパラメータをもち，引数を3として呼ばれると7という値を返す手続きであるとすると，

変数←(2×名前(3))+4;

という文を実行した結果は，変数の値が18になる．

複数の文は，DO; 〈文のリスト〉 END; という構成により複合文としてまとめることができる．このようなDO-END の対で囲まれると，文のリストは単文であるかのように扱われる．

ブール根源語 (Boolean Primitive) は，真，偽，および（代入文の右辺に現われるような）算術式を比較演算子で結合したものからなる．ここで比較演算子とは，より小さい (<)，より大きい (>)，等しい (=)，そしてそれらの反対 (≥, ≤, ≠) よりなる．ブール式（以下で〈ブール〉と書く）はブール根源語を，論理積(&)，論理和(|)，含意 (→) および否定 (¬) で結合することにより構成される．ブール式の値は，真または偽である．

2分岐条件は，次の形式をとる．

IF 〈ブール〉 THEN 文$_1$ ELSE 文$_2$

ここで文$_1$ と文$_2$ は，文または複合文である．通例のように，文$_1$ または文$_2$ が，ブールの真理値によって実行される．

繰返し文は，次の形式をとる．

DO WHILE 〈ブール〉; 〈文のリスト〉 END;

制御が DO WHILE 文にいたると，もし〈ブール〉の値が真であれば，文のリストが実行され，制御は DO WHILE にもどる．もし〈ブール〉が偽であれば，制御 END は文の直後の文に渡される．

上に述べたように，関数手続きは算術式の中で関数名が参照されることによって，呼び出される．パラメータは，あとでサブルーチン手続きについて論ずるところで説明するのと同様に，受け渡される．パラメータを通じてもたらされる変化のほかに，関数手続きは，これを呼び出した式の評価で使われる，1つの特別な値を返す．サブルーチンは，CALL 文によって呼び出され，パラメータが受け渡される．たとえば，

CALL 名前 $(a_1, a_2, a_3, \cdots, a_n)$;

という文によって，名前と名づけられたサブルーチンが呼び出され，形式パラメータ $p_1, \cdots, p_n$ の名前がそれぞ

* （訳注）4節以降は，次号に掲載する．

```
1  ABSOLUTE:
     PROCEDURE (X);
3      DECLARE X, Y INTEGER;
4      IF X<0
5        THEN Y←−X;
6        ELSE Y←X;
8      RETURN (Y);
9  END;
```

図 1 関数手続き *ABSOLUTE*

れ実引数 $a_1, \cdots, a_n$ に対応づけられる．パラメータの受け渡しは，PL/I の“参照による”流儀に従う．すなわち，実引数への参照がサブルーチン（または関数）へ渡される．引数が式のときは，その値が一時的な記憶場所にしまわれ，それへの参照が，サブルーチンに渡される．

正当表明の中で手続きのパラメータの初期値に言及する便宜上，パラメータの初期値は，プライム符号をつけた記号で表わされる特別な手続きの変数にしまわれるものとする．たとえば，上述の CALL 文の実行のあと，名前へ入ったときの $a_1, \cdots, a_n$ の値は，それぞれ変数 $p_1', \cdots, p_n'$ に保存される．

呼び出された手続きからの復帰は，RETURN 文によってなされる．RETURN 文は，手続き内のどこに置かれてもよい．各手続きは，唯一つの RETURN 文をもつ．RETURN 文は，サブルーチンにおいては，

```
RETURN;
```

という形式をとり，関数では，

```
RETURN(〈式〉);
```

という形式をとる．後者では，〈式〉の値を関数の値として返す．

2. プログラムの正当性

第 1 節で簡単なプログラミング言語を定義したので，ここでこの言語で書かれた手続きの“正当性”の意味について論ずることとする．

手続きの意図に従った動作を，正しく形式化する方法を用意しよう．とくに手続きへの入力に対する制約と，入力と出力間に成り立つべき関係が，プログラム変数の上での表明として表わされる．入力表明とは，次の形式の文である．

```
ASSUME (〈ブール〉);
```

