Vol. 21 No. 6　　情　報　処　理　　June 1980

解　説

# プログラミング言語の意味論†

## —入門的解説—

伊　藤　貴　康††

## 1. まえがき

計算機の登場以来，いろいろなプログラミング言語が提案され，利用されている．プログラミング言語は，プログラマの意図をプログラムとして表現し，計算機に実行させるために用いられる．FOPTRAN や ALGOL のようなコンパイラ言語が 1960 年代に普及してから，プログラミング言語と言えば，このような高級言語の事を指すようになっている．プログラミング言語は，ソフトウェア工学の最も基本的な概念であり，重要な手段の 1 つであると言える．プログラミング言語についての正しい理解は，ソフトウェアの設計，開発，利用，解析，改良などに不可欠である．

プログラミング言語は，シンタックス (syntax) とセマンティクス (semantics) を与える事によって規定されると言われる．シンタックスは言語* が（アルファベットと呼ばれる）基本記号のどのような系列から出来ているかを構文規則を与える事によって規定する．言語のシンタックスを定義する方法にも，FORTRAN に見られるように代表的な文形とその使用例で述べる方法，ALGOL 60 で用いられた BNF 記法のような生成文法を用いる方法，McCarthy などによる abstract syntax を用いる方法などがあり，目的・用途，抽象化のレベルにいろいろな特徴がある．このような言語のシンタックスは，周知のように，シンタックス記述のためのメタ言語を用いて書かれる．FORTRAN の場合は自然言語，ALGOL 60 の場合は BNF 記法，abstract syntax の場合には解析的述語と形式論理学の体系が文法記述のためのメタ言語である．言語のセマンティクスも，セマンティクス記述のためのメタ言語を用いて与えられるが，この記述体系と記述のレベルにいろいろなものがある．

プログラミング言語の意味論というのは，言語のセマンティクスを形式的な体系の下で取扱う研究と考えるのが通例であり，本論文は，このような立場からの入門的解説である．意味論に関しては，哲学・論理学・言語学の分野における旧い歴史があり，述語論理における Tarski の意味論のような重要な成果や様相論理における Kripke モデルのような興味深い概念も知られている．プログラミング言語の意味論も，これらの考え方や成果から大きな影響を受けて進められている．しかしプログラミング言語の意味論の場合には，言語による表現が計算機の上で機能するプログラムやデータとして構成的な意味を持つことが要求されるので論点が明確であるのが特徴であると言える．

プログラミング言語の意味論に関する最初の試み** を 1957 年のソ連の Yanov のプログラム図式に関する研究に見る事が出来る．Yanov は，当時ソ連で提案されていた特種なプログラム表現法を用い，基本的には，文の複合，条件文，**go-to** 文に対する意味を公理論的に与えた．Yanov の公理系の完全性の証明においては，(後に Floyd-Naur により提案された) 検証条件や不変検証条件に相当するものも用いられている．またプログラムをマトリックスを用いて記述するという代数的なアイディアも提案されているが，これはプログラムの仕様記述の先駆をなすと見なせる．Yanov の理論は，計算機科学やソフトウェアの分野における理論研究として画期的なものであったが，当時の研究状況は余りに未熟であった事，代入文や算術式といった文の中に立ち入った議論がなかった事，理論が当時としては難解であった事などから，次の大きな飛躍には 10 年余の歳月が必要であった．1950 年代の末から 1960 年代の初めにかけて，米国の McCarthy は，彼が中心になって設計・開発した記号処理言語 LISP の理論的考察と Yanov の理論をもとに，数学的計算の理論 (Mathematical Theory of Computa-

† Semantics of Programming Languages—An Introductory Survey— by Takayasu ITO (School of Engineering, Tohoku University).

†† 東北大学工学部通信工学科

* 以下では，言語と言えばプログラミング言語のこととする

** 米国などでは，最近，A. Turing "Checking a Large Routine" (1950) を最初の試みと主張する説もある．

tion) を提唱した．彼は，1962 年の IFIP Congress の論文で，状態ベクトルセマンティクス，abstract syntax，それらによる言語のセマンティクス，再帰的帰納法による証明法，トランスレータの正当性の概念などプログラミング言語の意味論のその後の展開に重要な役割を果している概念や方法を提案した*．1950 年代末から，FORTRAN や ALGOL に代表される高級言語に対するコンパイラの開発が行われたが，このようなコンパイラ言語とそのコンパイラの作成法を体系化する過程において必然的に言語の意味論の形式化の問題に直面する事となり，数多くの試みがなされた．1964 年に Vienna で開催された IFIP のシンポジウム "Formal Language Description Languages" にその後大きな影響を与えたいくつかの重要な研究発表が行われている．(このシンポジウム論文集は Steel (編) により 1966 年に出版されている．) 1966 年には 2 つの重要な発表がなされている．その 1 つは McCarthy-Painter によるコンパイラの正当性に関する研究論文であり，他の 1 つは Floyd-Naur によるプログラムの正当性証明法に関するものである．McCarthy-Painter は簡単な数式の場合を例にとって，ソース言語およびオブジェクト言語のアブストラクト・シンタックスと状態ベクトル・セマンティクス，コンパイラとその正当性の定義を与え，構造的帰納法の考え方を用いてコンパイラの正当性を証明した．McCarthy により提案されてきたアイディアが具体的にどのように使えるかを示したものと言える**．Naur と Floyd は検証条件によるプログラムの正当性の証明法に関する論文を発表している．その後，この方法は Hoare の検証文を用いて簡明なものとされ，言語のセマンティクスの定義やプログラムの正当性の公理化にも用いられている．1968 年には，述語論理，オートマトン理論，言語理論，論理学における公理的方法などを応用して，プログラムの同値性や正当性などについての決定問題や証明論などを論じる先駆的な研究論文が発表された***．これらによりプログラムに関する理論研究が盛んに行われる契機が開かれ，これ以降，プログラムに関する夥しい数の論文が欧米において発表され，著書や論文集を出版されている．このような研究の中で，プログラミング言語の意味論の研究史上重要な意義を持つものとして次のような研究がある．

① VDL による操作的意味論****．

IBM Vienna 研究所の Lucas 他の研究者は McCarthy のプログラム状態ベクトルおよびアブストラクト・シンタックスと Elgot-Robinson による RASP と呼ばれる抽象機械を基にして，PL/I のセマンティクスを定義する形式化されたインタプリタを与えている．この研究は，操作的意味論の基礎を与えると共に，大規模な言語に対して形式的意味論を展開した最初の試みで，VDL 法とも呼ばれている．なおこの試みはインタプリタで意味解釈をしているのに対し，コンパイラを作り，そのオブジェクトを VDL 的形式言語で表わす試みとして，Bjorner らによる VDM と呼ばれる方法もある*****．

② 検証文による公理的意味論

Hoare はプログラム $P$ に対する前提 $\alpha$ と帰結 $\beta$ を用いて，Floyd-Naur の検証条件を $\{\alpha\}P\{\beta\}$ なる検証文を用いて表現する記法を導入し，この記法を用いて言語のセマンティクスを公理的に記述する方法を与えた．この方法は，その簡明さとプログラミング言語 PASCAL の形式的定義における成功により，言語の公理的意味論の代表的な手法となっている******．

③ プログラム束に基づく指示意味論

Scott はプログラムおよびデータ・タイプに関する束論的な構造を持つ理論的枠組を考え，プログラムの意味をそのような体系の中で許容される単調連続関数によって与える事を提案した．Scott の方法は，再帰的に定義された許容関数の不動点としてプログラムの意味を与えるものとなっており，不動点による意味論 (Fixed-point semantics) とも呼ばれる．この Scott の理論と Strachey による指示意味論の考え方が一体となって，ALGOL や PASCAL などいろいろな言語のセマンティクスを与える試みがなされている．既述のように McCarthy の提案は指示意味論の先駆をなすものと言ってよい．指示意味論 (denotational semantics) は mathematical semantics と呼ばれる事もあるが，本質は denotation by functional elements の事である．また denotation by predicates は取りも直さず記述的意味論の事である．

