1 序論

プログラミング言語の理論は，計算機科学の主要な分野のひとつである．この本の主題は，これに対する，論理学的および意味論的なアプローチへの入門である．全体としては，大きく分けて以下のような四部構成になっている．

第 I 部　数理論理学入門
第 II 部　プログラムの不動点意味論
第 III 部　プログラミング論理入門
第 IV 部　リアクティブ・システムのための時相論理

各部は，基本的に独立して読めるように書かれている．しかし，おのおのの理論の進展を読みすすむに伴い，第 I 部における論理学的アプローチと第 II 部で展開される意味論的なアプローチとが相補的な役割を果たすことが明らかになる．同時に，各部で展開される話題もまたお互いに深く関連しあっていることがわかってくるであろう．とくに，第 III 部では，第 I 部と第 II 部で紹介する基本的な考え方が組み合わせて用いられる場面を紹介する．さらに，第 IV 部では，リアクティブな計算システムに対する論理学的・意味論的な研究の現況を垣間見ることになる．

本章では，本書全体を貫く，中心的な概念について入門的な解説を与える．本章の理解は後の章の技術的な内容を理解する上で必ずしも必須ではない．しかし，なぜこのような数学理論が計算機科学という分野のなかで必要とされ，盛んに研究されてきたのか知ることは，はじめて学ぶ人にとって，この本の意図すること

を理解するための重要な指針になるはずだ.

§1.1 プログラミング言語とその理論

第I部の数理論理学の入門の部分を除いて，この本での興味の中心は，**プログラミング言語**およびプログラムが表現する**計算**(computation)という現象の分析にある．ここで，プログラミング言語とは，コンピュータになんらかの仕事をさせるための命令を記述する言語のことである．仕事は，実用的な仕事でも，あるいはゲームのような娯楽的なものでもよい．本書でいう計算とは，そのコンピュータが行う仕事の通称と，いまのところは考えていただきたい．プログラミング言語は，20世紀の後半になって人工的に発明され，現在も発明・改良されつつある．

さて，現在普及している特定のプログラミング言語ないしソフトウェアについて語ることが，本書の目的ではない．本書では，一般のプログラミング言語と計算の概念について，数学的・理論的に厳密に取り扱うための枠組みやそれを用いた成果について語る．別の言葉でいうと，この本ではプログラムの書き方は教えないが，プログラムとプログラムが書き表している計算の内容をきちんと議論・推論できるようになるための重要な考え方を提供する．これらの点については，同様の意図で書かれている姉妹編『コンピュータサイエンス入門 アルゴリズムとプログラミング言語』も参照して欲しい.

巷にあふれかえる多彩なソフトウェアの圧倒的大多数は，なんらかのプログラミング言語を用いて書かれたプログラムである．そのプログラムという，ありふれているようで必ずしもよく理解されているとは言いがたい存在に秘められた，隠れたからくりや本質に，特に予備知識がなくても迫れる手法を，多くの方に紹介したい，というのが本書の執筆者たちの願いである.

より具体的には，すでに言及した論理学的な考え方と意味論的な考え方がその核となる．以下ではこれらの背景について解説する．数理論理学，プログラミング言語の意味論，これらはいずれも，20世紀の数学の本質的な成果を反映させた分野であり，計算機科学の独自のモチベーションにより，応用数学の一分野としてももっとも成功した話題である．(数理論理学は純粋数学の立場からももちろん重要な分野であるが，本書では，あくまでも計算機科学の立場にたった紹介を行う.) 本書で展開するのは，その美しい成果のごくはじめの部分にすぎないが，本書を読むことが，これら魅力あふれる研究分野に読者が興味をもつきっかけとなれば幸いである.

§1.2 数理論理学と形式化の技法

ここで，ちょっとプログラミング言語のことは忘れて，次のような，やや抽象的な話題を考えて欲しい.

証明とは何だろう？ あるいは，「正しい」とはどういうことだろう？ そのような疑問を，読者は持ったことがあるだろうか —— 実生活においても(「浮気してないことの証明」だとか，「君は正しい」とか)，あるいは数学の問題を考えているときにでも(入試問題の「○○を証明せよ」「△△が正しいことを示せ」など).

尋ねておいて無責任な話であるが，実のところ，これらの疑問に関して本当に心の底から納得のいく答えを筆者たちは知らない．実生活における疑問はもとより，問題を，曖昧さがないと信じられている数学における「証明」「正しさ」に限定しても，である．たとえば，ここに1000ページからなる，長い長い難解な数学の証明があるとしよう(そういう証明は実際に存在する)．おそらくそれを最初から最後まで読み通し，完全に理解して「わかった！」と言うことができる人はそう多くはないだろう．ほとんどの数学者は，その証明(と書いた人の良心)を信じているのである．逆に，極端に短い証明も問題である．筆者が採点する試験の答案で，「明らか」の一言で証明が片付けられていた場合，まれに天才が本当にそう思って書いた場合もないことはないだろうが，たいていの場合，これは単に解答者の怠慢に過ぎないので，筆者は情け容赦なく0点をつけてしまうのである(しかし，天才が本当に明らかと思って書いていることもまれにある)．こういったことは，日常の数学的な活動における「証明」という概念が，たとえ数学の専門家にとっても，案外あいまいなものになっていることを示唆している.