これは普通，PROCEDURE 文の直後に現われる．たとえば，入力表明

```
ASSUME (p1>0);
```

```
1  ABSOLUTE:
     PROCEDURE (X);
2      ASSUME (真);
3      DECLARE X, Y INTEGER;
4      IF X<0
5        THEN Y←−X;
6        ELSE Y←X;
7    PROVE ((Y=X'|Y=−X') &
                          Y≥0 & X=X');
8  RETURN (Y);
9  END;
```

図 2 正当表明つきの手続き *ABSOLUTE*

は，パラメータ p_1 の値が，手続きの入口において正であると仮定することを，表明している．出力表明は，次の形式の文である．

```
PROVE (〈ブール〉);
```

これは，普通，手続きの RETURN 文の直前に現われる．たとえば，出力表明

```
PROVE ((X=Y') & (Y=X'));
```

は，変数 X と Y の値が入れ替えられたことを示す．これが正しい入れ替えの手続きによって満たされるべき，入力と出力の関係になっていることに，注意されたい．

当然のことながら，手続きの“正当性”の考え方は，この入力表明，出力表明と手続きの本体とのあいだの関係を，反映せねばならない．

手続きが，（その入力表明と出力表明に対して）正当であるのは，手続きの入口で入力表明が真であれば，手続きの出口で出力表明が真であることが保証される場合である．

この定義では，プログラムの終止の問題が省かれていることに，注意されたい．直観的にいえば，手続きは，それが終止したときに，予期されたように動作すれば，正当であるとする．これは，“部分的正当性”と呼ばれることも多く，その場合は“正当性”あるいは“完全正当性”という用語は，部分的に正当であり，かつどんな入力に対しても必ず終止するような手続きに対して用いるようにする．

図 1 に簡単な手続きを示した．

ABSOLUTE という関数は，そのパラメータの絶対値を返すものである．*ABSOLUTE* への入力パラメータについて，何の仮定も必要としないので，入力表明は ASSUME (真) となる．出力表明は，RETURN 文が実行されたときに，手続きの変数の値が，パラメータ X の初期値の絶対値になっていることを，規定しなければ

ならない．したがって，出力表明として手続きが何をなしているかにつき記述するのに適切なもの(の一部)は，PROVE ((*Y*=*X*′|*Y*=−*X*′) & *Y*≥0) となる．あとでれるが，手続きが何をしないかということを規定することも重要なことである．手続き *ABSOLUTE* では，*X*の値は変化しないということも規定したより完全な出力表明は，PROVE ((*Y*=*X*′|*Y*=−*X*′) &*Y*≥0 & *X*=*X*′) となる．

正当表明の入った手続きは，したがって図2のようになろう．

この例では，手続きが正しいことは，明らかである．次節で，（入力および出力表明のついた）手続きが正しいことを証明する形式的な（formal）手法を，論ずることとする．

3. 記号実行と記号実行木

プログラムの正当性を証明するということは，あらゆるプログラム入力に対して証明するということである．実際，特定な入力の有限な（小さな）集合しか使わないのでは，一般にこのような証明はできない．証明は，あらゆる入力に関して，成り立つ事実に基づかねばならない．

任意のプログラム入力を表わすのに，数学でよく行なわれるように記号化するという技法を用い，それらの記号を使った証明を試みるというやり方ができる．もし，あらゆる入力に仮定される性質以外の特別な性質を，証明のために必要としないのであれば，その証明は，任意の特定な入力に対して成り立つものとなる．あるいはもし，証明を構成するために，記号に対し特別な性質を仮定せねばならないのなら，場合を尽くしたケース分析を行ない，各ケースにそれぞれ証明を与えて，それらの証明の集まりによって全体の証明を構成することができる．

図2の簡単なプログラム *ABSOLUTE* の正当性証明を行なうために，この戦略を単純に適用してみよう．

ABSOLUTE の通常の適用結果は，記号的な引数，たとえばαを用いて *ABSOLUTE*（α）と表わすことができる．この記号αを*X*の入力値として用いて，プログラムを実行させよう．