---

* Scott-Strachey による指示意味論 (denotational semantics) の先駆をなすものであった．

** 最近の指示意味論の応用や代数的理論はこの成果をもとに発展せられたものと言える．

*** これらの結果として，文献 17)～22) に見られるような著書が出版された．

**** operational semantics, interpretive semantics を操作的意味論に対応させている．

***** VDM による意味記述は指示意味論に属するが，数学的とは言い難い点に留意されたい．(VDM については D. Bjorner, Springer Lecture Notes, Vol. 61 参照)

****** 公理的意味記述は他の形式化でも可能なところから，以下では Hoare の方法は記述的意味論の部類に含めて考えている．

プログラミング言語の意味論に関しては，代表的なものとして上記のような3つのものがあるが，言語の種類によっては他の形式化が良い場合もある．例えば，様相論理や内包論理，直観主義論理，一般代数系の理論などを用いた試みもある．特に，並列プログラミング用言語の意味論に関しては新たな工夫も必要とされる．

上述のように，プログラミング言語の意味論の研究には，20年余の歴史があり，夥しい数の論文が抽象的な理論から実際的な側面にわたって発表されている．これらを網羅することは，この小論では不可能であり，また筆者の能わざる所である．本稿は，言語の意味論の入門的解説として，簡単な例によって基礎的な考え方を比較・対照的に示すと共に，最近の研究の断面についても簡単に述べる．専門的な詳細に興味のある読者は文献によって深めて頂きたいと考えている．

## 2. 意味論の目的と基礎概念

プログラミング言語の意味論の詳細に立入るまえに，意味論の目的と意味論の理解のための基礎概念について説明しておこう．

### 2.1 意味論の目的

プログラミング言語の意味論には，いろいろなものがあり，本稿で紹介される方法にも特徴・欠点がある．意味論の目的について知る事は言語についての理解を深める第一歩になると考えられる．

プログラミング言語の設計者は，計算機を用いて処理する問題のクラスを考えて言語の設計を行う．その結果は，言語の構造(シンタックスとセマンティクス)を記述する文書である．その厳密な形式的記述は，言語を理解し，矛盾や誤りを除去し，より簡明で能力のある言語を設計するのにも有用であると言われている．設計された言語が，プログラミング言語と言われるためには，計算機上で機能する言語処理系が実現され，利用者が活用できなければならない．言語処理系を実現するトランスレータ作成者にとっては，言語のシンタックスとセマンティクスが正しく厳密に与えられ，処理系（すなわちトランスレータ）が正しく実現される事が望まれる．セマンティクスの厳密な記述を与える事を目的とする言語の意味論は，トランスレータ作成者にとっても極めて有益なものとなる事が期待される．トランスレータ作成者にとっては，言語の各要素が処理系においてどのように実現され，機能すべきかという事が machine independent に記述され，処理系実現においても矛盾や曖昧さを生じない事が望まれる．また作成された言語処理系が利用者のプログラミングと正しくインタフェースする事も望まれる．すなわち，トランスレータ作成者にとっては，言語処理系実現のための操作的及至内包的意味記述と利用者に対する記述的及至外延的意味記述の両者が（矛盾なく）与えられる事が望ましい．言語の利用者にとっては，言語自身の使い易さと共に，言語の記述が簡明で使い易いものであり，構文的にも意味的にも正しいプログラムが簡明に作成でき，その正当性の検証を容易に行える事が望まれる．言い換れば，利用者にとっては言語の各要素がどのような機能を持ち，それらから正しいプログラムがどのように組み上げられるかが簡明かつ正確に与えられる必要がある．言語の意味論は，上述してきたような観点から，必要性・重要性が認識され，研究されている．これ以外にもプログラムの移植や言語間の翻訳，トランスレータの正当性の検証のように，言語間のインタフェースを正しく行う問題を考える場合にも重要である．またソフトウェアの維持・管理のようなソフトウェア工学上の問題も形式的意味記述の利用によって解決される点が少なくないと考えられる．

以上からも推察されるように，実際的な見地から考えると，プログラミング言語の意味論は言語についての一面的な記述では不十分である．言語のいろいろな側面を記述できる体系あるいは異なる特徴を持った方法を複合的に活用する事が必要である．本稿においてもいくつかの方法が説明されるが，意味記述体系の能力，抽象化のレベル，形式化の点などにおいて諸種のものがあり，目的や用途によって相補的に用いられるべきであろう．

### 2.2 言語の諸要素

プログラミング言語の意味論は，言語を構成する諸要素・諸機能に対する意味記述を与える事から始まる．ALGOL 60 の BNF 記法による文法定義を見ると分かるように，シンタックスはセマンティクス記述への1つの重要なステップである．しかし言語の意味論は，計算構造上で，式やデータ，文，制御構造などの言語構成要素に対する意味記述を与える事から始まる．例えば，次のような言語構成要素が意味記述の対象となる．

① 入力文，入力形式；出力文，出力形式
② データ（文字・記号列，数，信号）
③ データ・タイプ，データ構造

④ 宣言（データ，手続き，プロセス等に対して）
⑤ 基本関数と式（数式，論理式など）
⑥ 基本文（代入文，制御文（go-to 文，条件文，繰返し文など））
⑦ 文の複合，ブロック構造
⑧ 手続き文 (procedure, subroutine, coroutine)，再帰的定義関数
⑨ 並列処理・非決定処理機能
⑩ データ・関数・プロセス・制御などの定義機能

などがある．しかしこれらは FORTRAN, ALGOL, PL/I などの標準的なプログラミング言語に出て来る言語要素や機能を列記したものであり，人工知能用言語，図形処理用言語，シミュレーション言語などにおいては，より高次の言語要素を考慮する事が要求される場合もある．

プログラミング言語の意味論の研究は，主として，⑤，⑥，⑦，⑧に対して進められ，基礎が確立されつつある．③や⑨についても研究が盛んに行われており，興味深い結果も得られつつある．しかし実際のプログラミング言語では，類似の機能であっても，pragmatics の立場からいろいろな表記法を導入しており，一般的な解決を与えるのは困難である．FORTRAN の subroutine, ALGOL 60 の procedure, PL/I の entry, SIMULA の class, UNIX の coroutine, また少し趣は異なるが actor や cluster などに対して統一的な枠組を与えるのは困難な事である．言語の意味論の現状は，上述してきたような言語の諸要素を個々に取扱う数学的及至論理的基礎のいくつかが確立され始めた状況と見なして良かろう．

### 2.3 意味論の基礎的概念

言語のシンタックスを記述するのに BNF 記法や構文認識マシンなどのような構文記述のためのメタ言語やメタ構造を用いるのは周知の通りである．言語のセマンティクスの記述にも意味記述のためのメタ言語・メタ構造が必要とされる．このような意味記述体系に要求される最も基本的な条件は，論理的及至数学的性質がよく知られている事 (well-definedness) と記述能力が豊かな事 (universality) である．また意味記述の結果が理解し易く (understandability)，記述体系が論理的に完全 (completeness) である事が望まれる．

