解　説

プログラムの検証理論*

林　健志**　五十嵐　滋**

1. はじめに

プログラムの検証は，プログラムの理論の典型的な応用分野の1つである．検証の直接的な目的は，いうまでもなく，その機能について信頼性の高いソフトウェアを得ることであるが，間接的な効用として各種ソフトウェアに関する概念を明確にえぐり出して，工学的に厳密な取り扱いに耐えるものにするということがある．

前者については，たとえば，指導的なプログラマーの1人であるダイクストラは次のように論じている***．「プログラムのテストは，虫の存在を示すために極めて効果的に用い得るが，それが存在しないことを示すためには絶望的に無力であり，納得できる正当性の証明が，(プログラムに虫がないことの) 必要な確信のレベルに到達する唯一の方法であるように思われる．納得できる正当性の証明が存在するために，このような正当性の証明は次の2つの条件を満たすものでなければならない．

(1) それは1つの証明でなければならぬ．それ故一群の公理から出発することを要する．

(2) それは我々が納得できるものでなければならない．それ故，我々は証明を書き上げ，チェックし，理解し，評価できなければならない」(直訳．括孤内筆者)．

専門的な立場から補足すれば，(2)については大容量のファイルを伴った人間一機械系が必須の条件になっている．

一方，後者の，間接的効用はある意味では前者以上に重要だと考えられる．いずれにせよこのような背景から，76年10月サンフランシスコでの第2回国際「ソフトウェア工学」会議には，ホーア，マンナと共に筆者の1人が論文委員に加わるように委嘱されているし，また77年8月トロントでのIFIPコングレスの「情報処理の理論的基礎」部門**** についても形式的検証が重視され，筆者の1人がコングレス論文委員兼部門議長に指名されている．

ところで今日多少の成功を納めている検証理論はすべて公理論的ないしは形式主義的なものである．それには必然的な理由があるが，本稿の読者の大部分を占めるであろうプログラマーのために端的にいえば，このことはプログラム言語がすでに著しくフォーマルな対象である事実と無縁ではない．プログラム言語開発の目的は，ごく初期の50年代から60年代の初めにかけては，単に計算機にプログラムを誤り無く読ませることとしてしか一般には理解されていなかったが，次第に，アルゴリズム上の概念を人間が正しく効果的に操作することをいかに保証するかがむしろ重視されるようになった．そして明らかな意識をもって行うか否かに拘らず，一旦概念の表現や操作を追求しはじめると，我々の営みは最早半分はフォーマルな世界に入ってしまうことになる．歴史的にはIFIPのWG 2.2***** は，WG 2.1がALGOL 68の設定に当たったのと表裏一体の関係を保ちつつ，このフォーマルな世界の問題を技術的に検討して来たものである．その経験に則していえば，フォーマルな方法は一見最も迂遠に見える場合でも，実は唯一の正解一合理的な問題解決の方法であることが，ソフトウェアの工学的取り扱いでは起こるものである．

---

* Theory of Program Verification by Takeshi HAYASHI and Shigeru IGARASHI (Research Institute for Mathematical Sciences, Kyoto University)

** 京都大学数理解析研究所

*** "A Simple Axiomatic Basis for Programming Language Constructs," Indagationes Mathematicae Vol. 36, pp. 1～15 (1974). これは，1973年のIFIP WG 2.3 (Programming Methodology) の会合での講演に基づいている．

**** コングレスは8部門に分かれ，各部門は6セッション位になる予定である．なお「プログラム方法論」は理論部門に入れられることになった．

***** 1966-74 のタイトルは "Formal Language Description Language". 1974 以降 "Formalization of Programming Concepts". 意味論における公理的方法は日本で生まれ，WG 2.2 を通じてホーアの記法やスコットのカルキュラスを生み出す源流となった．

本稿ではフォーマルな対象を扱うための道具として1階述語論理を用いるが，これは我々に知られている道具の中では最も簡明なものである．この論理は固有の言語，すなわち1階言語を伴っている．動的な概念表現であるプログラム言語と，この静的な概念表現である1階言語とを巧妙に調和させることにより，検証のための言語である表明言語が構成される．表明言語は，ある意味では，在来のプログラム言語のコメントの部分を，プログラム本体の部分と同様に厳格な文法に従うものにし，さらに精密なセマンティクスを付与して得られる言語である．紙数の都合で1階述語論理を詳しく解説することはできないが，本文および付録に述べた範囲で一通り厳密な定義が理解できるように工夫してある．

2. 表　明

2.1 プログラムの検証とは

プログラムの**検証** (verification) とは，あるプログラム言語で書かれたプログラムが与えられるとき，そのプログラムが与えられた**仕様** (specification) を満たしていることを検査し証明することである．プログラムが与えられた仕様を満たしているとき，そのプログラムはその仕様に関して**正当** (correct) であるという．通常，プログラムは，与えられたデータを変換して，目的の結果を算出している．よって，あるプログラムについての仕様としては，まずデータがどのような領域から選ばれているかを示し，かつそのプログラムの実行が意味のあるものとなるために満たされるべきデータに関する制限を記述しておく必要がある．と同時に，計算結果がどのようなものになるかの記述も必要である．この様な記述のことを**表明** (assertion) と呼ぶ．

表明は，従来は，いわゆるドキュメントやコメントの形で自然言語で書き表されてきた．しかし仕様を満たしていることを証明するためには，自然言語では曖昧さが残り充分ではない．さらに，計算機を使って**正当性の証明**をさせることを考えた場合にはもちろんであるが，計算機によって**ソフトウェアのドキュメント管理・解析**を行う際ですらも，表明の簡潔かつ厳格な記述方法が必要となる．すなわちデータに関する制限や結果の記述のための言語が必要になってくる．この様な言語のことを**表明言語** (assertion language) と呼ぶ．表明言語は元のプログラム言語と双対的な関係にある．後者が**動的概念**を表現することに対して，前者は**静的**である．話を抽象的にしておけば，この関係が見易くなる上に，いつでも実用言語に適用できるから，我々としては，以下に見るように，1階述語論理に基礎を持つ表明言語を用いることにする．本稿で必要な1階述語論理に関する知識は付録に掲げておいた．

2.2 表明言語

前節での考察から分かるように，プログラムの検証のための理論を展開するためには，

（1） 取り扱うプログラムのクラス
（2） データや結果の表明方法
（3） 正当性の証明方法

を明確に定めて議論する必要がある．これら，(1)～(3)を統一的に扱うために，我々は1階述語論理に基づく方法をとることにする．その他の方法に関しては例えば LCF[1]をみよ．本節では，(1)を扱い，**2.3** で(2)，**3.** で(3)を扱う．

さて，(1)に関しては，PASCAL[2]のような現実のプログラム言語をそのまま取り扱うことも可能である(例えば，ILL システム[3])* が，本稿では理論的な話を述べることを主眼としているので，次のような考え方に基づいて抽象的に定義されたプログラムのクラスについて論述する．