ASSUME 文の実行は，その引数が真であって入力αに対し何らの制約もしていないので，意味をもたない．IF 文の実行は，もっと面白い．ここでは，*X*の値が負かどうか，すなわち α<0 か否かを定めねばならない．

ソフトウェア産業論(2)

小野勝章

次　郎　兄さん，ところでソフトウェアと言うのは売りものになるのかい。

一　夫　ソフトウェアと言ってもOSや言語処理系のようなものは，ハードの一部くらいに考えたほうがいいんじゃないの。

次　郎　ユーザ向けの，いわゆるアプリケーション・ソフトはどうなの。

一　夫　日本の市場はかなり貧しいと思うね。

次　郎　なぜ，どんな点で貧しいんだい。

一　夫　まず，既成服に身体を合わせるという努力をしたがらないことだ。事務処理システムではとくにそうだね。それから，他人のやっていることだけで満足できなくて，なにか一ひねりしたいという要求があるんだね。これは技術屋さんに多い。

次　郎　だけど，コンピュータを利用したいのだがプログラムを作るのが面倒くさいから，既存のソフトを使いたいと考えている潜在的ユーザはずいぶんいると思うよ。

一　夫　使用目的のバラエティが多すぎるんだね。たとえば特殊な目的の図形処理システムを作ると，開発費として何千万円もかかる。それを少数の販売で回収しようとするから，1本何百万もの値段になってしまうんだ。

次　郎　でも，統計パッケージや数学パッケージなどは数万円で手に入るのがあるよ。

一　夫　そういうのはすでに償却ずみのソフトだから安いのさ。

次　郎　そうなのか。でも兄さんの会社でも，パッケージの販売を計画しているんだろう。

一　夫　兄さんのところではいま非線形の最適化や高等数学計算パッケージに力を入れているんだ。また，小型計算機用の線形計画パッケージを3万円前後で販売するつもりだ。これらは償却ずみではなくて，次郎の言う潜在的ユーザのためにとくに開放するわけなんだよ。

次　郎　そうかい。では僕の課長にも買ってもらうように相談してみよう。

パッケージについてのお問合せは
〒105　東京都港区芝公園3-1-22 協立ビル1階
03-438-0845(代) 小野勝章事務所までどうぞ。

もし，αが3を表わしているなら答えはノーであり，αが-3ならイエスである．この質問に答えるには，αの値に関してある仮定をおかねばならず，ケース分析が必要となる．

ケース1：$\alpha<0$とする．このときはIF判定が真をもたらし，実行はTHEN節へ進む．そこではYはXの値の負（すなわち$-\alpha$）となる．PROVE文にきたとき，現在の値がこの場合について$((Y=X' \mid Y=-X') \ \& \ Y\geq 0 \ \& \ X=X')$を満たすことを，示さなければならない．$Y=-\alpha$かつ$X=X'=\alpha$であるから，この式は

$$(-\alpha=\alpha \mid -\alpha=-\alpha) \ \& \ -\alpha\geq 0 \ \& \ \alpha=\alpha$$

となり，これは簡単化すると$-\alpha\geq 0$，あるいはもっと簡単に$\alpha\leq 0$となる．したがって，PROVE文の正しさをいうには，$\alpha\leq 0$を示せばよいことになる．しかし，われわれは$\alpha<0$を仮定したから，証明は自明となる．すなわち，$\alpha<0$の場合，プログラムは正しい．

ケース2：$\alpha\geq 0$とする．このときはIF判定が偽をもたらし，実行はELSE節へ進む．そこではYは，Xの値すなわちαになる．PROVE文にきたとき，$Y=\alpha$，$X=X'=\alpha$に対して，$((Y=X') \mid (Y=-X') \ \& \ Y\geq 0 \ \& \ X=X')$が真であることを示さなければならない．それは

$$(\alpha=\alpha \mid \alpha=-\alpha) \ \& \ \alpha\geq 0 \ \& \ \alpha=\alpha,$$

