ま え が き

デジタルコンピュータを中核とする情報処理システムは，それ以前のアナログ機械と違い，複雑で大規模な問題も解くことが可能な，汎用の問題解決システムである．その基本原理は，問題解決に必要な情報や知識などをコード化，すなわち記号の列で表現し，その記号列を機械によって解釈し変換する，というものである．この問題解決パラダイムは，人が，記号の集合である文字（アルファベット）で構成された言語によって知的な活動を行うことに対比しうる，一般性を持つものである．この記号列として表現された情報や処理手順がプログラムである．

簡単な処理を行うプログラムであれば，コンピュータで直接実行可能な，機械命令の列として書くことも不可能ではないが，複雑で大規模な問題を解決するためには，より高水準のプログラムの記述システム，すなわち**プログラミング言語**が必要となる．自然言語が，情報の伝達手段にとどまらず，人間の思考活動の枠組みとしての役割を果たしているのと同様に，プログラミング言語も，コンピュータへ動作を指示するための手段にとどまらず，複雑で大規模なソフトウェアを構築するための枠組みを与える重要なものである．数学を典型とする近代の科学が，種々の抽象概念を組織的かつ階層的に導入することを可能にする記述システムを持つことによって急速な発展を遂げたように，ソフトウェア科学・工学が，複雑なソフトウェアを信頼性を持って効率よく開発する体系を確立し発展させていくためには，ソフトウェア構築上有用な種々の抽象化の概念や構造化の機構を効率よく構築していくことを可能にする，プログラムの記述システムが必要である．近代的なプログラミング言語の主な役割は，そのような記述システムを提供することである．

今日のソフトウェアシステムは，種々の形態のメディアおよび新しい計算パラダイムの出現により，急速にその複雑さを増しつつある．それに伴い，情報処理の種々の分野において，より高度な機能を装備した新しいプログラ

ミング言語の開発が望まれている．巨大な論理体系である大規模なソフトウェアシステムを記述する基礎となるプログラミング言語は，種々の豊富な機能を提供すると同時に，その言語で表現可能ないかなるプログラムの意味も厳密に定義されている，整合性ある体系でなければならない．そのようなプログラミング言語を実現するためには，プログラミング言語の持つ諸機能の間の相互関係の理論的分析や整合性の検証，およびそれに基づく厳密な意味の定義等が必要である．本書のテーマであるプログラミング言語の基礎理論の目的は，そのような分析や検証を可能にし，実用的なプログラミング言語の設計および実装の基礎を与えることである．

理論的な基礎に基づき開発された実用プログラミング言語の（数少ない）具体例として，ML 言語を挙げることができる．ML の定義 [38] は，誰が読んでも唯一の意味を持つ型理論の概念を用いて書かれており，その実装は，言語の意味の定義に基づき系統的に行われている．この厳密性は，ML で書かれたプログラムに極めて高い信頼性を与えている．さらに，ML は，多相型関数や型の自動推論機構などの高度な機能を提供している．これらの機能は複雑なソフトウェアを信頼性を持ってしかも効率よく開発していく上で有効性が高いが，これらは理論的基礎に基づく系統的な設計によって初めて可能になったものである．以上のような特徴を持つ ML（Standard ML）は，現時点では，最も良く設計された高水準の実用プログラミング言語の一つであり，またプログラミング言語の新しい機能等の研究の基礎ともなっている言語である．

この例からも伺われる通り，プログラミング言語の理論的基礎は，単なる理論的な興味の対象ではなく，望ましい機能を持つ実用プログラミング言語の系統的な開発を可能にする鍵となるものでもある．本書の目的は，プログラミング言語の数理的モデルを理解し，プログラミング言語の理論の基礎を習得することである．特に，ML を典型とする近代的なプログラミング言語の動作原理を理解するための基礎知識を獲得することを，具体的な目標とする．

プログラミング言語の理論は，**構文論**（syntax）と**意味論**（semantics）に大別される．構文論は，文としてのプログラムそのものの性質を扱う理論で