あるプログラム言語が与えられたとき，そのプログラム言語が持っている**制御構造**と**データ構造**に注目して，そのプログラム言語を1階述語論理を基礎にした記述方法の枠内に翻訳してやれば，正当性証明のための理論的展開が容易になる．我々は以下では制御構造として，**割当** (assignment)，**合成** (composition)，**条件** (if-then-else)，**リカーシブ・プロシージャー** (recursive procedure) を考え，データ構造としては説明を簡単にするために，1つのタイプ（例えば自然数）の単純な変数のみを仮定する．〔本稿で考えるプロシージャー宣言は，次の様な制限を満たしているものとする (PASCAL[2]，ホーア[4] 参照)．グローバルな変数は認めず，また仮引数を可変 (variable) 引数と定値 (value) 引数の2つに分類し，定値引数に現れている変数は宣言の本体の中で割当ステートメントの左側にも，プロシージャー・コールの可変引数部に実引数としても現れないように制限する．プロシージャー・コールにおいては，可変引数はコール・バイ・ネームで，定値引数はコール・バイ・ヴァリューで呼ばれ

* これは論文（文献3)）の著者達の頭文字から来た俗称であって，論文中でこう呼ばれているわけではないが，この呼び名はその後欧米でかなり使われるようになった．

る.〕

すると，上に述べた翻訳は次の様な形で行えばよい*．まずプログラム言語中の変数に対しては，1階言語における（自由）変数が対応し，プログラム言語中の定数に対しては，1階言語の定数が対応し，掛け算，たし算等の算術演算を表す記号（＊，＋等）には，関数記号が対応する．するといわゆる算術式には1階言語の項（term）が対応することが分かるであろう．等号や不等号などの関係を表す記号には，述語記号が対応し，論理演算には対応する論理記号を考える．そうするといわゆる論理式（Boolean expression）には，量記号なしの命題（quantifier-free formula）が対応していることが分かるであろう．すなわち，そのプログラム言語がもっている演算の記号を非論理記号と考え，その機能を公理の形で述べてやれば，そのプログラム言語に自然に対応した1つの1階理論Tを考えることができるわけである．〔例えば，データ構造として自然数の単純な変数をとった場合には，Tは自然数論を拡張したものになる．〕以下の議論ではこの1階理論Tを固定して考える．

さて次にそのプログラム言語の制御構造に対応する記号をL(T)（Tの1階言語）の記号の他に新しく導入する．我々は，制御構造としては，割当，合成，条件，リカーシブ・プロシージャーを考えているのだから，新しい記号として，$\leftarrow$，；，**if-then-else** をそれぞれ割当，合成，条件に対応して導入し，リカーシブ・プロシージャーに対しては，**プロシージャー記号**（procedure symbol）と呼ばれる新しい記号を導入する．各プロシージャー記号には予じめ対 (m,n) $(m\geq 0,\ n\geq 0)$ が一意に対応づけられる．ここに，m,n はそれぞれ可変引数，定値引数の個数である．対(m,n)が対応づけられたプロシージャー記号を (m,n) **変数プロシージャー記号**と呼ぶ．

以上の準備のもとで，我々の考えているプログラムは次の様にして形式的に記述できる．まずステートメントを帰納的に定義する．

定義 2.1 ステートメント

1) 空の記号列 Λ はステートメントである．
2) x を変数，t をL(T)の項とすれば，
$x\leftarrow t$ はステートメントである．
3) p を (m,n) 変数プロシージャー記号とし，$x_1,\cdots,x_m$ を互いに相異なる変数，$t_1,\cdots,t_n$ を $x_1,\cdots,x_m$ を含まないL(T)の項としたとき，
$p(x_1,\cdots,x_m;\ t_1,\cdots,t_n)$ はステートメントである．
4) A,B がステートメントならば，
$A;\ B$ もステートメントである．
5) A,B がステートメントで，F がL(T)の量記号なしの命題ならば，
if F **then** A **else** B もステートメントである．

定義2.1の(1)～(5)によって定義されるステートメントのことを，それぞれ，空ステートメント，割当ステートメント，プロシージャー・ステートメント，合成ステートメント，条件ステートメントと呼ぶ**．

さて次にプロシージャー宣言に対応するものを形式的に定義するのであるが，そのためにステートメント A に対して，A の中で割当ステートメントの左側に現われているか，プロシージャー・ステートメントの可変実引数になっている変数の集合 lef A を与える写像 lef を A の構成に関する帰納法で定義する．

定義 2.2

1) A が Λ のとき，lef $A=\phi$.
2) A が $x\leftarrow t$ のとき，lef $A=\{x\}$.
3) A が $p(x_1,\cdots,x_m;\ t_1,\cdots,t_n)$ のとき，lef $A=\{x_1,\cdots,x_m\}$.
4) A が B,C のとき，lef $A=$lef $B\cup$lef C.
5) A が **if** F **then** B **else** C のとき，lef $A=$lef $B\cup$lef C.

次に，プロシージャー宣言を示す新しい記号 **proc** を用意する．するとプロシージャーの定義（宣言）の定義は次の様になる．

定義 2.3 プロシージャーの定義

p を (m,n) 変数プロシージャー記号，$x_1,\cdots,x_m$，$y_1,\cdots,y_n$ を互いに相異なる変数とし，B をステートメントとしたとき，

$$p(x_1,\cdots,x_m;\ y_1,\cdots,y_n)\ \mathbf{proc}\ B \qquad (2.1)$$

なる形をしたものを**プロシージャーの定義**と呼ぶ．ただし，B の中に現れている変数の集合を occ B とすれば，occ $B-$lef $B=\{y_1,\cdots,y_n\}$，$\{x_1,\cdots,x_m\}\subseteq$lef B となっているとする．以後，見易くするために，ベクトル表記を使って，(2.1)式を，

$$p(\boldsymbol{x};\ \boldsymbol{y})\mathbf{proc}\ B \qquad (2.2)$$

---

* 以下の議論で必要な1階述語論理に関する知識は付録に述べてある．

** 記号 Λ と $\leftarrow$ をそれぞれ，基本的な (0,0) 変数プロシージャー記号，(1,1) 変数プロシージャー記号と考え，シンタックスの自明な拡張を行えば，定義2.1の(1)，(2)は(3)に含まれることに注意せよ．

と書く．ここで注意しておかねばならないことは，我々はリカーシブ・コールを許している点である．一般にステートメント A の中にたかだか $p_1, \cdots, p_n$ がプロシージャー記号として現れていることを強調するために，$A(p_1, \cdots, p_n)$ と書くことにする．すると，(2.2)式の B の中には，他のプロシージャー記号 $p_1, \cdots, p_n$ とともに p 自身が現れていてもよいので，B は $B(p_1, \cdots, p_n, p)$ と書ける．

定義 2.4 プログラム

プロシージャーの定義の有限列の後に1つのステートメント A をつけ加えた列：

$p_1(\boldsymbol{x}_1; \boldsymbol{y}_1)$**proc** $B_1(p_1)$, $p_2(\boldsymbol{x}_2; \boldsymbol{y}_2)$**proc** $B_2(p_1, p_2)$, $\cdots$, $p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)$**proc** $B_n(p_1, \cdots, p_n)$, $A(p_1, \cdots, p_n)$