あるいは簡単に$\alpha\geq 0$が真であることを，示すことである．$\alpha\geq 0$を仮定したから，証明は再び自明である．

IF文の性質により，この2つのケースは場合を尽くしており（$\alpha<0$であるかまたは$\alpha\geq 0$），両者は正しい結果をもたらす．ゆえにプログラムは正当である．

この例で，いくつかの点が指摘されよう．

ケース分析で使われた仮定は，分岐の定まらないIF文の実行によって，生じたものであった．この仮定は，IF判定を評価したものとその否定とにほかならず，いずれも入力αのみによるブール値をもつ式であった．これらの仮定は，各ケースについて，PROVE文の正しさをいうための前提として必要となった．

記号実行

この節では，前の例で漠然と用いた“記号実行”という基本的な手法を，より精細により完全に説明することを試みる．

ここで使っているプログラミング言語の範囲内で，この言語をインプリメントするのに用いられる計算の機能（すなわちプログラム実行の機構（mechanism））を，整数の上で算術演算をするというものから，記号式の上での代数的演算に変えて，その結果を考えてみよう．

たとえば，変数XとYがそれぞれ値として，記号αとβとを持つとしよう．$X\leftarrow Y+X$という文を実行した結果として，Xは$\alpha+\beta$という式になる．次に$Y\leftarrow 3\times X-Y$を実行すると，Yの新しい値として，$3\times\alpha+2\times\beta$という式が記号的に計算される．

記号を操作する計算機の上で，プログラムを実行すると，出力変数の値として，入力された記号の上での代数式が得られるであろう．そこで，これらの結果を出力表明と比べて調べることにより，プログラムの正しさ，あるいは誤りを立証できるように思われる．しかし，上に示したような*ABSOLUTE*というきわめて簡単な例をみても，このことが決してそれほどやさしいことではないことがわかる．この例では，記号を含んだブール式が，必ずしも真または偽に簡単化されないということから，ケース分析が必要となった．たとえば，$\alpha\geq 0$が正しいかどうかは，αについて何らかの情報がないと，決められない．

しかしながら，記号実行は，このようなケース分析と一般的な帰納法とを付加すれば，プログラムの正当性を立証するための完全な手段を提供する．第1節で与えたプログラム実行の定義を，プログラムが記号あるいは記号式を入力として受け取り，代数演算を実行できる計算機の上で実行されるものとして，みなおしてみよう．

手続き呼出し（CALLか関数参照）では，制御の移行と引数とパラメータの結びつけは，前と同じように働く．同様に，RETURN文の意味も変わらない．

もっと面白いことが生じる構成要素として，まず代入文をあげることができる．

通常の代入文の実行は，まず右辺の式の中のすべての変数をその値で置き換え，指定の演算を実行し，その結果得られた値を左辺の新しい値として，代入する．記号実行では，これに相当することを代数的に行なう．右辺の式中の変数は，その値（演算子の適当な有効範囲を保つために括弧つきになる）で置き換えられる．変数の値は式であるから，指定の算術演算は数値的に行なうことはできず，代数におけるように，単に記号的に表わされることになる．その結果の記号式が，左辺の変数の新しい値になる．

算術式が関数呼出しをともなう場合は，事情がもっと複雑になるので，手続き呼出しを直接取り扱う後節で論ずる．また，右辺の式中の変数に値を代入した結果の数

式に対し，代数的な式の簡素化をほどこすべきか否かという問題もある．もし簡素化が行なわれないと，数式は，入力が数値であった場合になされるはずの計算の過程を，正確に示すことになる．実際，これらの計算はこの数式に従って後から行なうこともでき，オーバーフローやその他の計算機の異常事態も含めて，同じ結果を与える．

しかしながら，簡素化をしないと，数式が大変繁雑になる．すぐ後で見るように，これらの式に対して定理の証明が必要となるが，式が大変繁雑であると，この証明がより困難になる．