本書では，そのような哲学的な(または倫理学的な)問題には答えない．そのかわりに，数学的な厳密な概念として「正しさ」および「証明」を扱う方法と，その利点について解説する．これが，第I部の主題である.

数理論理学では，曖昧な証明はいっさい排除する．形式的に，ある一定のルー

ル(文法といってもよい)にしたがって得られるものだけを証明とよび，またある証明の結論になっている論理式のことを証明された論理式，すなわち定理とよぶのである．また，「正しさ」を決める尺度についてもきちんと定式化する．すなわち，ある定められた数学的な構造(の集まり)に照らし合わせて，ある命題が正しいか否かを定めるのである(これは，この後で解説する意味論の考え方の一例でもある)．

具体的には，第I部では，はじめに一階述語論理の理論の基礎を学ぶ．これは，形式を重んじる考え方が成功をおさめた場面であるといえる．どういうことかというと，ある限定された数学理論について，その理論のための証明を形式的な記号操作として表現することができ，しかもその形式的な証明の概念が理論における正しさの概念とぴったり一致するという，完全性とよばれる性質が成り立っているのである．(数理論理学者たちは，さらに踏み込んだ考察の末，より実用的な数学の体系においては完全性は成り立たないことを突き止めた．この不完全性の話題もまた今世紀の数学や計算機科学の発展を理解するために重要な鍵となるものであるが，本書ではあえて立ち入らない．読後，さらに興味を持たれた読者は，巻末に挙げた数理論理学の教科書を参照されたい.)

ところで，口で言うのは簡単だが，形式的な証明を実際にやってみるとなかなか大変である —— ひとつひとつの論理的な推論をおこなうたびに，それが形式的に定められたルールで許されているかどうか常に確認しなくてはならないのだ．読者は第I部を学ぶなかで，そのわずらわしさを実感するであろう．

しかし，形式化の効果は絶大である．証明を形式的な記号のゲームに帰着させたことで，計算機科学への関連もぐっと深くなる．というのは，形式的な記号操作は，多くの場合人間の手にはおえない代物であるが，コンピュータにとってはもっとも得意とするものの一つだからである(逆に，厳密に形式化されていない概念の操作は，現在のコンピュータのもっとも苦手とするものである)．形式化された証明の概念を用いることで，定理のコンピュータによる自動証明や一部自動化への道がひらかれたと言ってよい．それはさらに，古典的な数学の理論のみならず，後で紹介するプログラミング言語の理論についてもあてはまる．プログラミング言語の理論とそのための証明の概念を形式化し，コンピュータ上で実装することにより，プログラムの正しさの(自動)検証という応用上大変重要な話題が大きく発展してきたのである．

第III部では，この魅力的な研究分野の，ほんの導入の部分を紹介する．プログラミング言語の理論と数理論理学のアイデア(形式主義)がうまく組合わさった例として理解してもらえれば幸いである．さらに，第IV部では，古典的な論理の枠組みではとらえきれない，計算の時相的な側面を分析するために，時相の入った論理体系，すなわち時相論理を導入し，とくにそのリアクティブ・システムとよばれる計算現象への応用を展開する．ここでもまた，計算機科学における論理学的アイデアと計算の意味論との興味深い(そしてより自由な)結合がみられるだろう．

§1.3 言語の意味論入門

現在の言語学では，言語を**構文論**(syntax)と**意味論**(semantics)との2つの補い合う側面から扱うことが，基本的な考え方となっている．

構文論とは，文法のむずかしい言い方と思えばよい．たとえば英語の場合

John love Mary.

は構文論上まちがっている．三人称単数現在の動詞にsがついていない．正しくは

John loves Mary.

である．つまらない話を始めたと思っても少し我慢していただきたい．

言語の始まりには必ずアルファベットというものがある．これはそれ以上分解できない(英語では atomic とか primitive とかいう)記号の(おそらくは有限)集合である．英語なら

{A, B, C, ..., Z, a, b, c, ..., z, !, ?, ', ...}

(厳密には，ここで space あるいは空字も含まれることに注意しよう．たとえば John loves Mary. が JohnlovesMary. になっては困る.) 日本語なら

{あ, い, う, ..., ん, ア, イ, ウ, ...,「,」, ...}∪当用漢字全部

である.

構文論とは，アルファベットの要素をどのように並べればその言語の正しい文章ができるかを定めた規則(およびそれに関する理論)である．言い換えると，その言語の文章とは，アルファベットの要素からなる(有限の)列で構文論の定める規則を満たすものをいう．もっと簡単な言語の例には，交通信号がある．そのアルファベットは{赤，青，黄}であり，3つの要素それぞれ一つが文章である．なお，この場合には，可能な文章は有限個(3つ)しかなく，しかもどれも限られた長さしか持たない．しかし，英語や日本語のようなより複雑な言語においては，可能な文章は無限に存在し，しかもいくらでも長くすることができる．このように，構文論によりさまざまな複雑さの言語を定めることができる．

一方，意味論とは，構文論が文章と認めるアルファベットの列に，何らかの人間の概念(人が考えうる対象)を対応させることである．たとえば交通信号の意味論は，赤に＜止まれ＞，青に＜進め＞，黄には＜早く行け＞を対応させる．また

John loves Mary.