をプログラムと呼ぶ．ここで $p_1, \cdots, p_n$ はすべて相異なる．この定義ではいわゆる相互リカージョン (mutual recursion) が排除されていることに注意せよ．

これで我々のプログラムが定義できたわけであるが，ここで制御構造においてループを導入するもの，例えば，**while-do** をなぜ取り入れなかったかについて簡単に説明しておく．任意の while ステートメント **while** F **do** A (PASCAL式) に対しては次の様にして，それと等価なリカーシブ・プロシージャーを書き下すことができるからである．whiledef$(\boldsymbol{x}; \boldsymbol{v})$**proc if** F **then** A ; whiledef$(\boldsymbol{x}; \boldsymbol{v})$**else** A, ここで $\boldsymbol{x}$ は，lef A の変数を並べた列であり，$\boldsymbol{v}$ は F または A に現れている $\boldsymbol{x}$ 以外の変数の列である．だから実は我々の立場は while ステートメントを除外したのではなく，むしろ，いわゆる**定義された記号**としてそれを取り込めるだけの力を持っているのである．つまり正当性を証明したいプログラムが与えられたら，その while 部分はリカーシブ・プロシージャーに書直して証明すれば済む．

2.3 表明の方法

プログラム $p_1(\boldsymbol{x}_1; \boldsymbol{y}_1)$**proc** $B_1, \cdots, p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)$**proc** B_n, A が与えられたとき，各プロシージャーの定義 $p_i(\boldsymbol{x}_i; \boldsymbol{y}_i)$**proc** B_i に対して，プロシージャー $p_i(\boldsymbol{x}_i; \boldsymbol{y}_i)$ の仕様と，A の仕様を与えてやれば，全体のプログラムに対して，仕様を与えたことにもなる．プロシージャー $p_i(\boldsymbol{x}_i; \boldsymbol{y}_i)$ の仕様とステートメント A の仕様を決めるためには，これらのプロシージャーまたはステートメントにどの様なデータが投入され，実行の後どの様な結果を算出するかを与えてやればよい．

* $\vdash_T F$ は形式的体系 T において F が証明可能の意．付録の8.を参照せよ．

このことを形式的に表現するために，次の定義を考える．

定義 2.5 表明式 (assertion formula)

1) L(T) の命題は表明式である．
2) F, G を L(T) の命題，A をステートメントとしたとき，$F\{A\}G$ は表明式である．
3) プロシージャーの定義 $p(\boldsymbol{x}; \boldsymbol{y})$**proc** B は表明式である．

(1)の定義は，仕様を書くための言語（表明言語）として，もともとの L(T) の命題を利用することを述べている．(2)の定義は直観的には，「F が真であるような開始状態に対して，A が**停止するならば**，G は終了状態において真である．」ことを表す表現である．すなわち A が F と G に関して**正当** (correct) であることを主張している一種の命題である．$F\{A\}G$ の解釈を次の様に変えたとき，A は F と G に関して**強正当** (strongly correct) と呼ばれる．「F が真であるような開始状態に対して，A は**必ず停止し**，かつ G は終了状態において真である．」本稿では強正当に関しては扱わないことにする．強正当の理論的扱いに関しては，例えば，五十嵐[5]，マンナ[6]を参照されたい．(3)の定義はある意味でプロシージャーの定義自体も仕様の一部であることを主張している．

プログラム $p(\boldsymbol{x}_1; \boldsymbol{y}_1)$**proc** $B_1, \cdots, p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)$**proc** B_n, A に対して，各プロシージャー p_i と A の開始状態における条件を命題 $F_i(\boldsymbol{x}_i, \boldsymbol{y}_i)$, F で与え，終了時における条件を $G_i(\boldsymbol{x}_i, \boldsymbol{y}_i)$, G で与えてやる．すると定義2.5の(2)の形をした表明式の列：

$F_1(\boldsymbol{x}_1, \boldsymbol{y}_1)\{p_1(\boldsymbol{x}_1; \boldsymbol{y}_1)\}G_1(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, F_n(\boldsymbol{x}_n, \boldsymbol{y}_n)\{p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)\}G_n(\boldsymbol{x}_n, \boldsymbol{y}_n), F\{A\}G$

は，$p_1, \cdots, p_n$ をサブルーチン，A を主プログラムと考えれば，それぞれサブルーチンの仕様，主プログラムの仕様を形式的に書き与えたものであると理解される．

3. 証明体系

我々の目的は，プログラム P: $p_1(\boldsymbol{x}_1; \boldsymbol{y}_1)$**proc** $B_1, \cdots, p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)$**proc** B_n, A と，それらに対する表明式の列 $F_1\{p_1(\boldsymbol{x}_1; \boldsymbol{y}_1)\}G_1, \cdots, F_n\{p_n(\boldsymbol{x}_n; \boldsymbol{y}_n)\}G_n, F\{A\}G$ が与えられたとき，L(T) の命題の列，$E_1, \cdots, E_m$ を求め，もし $\vdash_T E_i (1 \leq i \leq m)$ ならば，$\vdash_T F_j\{p_j(\boldsymbol{x}_j; \boldsymbol{y}_j)\}G_j (1 \leq j \leq n)$ かつ $\vdash_T F\{A\}G$ であるような形式的体系Cを作ることにある*．1階述語論理の命題である $E_1, \cdots, E_m$ (または $E_1 \wedge \cdots \wedge E_m$) のことをプロ

グラム P の**検証条件** (verification condition) と呼ぶ．すなわち，P の検証条件が数学的に証明されれば，P の正当性が論理的に保証される体系 C を作るわけである．**形式的体系** C はいくつかの**公理**および，**前提**と呼ばれる表明式の列を**結論**と呼ばれる常に定義 2.5 の(2)の形をした表明式に変換する規則である**推論規則**から成り立っている．各々の推論規則は，次の $R1$ または $R2$ の形をしている．

$$R1.\ \frac{H_1, \cdots, H_n}{K}, \quad R2.\ \frac{H_1, \cdots, H_n, J}{K}\ \ (I)$$

ここで，$H_1, \cdots, H_n, I, J$ は前提の表明式，K は結論の表明式，$R2$ 型の推論規則は1つしかないが，リカーシブ・コールに関する表明式を証明する際に用いられる．記法 $\overset{(I)}{J}$ は，論理学における自然演繹法（例えば，島内[7]，松本[8]参照）の場合と同様に，I を仮定して J が導かれることを意味している．つまりこの推論規則は，I を仮定して J が導びかれるとき，$H_1, \cdots, H_n$ が成り立っているならば，仮定 I とは無関係に K が成り立っていることを主張している．このとき，仮定 I はこの推論規則によって**取消される** (discharge, **落ちる**ともいう．) と言う．一般には $H_1, \cdots, H_n$ にも J にも他の仮定があり得るが，それらは結論に遺伝して行く．

推論規則を樹の枝の状態に積み重ねてできる図形を考える．この図形において，一番下に現れる表明式のことを**終表明式**と呼び，先端になっている表明式のことを**始表明式**と呼ぶ．ある表明式 K が C で**証明可能** (provable) であるとは，K を終表明式とする図形が存在して，しかも始表明式のうち公理でない表明式（すなわち仮定）すべてから K が独立である——すなわちすべての仮定が取り消されている——ときに言う．このときこの図形のことを K の**証明図**と呼ぶ．次に C の公理と推論規則を表にして示す．

公　理

$C1$.　割当公理：$F|_t^x\{x \leftarrow t\}F$.