この論文の目的は，プログラムの正当性証明の基本的考え方を，できるかぎり，簡単に説明することであるから，この難しい問題については論じない．例題の中では，そのほうがよければ数式を簡素化することがある．理論的には，数式が簡素化されようともされまいとも，基本的な手法は正しい．極端にいえば，大変難しい定理証明と仕様記述（簡素化をしない場合）をするか，簡素化をした場合には，実際の計算機の実行には必ずしも常に正確には対応しない結果となるかの，両者のうちの一方を選ばなければならない．

条件分岐文の記号実行も，通常の実行に並行するが，さらに複雑なところがある．

まず IF 文を考えよう．この記号実行は，ブール式中のすべての変数を括弧つきの値で置き換えることから始まる．その結果の式は，真，偽，または記号的なプログラム入力の上のブール式と等しくなろう．この最後の場合は，*ABSOLUTE* の例題で行なったようなケース分析に帰着する．ブール値の結果が真でも偽でもない場合は必ず，その結果が偽となるようなプログラム入力の数値が少なくとも一つはある．実行が，THEN 節に行っても ELSE 節に行っても，あらゆる入力に対して正しいということにはなり得ない．

前の例題において，ケースを決める仮定が，後で PROVE 述語の真であることを証明するのに必要となったことを思い起こされたい．

この仮定はまた，後に続く条件文の実行によって生じうる副ケースのうちあり得ないものについて考えることを避けるためにも必要となる．たとえば，次の連続する二つの IF 文の実行を考えてみよう．

IF $X<0$ THEN $Y\leftarrow 88$;
IF $X=3$ THEN $Y\leftarrow 99$;

ここで，Xの値$=\alpha$とする，構文的には，この二つの文

を通る路は4種類あるが，意味的には，そのうち3つが可能である．不可能な副ケースは，最初の文における選択を実行するために必要となるαに対する条件（すなわち，$\alpha<0$，$\alpha\geq 0$）を覚えておいて，2つめの文における選択の際に，整合するものをとるように使うことによって，検出することができる．

この議論から，“路の条件（path condition）”略してpcという概念が導かれる．これは記号実行の状態の一部をなし，それぞれのケース，副ケース，副々ケースなどを決定するプログラムの記号入力に対する条件を，その値としてとる．pcは，記号実行の初めでは真に初期化され，新しいケースが考慮されるたびに更新される．

IF 〈ブール〉 THEN 文$_1$ ELSE 文$_2$

という形のIF文の記号実行の完全な記述は，次のようである．

1) 〈ブール〉を記号入力に対して評価し，値Bを得る．

2) Bと$\neg B$という副ケースをつくるべきか否かを，決定する．もしpc→Bであれば，すでにある仮定により（pcに記録されている）文$_1$が常に次に実行されると判定できるので，新たな副ケースは必要とならない．記号実行は，直接文$_1$へ進む．同様に，もしpc→$\neg B$であれば，記号実行は直接文$_2$へ進む．もし（pc→B）でも（pc→$\neg B$）でもない場合は，それぞれ手順3と4に述べるように，Bおよび$\neg B$の新しい副ケースが必要となる．

3) Bを仮定して，副ケースをつくる．pcを新たな条件Bによって，(pc_{old} & B) に置き換えることによって更新する．ただし，pc_{old}はpcの一番最近の値である（すなわち，pc←pc & B という代入を行なう）．この場合は，記号実行は変更されたpcをもって，文$_1$へ進む．

4) $\neg B$を仮定して，副ケースをつくる．pcを新たな条件$\neg B$によって，(pc_{old} & $\neg B$) に置き換えることによって更新する．この場合は，記号実行は変更されたpcをもって，文$_2$へ進む．

IF文の実行で，評価されたブールが直接真あるいは偽に帰着される場合は，手順2で終わる．pcが偽であること（不可能な路）は決して許されないから，どのようなpcに関しても，(pc→真) は真であり（pc→偽）は偽である．