プログラミング言語の意味論は，言語を構成する要素を解釈領域上での要素や演算として解釈し，意味付けを行う事によって展開される．すなわち解釈領域 $\boldsymbol{D}$ とその上での意味付け関数 $\boldsymbol{F}$ とによって解釈 $\boldsymbol{I}$ が与えられ，これを $\boldsymbol{I}=\langle \boldsymbol{D}, \boldsymbol{F}\rangle$ と書く．解釈 $\boldsymbol{I}$ を与えるに当っては，論理的あるいは数学的な性質が良く知られた領域 $\boldsymbol{D}$ やその上での関数 $\boldsymbol{F}$ を使う事が要求される．ここに，いろいろの意味論の形式化が出て来る所以がある．例えば，1つの演算 $f$ を考えたとき，$f$ を領域 $\boldsymbol{D}$ 上の関数とし

$$\mathcal{S}[\![f]\!]: \boldsymbol{D}\to\boldsymbol{D} \qquad (1)$$

のように考える事ができる．ここに，$\mathcal{S}$ は演算 $f$ に対して $\boldsymbol{D}$ 上での対応する関数を与える意味関数である．式（1）は，次のようにも書ける：

$$\xi'=\mathcal{S}[\![f]\!](\xi) \quad (\xi,\xi'\in\boldsymbol{D}) \qquad (2)$$

このとき $\boldsymbol{D}$ の要素 $\{\xi, \xi'\}$ は状態ベクトルと言われる．2つの演算 $f$ と $g$ に対して，$f$ の実行後に $g$ を実行するという事を $f\circ g$ と書いたとき，この意味は，次のような状態ベクトルの変換として考える事ができる．

$$\xi'=\mathcal{S}[\![f]\!](\xi),\ \xi''=\mathcal{S}[\![g]\!](\xi') \qquad (3)$$

$$\xi''=\mathcal{S}[\![f\circ g]\!](\xi)=\mathcal{S}[\![g]\!](\mathcal{S}[\![f]\!](\xi)) \qquad (4)$$

このような記法のかわりに演算の系列を状態ベクトルも含めて，次のように書く事ができる．

$$\xi f \xi' g \xi'' \qquad (3)'$$

$$\xi f\circ g \xi'' \qquad (4)'$$

ここで，（3）や（3）′ のように演算を行うごとに，その状態の履歴がどのようになるかを（明記し）考慮する立場が操作的意味論の考え方である．一方，（4）や（4）′ のように，途中の状態の履歴には関係なく数学的な関数と見なしてしまうのが（関数的）指示意味論の考え方である．

演算 $f$ を関数と考えるかわりに，（式（2）の）$f$, $\xi$, $\xi'$ がある論理関係を満足するという事を主張する事によって意味記述する事も考えられる．例えば

$$\mathcal{R}(f,\xi,\xi'),\ \mathcal{R}(f,\xi)$$
$$\mathcal{L}(f,P_\xi,P_{\xi'}),\ \mathcal{L}(f,P_\xi) \qquad (5)$$

$\mathcal{R}(f,\xi,\xi')$ は，$f$ は入力状態ベクトルを $\xi$ としたとき，$\mathcal{R}$ によって関係付けられる出力状態を生じるものであり，$\mathcal{R}(f,\xi)$ は $\xi$ の上での $f$ の演算（結果）は条件 $\mathcal{R}$ を満たすものである事を意味している．$\mathcal{L}(f, P_\xi, P_{\xi'})$ は状態 $\xi$ および $\xi'$ に関する論理的記述（$P_\xi$ および $P_{\xi'}$）が演算 $f$ に関する論理的記述 $\mathcal{L}$ を満足する事を主張する事により $f$ の意味記述を与えるものである．$\mathcal{L}(f,P_\xi)$ についても同様である．例えば $\pi(\xi)$ を状態ベクトルに関する述語としたとき，$\pi(\xi)$ が真 (**true**) なら $f(\xi)$ は $\varphi$ を満たし，$\pi(\xi)$ が偽 (**false**) なら $f(\xi)$ は $\psi$ を満たすという関係は，論理演算記

号を用いて次のように書ける.

$$[\pi(\xi)\wedge\varphi(f(\xi))]\vee[>\pi(\xi)\wedge\psi(f(\xi))] \quad (6)$$

$$[\pi(\xi)\supset\varphi(f(\xi))]\wedge[>\pi(\xi)\supset\psi(f(\xi))] \quad (6)'$$

また入力条件 $\pi$ が真なら，$f$ の実行後，出力条件 $\varphi$ が成立するという事は次のように書ける.

$$\pi(\xi)\supset\varphi(f(\xi)) \quad (7)$$

Hoare はこの式の代りに，次の表記を提案した:

$$\{\pi\}f\{\varphi\} \quad (7)'$$

これは Hoare の検証文と呼ばれ，$\pi$ および $\varphi$ はそれぞれ入力ベクトルおよび出力ベクトルに関する記述であるから，(4)′ に対応した表現とも見なせる．このような論理的記述において，記述に用いられる論理体系 (underlying logic) として，述語論理，第一階論理体系，高階論理，様相論理，直観主義論理，代数的論理などの論理体系を用いる事が考えられ，それによって，記述能力や理論的扱いもさまざまである．このような論理的記述を用いる場合，$f\circ g$ のような演算の合成も論理体系の中で，

$$\frac{\{\pi\}f\{\delta\},\{\delta\}g\{\varphi\}}{\{\pi\}f\circ g\{\varphi\}} \quad (8)$$

のような推論図式として与えられる．論理的記述によって意味記述を与える方法を記述的意味論と言うが，通常，公理的な枠組として与える事が多いところから公理的意味論とも呼ぶ.

操作的意味論や指示意味論も，それぞれ論理的な枠組の中で組み立てられていると見なせるから広い意味での記述的意味論に属する．したがって，ここでの分類は理論の形式化の見かけ上の違いに準拠したものと考えてよいであろう．実際，高階論理，様相論理，内包論理の応用においては，関数的あるいは操作的な側面を記述できるように考えた試みが行われつつあり，截然とした区別は行い難いと言える.

操作的意味論，指示意味論，記述的意味論の考え方について説明してきたが，これらはコンパイラ作成時における意味記述と較べると抽象的なものである．コンパイラなどのトランスレータを作成する立場からは，解釈領域として計算機（あるいはそのモデル）を基にした具体的なものが要求される．すなわち，演算や式が計算機上でどのように表現され，どのような命令語系列が対応付けられるかを考える必要がある．構文解析，構文変換を行うごとに意味付けルーチンや意味的属性を割り当て，意味記述を行う方法も考えられ，例えば，生成システム (production system) や構文木に意味的属性を付与する方法などもある*．これらはコンパイラ指向の意味論であり，構文解析と関連付けられたものを構文指向形意味論ともいう.

関数的指示意味論は $\lambda$-calculus, combinatory logic, 関数に関する代数的理論などを基にしている．記述的意味論（あるいは公理的意味論）は各種の公理的な論理体系を用いている．言語の意味論は論理学や数学における成果を拠所として意味記述を行うものであるという見方もできる．しかしプログラミング言語の意味論を展開するには，言語のモデル化をどのように考えるかという本質的な課題が存在する．また，従来の論理学や数学の成果の応用だけでは不十分であり，新しい理論が必要とされる場面も多い．Scott-Strachey により理論的基礎が与えられた指示意味論は，再帰的定義関数の意味を自然に表現し得る新しい理論である．Pratt らによって展開されている Dynamic Logic は様相論理のプログラム理論流の形式化の例であると言える．またプログラミング言語は，計算機という構成的な存在の上で機能すべきものであるという立場を強く意識した理論，計算機上で有限的に構成可能な対象のみから出発して有限的な手法で検証可能な対象のみを意味あるものと認める考えなど，新しい構成的理論を追求する立場も考えられる.

次節では，数式と代入文というプログラミング言語の最も基本的な構成要素の意味記述とその問題点などについて説明する．そのまえに，プログラミング言語の意味論と関係の深いコンパイラの正当性の概念について簡単に説明しておこう.

### 2.4 コンパイラの正当性の概念

コンパイラは，ソース言語で書かれたプログラムを解析し，対応するオブジェクト言語のプログラムに正しく翻訳するのが目的である．コンパイラが正しいというのは，ソース・プログラムの計算と，それに対応するオブジェクト・プログラムの計算とが同じ意味を持つ事であると考えられる．**source** $[P](\xi)$ は，ソース・プログラム状態ベクトル $\xi$ の下でのソース・プログラム $P$ の意味，**object** $[p](\eta)$ はオブジェクト・プログラム状態ベクトル $\eta$ の下でのオブジェクト・プログラム $p$ の意味，**compile** $[P]$ はソース・プログラム $P$ をコンパイラ **compile** によって翻訳して得られるオブジェクト・プログラムであるとする．このとき，与えられた言語 $\mathcal{L}$ に対するコンパイラ **compile** の正当性は，この言語で書かれたあらゆるプログラム

* 例えば，Feldman の FSL や Knuth の semantic attribute を用いる方法.