$C2$.　枠組公理 (frame axiom)：$F\{A\}F$, lef $A \cap$ occ $F = \phi$ のとき．A が Λ のときは，lef $A = \phi$ だから，任意の F について，$F\{\Lambda\}F$ となっている点に注意せよ．

$C3$.　プロシージャーの定義：$p(\boldsymbol{x}; \boldsymbol{y})$**proc** B

$C4$.　1階理論 T の任意の定理 F（データによって定まる公理群である）

推論規則

$C5$.　論理的帰結：$\dfrac{E \supset F, F\{A\}G}{E\{A\{G}, \quad \dfrac{E\{A\}F, F \supset G}{E\{A\}G}$

$C6$.　and/or：$\dfrac{E\{A\}F, G\{A\}H}{E \wedge G\{A\}F \wedge H}, \quad \dfrac{E\{A\}F, G\{A\}H}{E \vee G\{A\}F \vee H}$

$C7$.　合成：$\dfrac{E\{A\}F, F\{B\}G}{E\{A;\ B\}G}$

$C8$.　条件式：$\dfrac{E \wedge F\{A\}G, E \wedge \neg F\{B\}G}{E\{\textbf{if } F \textbf{ then } A \textbf{ else } B\}G}$

（ただし F は量記号なしの命題）

$C9$.　代入：（左）$\dfrac{F(\boldsymbol{x}, \boldsymbol{y})\{q(\boldsymbol{x}; \boldsymbol{y})\}G(\boldsymbol{x}, \boldsymbol{y})}{F(\boldsymbol{z}, \boldsymbol{y})\{q(\boldsymbol{z}; \boldsymbol{y})\}G(\boldsymbol{z}, \boldsymbol{y})}$

（右）$\dfrac{F(\boldsymbol{x}, \boldsymbol{y})\{q(\boldsymbol{x}; \boldsymbol{y})\}G(\boldsymbol{x}, \boldsymbol{y})}{F(\boldsymbol{x}, \boldsymbol{s})\{q(\boldsymbol{x}; \boldsymbol{s})\}G(\boldsymbol{x}, \boldsymbol{s})}$

ただし，$C9$. は次の制限を満たしているときのみ使用可能：(i) $\boldsymbol{s}$ は $\boldsymbol{x}$ の変数を含んでいない．(ii) $\boldsymbol{z}$ は互いに相異なる変数からなり，かつ $F(\boldsymbol{x}, \boldsymbol{y})$, $G(\boldsymbol{x}, \boldsymbol{y})$ の変数のうち，$\boldsymbol{x}$ に属するものは含んでいてもよいが，それ以外の変数は含んでいない．

$C10$.　プロシージャー・コール：$\dfrac{[F\{r(\boldsymbol{x}; \boldsymbol{y})\}G]}{p(\boldsymbol{x}; \boldsymbol{y})\textbf{proc } B(p_1, \cdots, p_n, p), F\{B(p_1, \cdots, p_n, r)\}G \over F\{p(\boldsymbol{x}; \boldsymbol{y})\}G}$

ただし p は $F\{B(p_1, \cdots, p_n, r)\}G$ を証明するための部分証明図の中には現れず，かつ r はその証明図中の $F\{r(\boldsymbol{x}; \boldsymbol{y})\}G$ 以外の仮定には現れていない．ここで r は任意のプロシージャーを表現している変数で，いわゆる eigenvariable の一種である[7),8)]．

この表では次の表記法を採用した．E, F, G, H—L(T) の命題；A, B—ステートメント；$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$—変数の列；$p, p_1, \cdots, p_n, q, r$—プロシージャー記号；$\boldsymbol{s}, \boldsymbol{t}$—L(T) の項の列と項；$p(\boldsymbol{x}; \boldsymbol{y})$ 中の $\boldsymbol{x}(\boldsymbol{y})$ は p の可変（定値）引数；$F|_t^x$ は，F の x に t を代入して得られる命題（付録の **5.** を参照）；occ F は F の自由変数の集合；$B(p_1, \cdots, p_n, r)$ は $B(p_1, \cdots, p_n, p)$ の p の出現をすべてプロシージャー記号 r で置き換えたものを示す．

例：ここで次の様な簡単なプログラムについてCにおける証明を行ってみよう．この場合1階理論Tとしては，定数記号0を含む通常の自然数論に，2変数関数記号 mod および gcd を公理とともに添加して拡張したものをとる．ここに，mod(m, n) は m を n で割った余り，gcd(m, n) は m と n の最大公約数と解釈する．証明図の形で証明を与えるスペースがないので，番号を付けて記してある（計算機のディスプレイでやるときはむしろこれに近い形が使われる）．

プログラム: $g(x;\ m, n)$**proc if** $\mathrm{mod}(m, n)=0$ **then** $x \leftarrow n$ **else** $g(x;\ n, \mathrm{mod}(m, n)),\ g(x;\ m, n)$

証明すべき表明式: **true**$\{g(x;\ m, n)\}\ x=\mathrm{gcd}(m, n)$

この場合，ここに掲げたものはいわば縮退したものであって，本章の最初に掲げた一般の形ではもう1つ同じ表明式を書き並べたものになる*.

1. $\mathrm{mod}(m, n)=0 \supset n=\mathrm{gcd}(m, n)$ 補題1 (C4)
2. $n=\mathrm{gcd}(m, n)\{x \leftarrow n\}\ x=\mathrm{gcd}(m, n)$ C1
3. $\mathrm{mod}(m, n)=0\{x \leftarrow n\}\ x=\mathrm{gcd}(m, n)$ C5 (1, 2)
4. **true**$\wedge \mathrm{mod}(m, n)=0 \supset \mathrm{mod}(m, n)=0$ C4
5. **true**$\wedge \mathrm{mod}(m, n)=0\{x \leftarrow n\}\ x=\mathrm{gcd}(m, n)$ C5 (3, 4)
6. **true**$\{r(x;\ m, n)\}\ x=\mathrm{gcd}(m, n)$ 仮定
7. **true**$\{r(x;\ n, \mathrm{mod}(m, n))\}\ x=\mathrm{gcd}(n, \mathrm{mod}(m, n))$ C9右(6)
8. $\mathrm{mod}(m, n) \neq 0\{r(x;\ n, \mathrm{mod}(m, n))\}\ \mathrm{mod}(m, n) \neq 0$ C2
9. **true**$\wedge \mathrm{mod}(m, n) \neq 0\{r(x;\ n, \mathrm{mod}(m, n))\}\ x=\mathrm{gcd}(n, \mathrm{mod}(m, n)) \wedge \mathrm{mod}(m, n) \neq 0$ C6 (7, 8)
10. $x=\mathrm{gcd}(n, \mathrm{mod}(m, n)) \wedge \mathrm{mod}(m, n) \neq 0 \supset x=\mathrm{gcd}(m, n)$ 補題2 (C4)
11. **true**$\wedge \mathrm{mod}(m, n) \neq 0\{r(x;\ n, \mathrm{mod}(m, n))\}\ x=\mathrm{gcd}(m, n)$ C5 (9, 10)
12. **true**$\{$**if** $\mathrm{mod}(m, n)=0$ **then** $x \leftarrow n$ **else** $r(x;\ n, \mathrm{mod}(m, n))\}\ x=\mathrm{gcd}(m, n)$ C8 (5, 11)
13. $g(x;\ m, n)$**proc if** $\mathrm{mod}(m, n)=0$ **then** $x \leftarrow n$ **else** $g(x;\ n, \mathrm{mod}(m, n))$ C3
14. **true**$\{g(x;\ m, n)\}\ x=\mathrm{gcd}(m, n)$ C10(12, 13)