と

ジョンはメアリにほれている.

は意味論からはほぼ同等の文章である．すなわち翻訳とは，たとえば英語の文章を，意味論の視点で同等の日本語の文章に変換することである．

以上は自然言語の話であったが，自然言語については構文論も意味論も数学的に議論するのはなかなか容易でないことが知られている．幸い，本書で問題とするプログラミング言語では，厳密に構文論および意味論を展開する道が開かれており，その結果から得るところも大きい．この，意味論を基礎に置いたプログラミング言語に関する理論的考察とその成果について解説することが，第II部の主題である．その基本方針は，第IV部においても，より自由な計算のモデルの理論を展開していくうえで受け継がれてゆく.

§1.4 プログラミング言語の意味論

以上の一般論をふまえたうえで，この節ではプログラミング言語の意味論の具体例を紹介しよう.

(a) プログラミング言語 petit の構文

ここでは，非常に簡単なプログラミング言語 **petit**(プチ)を定義する．言語を定義するには，まずその構文規則(文法)を決めなくてはならない．これは数理論理学の場合でもプログラミング言語の場合でも同じである．プログラミング言語の場合，文章に相当するのがプログラムである.

petit の構文領域

まず，**構文領域**とよばれる，記号の列からなる集合 **Var**，**Exp**，**Pro** を導入する(これらの具体的な定義は後述する構文規則により与えられる).

ξ : **Var**　プログラム(の中で用いられる)変数の記号全体

ε : **Exp**　式(expression)の全体

τ : **Pro**　プログラムの全体

ξ，ε，τ はそれぞれその上を動く(メタな)変数である．なお，$x : X$ のような表記法は，数学で用いられる $x \in X$ (x は集合 X の要素)のことだと考えて読んでかまわない．この記法はプログラミング言語の理論において広く用いられているものである.

petit の構文規則

これらの構文領域の要素がいかなる記号の列であるかを定めるために，次のような規則を与える.

(i) ξ ::= A | B | ⋯ | Z

(ii) ε ::= 0 | ξ | suc ε

(iii) τ ::= $\xi := \varepsilon$ | $\tau;\tau$ | for ε times do τ end

このような構文規則の与え方は形式言語(formal language)の理論で文脈自由文法(context-free grammar)とよばれる．ここで用いられた規則の表記法はバッカス-ナウア記法(Backus-Naur Form，BNF)または単にバッカス記法などとよばれるものである．バッカス記法は，本書のさまざまな場面で使用される．

規則(i)は，「::=」の左辺にあらわれるメタ変数ξが，右辺にあらわれるAかBか，…，またはZのいずれかを表すということを述べている．すなわち，A，B，…，Zが**petit**で用いられる変数の記号全体である．

規則(ii)では，左辺のメタ変数εは，右辺の，0か，またはメタ変数ξで表されるもの((i)で規定されたように，これはAかBか，...，またはZのいずれかである)か，あるいはsuc εと表されるもののいずれかであることを定めている．たとえば，εはsuc Aやsuc suc 0などの上を動く．いいかえると，suc Aやsuc suc 0は**petit**の式である．

規則(iii)も同様に理解される．そして，**petit**のプログラムは**Pro**の任意の要素，すなわちメタ変数τによって表されるものと定義する．プログラムの例として，たとえば以下のようなものがある．

X:=A; X:=B; for C times do X:=suc X; Y:=suc Y end

問 1.1 このプログラムが上に与えた構文規則によってどのように生成されるかを考えよ．

(b) petit の意味論

petitのプログラムがどんな記号の列であるかはわかったが，この記号列をコンピュータで実行させるとき，何らかの効果をもたらしてほしい．この実行の効果をプログラムの意味とよぶことにしよう．プログラムがどんな記号の列かをみることによってその意味を決定できる系統的な方法があると望ましい．

プログラム言語のマニュアル(使用手引き)でよく用いられる意味の記述の方法は，日本語や英語を用いて説明することである．**petit**の各構文領域の要素に対応する意味を，この方法で記述すると表1.1のようになる．

petitは簡単だから，このような意味の定め方で十分だと感じられるだろうが，実際に世の中で使われるプログラム言語があつかうデータや機能は多様かつ複雑で，プログラムを実行して得られる効果を正確に曖昧さなく記述することは上の

表 1.1 **petit** の直感的な意味

構文要素	意味
0	整数値 0
suc 0	整数値 1
suc A	プログラム変数 A が現在もつ値に 1 を加えて得られる整数値
X:=ε	変数 X に ε の値を入れる
$\tau_0;\tau_1$	まず τ_0 を実行し，その後に τ_1 を実行する
for ε times do τ end	τ を ε 回実行する

ような方法では困難である．

正確さや厳密さに加え，プログラム言語の意味の記述法が満たすべき大切な要件は抽象性である．プログラムの意味とはその実行がもたらす効果であるとしたが，同じ効果をもつプログラムが必ずしも同一の記号の列であるわけではない．構文上は相異なるプログラムがまったく同一の働きをもつこともある．簡単な例を考えよう．