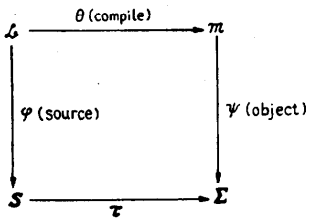


図-1　コンパイラ正当性の図式的表明

$P$ に対して次式が成り立つ事である*.

$$\tau[\mathbf{source}[P](\xi)]=\mathbf{object}[\mathbf{compile}[P]](\tau(\xi)) \qquad (9)$$

ここに $\tau[\xi]$ はソース・プログラム状態ベクトルをオブジェクト・プログラム状態ベクトルに対応させる写像であるとする.

コンパイラの正当性を図-1のように表明することができる. すなわち $\theta$ を演算してから $\psi$ を演算したもの $(\theta\circ\psi)$ が, $\varphi$ を演算してから $\tau$ を演算したもの $(\varphi\circ\tau)$ に等しくなることがコンパイラ $\theta$ (すなわち **compile**) の正当性の条件となる. $\mathcal{L}$, $M$, $S$, $\Sigma$ を代数構造とみなしたとき, $\theta$, $\varphi$ が準同型 (homomorphism) で, $\psi$, $\tau$ も準同型 (homomorphism) であり, $\theta\circ\psi=\varphi\circ\tau$ が成立する事がコンパイラ正当性の条件となる.

コンパイラの正当性の証明を行うには, ソース言語とオブジェクト言語の意味記述, コンパイラの記述, ソース・プログラム状態ベクトルとオブジェクト・プログラム状態ベクトル間の関連付けが前提とされる. 意味論の代数的な形式化がこのような場面で有用となる事が知られている.

## 3. 数式と代入文の意味論

プログラミング言語の意味論の例として, 数式 (arithmetic expression) と代入文 (assignment statement) の意味記述を考えてみよう.

数式のセマンティクスは旧くから研究されており, コンパイラ作成の立場からの技法も確立されている. 数式のセマンティクスは解釈領域上での値を求める事により行われる. これに対し, 代入文のセマンティクスは状態ベクトル (あるいは, その記述) に関して与えられる.

### 3.1 数式の意味論

数式 $e$ のセマンティクスを状態ベクトル $\xi$ の下での値 value$[\![e]\!](\xi)$ として定義する方法を明確な形で提案したのは McCarthy である. 簡単な例を考えてみよう. 数式 $e$ が定数, 変数, 加算, 乗算のみからなるとしたとき, value$[\![e]\!](\xi)$ を次のように定義する.

value $[\![e]\!](\xi)$=**if** const? $[\![e]\!]$ **then** val $[\![e]\!]$
**else if** var? $[\![e]\!]$ **then** $C[\![e]\!](\xi)$
**else if** sum? $[\![e]\!]$ **then** value [addend $[\![e]\!]](\xi)$
+value [augend $[\![e]\!]](\xi)$
**else if** product? $[\![e]\!]$ **then** value [mplier$[\![e]\!]](\xi)$
×value [mpcand $[\![e]\!]](\xi)$　　(9)

ここに const?, var?, sum?, product? は数式 $e$ のシンタックスを問う述語で abstract syntax と言われ, val $[\![e]\!]$ は解釈領域上での $e$ の定数としての値, 内容関数 $C[\![e]\!](\xi)$ は状態ベクトル $\xi$ の下での変数 $e$ に割り当てられた値, +および×は解釈領域上での加算および乗算とする. 数式の状態ベクトル $\xi$ の下でのセマンティクス **value** $[\![e]\!](\xi)$ は, **if-then-else** を用いた再帰的な関数として定義され, そのために, いくつかの補助的関数と解釈領域上での算術体系が用いられている. 一方, 数式の意味を自然数論の公理系 (例えばペアノの公理) から誘導する事が考えられる. これは自然数論の公理系を記述体系として用いるものであり, Hoare による PASCAL の意味記述では, このような立場が取られていると考えて良い. 数式の意味記述に関しては, このような2つの試みがなされているが, 計算機上で具体的に演算されるものとして数式の意味を考えると, これらの扱いでは不十分と言わざるを得ない. 計算機上では, 数は浮動小数点形式あるいは固定小数点形式を持つ有限精度の数として表現され, 演算によっては overflow や underflow を起す. これらの事を正しく記述できなければ, computational arithmetic に対する数式の意味を正しく与えたものとは言えない. このような側面は既存の数学や論理学の中では体系化されておらず, 新しい論理体系を構築する必要がある.

プログラミング言語のレベルで表現される外部表現と計算機内部での情報の内部表現にあるこのような gap を埋める問題は, プログラミング言語の意味論を考える上での重要な課題の1つと言える.

### 3.2 代入文の意味論

変数 $x$ についての代入文 $x\leftarrow\alpha$ に対して, 状態ベクトル $\xi$ の下でのセマンティクスを $s[\![x\leftarrow\alpha]\!](\xi)$ と記す事にする. McCarthy は代入文の意味記述が次の公理によって与えられると主張した.

$s[\![x\leftarrow c[\![x]\!](\xi)]\!](\xi)=\xi$
$s[\![x\leftarrow\alpha;\ y\leftarrow\beta]\!](\xi)$=**if** $x=y$ **then** $s[\![x\leftarrow\beta]\!](\xi)$

* この定義は McCarthy (1963) による.

**else** $s[\![y\leftarrow\beta;\ x\leftarrow\alpha]\!](\xi)$ (10)

$c[\![x]\!](s[\![y\leftarrow\alpha]\!](\xi))=$**if** $x=y$ **then** $\alpha$
**else** $c[\![x]\!](\xi)$

ここに $s[\![\cdot]\!](\xi)$ は状態ベクトルを値として取る意味関数．また $s[\![x\leftarrow\alpha;\ y\leftarrow\beta]\!](\xi)=s[\![y\leftarrow\beta]\!](s[\![x\leftarrow\alpha]\!](\xi))$ であるとする．この McCarthy の公理は代入文の意味記述として十分とは言えない．$\alpha$ と $\beta$ が定数のときには成立するが，例えば，次の場合には成立しなくなる*：

$$s[\![x\leftarrow f(x);\ x\leftarrow g(x)]\!](\xi)\neq s[\![x\leftarrow g(x)]\!](\xi)$$
$$s[\![x\leftarrow f(x,y);\ y\leftarrow g(x,y)]\!](\xi)\neq s[\![y\leftarrow g(x,y);\ x\leftarrow f(x,y)]\!](\xi) \quad (11)$$

代入文の意味記述を正しく与える事の難しさは，1964年の IFIP Vienna Symposium において Strachey によって指摘され，その事に端を発するアイディアが，Scott-Strachey による指示意味論の1つの背景となっている．以下では代入文に対する意味記述のいくつかを示す．

① $s[\![x\leftarrow e]\!](\xi)=[\text{value }[\![e]\!](\xi)/x]\xi$ (12)

右辺の $[y/x]\xi$ なる記法は $\xi$ にあらわれる変数 $x$ を $y$ で置き換える事を示している．

② $\{[y/x]P\}\,x\leftarrow y\,\{P\}$ (13)

$P$ 内に自由変数としてあらわれるすべての $x$ を $y$ に置き換えて得られる論理式 $[y/x]P$ が成立すれば，代入文 $x\leftarrow y$ の実行の後，論理式 $P$ が成り立つ事を意味している．$[y/x]P$ を $P^x_y$ と記し，次のようにも書く．

$$\{P^x_y\}\,x\leftarrow y\,\{P\} \quad (13)'$$

次の公理はこれらの special case であるが分かり易い．

$$\{p(f(x))\}\,x\leftarrow f(x)\,\{p(x)\} \quad (13)''$$

③ $\{P\}\,x\leftarrow y\,\{\exists t([t/x]P\wedge x=[t/x]y\}$ (14)