あり，プログラムの文法構造やその構成要素の持つべき型に関する性質，およびプログラムの同値性，プログラムが具体的にどのような操作を表現するか，等を対象とする．プログラムの表現する操作は，プログラムの意味にかかわることであるが，プログラムの文法構造に則して具体的に定義される意味は，プログラムの**操作的意味論**（operational semantics）と呼ばれ，プログラムの構文論の一部として論じられる．これに対してプログラムの意味論は，プログラムが表現（表示）する計算そのものを，プログラムの文法的な構造に依存しない抽象的数学的対象として表現し，それを通じてプログラムの持つ性質を調べる理論であり，より厳密には**表示的意味論**（denotational semantics）と呼ばれる．

これらのなかで，プログラミング言語の設計や実装の直接的な基礎となるものは，構文論的諸性質，特にプログラムの持ちうる型に関する性質，プログラムの操作的意味論，およびその両者の関係である．そこで，本書では，プログラミング言語の構文論的性質を詳しく取り扱う．プログラミング言語の表示的意味論については，その一般的な枠組み，およびその構文論との関係等を主に取り扱う．プログラミング言語の具体的な表示的意味論（モデル）の構築も，種々の興味深い問題を含んだ重要な研究対象であるが，このテーマについては，例えば文献 [54, 63, 49, 26, 64] などの教科書で，すでに詳しく取り扱われているので，詳しい内容はそれら良書に譲ることにする．

以下第 1 章で，本書で取り扱う基礎理論の構造，および他の理論との関連について説明した後，次章以降で展開するプログラミング言語の基礎理論の準備として，形式言語の帰納的な定義，およびラムダ計算に基づく計算モデルを概説する．さらに，その後の章で扱う種々の型システムの構造およびそれ以降の本書の展開について説明する．

本書は，情報科学関連の学部や大学院初年度のレベルを念頭においているが，プログラミング言語の基礎に興味のある者であれば，情報科学の基礎知識がなくても理解できるよう配慮し，必要な概念はすべて定義するように心がけた．そのような本書の性格上，取り上げることができなかった重要な概念や手法が数多くある．より深い理解のためには，上に挙げた表示的意味論の教科書の他に，本書のテーマに関連の深い計算モデルに関する教科書

[60, 62] や，型理論が詳しく扱われている教科書 [40] などを参考にされることを勧める．また，より専門的な学習の手がかりとして，本文中になるべく多くの関連する論文の引用を含めた．

最後に，本書の草稿を一緒にセミナーで読み，種々の誤り等を指摘下さった加藤岳臣氏，田村広樹氏，および草稿を注意深く読んで頂いた南出靖彦氏，小林孝次郎先生，本書の執筆をすすめて下さった武市正人先生，本書執筆のあいだお世話になった共立出版株式会社の坂野一寿氏に深謝いたします．

１９９７年１月

大堀　淳

目　　次

第1章　プログラミング言語のモデル

プログラミング言語は，実際に計算機で実行可能な計算の記述言語としての側面と，複雑な処理を行うソフトウェアの論理構造の記述システムとしての側面を持つ．この二つをうまく統合したものが，望ましいプログラミング言語といえる．プログラミング言語の基礎理論を展開するには，これら二つの側面を併せ持ったモデルの構築が必要である．本章では，プログラミング言語の基礎理論の土台となる計算のモデルについて概説する．

1.1　計算モデルの必要性

実際に計算機で実行可能な計算の最も直接的なモデルはvon Neumann型のコンピュータそのものであり，その動作は命令コードの列として記述される．このモデルは確かに実行可能であるという要件を満たしているが，高水準プログラミング言語が表現する計算のモデルとして，ふさわしいものではない．プログラミング言語の基礎理論の構築のためには，数学的，論理的な記述が可能な，より抽象度が高い計算モデルが必要である．一方，現在の数学や論理学で採用されている諸概念は，実際に計算機で実行されることを意図したものではなく，計算機で実行可能な処理の記述言語としての要件を必ずしも満たしていない．例えば，プログラムを入力から出力を計算するものと考えると，ほぼ数学で扱われる関数としてモデル化可能と期待されるが，数学で扱われる関数は，グラフとしての関数であり，実際に入力が与えられたらそれに対応する出力が計算可能であるというプログラムの持つ重要な