(A) x:=x+y; y:=y+x; x:=2

(B) y:=2*y+x; x:=2　　(* は掛け算を表す)

(A)に与えたプログラムと(B)に与えたプログラムは，字面は異なっても同じ意味をもってほしい．実際の世界では，プログラムを(記号の列として)書く方法にはいろいろな規則があり，その実行にはいろいろなこまかい仕組みや面倒な構造が必要だが，我々が知りたいプログラムの意味は，そのような表面的な現象を超えた，プログラムが本来的に期待されている働きである．意味の記述が抽象的であってほしいとは，これらの非本質的な具体的な構造や仕組みを，意味論に混入させたくない，ということである．

意味領域

この目標を実現するため，集合や関数のような簡単な数学の概念を用いて**petit**に意味を与えよう．基本的な方針は，**petit**の構文要素のそれぞれを，適当な数

学的領域の要素に対応させることである．この数学的領域のことは第II部で順序構造に基づいたデータ領域として議論されるのだが，ここでは単に集合を考えれば十分である．**petit** の意味記述のためには，次のような数学的領域(意味領域という) **N**, **S** および **C** を考える．

$\mathbf{N}$	自然数の全体 (0 を含む)
$\mathbf{S} = (\mathbf{Var} \to \mathbf{N})$	コンピュータの状態全体
$\mathbf{C} = (\mathbf{S} \to \mathbf{S})$	コンピュータの状態の変換全体

ただし，$A \to B$ は集合 A から集合 B への関数全体を表す(章末の表記法を参照)．ここでは，コンピュータの状態は，その時点で記憶されている各プログラム変数の値により定まると考えるので，プログラム変数の集合 **Var** からそれらが取りうる値の集合 **N** への関数全体 $\mathbf{S} = (\mathbf{Var} \to \mathbf{N})$ をコンピュータの状態全体とみなすのである．

プログラムの実行につれて，プログラム変数の値は変化する．ここでは **petit** のプログラムの実行の効果を，「各プログラム変数がもつ値が，実行を開始してから停止するまでに，どのような変化をするか」によって表す．すなわち，コンピュータの状態の変換，**C** の要素として解釈するのである．コンピュータの状態とは，変数全体が取りうる値への関数 $\sigma \in (\mathbf{Var} \to \mathbf{N})$ である(**petit** では変数の値として $\mathbf{N} = \{0, 1, 2, \ldots\}$ だけが可能である)．ある時点でのコンピュータの状態を σ とすると，そのときの変数 $\xi \in \mathbf{Var}$ の値は，$\sigma(\xi) \in \mathbf{N}$ である．

意味関数

意味領域が決まったところで，今度は **petit** の構文要素が意味領域のどんな要素に対応するかを定めたい．そのためには **Pro** や **Exp** のような構文領域から意味領域への関数(意味関数という) $\mathcal{E}$ および $\mathcal{C}$ を導入する．

$$\mathcal{E} : \mathbf{Exp} \to (\mathbf{S} \to \mathbf{N})$$
$$\mathcal{C} : \mathbf{Pro} \to \mathbf{C}$$

以下，意味関数を定義するために，構文領域上を動くメタ変数 ξ, ε, τ に加えて，意味領域上を動くメタ変数

$$\nu : \mathbf{N}, \quad \sigma : \mathbf{S}, \quad \theta : \mathbf{C}$$

を導入しておく．

意味関数 $\mathcal{E}$ の定義

意味関数 $\mathcal{E} : \mathbf{Exp} \to (\mathbf{S} \to \mathbf{N})$ は以下のように定められる．

(i) $\mathcal{E}[\![0]\!](\sigma) = 0$
(ii) $\mathcal{E}[\![\xi]\!](\sigma) = \sigma(\xi)$
(iii) $\mathcal{E}[\![\mathrm{suc}\ \varepsilon]\!](\sigma) = \mathcal{E}[\![\varepsilon]\!](\sigma) + 1$

構文規則により，**Exp** の要素は 0 と ξ (**Var** の任意の要素)から始まって，suc をほどこして得られるものであった．したがって(i)～(iii)によって任意の **Exp** の要素に対する $\mathcal{E}$ の定義が得られる．このように，各部分の表現に関する定義からはじめて全体の表現に関する定義が定まるような定義の仕方を，(構文規則に関して)帰納的な定義であるという．

$\mathcal{E} : \mathbf{Exp} \to (\mathbf{S} \to \mathbf{N})$ であるから，$\varepsilon \in \mathbf{Exp}$ および $\sigma \in \mathbf{S}$ に対し，$\mathcal{E}(\varepsilon) \in \mathbf{S} \to \mathbf{N}$ であり，$\mathcal{E}(\varepsilon)(\sigma) \in \mathbf{N}$ となる．構文上の概念と意味上の概念の区別を明確にするために，**Var**, **Exp**, **Pro** のような構文領域の引数は，(–) のかわりに $[\![-]\!]$ を用いて明示する．たとえば，$\mathcal{E}(\varepsilon)$ のかわりに $\mathcal{E}[\![\varepsilon]\!]$ と書くことにする．同様に $\mathcal{C}(\tau)$ は $\mathcal{C}[\![\tau]\!]$ と表す．$\mathcal{E}[\![\varepsilon]\!]$, $\mathcal{C}[\![\tau]\!]$ をそれぞれ ε, τ の表示(denotation, 「意味をさし示すもの」の意)とよぶことにする．