ここに補題1と補題2と記したものは，この証明のために必要な，理論Tに依存した（gcd に関する）数学的命題である．本章の始めで述べたように，この様な命題のことを検証条件と呼んでいる．我々の目的はこの様な検証条件を（計算機を使って）自動的に作り出すことにある．検証条件を自動的に作り出すシステムのことを **VCG** (verification condition generator) と呼ぶ．次章で VCG について論じる．

* **true** は $0=0$ の省略形（付録の4.参照）．証明中の $\mathrm{mod}(m, n) \neq 0$ は $\neg \mathrm{mod}(m, n)=0$ の省略形である．

** ILL システムでは実際的アルゴリズムを反映するように VCG が定義されているが，本稿では原理的にバックトラッキングが可能であることを示すことのみを目的として，数学的にはより簡単な VCG を与える．

*** すなわち $F\{p(x;\ y)\}G$ が C で証明可能であることの（メタな）証明．

4. V C G

以下誤解のおそれのない限り，表明式を単に表明ということにする．さていま1つのプロシージャー・コールだけからなるプログラム P: $p(x;\ y)$proc $B(p)$, $p(x;\ y)$ と（縮退した）表明 $F\{p(x;\ y)\}G$ が与えられているとする．そのときの VCG の働きを考えてみよう**．検証条件は1階理論 T の命題 I であり，$\vdash_T I$ ならば $\vdash_C F\{p(x;\ y)\}G$ を満たしていなければならない．さて $\vdash_C F\{p(x;\ y)\}G$ を証明***するためには，**3.** の表中のプロシージャー・コール規則 C10 を使えば，プロシージャー p の定義は与えられているので，

$$F\{r(x;\ y)\}G \vdash_C F\{B(r)\}G \tag{4.1}$$

が証明出来ればよい．式 (4.1) は，C に新たに表明 $F\{r(x;\ y)\}G$ を公理として添加して得られる形式的体系を C′ とするとき，$\vdash_{C'} F\{B(r)\}G$ であることを意味する．また r は p に対応する新しいプロシージャー記号 (eigenvariable) である．よって，

$$\vdash_T I \text{ ならば } F\{r(x;\ y)\}G \vdash_C F\{B(r)\}G \tag{4.2}$$

なる1階理論Tの命題 I を求めればよい．I を求めることが VCG の働きである．すなわちこの場合，VCG は仮定 $F\{r(x;\ y)\}G$ と表明 $F\{B(r)\}G$ をデータとして動き，結果として I を算出する．

より一般的なプログラム P: $p(x;\ y)$**proc** $B(p), A(p)$ と表明の列 $F_1\{p(x;\ y)\}G_1$, $F\{A(p)\}G$ が与えられたときには，上記の考察から分るように VCG はまず，

$$\vdash_T I \text{ ならば } F_1\{r(x;\ y)\}G_1 \vdash_C F_1\{B(r)\}G_1 \tag{4.3}$$

なる L(T) の命題 I を算出し，次に，

$$\vdash_T K \text{ ならば } F_1\{p(x;\ y)\}G_1 \vdash_C F\{A(p)\}G \tag{4.4}$$

なる L(T) の命題 K を算出する．$I \wedge K$ がプログラム P の検証条件になっていることは容易に分る．(4.4) 式を満たす K を算出するための VCG のアルゴリズムが(4.3)式を満たす I を算出する際にも適用できることは明らかであるから，以下このアルゴリズムについて記述する．

さて VCG は形式的体系Cをもとにして動くものであるが，ステートメント $A(p)$ 内のプロシージャー・ステートメントに関しては，次の（適合規則と呼ばれる）規則を使用している．

適合 (adapatation) 規則:

$$\frac{F(z, t)\{p(z;\ t)\}G(z, t)}{F(z, t) \wedge \forall z(G(z, t) \supset H(z, t))\{p(z;\ t)\}H(z, t)}$$

ここで，F, G, H は L(T) の命題，p はプロシージャー記号，z, t はそれぞれ変数と項の列であり，$\forall z$ は，z を $z_1, \cdots, z_n$ としたとき，$\forall z_1 \cdots \forall z_n$ を略記したものである．この規則が形式的体系Cで導出できることを次に示す．

1. $F(z,t)\{p(z;t)\}G(z,t)$ 仮定(適合規則の前提)
2. $\forall z(G(z,t)\supset H(z,t))\{p(z;t)\}$
   $\forall z(G(z,t)\supset H(z,t))$ $C2$
3. $F(z,t)\wedge\forall z(G(z,t)\supset H(z,t))\{p(z;t)\}G(z,t)$
   $\wedge\forall z(G(z,t)\supset H(z,t))$ $C6\ (1,2)$
4. $G(z,t)\wedge\forall z(G(z,t)\supset H(z,t))\supset H(z,t)$ $C4$
5. $F(z,t)\wedge\forall z(G(z,t)\supset H(z,t))\{p(z;t)\}H(z,t)$
   $C5\ (3,4)$

(4.4)式を満たす K を算出するために用いられている考え方は，ステートメント $A(p)$ をパーズして一番外側の構成要素に関して C の規則を用いて，順に内側に入っていくやり方で，一種の**逆向き導出** (backward derivation) である．適合規則は，プロシージャー・ステートメントが実行された後の時点の内部状態を表現している命題 $H(z,t)$ が与えられたとき，実行前に成り立っていなければならない命題の1つを与えていることに注意せよ．

さて任意のステートメント $A(=A(p))$ と L(T) の命題 H に対して，A を終了後 H が満たされているとき，A を実行する前に満たされていなければならない L(T) の命題 $\Phi(A,H)$ を与える写像 Φ は A の構成に関する (一般化された) 帰納法で次の様に定義できる．右端の ( ) 内は対応する C の公理または推論規則である．

定義 4.1

1) A が Λ のとき，$\Phi(A,H)=H$ $(C2)$
2) A が $x\leftarrow t$ のとき，$\Phi(A,H)=H|^x_t$ $(C1)$
3) A が $p(z;t)$ のとき，
   $\Phi(A,H)=F_1(z,t)\wedge\forall z(G_1(z,t)\supset H(z,t))$
   (適合規則)
4) A が $B;C$ のとき，
   $\Phi(A,H)=\Phi(B,\Phi(C,H))$ $(C7)$
5) A が **if** E **then** B **else** C のとき，
   $\Phi(A,H)=(E\supset\Phi(B,H))\wedge(\neg E\supset\Phi(C,H))$
   $(C8)$