プログラムの正当性を示すASSUMEとPROVE文の記号実行もまた，次のように定義できる．

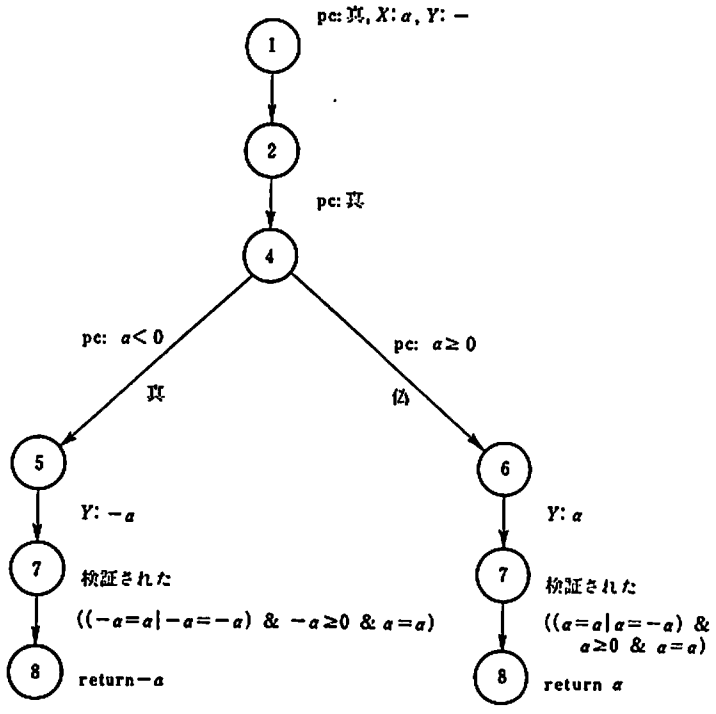

図 3 手続き *ABSOLUTE* の記号実行木

ASSUME (〈ブール〉):

1) 〈ブール〉を，変数に括弧つきの値を代入することによって，評価する．その結果をBと呼ぶ．

2) pcを (pc_{old} & B) に更新する．

これは，後に続く記号実行を，〈ブール〉が真であるケースに制約する効果をもつ．このことはまさに，入力表明の意図するところである．

PROVE (〈ブール〉):

1) 〈ブール〉を，変数に括弧つきの値を代入することによって，評価する．その結果をBと呼ぶ．

2) もし (pc→B) なら，“検証された”と印字し，そうでなければ“検証されない”と印字する．

この文は，このケースにおいてプログラム変数の値が出力表明を満たすか否かによって，“検証された”あるいは“検証されない”という印字をする．このケースを定義する条件は，pc で与えられる．

図2の *ABSOLUTE* のようなプログラムの完全な記号実行は，“記号実行木”によって簡潔に表わすことができる．この例に対する木は，図3に示されている．

これはプログラムのフローチャートに似て，各文の実行が節点で表わされ，文の実行間の制御の移行が枝で表わされる．節点はプログラムの文番号でラベルづけられ，文から出る枝は，先行する文の実行による実行状態の変化がもしあればそれによって，ラベルづけられる．もちろん，条件文実行の節点は，後続の文の選択が未決定のときは2本以上の枝が出ることになる．非実行文

```
1  GCD:
     PROCEDURE (M, N);
2      ASSUME (M>0 & N>0);
3      DECLARE M, N, A, B INTEGER;
4      A←M;
5      B←N;
6      DO WHILE (A≠B);
8        IF A>B
9          THEN A←A−B;
10         ELSE B←B−A;
11     END;
12     PROVE (A=(M, N));
13     RETURN (A);
14   END;
```

図 4 正当表明の入った手続き *GCD*

(たとえば **DECLARE**) の節点は，場所の節約のために，ここに示した木からは省いてある．

図3の木はプログラム *ABSOLUTE* のあらゆる可能な実行を尽くしており，そのそれぞれで“検証された”と印字されているので，*ABSOLUTE* は正しい．