$\mathcal{E}$ による **Exp** の要素の表示とは単なる整数値ではない．たとえば $\mathrm{suc\ A} \in \mathbf{Exp}$ の値はプログラム変数 A の値が定まって初めて決まる．したがって，suc A の表示 $\mathcal{E}[\![\mathrm{suc\ A}]\!]$ は，A の値を決定するコンピュータの状態が与えられたときに整数値を返すような関数 $\mathcal{E}[\![\mathrm{suc\ A}]\!] \in (\mathbf{S} \to \mathbf{N})$ であると考えられる．$\varepsilon \in \mathbf{Exp}$ に対し，$\mathcal{E}[\![\varepsilon]\!](\sigma) \in \mathbf{N}$ を，状態 σ における ε の値とよぶ．

定義(i)～(iii)の内容を説明しよう．(i)では，式 0 の表示(意味)は，コンピュータの状態にかかわらず 0 であることを定めている．ここで，左辺の $\mathcal{E}[\![0]\!]$ の 0 は **Exp** の要素，すなわち記号であるが，右辺の 0 は記号としての 0 ではなく，整数値という(インドで発見されたといわれる)概念上のゼロを示していることに注意しよう．

(ii)は，プログラム変数の表示 $\mathcal{E}[\![\xi]\!] \in (\mathbf{S} \to \mathbf{N})$ は，コンピュータの状態 σ が決まると整数値 $\sigma(\xi)$ を返すような関数であることをいっている．

(iii)では，suc ε の表示を，ε の表示に 1 を加えたものとして定めている．

例 1.1 式 suc suc 0 の表示を計算してみよう．

$$\begin{aligned}
\mathcal{E}[\![\mathsf{suc\ suc\ 0}]\!](\sigma) &\overset{\text{(iii)}}{=} \mathcal{E}[\![\mathsf{suc\ 0}]\!](\sigma) + 1 \\
&\overset{\text{(iii)}}{=} (\mathcal{E}[\![\mathsf{0}]\!](\sigma) + 1) + 1 \\
&\overset{\text{(i)}}{=} (0 + 1) + 1 \\
&= 2
\end{aligned}$$

すなわち，式 suc suc 0 の表示は，コンピュータの状態 σ に依存せず 2 になることがわかる． □

意味関数 $\mathcal{C}$ の定義

続いて，**Pro** の要素の表示の定義を与えよう．**Pro** の要素の表示はコンピュータの状態の変換であったことを思い出してほしい．意味関数 $\mathcal{C} : \mathbf{Pro} \to \mathbf{C}$ を以下のように定める．

(iv) $\mathcal{C}[\![\xi := \varepsilon]\!](\sigma) = \mathit{update}(\xi, \mathcal{E}[\![\varepsilon]\!](\sigma))(\sigma)$

(v) $\mathcal{C}[\![\tau_0; \tau_1]\!](\sigma) = \mathcal{C}[\![\tau_1]\!](\mathcal{C}[\![\tau_0]\!](\sigma))$

(vi) $\mathcal{C}[\![\mathsf{for}\ \varepsilon\ \mathsf{times\ do}\ \tau\ \mathsf{end}]\!](\sigma) = \mathit{iterate}(\mathcal{C}[\![\tau]\!], \mathcal{E}[\![\varepsilon]\!](\sigma))(\sigma)$

ここで，関数 $\mathit{update} : \mathbf{Var} \times \mathbf{N} \to \mathbf{C}$ と $\mathit{iterate} : \mathbf{C} \times \mathbf{N} \to \mathbf{C}$ は次のように定義する．

$$\begin{aligned}
\mathit{update}(\xi, \nu)(\sigma) &= \sigma\{\xi : \nu\} \\
\mathit{iterate}(\theta, \nu) &= \underbrace{\theta \circ \theta \circ \ldots \circ \theta}_{\nu\text{回}}
\end{aligned}$$

ただし，$\sigma \in \mathbf{S}$ に対し，$\sigma\{\xi : \nu\} \in \mathbf{S}$ は

$$\sigma\{\xi : \nu\}(\eta) = \begin{cases} \nu & \mathit{if}\ \ \eta = \xi \\ \sigma(\eta) & \mathit{if}\ \ \eta \neq \xi \end{cases}$$

また，$\circ$ は関数の合成を表す $((g \circ f)(x) = g(f(x)))$．

$\mathcal{C}$ の定義は $\mathcal{E}$ と同様に構文規則に対して帰納的であることに注意しよう．いいかえると，

- まず，各原子的構文要素(0 や変数 A, ⋯, Z)の表示が定まる．
- 合成されてできる構文要素の表示は，それを構成する各部分の表示が決まれば定まる(たとえば，ε と τ の表示が決まれば，for ε times do τ end の表示が決まる)．