この公理は (13) に対応して考えられるもので，この意味記述を forward rule，(13) を backward rule と言う事がある．

④ $\text{value }[\![x]\!](s[\![x\leftarrow e]\!](\xi))=\text{value }[\![e]\!](\xi)$
$x\neq y\supset\text{value }[\![y]\!](\xi)=\text{value }[\![y]\!](s[\![x\leftarrow e]\!](\xi))$ (15)

これは代入文の実行後に成り立つ関係を状態ベクトルを構成する要素について与えたものである．

⑤ $s.[\![x\leftarrow e]\!](\xi)=[\lambda_s.\,\xi(x)](e)$ (16)

これは代入文の意味を $\lambda$-calculus の式で与えたもので，意味記述を $\lambda$-calculus で行う考え方である．

しかし，式 (16) の記述は (LISP や $\lambda$-calculus の利用者が直ちに気付くように) 不十分であり，Scott-Strachey 流の考察が必要となる．

代入文の意味記述に対してもいろいろなものが考えられるが，上述の議論は，代入文 (assignment statement) を数学や論理学における変数の代入 (substitution) によって理解しようとするものである．このような記述は代入文の左辺が単純変数の場合には問題はないが，複雑な代入文を許す言語の場合には再考の要がある．

$$x[x[i]]\leftarrow t$$
$$(\textbf{if } p \textbf{ then } x \text{ else } y)\leftarrow t \quad (17)$$
$$f(x,y)\leftarrow t$$

などのように，代入文の左辺が単純変数でない，いわゆるテキスト形式を取る場合には，上述の①～⑤では不十分である．

代入文の意味記述に対しては，代入記号 ← の左辺と右辺をそれぞれ評価し，変数間の参照に注意を払う事によって，より一般的に取扱う事ができる．例えば，

ⓐ 代入規則 $[y/x]$ を対象とする言語に見合うように定義する方法

ⓑ 内包論理 (Montague's intensional logic) を用いる方法

ⓒ Strachey の $\mathcal{R}$-value, $\mathcal{L}$-value を用いる方法

ⓓ VDL (あるいは VDM) のような記述を用いる方法

などがある．ⓒやⓓの方法は記述能力は大きいが繁雑な面も多い．紙数の関係で詳論はできないが，Scott-Strachey と Montague logic を用いた代入文の意味記述の概要を示そう．

・Scott-Strachey 流の代入文の意味記述例

$$\mathcal{O}[\![e_1\leftarrow e_2]\!]\rho\theta=\mathcal{L}[\![e_1]\!]\rho(\lambda\alpha.\,\mathcal{R}[\![e_2]\!]\rho(\lambda\beta.\,\text{assign }\alpha\beta\theta)) \quad (18)$$

ここに $\rho$ は enviroment, $\theta$ は continuation, $\alpha$ は $L$-value, $\beta$ は $R$-value であり，$\mathcal{L}$ は，$L$-mode evaluation, $\mathcal{R}$ は $R$-mode evaluation である．ここに

assign $\alpha\beta\theta=$**if** $\alpha\in L\wedge\beta\in V$ **then** $\theta\circ$update $(\alpha,\beta)$ **else** error

update $(\alpha,\beta)\sigma$ は，$\alpha$ の内容が $\beta$ であるような新しい状態 $\sigma'$ を状態 $\sigma$ から生成する．直感的には，$\mathcal{L}$ によって $e_1$ の generalized location $L$ を求め，$\mathcal{R}$ によって $e_2$ の値 $E$ を求めて，$L$ に $E$ を代入 (substitution) して得られる新しい状態が代入文

* $f$, $g$ が特殊な場合は別として．

"$e_1 \leftarrow e_2$" の意味記述であると考えるものである*.

・Montague logic 流の代入文の意味記述例

Janssen-Boas** は Montague の intensional logic を用いて次のような意味記述を与えている.

A1 (simple assignment)

$$\pi[x \leftarrow e] = \lambda P.\, \wedge \exists z([z/\vee x] \vee P \wedge \vee x = [z/\vee x]e) \qquad (19)$$

A2 (conditional assignment)

$$\pi[\textbf{if } \beta \textbf{ then } x \textbf{ else } y \textbf{ fi} \leftarrow e] = \Pi[\textbf{if } \beta \textbf{ then } x \leftarrow e \textbf{ else } y \leftarrow e \textbf{ fi}] \qquad (19)'$$

ここに

$$\Pi[\textbf{if } \beta \textbf{ then } S_1 \textbf{ else fi}] = \lambda P.\, \wedge[S_1 \wedge (\beta \wedge P) \vee S_2 \wedge (\neg\beta \wedge P)]$$

A3 (array assignment)

$$\pi[a[v] \leftarrow e] = \pi[a \leftarrow \lambda n.\ \textbf{if } n = v \textbf{ then } e \textbf{ else } a[n] \textbf{ fi}] \qquad (19)''$$

ここで, $\vee\alpha$: extension of $\alpha$ ($\alpha \in T_{s \to a}$ なら $\vee\alpha \in T_a$)

$\wedge\alpha$: intension of $\alpha$ ($\alpha \in T_a$ なら $\wedge\alpha \in T_{s \to a}$)

並列プログラムの場合には更に複雑となる. 例えば $(x \leftarrow y+1) \| (y \leftarrow x+1)$ の場合には, $(x, y) = (1, 1)$ に対して, $S_1 \| S_2$ の意味を $S_1 \circ S_2 \sqcup S_2 \circ S_1$ のように非決定的に解釈したとき, その値は次のようになる.

・$x \leftarrow y+1$; $y \leftarrow x+1$ に対し $(x, y) = (2, 3)$

・$y \leftarrow x+1$; $x \leftarrow y+1$ に対し $(x, y) = (3, 2)$

しかし $x$ と $y$ に同時に 1 を代入して計算すれば***, $(x, y) = (2, 2)$ を得る. このような問題はオペレーティング・システムにおける並行プロセスの意味論において生じる基本的な問題の 1 つである.

また数式の意味論のところで述べた computational arithmetic の問題を考えると, $x$ が単純変数でも

$$\left\{a \cdot \frac{b}{a} = b\right\} x \leftarrow \frac{b}{a} \{a \cdot x = b\} \qquad (20)$$

が一般には成立しない事が起る. すなわち $\frac{b}{a}$ を計算すると overflow や underflow のために, 数学的な意味での $\frac{b}{a}$ と異なったものが計算され, それを $\left[\frac{b}{a}\right]$ と書くと, $a \cdot \left[\frac{b}{a}\right] \neq b$ となる.

プログラミング言語の構成要素の中で最も基本的なものとして, 数式と代入文を例に取って, 意味論の方法と問題点のいくつかを述べてきた. 条件文, **go-to** 文, 繰返し文などの基本的なものについても論じたかったが紙数の関係で割愛した. 次章で簡単な例の紹介があるが, 詳細は文献によって補われたい. なお, これらの場合についても, 代入文の場合と同様に, 問題点がすべて解決済とは言えない点に留意されたい.

## 4. 操作的意味論・指示意味論・記述的意味論の例

代入文, 文の複合, 条件文, 繰返し文といった基本的な文形を対象として意味記述を比較・対照的に与えることにしよう. なお文形は次のように書くことにする:

① 代入文 $x \leftarrow e$

② 文の複合 $S_1$; $S_2$

③ 条件文 IF $p$ THEN $S_1$ ELSE $S_2$

④ 繰返し文 WHILE $p$ DO $S$

代入文についても (他の文形についても) 前節で考察したような微妙な困難な問題はないと仮定し, これらに対する操作的意味論, 指示意味論, 記述的意味論を与えると**図-2** のようになる.