すると任意のステートメント $A(p)$ と，任意の L(T) の命題 H について次式が成立する (証明は略す)．

$$F_1\{p(x;y)\}G_1\vdash_{TC}\Phi(A(p),H)\{A(p)\}H \qquad (4.5)$$

$\Phi(A(p),H)$ は上記の構成から直ちに証明できるように L(T) の命題であることに注意せよ．

よって(4.4)式を満たす検証条件 K は，

$$F\supset\Phi(A(p),G))$$

である．また(4.3)式を満たす検証条件 I は，

$$F_1\supset\Phi(B(p),G_1)$$

である ($\Phi(B(p),G_1)$ と $\Phi(B(r),G_1)$ が同一の命題になることを注意せよ)．すなわち，

$$\vdash_{TC}(F_1\supset\Phi(B(p),G_1)\wedge(F\supset\Phi(A(p),G))$$

ならば，$\vdash_{TC}F_1\{p(x;y)\}G_1$ かつ $\vdash_{TC}F\{A(p)\}G$ である (証明は読者に委ねる)．

一般に，$n\geq 2$ の場合についての，プログラム P: $p_1(x_1,y_1)$**proc** $B_1,\cdots,p_n(x_n;y_n)$)**proc** B_n, Aと表明式の列 $F_1\{p_1(x_1;y_1)\}G_1,\cdots,F_n\{p_n(x_n;y_n)\}G_n, F\{A\}G$ が与えられたときの VCG の構成については読者自ら試みられよ (ヒント：相互リカージョンがないことに注意)．

検証条件は純粋に数学的な命題であるから，人間が証明してやるか，または Theorem Prover に証明させればよい．なおここに採用した体系とその原型である ILL の体系では相互リカージョンの場合の証明能力が弱いことが指摘されている*．

5. 実　例

いままでに述べて来た方法を，プログラム言語 PASCAL の部分集合について実際に適用し，VCG を PDP-10 の上で走らせたものに，ILL システムがある[3]．ILL では先に述べた制御構造の他に，**while, goto, function** 宣言とコールを扱い，またデータ構造としては配列を含めている．**while** や **goto** をステートメントとして許した場合には，**表明付きプログラム** (asserted program) の考え方が必要になって来る．すなわちプログラム中の適当な場所に，注釈として表明式を入れてやったものが表明付きのプログラムである．例えば，while ステートメントに関する推論規則は，

$$\frac{F\wedge G\{A\}F,\ F\wedge\neg G\supset H}{F\{\textbf{while}\ G\ \textbf{do}\ A\}H}$$

として与えられる．ここで，F, H は L(T) の命題，G は L(T) の量記号なしの命題，A は (拡張された) ステートメントである．(この規則は **2.2** の最後で述べた考え方を用いれば C で証明できる．) この規則においてループ中で不変量になっている命題 F (キー・アサーションと呼ばれることがある) は計算機で

* 笠井琢美 (1973) による．

見付け出すことが一般にはできないので，あらかじめ人間が与えてやらねばならない．この様な表明Fをプログラム中に注釈として，対応する **while** ステートメントの直前に入れてやる必要がある．ILLでは，このことは ASSERT ステートメントとして，PASCALのシンタックスの中に取り入れられている．goto に関しても同様の取り扱いが必要である．

配列への割当 $B[j]\leftarrow t$ に関する公理は次の様な形になっている．

$F(\textbf{if } i=j \textbf{ then } t \textbf{ else } B[i])\{B[j]\leftarrow t\}F(B[i])$

goto に関する推論規則，**function** 宣言とコールを許した場合の公理や推論規則の拡張の仕方，および拡張された VCG の定義に関しては，文献 3) を参照されたい．

次の例は ILL システムの証明結果の実例である*．プログラムは，データ X が配列 $A[1..P]$ の何番目に格納されているかを調べる表引きのルーチンを表している．Aは互いに異なる要素から成るソート済みの配列である．この事実は SORTED(A) という表明で示されている．X が配列内にないときは，ERROR がTRUE，あるときは，FALSE になるように作られている．ENTRY および EXIT の後についているのは，それぞれ，入口と出口での表明を示している．また NOTFOUND(X, M, N) は，$A[M]$ から $A[N]$ に至る配列の部分には，Xが存在しないことを示している．

```
PASCAL ENTRY(1<P)∧SORTED(A)∧(A[1]≦X)∧
    (X<A[P]);
EXIT (A[LOOKUP]=X)∧(ERROR=FALSE)∨
    NOTFOUND (X,1,P)∧(ERROR=TRUE);
BEGIN M←1; N←P; ERROR←FALSE;
ASSERT (M<N)∧(A[M]≦X)∧(X<A[N])∧
    SORTED(A)∧(ERROR=FALSE);
WHILE M+1<N DO BEGIN
    I←(M+N)DIV 2;
    IF X<A[I]THEN N←I ELSE IF A[I]<X THEN
        M←I ELSE BEGIN LOOKUP←I; GO TO 1 END
    END;
IF A[M]≒XTHEN GO TO 2 ELSE BEGIN LOOKUP←
    M; GO TO 1 END;
2: ASSERT NOTFOUND (X,1,P); ERROR←TRUE;
1: ASSERT (A[LOOKUP]=X)∧(ERROR=FALSE)∨
    NOTFOUND (X,1,P)∧(ERROR=TRUE)
END.;
*****
PASCAL PROGRAM SUCCSSFULLY PARSED
FOR THE MAIN PROGRAM THE
8 VERIFICATION CONDITIONS ARE:
# 1 (1<P)∧SORTED(A)∧(A[1]≦X)∧(X<A[P])
    ⊃
    (1<P)∧(A[1]≦X)∧(X<A[P])∧SORTED(A)∧
        (FALSE=FALSE)
# 2 (M<N)∧(A[M]≦X)∧(X<A[N])∧SORTED(A)∧
    (ERROR=FALSE)∧(M+1<N)⊃X<[(M+N)DIV 2]
    ⊃
    (M<(M+N)DIV 2)∧(A[M]≦X)∧
    (X<A[M+N)DIV 2])∧SORTED(A)∧
    (ERROR=FALSE)
# 3 (M<N)∧(A[M]≦X)∧(X<A[N])∧SORTED(A)∧
    (ERROR=FALSE)∧(M+1<N)⊃
    ¬(X<A[(M+N)DIV 2]⊃A[(M+N)DIV 2]<X
    ⊃
    ((M+N)DIV 2<N)∧(A[(M+N)DIV 2]≦X)∧
    (X<A[N])∧SORTED(A)∧(ERROR=FALSE)
# 4 (M<N)∧(A[M]≦X)∧(X<A[N])∧SOTED(A)∧
    (ERROR=FALSE)∧(M+1<N)⊃
    ¬(X<A[(M+N)DIV 2])⊃¬(A[(M+N)DIV 2]<X)
    ⊃
    (A[(M+N)DIV 2]=X)∧(ERROR=FALSE)∨
    NOTFOUND(X,1,P)∧(ERROR=TRUE)
# 5 (M<N)∧(A[M]≦X)∧(X<A[N])∧SORTED(A)∧
    (ERROR=FALSE)∧¬(M+1<N)⊃A[M]≒X
    ⊃
    NOTFOUND(X,1,P)
# 6 (M<N)∧(A[M]≦X)∧(X<A[N])∧SORTED(A)∧
    (ERROR=FALSE)∧¬(M+1<N)⊃¬(A[M]≒X)
    ⊃
    (A[M]=X)∧(ERROR=FALSE)∨
    NOTFOUND(X,1,P)∧(ERROR=TRUE)
# 7 NOTFOUND(X,1,P)
    ⊃
    (A[LOOKUP]=X)∧(TRUE=FALSE)∨
    NOTFOUND(X,1,P)∧(TRUE=TRUE)
# 8 (A[LOOKUP]=X)∧(ERROR=FALSE)∨
    NOTFOUND(X,1,P)∧(ERROR=TRUE)
    ⊃
    (A[LOOKUP]=X)∧(ERROR=FALSE)∨
    NOTFOUND(X,1,P)∧(ERROR=TRUE)
*****
```