(iv)で与えられる代入文 $\xi := \varepsilon$ の表示は，状態 σ から新しい状態 $\sigma\{\xi : \mathcal{E}[\varepsilon](\sigma)\}$ を作り出す関数(すなわち **C** の要素)である．

(v)では，τ_0, τ_1 の表示をそれぞれ θ_0, θ_1 とすれば，$\tau_0; \tau_1$ は τ_0 を実行してから τ_1 を実行するので，状態 σ を $\theta_1(\theta_0(\sigma))$ に変化させる．いいかえると，$\tau_0; \tau_1$ の表示は $\theta_1 \circ \theta_0$ である．

(vi)でも同様に考えられる．for ε times do τ end は，τ を $\nu = \mathcal{E}[\![\varepsilon]\!](\sigma)$ 回くり返すことであるから，その表示は，$\mathcal{C}[\![\tau]\!]$ を ν 回合成したものである．

例 1.2 次のプログラムに表示を与えよう：

X:=A; for C times do X:=suc X; end

直感的には，このプログラムは，はじめに変数 X に変数 A の値を代入し，続いて X の値を 1 増やす操作を C の値の回数だけ繰り返すものだと考えられる．その正確な表示は，以下のように計算される．

$$\begin{aligned}
&\mathcal{C}[\![\mathsf{X := A;\ for\ C\ times\ do\ X := suc\ X\ end}]\!](\sigma) \\
&\overset{\text{(v)}}{=} \mathcal{C}[\![\mathsf{for\ C\ times\ do\ X := suc\ X\ end}]\!](\mathcal{C}[\![\mathsf{X := A}]\!](\sigma)) \\
&= \mathcal{C}[\![\mathsf{for\ C\ times\ do\ X := suc\ X\ end}]\!](\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}) \\
&\quad \left(\text{ここで } \mathcal{C}[\![\mathsf{X := A}]\!](\sigma) \overset{\text{(iv)}}{=} \mathit{update}(\mathsf{X}, \mathcal{E}[\![\mathsf{A}]\!](\sigma))(\sigma) = \mathit{update}(\mathsf{X}, \sigma(\mathsf{A}))(\sigma) = \sigma\{\mathsf{X} : \sigma(\mathsf{A})\} \right) \\
&\overset{\text{(vi)}}{=} \mathit{iterate}(\mathcal{C}[\![\mathsf{X := suc\ X}]\!], \mathcal{E}[\![\mathsf{C}]\!](\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}))(\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}) \\
&= \mathit{iterate}(\mathcal{C}[\![\mathsf{X := suc\ X}]\!], \sigma(\mathsf{C}))(\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}) \\
&\quad \left(\text{ここで } \mathcal{E}[\![\mathsf{C}]\!](\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}) \overset{\text{(ii)}}{=} \sigma\{\mathsf{X} : \sigma(\mathsf{A})\}(\mathsf{C}) = \sigma(\mathsf{C}) \right)
\end{aligned}$$

$$
\begin{aligned}
&= \underbrace{\mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!] \circ \cdots \circ \mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!]}_{\sigma(\mathsf{C})}(\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}) \\
&= \underbrace{\mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!] \circ \cdots \circ \mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!]}_{\sigma(\mathsf{C})-1}(\sigma\{\mathsf{X} : \sigma(\mathsf{A})\}\{\mathsf{X} : \sigma(\mathsf{X}) + 1\}) \\
&\quad \left(\begin{array}{ll} \text{ここで } \mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!](\sigma) & \overset{\text{(iv)}}{=} \ \mathit{update}(\mathsf{X}, \mathcal{E}[\![\mathsf{suc\ X}]\!](\sigma))(\sigma) \\ & \overset{\text{(iii)}}{=} \ \mathit{update}(\mathsf{X}, \mathcal{E}[\![\mathsf{X}]\!](\sigma) + 1)(\sigma) \\ & \overset{\text{(ii)}}{=} \ \mathit{update}(\mathsf{X}, \sigma(\mathsf{X}) + 1)(\sigma) \\ & = \ \sigma\{\mathsf{X} : \sigma(\mathsf{X}) + 1\} \end{array} \right) \\
&= \underbrace{\mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!] \circ \cdots \circ \mathcal{C}[\![\mathsf{X} := \mathsf{suc\ X}]\!]}_{\sigma(\mathsf{C})-1}(\sigma\{\mathsf{X} : \sigma(\mathsf{A}) + 1\}) \\
&= \cdots \\
&= \sigma\{\mathsf{X} : \sigma(\mathsf{A}) + \sigma(\mathsf{C})\}
\end{aligned}
$$

すなわち，

$$\mathcal{C}[\![\mathsf{X} := \mathsf{A};\ \mathsf{for\ C\ times\ do\ X} := \mathsf{suc\ X\ end}]\!](\sigma) = \sigma\{\mathsf{X} : \sigma(\mathsf{A}) + \sigma(\mathsf{C})\}$$