操作的意味論と指示意味論の違いは, 操作的意味論が状態系列を忠実に記録・再現し得る点にある. 複合文の指示的セマンティクス $\mathcal{S}_{ST}[\![S_1; S_2]\!](\xi)$ に対して, $\mathcal{S}_{ST}[\![S_1]\!](\xi) * \mathcal{S}_{ST}[\![S_2]\!](\mathcal{S}_{ST}[\![S_1]\!](\xi))$ を考えると操作的意味論と同じ議論が展開できる. 指示意味論では, $\mathcal{S}_{ST}[\![S_1; S_2]\!](\xi)$ は, ($\mathcal{S}_{ST}[\![S_1]\!](\xi)$ は表面に出す事なく) $\mathcal{S}_{ST}[\![S_2]\!](\mathcal{S}_{ST}[\![S_1]\!](\xi))$ のみによって定まるという訳であるから指示意味論は操作的意味論から誘導できると考えてよい. 指示意味論の考え方は, 関数的に意味記述を行う所から, 豊かな数学的体系を構成できる利点がある. 指示意味論に対する数学的基礎を与えた代表的なものとして Scott によるプログラム束の理論がある****. 指示意味論に用いた $\mu_f[\mathcal{F}(f)]$ はそのような理論における汎関数 $\mathcal{F}$ の最小不動点を意味している. すなわち, $\mu_f[\mathcal{F}(f)]$ は $f = \mathcal{F}(f)$ の最小解であり, $\mu_f[\mathcal{F}(f)] = \textbf{lub}\{f^n(\perp) | n = 0, 1, 2, \cdots\}$ で与えられる. (ここに **lub** は, least upper bound, $\perp$ は

---

* L-value, R-value を用いる方法は Strachey により提案され, Scott-Strachey の指示意味論により厳密化された. 詳細については Stoy[35] あるいは Milne-Strachey[24] を参照されたい.

** Springer Lecture Notes in Computer Science Vol. 52, pp. 282-300 を参照.

*** $S_1 \| S_2 = S_1 \circ S_2 \sqcup S_2 \circ S_1 \sqcup S_1 \sharp S_2$ (♯: 同時実行) のように書ける.

**** Scott による Lattice theory of data types が, Strachey の指示意味論の考え方と一体となって現在の形式化が完成されている.

<操作的意味論の例>

A1:

$\mathcal{S}_{OP}[x \leftarrow e](\xi) = [\text{value}[e](\xi)/x]\xi$

A2:

$\mathcal{S}_{OP}[S_1;\ S_2](\xi) = \mathcal{S}_{OP}[S_1](\xi) \circledast \mathcal{S}_{OP}[S_2](\xi')$

(ここに $\xi' = \mathcal{S}_{OP}[S_1](\xi)$, $\circledast$ は状態系列をならべる演算)

A3:

$\mathcal{S}_{OP}[\text{IF } p \text{ THEN } S_1 \text{ ELSE } S_2](\xi)$
$= \textbf{if value } [p](\xi) \textbf{ then } \mathcal{S}_{OP}[S_1](\xi)$
$\textbf{else } \mathcal{S}_{OP}[S_2](\xi)$

A4:

$\mathcal{S}_{OP}[\text{WHILE } p \text{ DO } S](\xi)$
$= \textbf{if value } [p](\xi)$
$\textbf{then } \mathcal{S}_{OP}[S](\xi) \circledast \mathcal{S}_{OP}[\text{WHILE } p \text{ DO } S](\xi')$
$\textbf{else } \xi$
(ここに $\xi' = \mathcal{S}_{OP}[S](\xi)$)

<指示意味論の例>

B1:

$\mathcal{S}_{ST}[x \leftarrow e](\xi) = [\text{value}[e]/x]\xi$

B2:

$\mathcal{S}_{ST}[S_1;\ S_2](\xi) = \mathcal{S}_{ST}[S_2](\mathcal{S}_{ST}[S_1](\xi))$

B3:

$\mathcal{S}_{ST}[\text{IF } p \text{ THEN } S_1 \text{ ELSE } S_2](\xi)$
$= \textbf{if value } [p](\xi) \textbf{ then } \mathcal{S}_{ST}[S_1](\xi)$
$\textbf{else } \mathcal{S}_{ST}[S_1](\xi)$

B4:

$\mathcal{S}_{ST}[\text{WHILE } p \text{ DO } S](\xi)$
$= \textbf{if value } [p](\xi)$
$\textbf{then } \mathcal{S}_{ST}[\text{WHILE } p \text{ DO } S](\mathcal{S}_{ST}[S](\xi))$
$\textbf{else } \xi$

または

$\mathcal{S}_{ST}[\text{WHILE } p \text{ DO } S](\xi)$
$= \mu f[\lambda\eta.\ (\textbf{if value } [p](\eta) \textbf{ then } f(\mathcal{S}_{ST}[S](\eta))$
$\textbf{else } \eta)][\xi]$

ここに $\mu f[\mathcal{F}(f)]$ は $\mathcal{F}$ の最小不動点となるような $f$

<記述的意味論の例>

C1:

$\{P^x_e\}\, x \leftarrow e\, \{P\}$

C2:

$$\frac{\{P\}S_1\{Q\},\ \{Q\}S_2\{R\}}{\{P\}S_1;\ S_2\{R\}}$$

(これは上の2つの式から下の式を推論する規則)

C3:

$$\frac{\{P \wedge p\}S_1\{Q\},\ \{P \wedge \neg p\}S_2\{Q\}}{\{P\}\ \text{IF } p \text{ THEN } S_1 \text{ ELSE } S_2\{Q\}}$$

C4:

$$\frac{\{P \wedge p\}S\{P\}}{\{P\}\ \text{WHILE } p \text{ DO } S\{P \wedge \neg p\}}$$

**図-2** 代入文，複合文，条件文，繰返し文のセマンティクスの例

totally undefined element とする.)ある言語の要素をプログラ束の構造を持った領域上で表現すると，その要素を $f = \mathcal{F}(f)$ なる方程式の最小不動点 $\mu_f[\mathcal{F}(f)]$ として表現できる．この意味で，Scott の指示意味論を不動点意味論と呼ぶ事もある．Scott の貢献は，そのような計算モデルを数理論理学的な観点から elegant に構成した点にある*.

Scott の理論では，プログラム束の要素間の半順序関係を表わすのに "$x \sqsubseteq y$" という記法を用いる．これは，"$x \sqsubseteq y \Leftrightarrow x = \perp$ または $x = y$" の意味である．この事から，Hoare の検証文 $\{\alpha\}S\{\beta\}$ は次のように表現できる**.

〔1〕 意 味 領 域

Int, Bool, Udef, Val=Int+Bool+Udef, Arg=Val+[Int→Val]
Func=Arg*→Val, Arg=Val+[Int→Val], Lab=C,
$C = S \to \text{Val}^*$, $S = [\text{Id} \to [\text{Val} + [\text{Int} \to \text{Val}]]] \times \text{Val}^* \times \text{Val}^*$
$\text{Proc} = I_d^* \to [\text{Arg}^* \to [C \to [S \to \text{Val}^*]]]$
$\text{Env} = I_d \to [\text{Func} + \text{Proc} + \text{Lab}]$
(注) $[D_1 \to D_2]$ は $D_1$ から $D_2$ への関数空間を示す

〔2〕 意 味 関 数

$\mathcal{S}_e$: 式の意味, $\mathcal{S}_a$: argument の意味, $\mathcal{S}_s$: statement の意味,
$\mathcal{S}_p$: プログラムの意味

〔3〕 意味記述 ($e \in \text{Env}$, $c \in C$, $s \in S$ に対して)