性質を含んでいない．ごく簡単な例として，任意の自然数xと任意の自然数の集合Xを引き数とする以下の関数fを考えてみよう．

$$f(X,x) = \begin{cases} 1 & (x \in X \text{ の場合}) \\ 0 & (x \notin X \text{ の場合}) \end{cases}$$

数学的には，関数fは厳密に定義され，その存在は疑い得ない．しかしこのごく簡単な関数でさえ，それを実現するコンピュータプログラムは存在し得ないことを，以下の簡単な議論で示すことができる．もしfを計算するプログラムがあれば，その一つの入力Xを固定することによって，関数$f_X(x) = f(X,x)$を計算するプログラムが構成できる．f_XはXが異なれば異なる関数であるから，fを計算するプログラムは，可算無限個を超える数の相異なるプログラムを生成しうることになる．しかしながら，およそプログラムは有限の記号列であり，異なったプログラムの数は可算無限個を超えることはない．したがって，関数fを実現するプログラムは存在し得ない．同様に，論理学で使われている法則も，計算機で実行可能なプログラムの記述システムの要件を必ずしも満たしていない．例えば論理学では，$\exists x.P(x)$の否定を取れば論理式$\forall x.\neg P(x)$を得ることができる．そこで，もし論理式がプログラムの記述に対応するなら，入力によっては結果が誤りであるプログラムを$\exists x.wrong_result(x)$のように記述し，その否定を取ることによって，たちどころに，常に正しいプログラム記述が得られることになる．しかしいうまでもなく，誤りのあるプログラムから正しいプログラムを構成するような方法は存在しない．

プログラミング言語の基礎を確立するためには，数学や論理学で扱われている対象と同様に抽象的な記述が可能で，かつ実際に計算機で実行可能な計算のモデルを構築する必要がある．今日のプログラミング言語の理論的研究の基礎となっている計算モデルの代表的なものは以下の二つである．

- **関数型計算モデル**
  プログラムを計算可能な関数で表現しようとするモデルである．このモデルでは，関数の値を求めることが計算の実行に対応する．このモデルに基づく言語においては，必要な抽象化の概念および構造化の機構は，

関数の定義機構を通じて実現される．とくに，関数を引き数として受け取る関数や，関数を値として返す関数を使用することにより，種々のプログラム構造を簡潔に表現可能である．

- **論理型計算モデル**
  プログラムを，自動証明可能な論理式で表現しようとするモデルである．このモデルでは，論理式の証明を探索することが計算の実行に対応する．このモデルを基礎とする言語では，必要な抽象化の概念および構造化の機構は，種々の定理や推論規則の定義機構として提供される．

両者とも抽象度が高くかつ数学的な分析に適した概念を基礎にした計算モデルであるが，汎用プログラミング言語のための計算モデルとしては，関数型計算モデルが適している．そこで本書では，関数型計算モデルに基づくプログラミング言語の基礎理論を取り扱う．このモデルは，関数の概念を基本としているが，手続き型プログラミング言語における記憶領域などの概念も表現可能であり，汎用のプログラミング言語の基礎となっている．論理型計算モデルは，解空間の探索を基本とする論理型プログラミングのための計算モデルである．PROLOG[11]等，このモデルに基づくプログラミング言語も開発されているが，これらについては，本書では扱わない．興味ある読者は，文献[31, 10]などの定理の自動証明に関する教科書を参照されたい．