したがって，このプログラムの意味は，変数 A と変数 C の和を変数 X に代入することである． □

(c) 表示的意味論

以上述べた **petit** のプログラムへの意味の与え方は，さまざまなプログラミング言語に対して有効である．その原則を一般化して述べると次のようになる．

> プログラミング言語の意味は，**Syn** をその言語の構文領域とすると，意味領域 **D** と意味関数 $\mathcal{V} : \mathbf{Syn} \to \mathbf{D}$ の対 $\langle \mathbf{D}, \mathcal{V} \rangle$ によって規定される．構文要素 $s \in \mathbf{Syn}$ に対し，$\mathcal{V}(s)$ (いままで使用した記法によれば $\mathcal{V}[\![s]\!]$) を s の表示とよぶ．

このような意味の与え方をプログラミング言語の**表示的意味論**(denotational semantics)とよぶ．第 II 部では，とくにプログラミング言語の本質的な要素である再帰プログラムに限定して，表示的意味論を展開するが，この方針を適用して，より実際的で複雑な言語，たとえば Java などにも，表示的意味が定義できるのである．また，第 I 部の論理学的なアプローチと結び付いた例として，第 III 部では，while 文を用いて繰り返しを記述できるようなプログラミング言語のプログラムについて推論するための論理を，表示的意味論を土台にして構築する．

表示的意味論は，今までにコンピュータサイエンスが生んだ最も深く美しい数学的体系のひとつである．実際，コンピュータサイエンスでは，いろいろな種類の意味論をプログラムやそのモデルに与えてきた．その中で表示的意味論は最も数学的にアピールする理論のひとつである．(誤解を避けるため断っておくが，コンピュータサイエンスは数学者に誉めてもらうために存在する学問だと言おうとしているのでは，決してない．コンピュータサイエンスは，コンピュテーションという現象の中から本質的な構造や理論を，いろいろなテクニックを用いて抽出し，議論し，その結果をコンピュータやネットワークのシステムへ応用しようとする，独自の問題意識を持つ数理科学である．そこでは，既存の数学を利用するとともに，数理論理学や代数，幾何などに新たな問題領域を提供している．物理学の数学への関係の仕方に似ているところが多少あるかもしれない．)

プログラミング言語の研究の流れから見ると，現在では，表示的意味論は，古典的・基礎的な技法として定着した一方， 主要な研究対象であるとはいえなくなった(ここ数十年で，表示的意味論に関する主要な問題はほぼ解決されたと考えられている)．しかし，表示的意味論の影響を受けつつ生まれ，発展してきた関数型プログラミング言語とよばれる一連の言語群は，近年多くの研究者やプログラマを惹きつけている．また，表示的意味論の数学的基礎である領域(第 5 章)とその理論，すなわち**領域理論**(domain theory)は，計算現象の数学的な精妙さを体現する独立した一分野を形成して，盛んに研究されている．

§1.5 リアクティブな計算の世界へ

現在のコンピュータにおいては，プログラムの実行の意味は，与えられた入力に対しコンピュータが状態変化を行なって停止する，というような，**petit** で用いたような単純な説明だけではとらえきれなくなっている．現実には，それぞれのプログラムが，なんらかのシステムないしプロセスとよばれる仕事の単位を記述し，それらのシステムが互いにインタラクティブなやりとりを行うことを通じ

て，より大きなシステムを形成していく，という現象が，多くの場面で，当たり前のようにおきている．たとえば，身近な例として，銀行の ATM システムや交通機関の管理システムなどを挙げることができるだろう．

そのような，相互に作用しあいながら，計算の実行やサービスの提供を(継続的に)行なうシステムのことを，**リアクティブ・システム**(reactive system)とよぶ．本書の第 IV 部では，第 III 部までに学んだ論理学的および意味論的な考え方を拡張することによって，これらリアクティブ・システムの性質および意味に対する数学的なアプローチを与える．

リアクティブ・システムの研究は，コンピュータサイエンスの歴史のなかでは比較的最近のものに位置づけられる．必ずしもまだ完全に確立されたとはいえない分野であるが，それだけに，新しいチャレンジングな問題を豊富に孕んでいるといえる．本書では，この新しい研究領域への，やさしい導入を与える．直感的に理解できる例を用いることにより，リアクティブ・システムに関する主要な問題意識が伝わるように配慮した．数学的な厳密な理解もさることながら，数学的な定義の背後にある計算現象の，内容的な理解に注意を払って読み進んでほしい．

§1.6 本書で使用する数学の概念および表記法

集合　一般的な用法に従う．たとえば，$\{a,b,c\}$ (a, b, c を要素にもつ集合)，$\{x \in A \mid P(x)\}$ (性質 $P(x)$ をみたすような集合 A の要素 x を集めてできる集合)，$a \in B$ (a は B の要素である)，$A \subseteq B$ (A は B の部分集合である)など．

関係　集合 A と B が与えられたとき，直積集合 $A \times B$ の要素を $\langle a,b\rangle$ ($a \in A$, $b \in B$)で表す．また，$A \times B$ の部分集合を A と B の間の**関係**とよぶ．A と B の間の関係 R について，$\langle a,b\rangle \in R$ であることを，a と b は R によって関係付けられているといい，aRb と書くことがある．

同値関係　集合 A が与えられたとき，以下の 3 条件を満たす A 上の(すなわち，A と A の間の)関係 R を，A 上の**同値関係**という．

(i) 任意の $a \in A$ について，aRa.