$\mathcal{S}e[n](e;\ s) = \textbf{Int } [n]$, $\mathcal{S}e[t](e;\ s) = \textbf{Bool}[t]$,
$\mathcal{S}e[\text{id}](e;\ s) = \textbf{if } sv[\text{id}] \in \textbf{Val then if } sv[\text{id}] \in \text{U}_{def} \textbf{ then T}$
$\textbf{else } sv[\text{id}]$
$\textbf{else T}$ (T: 束の Top element, $sv$: state の variable 要素)
$\mathcal{S}_a[\text{id}(\text{varg}^*)](e;\ s) = \textbf{if } e\ [\text{id}] \in \text{Func then let } a =$
$\mathcal{S}_a[\text{varg}^*](e;\ s) \textbf{ in value } a)\colon e[\text{id}](a) \textbf{ else T}$
$\mathcal{S}e[E_1\ op\ E_2](e;\ s) = \text{OP}[op](\mathcal{S}e[E_1](e;\ s),\ \mathcal{S}e[E_2](e;s))$
(OP: Val×Val→Val)
$\mathcal{S}a[\text{id}](e;\ s) = sv[\text{id}]$, $\mathcal{S}a[\text{exp}](e;\ s) = \mathcal{S}e[\text{exp}](e;\ s)$
$\mathcal{S}_s[\text{id} \leftarrow \text{exp}](e;\ c;\ s) = \textbf{let } v = \mathcal{S}e[\text{exp}](e;\ s) \textbf{in (value } v)\colon$
$\textbf{if } sv[\text{id}] \in \textbf{Val then } c([v/\text{id}]s) \textbf{ else T}$
$\mathcal{S}_s[\text{id}[E_1] \leftarrow E_2](e;\ c;\ s) = \textbf{let } v_1 = \mathcal{S}_e[E_1](e;\ s) \textbf{ to Int};$
$v_2 = \mathcal{S}_e[E_2](e;\ s) \textbf{ in (value } v_1, v_2)\colon \textbf{ if } sv[\text{id}] \in [\text{Int} \to \text{Val}]$
$\textbf{then} c([v_2/\text{id}(v_1)]s) \textbf{ else T}$
$\mathcal{S}_s[\textbf{null}](e;\ c;\ s) = c(s)$
$\mathcal{S}_s[\textbf{read id}](e;\ c;\ s) = \textbf{if } si \in \text{Val}^* \textbf{ then T else if } sv[\text{id}] \in \text{Val}$
$\textbf{then } c(([\textbf{hd}(si)/\text{id}], \textbf{tl } (si), so) \textbf{ else T}$
$\mathcal{S}_s[\text{write exp}](e;\ c;\ s) = \textbf{let } v = \mathcal{S}_s[\text{exp}](e;\ s) \textbf{ in (value } v)\colon$
$\textbf{if } so \in \text{Val}^* \textbf{ then } c((sv, si, \textbf{append } (so, v)) \textbf{ else T}$
$\mathcal{S}_s[\textbf{go to id}](e;\ c;\ s) = \textbf{if } e[\text{id}] \in \textbf{Lab then } e[\text{id}](s) \textbf{ else T}$
$\mathcal{S}_s[\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2](e;\ c;\ s) = c'(s)$
$c' = \textbf{if } \mathcal{S}_e[E](e;\ s) \textbf{ then } \mathcal{S}_s[S_1](e;\ c) \textbf{ else } \mathcal{S}_s[S_2](e;\ c)$
$\mathcal{S}_s[\textbf{while } E \textbf{ do } S](e;\ c;\ s) = \textbf{rec } c'(s)$
$C' = \textbf{if } \mathcal{S}_e[E](e;\ s) \textbf{ then } \mathcal{S}_s[S_1](e;\ \mathcal{S}_s[\textbf{while } E \textbf{ do } S](e;\ c))$
$\mathcal{S}_s[S_1;\ S_2](e;\ c;\ s) = \mathcal{S}_s[S_1](e;\ \mathcal{S}_s[S_2](e;\ c);\ s)$
$\mathcal{S}_s[\textbf{begin } S \textbf{ end}] = \mathcal{S}_s[S]$
$\mathcal{S}_s[\textbf{procedure id}(\text{id}_1\text{*}\colon \text{id}_2\text{*})\ v \text{ block};\ p \text{ block}](e;\ c;\ s)$
$= \textbf{if notin } (\text{id}_1\text{*}, \text{id}_2\text{*}) \textbf{ then } (\textbf{if distinct } (\text{id}^{1*})$
$\textbf{then } (\textbf{if distinct } (\text{id}_2\text{*}) \textbf{ then}$
$(\textbf{if assignsts } (v \text{ block}, \text{id}_2\text{*}) \textbf{ then T}$
$\textbf{else } \mathcal{S}_s[p \text{ block}]([p/\text{id}]e;\ c;\ [T/\text{id}]s))$
$\textbf{else T}) \textbf{ else T}) \textbf{ else T}$
$\textbf{rec fun } p(\text{Id}^*\ x;\ \text{Arg}^*\ y;\ \text{C } op;\ \text{S } sp)\colon$
$\textbf{begin } \mathcal{S}_s[v\text{block}]([p/\text{id}]e';\ cp';\ sp') \textbf{ end}$
$e' = \textbf{func } (\textbf{Id } i)\colon \textbf{if } e[i] \in \textbf{Lab then T else } e[i]$
$cp' = \textbf{func}(\ s_1)\colon cp([s_1 v[\text{id}_1\text{*}]/x]\text{spv},\ s_1 i,\ s_1 o))$
$sp' = [T;\ \text{spi};\ \text{spo}]$
⋮
(注) この例は，J. E. Donahue からの抜粋であり，完全なものは文献 23) を見よ．この定義では "continuation $c$" を用いている．

**図-3** $\mu$-PASCAL に対する指示的セマンティクスの例

---

* 不動点意味論の考え方は，1960年代に McCarthy, Burstall, 筆者らによって用られている．

** $\alpha, \beta$ は total predicate と考えている事に注意せよ．$t$ に $e$ (identity operator), $f$ に $\perp$ を対応させると Hoare の検証文に次式が対応する．$S \sqsubseteq \alpha \circ S \circ \beta \sqcup \overline{\alpha} \circ S$.

〔1〕 データ・タイプの記述

① 自然数タイプに対する公理

$0 \in I$ if $n \in I$ then $n^+, n^- \in I$

$n=(n^-)^+,\ n=(n^+)^-,\ n \leq n,\ n<n^+,\ n \leq m \supset n \leq m^+,$

$n<m \supset n<m^+$

$n+0=n,\ n+m=n^+ +m^- =n^- +m^+,\ n-0=n,$

$n-m=n^+ -m^+ =n^- -m^-$

$n \times 0=0,\ n \times m=n \times m^+ -n=n \times m^- +n$

$n>0 \supset m-n<(m \text{ div } n) \times n \leq m$

$n<0 \supset m \leq (m \text{ div } n) \times m<m-n$

$m \text{ mod } n=m-((m \text{ div } n) \times n)$

⋮

② その他論理データ・タイプ，キャラクタ・タイプ，実数タイプ，配列タイプ，レコードタイプ，ファイルタイプ，ポインタタイプなどに対する公理（省略）

〔2〕 文の意味記述

代入文，複合文，条件文，繰返し文の記述は 図-1の C1，C2，C3，C4 を参照．

$$\frac{\{P\}S\{Q\}}{\{P\}\textbf{begin } S \textbf{ end } \{Q\}}$$

$$\frac{\{P\}S\{\textbf{assertion at } (L)\}}{\{P\}S;\ \textbf{go to } L\{Q\}}$$

$$\frac{\{P\}S\{R\},\ \{P\}\ \textbf{go to } L\{\textbf{false}\} \vdash \{Q\}SL\{R\}}{\{Q\}L:\ S;\ SL\{R\}}$$

$$\frac{p(x;\ y)\textbf{procedure } B(p),\ \{P\}r(x,y)\{Q\} \vdash \{P\}B(r)\{Q\}}{\{P\}p(x,y)\{Q\}}$$

（注） PASCAL 全体の公理的記述については Hoare-Wirth の論文を参照せよ．

**図-4** PASCAL の公理的記述の形態

$$v[\![\beta]\!](\mathcal{S}[\![S]\!](\xi)) \sqsubseteq \textbf{if } v[\![\alpha]\!](\xi) \textbf{ then } t \textbf{ else } v[\![\beta]\!](\mathcal{S}[\![S]\!](\xi)) \qquad (21)$$

ここに $v[\![\cdot]\!](\xi) \in \{t, f, \perp\}$，$\mathcal{S}[\![\cdot]\!](\xi) \in \{\text{states}\}$．この対応関係を用いると，Hoare の検証文による記述的意味論の議論は，指示意味論でモデル化できる訳である．