さらにまだ，**DO WHILE** 文の記号実行について論じなければならない．

この文がなければわれわれの言語ではプログラムの繰返しをつくることができず，繰返しのないプログラムのもつ実行木は，常に有限である．*ABSOLUTE* 手続き

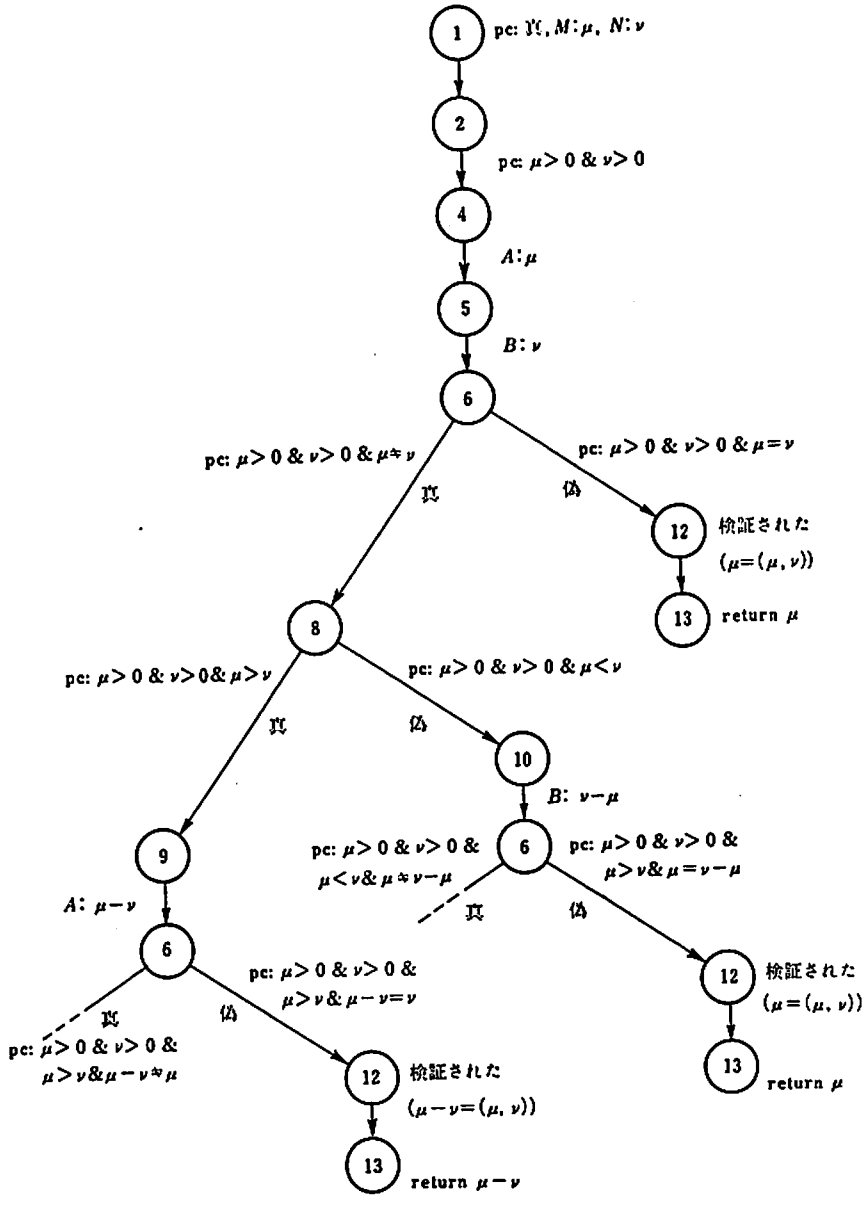

図 5 手続き *GCD* の記号実行木

の証明にあったように，記号実行は有限の記号実行木をもつ手続きの正当性証明には，都合のよい手段となる．そのようなプログラムは，その記号実行木の各葉*において“検証された”と出れば，正しい．

しかし，次のような DO WHILE という反復文

DO WHILE 〈ブール〉; 〈文のリスト〉 END;