関数型計算モデルを数学的に表現する形式的体系には，**ラムダ計算**（lambda calculus）と**部分帰納的関数**（partial recursive function）の二つがあるが，ラムダ計算のほうがより構文論的な定義となっており，プログラムとの親和性が高い．このため，ラムダ計算が，プログラミング言語の理論的基礎の研究に広く使用されている．ラムダ計算は，型無しラムダ計算と型付きラムダ計算に大別される．型無しラムダ計算は，プログラムの表現する計算そのもののモデルと考えることができる．これに対して，型付きラムダ計算は，型無しラムダ計算に，プログラミングにとって有用な種々の構造を「型」として導入したシステムに相当し，プログラミング言語の数学的モデルと考えることができる．プログラミング言語設計の最も重要な部分は，プログラムの種々の要素間の関係を規定することであり，その多くは型付きラムダ計算の型システムを通じて調べることができる．本書の主題は，型付き

ラムダ計算，特にその型システムの原理とその諸性質を詳しく調べ，それを通じてプログラミング言語の原理を学ぶことである．本章ではまずその準備として，本書で使用する数学的な記法を定義した後，形式言語の帰納的な定義と関数の再帰的定義の原理，および計算モデルとしての型無しラムダ計算を概説する．

1.2　本書で使用する集合に関する記法

A と B を任意の集合とする．A と B の差集合 $\{x|x \in A, x \notin B\}$ を $A \setminus B$ と書く．集合 $A_1, \ldots, A_n$ の直積 $A_1 \times \cdots \times A_n$ を以下のように定義する．

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n)|a_i \in A_i \ (1 \leq i \leq n)\}$$

また，$A_1 = \cdots = A_n$ のとき，$A_1 \times \cdots \times A_n$ を A^n と書き，A の n 次の直積と呼ぶ．A の要素の有限列の集合を A^* で表わす．ここで A^* は常に長さ 0 の空列 ϵ を含むと約束する．$a, b \in A^*$ のとき a と b の連結を ab と書く．ϵ は連結操作に関する単位元である．

集合 A と B の（二項）関係 r とは，$A \times B$ の部分集合である．r を A 上の与えられた関係（すなわち A と A の関係）とする．r が関係を表わすとき，$(a, b) \in r$ を $a\ r\ b$ と書くことがある．任意の $a \in A$ に対して $a\ r\ a$ であるとき，r は反射的であるといい，$a\ r\ b$ かつ $b\ r\ c$ となる任意の $a, b, c \in A$ に対して $a\ r\ c$ となるとき，r は推移的であるという．r の推移的（反射的推移的）閉包とは，r を含み推移的（かつ反射的）である最小の関係であり，r^+（r^*）あるいは $\overset{+}{r}$（$\overset{*}{r}$）と書く．

集合 A から集合 B への関数 f は，$f \subseteq A \times B$ でかつ，任意の $a \in A$ に対して $(a, b) \in f$ となる $b \in B$ が存在し，任意の $a \in A$ について，もし $(a, b) \in f$ かつ $(a, c) \in f$ なら $b = c$ となる関係である．f が集合 A から集合 B への関数であるとき，$f \in A \to B$ と書く．また，関数 f の定義域を $dom(f)$ と書く．f が移す a の値を $f(a)$ または単に $f\ a$ と書く．任意の $a \in A$ に対して，$f(a)$ の値が a を含む式 X で表わされるような関数 f を $\lambda a \in A.X$ と書く．例えば，自然数の集合を N とし N 上の加算演算を $+$ で表わすと，与えられた

自然数に1を足す関数は$\lambda a \in N.a+1$と表わすことができる．この記法は，集合論的な関数を表現するためのものであり，後に定義するラムダ式とは関係がない．関数$f \in A \to B$と$g \in B \to C$の合成$\lambda x \in A.g(f(x))$を$g \circ f$と書く．$X \subseteq dom(f)$のとき，集合$\{f(x)|x \in X\}$を$f(X)$と書く．関数fのXへの制限とは，集合$\{(a,b)|(a,b) \in f, a \in X\}$で表わされる関数であり，$f|_X$と書く．さらに，$f|_{dom(f)\setminus\{x\}}$を$f|_{\overline{x}}$と書く．$f$が関数のとき，$f\{x:v\}$で以下の関数$f'$を表わす．

$$dom(f') = dom(f) \cup \{x\} \quad \text{かつ} \quad f'(y) = \begin{cases} f(y) & (x \neq y \text{のとき}) \\ v & (x = y \text{のとき}) \end{cases}$$