これらから分かるように操作的意味論が最も能力があるが数学的性質を enjoy しないために一般的議論を展開する困難さ・繁雑さがある．指示意味論は数学的な構造は豊かであるがプログラムの正当性などを言語の利用者に伝えるには難点がある*．また **do** $S$ **forever** のような non-terminating program の意味論を展開するには数学的構造の選択に注意が必要とされる**．Hoare の検証文は分かり易いが，部分的正当性しか論じられない事や前節で代入文に対して述べたような問題点がある．記述的意味論においては，Hoare の検証文より更に豊かな記述能力を持つ論理体系を用いる試みもあるが，(Montague logic による代入文の意味記述に見たように）繁雑さは逃れ難い事となる．

最後にプログラミング言語の他の要素なども含めた言語の意味記述の例の断片を**図-3**，**4** に示し，本稿を終る．(並列プログラム，非決定性プログラムについては文献で補って頂きたい．)

* これは前頁の脚注に示したような議論を展開する事により，ある程度，解消できる（筆者による未発表の試論がある）

** この種の議論には Fixed-point semantics より，Unfolding semantics の方が better であると思われる．Unfolding semantics については筆者らによる試論がある（文献へのコメントを参照）

## 5. むすび

プログラミング言語の意味論の考え方を操作的意味論，指示意味論，記述的意味論に分類して，入門的解説をしながら，基本的な問題点についても指適してきた．しかし本稿で考えてきたプログラミング言語は FORTRAN，ALGOL に始まる古典的なスタイルのものを想定しており，Kowalski らの述語論理プログラミング，APL-Backus 流の関数的プログラミング，代数的プログラム仕様記述言語，人工知能用言語，知識データベース用言語，(制限付きの)自然言語プログラミングなどのような新しい傾向のプログラミング言語については考慮していない．言語の枠組をこのように拡大したとき，Scott-Strachey の理論では困難が予想され，Montague logic は computational な立場から問題がある．また VDL（あるいは VDM）を用いる方法も考えられるが繁雑さは避け難いと言える．

本稿を執筆するに当って意味論の研究の全貌を網羅す事も考えられ，あるいは本会編集部の意図もそのような観点からの展望解説にあったかも知れないが，筆者の独断でこのような入門的解説にしてしまった．意味論の目的や役割，意味論の理論化上の問題点，理論の相互関係・特徴・欠点，理論の限界など，プログラミング言語の意味論についての概要を初心者が知る上での参考となれば幸いである．また本稿では，形式的あるいは数学的な議論の詳細は殆んどすべて省略してしまったが，これについては文献を参照されたい．

## 参考文献


1) Yanov, Y. I.: The logical schemes of algorithms, Problems of Cybernetics I (1958).
2) McCarthy, J.: Towards a math, theory of computation, IFIP Congress '62 (1963).
3) Lucas, P. and Walk, K.: On the formal def. of PL/I, Automatic Programming, Vol. 6, No. 3 (1969).
4) Burstall, R.: Formal description of program structure and semantics in first-order logic, Machine Intelligence 5 (1970).
5) Scott, D. and Strachey, C.: Towards a math. semantics for computer lang., Proc. Symp. Computer and Automata (1973).
6) Hoare, C. and Wirth, N.: An axiomatic def. of the programming language PASCAL, Acta Informatica, 2 (1973).

7) Hoare, C. and Lauer, P.: Consistent and complementary formal theories of semantics of programming languages, Acta Informatica, 3 (1974).
8) Pratt, V.: Semantical considerations on Floyd-Hoare logic, Proc. 17th IEEE Symp. FOCS (1976).
9) Cook, S.: Soundness and completeness of an axiom system for program verification, SIAM J. Computing, Vol. 7, No. 1 (1978).
10) Owicki, S.: A consistent and complete deductive system for the verification of parallel programs, Proc. 8th ACM Symp. on Theory of Computing (1976).
11) Plotkin, G.: A powerdomain construction, SIAM J. Computing Vol. 5 (1976).
12) Flon, L. and Suzuki, N.: Consistent and complete rules for the total correctness of parallel programs, Proc. 19th IEEE Symp. FOCS (1978).
13) de Bakker, J. W.: Semantics of programming languages, Advances in Information Systems and Science (ed. J, T. Tou) (1969).
14) Kaplan, D.: Proving thingsabout programs, Proc. Fourth Princeton Conference on Information Science & Systems (1970).
15) Tennet, R. D.: The denotational semantics of programming languages, CACM Vol. 19, No. 8 (1976).
16) Manna, Z. and Waldinger, R.: The logic of computer prog., IEEE Trans. on SE-4 (1978).
17) de Bakker, J. W.: Recursive Procedures, Math. Centre Tract 24, Amsterdam (1971).
18) Engelfeit, J.: Simple Program Schemes and Formal Lang., Springer Lecture Notes in Computer Science, 20 (1974).
19) Manna, Z.: Mathematical Theory of Computation, McGraw-Hill (1974). (五十嵐滋訳：プログラムの理論，日本コンピュータ協会 (1975)).
20) 伊藤：プログラム理論，コロナ社 (1975).
21) 小野：プログラムの基礎理論，サイエンス社 (1975).
22) Greibach, S.: Theory of Program Structures, Springer Lecture Notes in Computer Science, 36 (1975).
23) Donahue, J.: Complementary Definitions of Programming Language Semantics, Springer Lecture Notes in Computer Science, 42 (1976).
24) Milne, R. and Strachey, C.: A Theory of Programming Language Semantics, Chapman and Hall (1976).
25) Stoy, J.: Denotational Semantics, MIT Press (1977).
26) Constable, R.: A Programming Logic, Winthrop (1978).
27) Harel, D.: First-Order Dynamic Logic, Springer Lecture Notes in Computer Science, 68 (1979).
28) Steel, T.: Formal Language Description Languages for Computer Programming, North-Holland (1966).
29) Schwartz, J.: Mathematical Aspects of Computer Science, Proc. Symp. Applied Math. Vol. 19, AMS (1967).
30) Engeler, E.: Semantics of Algorithmic Languages, Springer (1971).
31) Rustin, R.: Formal Semantics of Programming Languages, Prentice-Hall (1972).
32) Ershov, A.: Symposium on Theoretical Programming, Springer Lecture Notes on Computer Science, Vol. 5 (1974).
33) Newhold, E.: Formal Description of programming Concepts, North-Holland (1978).
34) Yeh, R.: Current Trends in Programming Methodology, I, II, III, IV Prentice Hall (1978).
35) Kahn, G.: Semantics of Concurrent Computation, Springer Lecture Notes in Computer Science, 70 (1979).

〈コメント〉

1)—7) はセマンティクスに関する古典的とも言える論文の例；10)—12) は並列プログラムのセマンティクスにする論文の例；13)—16) は解説論文の例；17)—27)は著書の例；28)—35)はシンポジウム論文集の例．文献 8) は dynamic logic, 文献 9) は relative completeness に関する論文である．最近，カテゴリや関手を用いた代数的理論も盛んになりつつある．例えば，文献 34) の第Ⅳ巻には Initial Algebra Semantics に関する Goguen-Thatcher らの試みがあり，筆者らは Scott 理論のカテゴリ論的拡張に対する試み（信学会オートマトンと言語研究会 AL 76-55 (1976)）で Unfolding semantics の形式化の基礎を与えたりしている．

なお Springer Lecture Notes in Computer Science, ACM Symp. on Theory of Computing, IEEE Symp. on Foundations of Computer Science, Acta Informatica, CACM, JACM, IFIP の 1974 年以降の論文集や学会誌にはセマンティクスに関する多数の理論的論文が見られるから参照されたい．なお本会誌 Vol. 20 の No. 1 にプログラミング方法論（鳥居他），No. 8 に算法理論（沢村），No. 11 にスコット理論（高橋）に関する解説があるから参照されたい．

（昭和 54 年 10 月 3 日受付）