を含む手続きに対しては，無限の記号実行木ができるであろう．この記号実行は，IF 文の記号実行から自然に導かれる．文のリストを実行するか，END の後へ続く文へ行くか，あるいはこの両者の選択を副ケースとして展開するかの決定は，IF 文でなされたようにブール式を調べることによって定められる．

図4の手続きに対する無限実行木の例を図5に示す．

図4の手続きは，正の数の入力 M と N に対し，その最大公約数を計算するものである．手続きの正しさは，標準的な数学の記法で仕様として書かれており，そこで，(M, N) は M と N の最大公約数を意味する．たとえば $(3, 12)=3$, $(20, 15)=5$ そして $(4, 4)=4$ である．

* (訳注) 木における葉 (leaf) とは，木の上の節点で，そこから出る枝のないものをいう．

a と b を整数としよう．最大公約数は，次の3つの公理で特徴づけられる．

$(a, a)=a$　ただし $a>0$ の場合

$(a, b)=(b, a)$

$(a, b)=(a+b, b)$

図5でわかるように，木の無限をなす部分は，記号入力を含む一つの条件の無限列によってできていることに注意されたい．

無限記号実行木をもつプログラムに対して，無限のプログラムの実行をひき起こすような特定の入力が存在するわけでは，必ずしもない．記号実行木が無限になるのは，ある実行に対し常に別の，より文の実行回数の多い実行があるからである．もちろん，停止しないような実行のあるプログラムは，無限実行木をもつ．

さて，すでに述べた手法が，無限実行木を生成するようなプログラムに，どのように適用できるであろうか．

この問題に一般的に答えるのは，無限路を“横切る”帰納技法である．これは4節で詳しく論じる．これを用いないとすれば，無限実行木の中の有限部分木に注意を制限することにより，すでに論じた問題に帰着できる．記号実行は有限実行木をもつ手続きに関しては，実際に正当性の証明を与えることを思い起こされたい．

この点を，図4の手続き *GCD* の変形を考えることによって示してみよう．

たとえば，この手続きの最初の ASSUME 文を ASSUME (偽) と置き換えたとしよう．その結果えられる手続きは，有限実行木をもつだけでなく（事実，空の木である），正しいことも保証される．無限実行木の空の部分木 (subtree) は，もちろん極端なものであり，部分木として調べる意味のないものである．プログラムの初期表明により巧みな制約を与えると，もっと意味のある部分木を選ぶことができる．

最初の ASSUME 文を，ASSUME ($M>0$ & $N>0$ & $M=C\times N$ & $C\leq 1000$) に変えたとしよう．そうすれば，1つの変数が他方の小さな倍数であるケースに対応する *GCD* 手続きの有限部分木に，注意を制限することになる．帰納法といった手段の助けをいっさい借りずに，記号実行を用いてこの変更された手続きの正当性証明を行なうことができる．

われわれの主要な関心が，この変更された手続きでなく，もとの *GCD* 手続きにあるかぎり，無限記号実行木の有限部分木を考慮することは，むしろ検査 (testing) の一つの形と考えたほうが，おそらくよいであろう．有限部分木が，関心のあるテストケースを表わすことになる．記号実行を用いる検査は，従来の検査技法と比べて少なくとも次の2つの点で異なることに注意したい．

一つは，通常の検査は，多くとも有限個の特定の入力を取り扱うのに対し，記号実行による検査は無限個の特定入力を取り扱う．二番目に，正当証明が手続きに入れられた場合，記号実行の検査は，単に考慮している各テストケースに対し出力値を与えるのではなく，考慮しているテストケースに対し正当性の証明を与える，ということがいえる．

検査技法として，記号実行はきわめて有望な新しい道具であるように思われる．これは最近の Clark [4] による Ph. D. 論文の題材であり，また，Boyer たち [1] は，記号実行を用いたテストケースの生成を研究している．また，筆者らとその同僚は，プログラムの証明とともに，プログラムの検査をする機能を含んだ，Effigy というプロトタイプ記号実行システムを開発した．

（つづく，全2回）

（文献は次号に一括して掲載します．）

（訳　たまい　てつお　(株)三菱総合研究所）