$f\{x:v\}$においてxは$dom(f)$の要素であってもなくてもよいことに注意．

1.3　言語の文法構造の定義

プログラミング言語を含めた計算機科学が対象とする集合や形式言語のほとんどは，定義される言語自身に言及する文法（生成規則の集合）によって定義される．また，そのようにして定義された言語の意味やその性質を規定する関数は，やはり，定義される関数自身に言及する規則の集合によって定義される．本書では，前者を言語の**帰納的**（inductive）な定義，後者を関数の**再帰的**（recursive）な定義と呼ぶ．本節では，計算機言語学の基礎をなすこれら二つの定義機構について解説する．

1.3.1　形式言語の帰納的な定義

プログラミング言語は，通常，**BNF 文法**と呼ばれる文法を用いて定義される．この方法では，定義しようとする言語の要素を代表する**メタ変数**と呼ばれる変数を定め，その言語の要素を生成する規則を，この変数を含んだ文法として定義する．例えば，nの次の自然数を表わす式を$succ(n)$，n_1とn_2の加算および乗算を表わす式をそれぞれ$plus(n_1,n_2)$および$times(n_1,n_2)$とすると，自然数の加算と乗算の組み合わせで作られる算術式の全体を表わす言語は，その言語を代表するメタ変数をAとして，以下のBNF文法で与え

られる．

$$A ::= 0 \mid succ(A) \mid plus(A, A) \mid times(A, A)$$

ここで，“ | ”は「または」の意味であり，この文法全体は，以下の生成規則の集合を表現している．

- 0は算術式である．
- Aが算術式なら，$succ(A)$も算術式である．
- A_1, A_2が算術式なら，$plus(A_1, A_2)$も算術式である．
- A_1, A_2が算術式なら，$times(A_1, A_2)$も算術式である．

例えば$times(plus(0, succ(0)), plus(succ(succ(succ(0))), 0))$は算術式である．

BNF文法による言語の定義は，より一般的には，複数のメタ変数を使用して，階層的になされるが，ここでは，上記のようにメタ変数を一つだけ使用する場合について解説する．メタ変数を複数使用する場合への一般化は容易に可能である．

定義する言語の要素を表現するメタ変数をEとするBNF文法

$$E ::= X_1 \mid \cdots \mid X_n$$

を考える．ここで，各文字列$X_1, \ldots, X_n$は，Eを含んでも含まなくてもよい．この文法で定義される言語を$L(E)$とし，各X_iの中のEの出現を$E_1^i, \ldots, E_m^i$とする．X_iの中の$E_1^i, \ldots, E_m^i$を，それぞれ文字列$a_1, \ldots, a_m$で置き換えて得られる文字列を$[a_1/E_1^i, \ldots, a_m/E_m^i](X_i)$と表わすことにする．すると，BNF文法の各式$X_i$は，$L(E)$に関する以下の規則を表現していると理解される．

> もし要素$a_1, \ldots, a_m$が$L(E)$の要素なら，$[a_1/E_1^i, \ldots, a_m/E_m^i](X_i)$も$L(E)$の要素である．

この形をした規則は，その中に$L(E)$自身が含まれているため，集合$L(E)$の性質を定めているに過ぎない．BNF文法が定義する言語は，その文法が表現する規則をすべて満たすものの中の最小のものである．数学的には，文

法が表現する生成規則から生成される帰納的な集合と特徴付けられる．以下，その厳密な定義を与える．

U を与えられた集合とし，F を以下の形の関数の集合とする．

$$f \in U^n \to U$$

このとき n を f **のランク**と呼び，$rank(f)$ と書く．以下，ランクが n である F の要素を $f^{r(n)}$ と書く．集合 $X \subseteq U$ に対して，集合 $\{f^{r(n)}(x_1, \ldots, x_n) | x_i \in X\}$ を $f^{r(n)}(X)$ と書き，集合 $\{f^{r(n)}(x_1, \ldots, x_n) | x_i \in X, f^{r(n)} \in F\}$ を $F(X)$ と書く．$F(X) \subseteq X$ のとき，X は関数集合 F **に関して閉じている**という．$C \subseteq U$ を与えられた定数の集合とする．C の F に関する**帰納的閉包**を，集合 C を含み関数集合 F に関して閉じている最小の集合と定義し，$Ind(C, F)$ と書く．