* Acta Informatica, Vol. 4, pp. 178~179 (1975) より転載した．

6. 展 望

プログラムの正当性の証明を計算機に手伝わせることの実験は 70 年以来，ウィスコンシン大，スタンフォード大，テキサス大，南カリフォルニア大，エジンバラ大で行われて来た．日本でも東京教育大一筑波大一慶応大のグループが独自のシステムを開発して実験をはじめている．京大（数理解析研）では未だディスプレイを介したインタラクティヴ・システムのサポーティング・ソフトウェアの開発をしている段階である．京大での構想は，今のところ，証明体系 directed な証明検査システムにすることと，そのシステムに証明および仕様に関する文書管理機能を伴わせることに重点をおいている．これらは，スタンフォード大，エジンバラ大グループとの交流・共同研究の経験から得

られた必須の要点である．というのは，証明の理論は日進月歩であり，それが扱うプログラムの範囲も，その言語も広がりつつあるので，1つの固定された体系による証明システムでは国際水準に遅れを取る．また証明を全部のレベルについてやる必要はなく，たとえば本稿中で gcd の検証条件として与えた程度のことは，人間が承認したことか出典・引用の記録でも残しておけば十分であって，むしろこういったことに関するドキュメントの記憶・管理を機械にさせることが大切だと考えられる．実際 $n!$ の計算のプログラムの証明位でないと，多くの人は，機械が補題，定理（すなわちプログラムの性質に関する段階的または部分的な知識群）を保管して，必要に応じてディスプレイしたり，プリント・アウトしてくれなければ，証明を書き上げることにいやけがさしてしまうだろう（LCF はすでに，ある程度このような機能を持っていた）．証明もソフトウェア完成までの1工程となるべきものだから，多数の人々が仕事を分担できるようなシステムにして置かねばなるまい．

さて実用的なソフトウェア群についてこのような証明と文書管理ができるようにするためには，どれだけの開発努力が必要であろうか．理想的なものを目指す計画は，おそらくアポロ計画と同じ規模にはならざるを得ないのではなかろうか〔証明技法の開発については，数値解析のように数学的には正当なアルゴリズムが証明と共に与えられている分野と，オペレーティング・システムやデータ・ベースのように概念を厳格化することからはじめなければならぬ場合を別々に考えなければならぬ．後者は要するに1つの工学理論を作ることに相当することはいうまでもない〕．その反面，ソフトウェアの正当性の証明を比較的インフォーマルに行うことは，従来の工学の知的レベルに比べて質的に格別高いものではなく，ただ問題も方法も新しいタイプの問題であることと，量的な大きさと多様性に違いがあるだけのように見える．従って，インフォーマルな証明を通じてのソフトウェアの信頼性の向上はきわめて近い将来に現実のものとなる可能性が高いと考えられる．その場合，証明まで見通して，あるいは少なくとも証明と交互に，プログラムを作る* ことが最も望ましく，それは現在のレベルでのスタイル論争を止揚した段階といえるであろう．

* ILL システムの VCG はそのような態度で開発された．本稿執筆に際してよりアブストラクトな定義を与えることにより，我々はこのソフトウェアについてさらに新しい知見を得た．

参考文献

1) R. Milner: Implementation and Application of Scott Logic for Computable Functions, Proc. ACM Conf. Proving Assertions about Programs, pp. 1～6 (1972).
2) K. Jensen & N. Wirth: PASCAL User Manual and Report, p. 170, Lecture Notes in Computer Science, No. 18, Springer-Verlag (1974).
3) S. Igarashi, R.L. London & D.C. Luckham: Automatic Program Verification I: A Logical Basis and its Implementation, Acta Informatica, Vol. 4, pp. 145～182 (1975).
4) C.A.R. Hoare: Procedures and Parameters: An Axiomatic Approach, Symposium on Semantics of Algorithmic Languages (ed. by E. Engeler), Lecture Notes in Mathematics, No. 188, Springer-Verlag, pp. 102～116 (1971).
5) S. Igarashi: A Natural Deduction System for Assertions, IFIP WG 2.2 Stuttgart Meeting (1974), および Séminaires IRIA, M. Nivat (ed.), pp. 39～45(1974).
6) Z. Manna: Mathematical Theory of Computation, p. 448, McGraw-Hill (1974). 五十嵐訳，プログラムの理論，p. 478，日本コンピュータ協会 (1975).
7) 島内剛一：数学の基礎，p. 520，日本評論社 (1971).
8) 松本和夫：数理論理学，p. 188，共立講座現代の数学1，共立出版 (1970).
9) Shoenfield: Mathematical Logic, p. 344, Addison-Wesley (1967).

展望と文献リストのために次のものを掲げておく．

10) 五十嵐滋：計算機科学における論理的検証，記号・情報・論理（科学哲学7），日本科学哲学会編，pp. 151～165，早稲田大学出版部 (1974).
11) 伊藤貴康：プログラム理論，p. 268，コロナ社 (1975).
12) 伊藤，五十嵐，野崎，金山，嵩，伊沢，房岡：プログラム理論，昭和50年電気四学会連合大会講演論文集，pp. 973～992 (1975).
13) E. Jensen: Prospects for Automatic Verification of Programs, p. 66, Technical Report, University of Michigan, distributed by NTIS (1974).

（昭和51年1月6日受付）

◇

付　録

本稿で用いた**1階述語論理** (first-order logic, 1階論理)の概要を簡単にまとめておく．詳しくは，Shoe-

nfield[9], 松本[8], 島内[7]などを参照されたい. ここでは Shoenfield の流儀に従うことにする.

1. 記 号

1階述語論理の言語の記号は次のものから成る.

a) (対象) **変数**: $x, y, z, x_1, y_1, z_1, \cdots$

b) 各 $n \geq 0$ について, **n 変数関数記号**と **n 変数述語記号**. 特に2変数述語記号として$=$ (等号) を含む. 0変数関数記号のことを (対象) **定数**と呼ぶ. $=$ 以外の関数記号と述語記号のことを**非論理記号** (nonlogical symbol) と呼ぶ.

c) 論理記号: $\neg, \vee, \exists$

(なお, 補助記号として括弧 ( ) 及びコンマをあいまいさを無くすために使う.)