(ii) aRb ならば bRa.

(iii) aRb かつ bRc ならば aRc.

関数　A と B の間の関係 f で，任意の $a \in A$ に対し $\langle a,b\rangle \in f$ となるような $b \in B$ がただひとつ定まるものを，A から B への関数とよび，b を $f(a)$ と表す．$A \to B$ で集合 A から集合 B への関数全体のなす集合を表し，**関数空間**とよぶ．$f \in A \to B$ を $f : A \to B$ とも表す．

- $f : A \to B$，$X \subseteq A$ に対し，$\{f(x) | x \in X\} \subset B$ を $f(X)$ と表す．
- 関数 $f : A \to B$ について "$f(a) = f(b)$ ならば $a = b$" が成り立つとき，f を**単射**とよぶ．また，$f(A) = B$ であるとき，f を**全射**とよぶ．両方が成り立つとき全単射とよぶ．
- 関数 $f : A \to B$，$a \in A$，$b \in B$ に対し，$f\{a : b\}$ とは次のような関数 $f' : A \to B$ のことである．

$$f'(x) = \begin{cases} b & (x = a) \\ f(x) & (x \neq a) \end{cases}$$

また，$f\{a_1 : b_1, \ldots, a_n : b_n\}$ とは，$f\{a_1 : b_1\} \ldots \{a_n : b_n\}$ の略記である．

- $f, g : A \to B$ について，$f = g$ とは任意の $a \in A$ について $f(a) = g(a)$ が成り立つことである．これを関数空間の**外延性**(extensionality)とよぶ．
- $f : A \to B$，$g : B \to C$ に対し，関数 $g \circ f : A \to C$ を $g \circ f(x) = g(f(x))$ で定義し，f と g の合成とよぶ．
- (**ラムダ記法**)　x を A 上を動く変数，$E(x)$ を(変数 x を含むかもしれない)B の要素を表現する式とするとき，

$$\lambda x \epsilon A.E(x)$$

は $f(a)=E(a)$ $(a\in A)$ であるような関数 $f:A\to B$ を表現する．たとえば A を非負整数の集合で，B を実数の集合とすると，

$$\boldsymbol{\lambda} x\epsilon A.\sqrt{x}$$

は $f(a)=\sqrt{a}$ なる関数 $f:A\to B$ を表す．A が明らかなとき，単に $\boldsymbol{\lambda} x.E(x)$ と表すことにする．$\boldsymbol{\lambda}$ は重複して用いることができる．すなわち，

$$\boldsymbol{\lambda} x\epsilon A.\boldsymbol{\lambda} y\epsilon B.\sqrt{y+x}$$

は $f:A\to(B\to B)$ なる関数であり，$f(a)=\boldsymbol{\lambda} y\epsilon B.\sqrt{y+a}:B\to B$，$f(a)(b)=\sqrt{b+a}\in B$ となる．一方，

$$\boldsymbol{\lambda}\langle x,y\rangle\epsilon A\times B.y+\sqrt{x}$$

は $f:(A\times B)\to B$ なる関数 $f(a,b)=b+\sqrt{a}$ である．この $\boldsymbol{\lambda}$ を用いた関数の表記法をラムダ記法とよぶ．

- 関数の定義には，where（ここで），if，iff（＝“if and only if”），otherwise を適宜用いる．たとえば，

$$g(x,y)=x+f(y)\ \text{ where }\ f(y)=\sqrt{y}$$

は $g(x,y)=x+\sqrt{y}$ ということを表す．また，if や otherwise は

$$g(x)=\begin{cases} x & \text{if}\quad x>0\\ 0 & \text{otherwise}\end{cases}$$

などのように用いる．一方，

$$f:A\to B;\ \ x\mapsto\sqrt{x}$$

なる表現で，定義域，値域を含めた関数の定義を与えることが多い．この例では，f は定義域 A と値域 B を持ち $f(x)=\sqrt{x}$ で定められる関数，ということになる．

- $c\in B$ に対し，$\boldsymbol{\lambda} x\epsilon A.c:A\to B;\ \ x\mapsto c$ を定値関数とよび，$\boldsymbol{\lambda} x\epsilon A.x:A\to A;\ x\mapsto x$ を恒等関数とよぶ．

- $f(x)$ や $f(E)$ の，x や E を関数 f の**引数**（英語の argument の訳）とよぶ（引数をプログラミング用語の parameter の訳語として用いることもある）．

部分関数　集合 A と B の間の関係 f で，A の各要素 a について $\langle a,b\rangle\in f$ なる $b\in B$ が高々ひとつしかないものを**部分関数**という（必ずひとつ決まるものは，先に述べた通常の関数に他ならない．通常の意味での関数を，部分関数との区別を強調して**全域関数**とよぶことがある）．f が A から B への部分関数であることを，$f:A\rightharpoonup B$ と書くことがある．また，$\langle a,b\rangle\in f$ なる b が存在すれば b を $f(a)$ で表し，存在しない場合には $f(a)$ は未定義である，などという．部分関数についても，ラムダ記法や $f\{a:b\}$ のような記法は関数の場合と同様に用いられる．