2. 項 (term)

次の a), b)によって定義されたもののみを項という.

a) 変数は項である.

b) $t_1, \cdots, t_n$ が項で, f が n 変数関数記号のとき $f(t_1, \cdots, t_n)$ も項である ($n=0$ のときは f 自身が項である).

この様な定義の仕方は, (一般化された)**帰納的定義** (generalized inductive definition) と呼ばれる. 次の命題の定義も帰納的定義である.

3. 命題 (well-formed formula)*

次の a), b), c), d)によって定義されたもののみを命題という.

a) $t_1, \cdots, t_n$ が項で, p が n 変数述語記号のとき, $p(t_1, \cdots, t_n)$ は命題である ($n=0$ のときは p 自身が命題である).

b) F が命題ならば, $\neg F$ も命題である.

c) F, G が命題ならば, $F \vee G$ も命題である.

d) F が命題で, x が変数のとき, $\exists x F$ も命題である.

$\exists$ の記号を全く含まない命題のことを**量記号なしの命題**と呼ぶ. すべての命題の集合と**1.**～**3.**で述べた文法的概念を結合したものを**1階言語** (first-order language) と呼ぶ. 1階言語は非論理記号を定めてやれば完全に決まる.

4. 定義された記号 (defined symbol)

左の命題は右の命題の**省略形** (abbreviation) である.

$F \wedge G$	$\neg(\neg F \vee \neg G)$
$F \supset G$	$\neg F \vee G$
$\forall x F$	$\neg \exists x \neg F$

なお例えば $+(x, y)$, $=(x, y)$ は $x+y$, $x=y$ と書くことにする.

5. 代入 (substitution)

命題 F における変数 x の**出現** (occurrence) が F において**束縛** (bound) であるとは, その x の出現が, F の $\exists x G$ という形をした部分の中にあるときにいう. 束縛でないとき, **自由** (free) であるという. 変数 x は, 命題 F のどこかに x の自由な (束縛された) 出現があるならば, F の**自由 (束縛) 変数**であるという (x が同時に F の自由変数かつ束縛変数となることもある). 自由変数を1つももたない命題は**閉じている** (closed) といわれる. F を命題, t を項としたとき, t が F の変数 x に対して**代入可能** (substitutible) であるとは, t に出現している各変数 y に対して, F の $\exists y G$ の形をした部分の中には, x の自由な出現が存在しないときにいう. このとき, F における x の自由な出現をすべて t で置き換えて得られる命題を $F|_t^x$ で示す.

6. 公理 (axiom)

明らかに成立していると考えられる命題のことを**公理**と呼ぶ. 公理には**論理的公理** (logical axiom) と呼ばれている公理と, 考えている対象に固有な公理 (**非論理的公理**, nonlogical axiom) とがある. 次に1階述語論理の論理的公理を掲げる (論理的公理の与え方は一意的でなく, 同値なものが様々な人々により与えられている).

1)	$\neg F \vee F$	命題的公理 (排中律)
2)	$F|_t^x \supset \exists x F$	代入の公理
3)	$x=x$	同一性の公理
4)	$x_1=y_1 \supset \cdots \supset x_n=y_n \supset f(x_1, \cdots, x_n)=f(y_1, \cdots, y_n)$ および $x_1=y_1 \supset \cdots \supset x_n=y_n \supset p(x_1, \cdots, x_n) \supset p(y_1, \cdots y_n)$	等号の公理

ここで, F は任意の命題, f は任意の n 変数関数記号, p は任意の n 変数述語記号である.

7. 推論規則 (inference rule)

推論規則とは, **前提** (premise) と呼ばれる命題の列, $F_1, \cdots, F_n$ を**結論** (consequence) と呼ばれる命題 G に変換する規則のことで, 普通

$$\frac{F_1, \cdots, F_n}{G}$$

の様な形をしている. 紙面の都合上, これを以下 $F_1, \cdots, F_n \Rightarrow G$ と書くことにする. 1階述語論理の推論規則を次に掲げる (推論規則の与え方は一意的でなく,

* 伝統的には論理式と呼ぶ人が多い.

同値なものがいろいろ与えられている．また公理と推論規則の間には密接な関係があり，例えば Gentzen の LK の様に論理的公理は1つで，他はすべて推論規則を用いる体系もある[8])．

1) $F \Rightarrow G \vee F$　　拡張規則
2) $F \vee F \Rightarrow F$　　縮約規則
3) $F \vee (G \vee H) \Rightarrow (F \vee G) \vee H$　　結合規則
4) $F \vee G, \neg F \vee H \Rightarrow G \vee H$　　切断規則
5) $F \supset G \Rightarrow \exists x F \supset G$　　ヨ-導入規則

ただし x は G の自由変数ではない．

8. 証明 (proof)

命題の有限列 $F_1, \cdots, F_n$ において，各 F_i が公理であるか，または F_i より前にある命題のいくつかを前提として，ある推論規則の結論に F_i が成っているとき，列 $F_1, \cdots, F_n$ のことを F_n の**証明**と呼ぶ．命題 F に関して，証明が存在するとき**証明可能**(provable)と呼ぶ．証明可能な命題を**定理** (theorem) と呼ぶ．それ故公理はまた定理である．特に公理としては**6.**で与えた論理的公理だけを用い，また推論規則としては**7.**で与えたものだけを用いて，命題 F が証明可能であるとき，$\vdash F$ と記すことにする．命題 F が証明可能のとき，使用された各推論規則を樹の枝の状態につみ重ねてできる図形を考えることができる．この図形では，F が1番下の根の部分に現れており，また葉に相当する部分には，すべて公理が現れている．このような図形のことを F の**証明図** (proof figure) と呼んでいる．

9. 1階理論 (first-order theory)

一般に，そこで使う記号を定め，命題を定義し，命題の形をしたいくつかの公理を与え，いくつかの推論規則を与え，**8.**で述べた証明の概念を用いれば，証明可能な命題（定理）について議論することができる．このような，定理に関する議論をするための枠組を**形式的体系** (formal system) という．次の条件を満たす形式的体系 T のことを**1階理論**という．

1) T の言語 (L(T) と記す) は1階言語である．すなわち，Tで用いられる非論理記号を定めてやれば，L(T) は，**2.** および **3.** を使って定義されるすべての命題の集合（に文法的概念を伴わせたものに）になっている．
2) T の公理は L(T) の論理的公理 (**6.**で与えたもの) とその体系 T に固有な非論理的公理と呼ばれる公理から成る．
3) T の推論規則は**7.**であげたものだけから成る．

よって1階理論 T は，Tで用いられる非論理記号と非論理的公理を与えれば完全に定まる．L(T) の命題が証明可能のとき，F のことを T の定理と呼び，$\vdash_T F$ と記す．1階理論の代表的なものは，自然数論と集合論である．またプログラムで扱う程度の整数，実数*，ストリング，リスト等は1階理論で表現できる．

* 浮動小数点の数については辻尚史の学位論文（東大・工学系）がある．