U の部分集合が関数 F に関して閉じているという性質は，集合属の共通部分を取る操作 $\bigcap$ によっても保存される．すなわち，Δ が F に関して閉じている集合の集合なら，$\bigcap \Delta$ も F に関して閉じている．また U 自身は，定義上，C を含みかつ F に関して閉じているから，$Ind(C, F)$ は確かに存在し，以下のように書ける．

$$Ind(C, F) = \bigcap \{V | V \subseteq U, C \subseteq V, F(V) \subseteq V\}$$

問 1.3.1　Δ が F に関して閉じている集合の集合なら，$\bigcap \Delta$ も F に関して閉じていることを示せ．

さらに，$Ind(C, F)$ をより具体的に生成することが可能である．集合の系列 $X_0, X_1, \ldots$ を以下の漸化式で定義する．

$$\begin{aligned} X_0 &= C \\ X_{i+1} &= F(X_i) \cup X_i \quad (0 \leq i) \end{aligned}$$

$Ind(C, F)$ は以下のように特徴付けられる．

命題 1.3.1　$Ind(C, F) = \bigcup_{0 \leq i} X_i$

証明　まず$\bigcup_{0\leq i} X_i \subseteq Ind(C,F)$を示す．そのために，$X_i \subseteq Ind(C,F)$が任意の$i$について成立することを，$i$に関する帰納法で示す．定義より，$X_0 \subseteq Ind(C,F)$である．$X_i \subseteq Ind(C,F)$と仮定する．$Ind(C,F)$は$F$に関して閉じているから，$F(X_i) \subseteq Ind(C,F)$であり，$X_{i+1} \subseteq Ind(C,F)$である．

次に$Ind(C,F) \subseteq \bigcup_{0\leq i} X_i$を示す．$Ind(C,F)$の定義より，$\bigcup_{0\leq i} X_i$が$F$に関して閉じていることを示せば十分である．$x_1,\ldots,x_n$を$\bigcup_{0\leq i} X_i$の任意の$n$個の要素，$f^{r(n)}$を$F$の任意の要素とする．ある$k$が存在して$x_1,\ldots,x_n \in X_k$である．よって，$f^{r(n)}(x_1,\ldots,x_n) \in X_{k+1} \subseteq \bigcup_{0\leq i} X_i$であり，$\bigcup_{0\leq i} X_i$は$F$に関して閉じている．■

BNF文法で与えられる言語は，文法規則が定める関数集合に関する帰納的閉包として定義できる．Uをすべての文字列の集合とする．与えられたBNF文法の各生成規則X_iに対して，もしX_iがメタ変数Eの出現$x_1,\ldots,x_n$を含めば，以下のように定義される関数$f_{X_i} \in U^n \to U$を対応させる．

$$f_{X_i}(a_1,\ldots,a_n) = [a_1/x_1,\ldots,a_n/x_n](X_i)$$

すると，BNF文法で定義される言語は以下のように定義できる．

$$L(E) = Ind(\{X_i | X_i \text{は} E \text{を含まない}\}, \{f_{X_i} | X_i \text{は} E \text{を含む}\})$$

帰納的閉包$Ind(C,F)$の種々の性質の証明は，多くの場合帰納法によって行われる．$Ind(C,F)$の任意の要素がある性質Pを持つことを示すためには，以下の二つの性質を示せばよい．

1. 任意の$x \in C$が性質Pを持つ．
2. Fの各生成規則は性質Pを保存する．すなわち，各$f^{r(n)} \in F$について，もしUの要素$x_1,\ldots,x_n$がそれぞれ性質Pを持てば，$f^{r(n)}(x_1,\ldots,x_n)$も性質Pを持つ．

この証明方法を，帰納的に定義された集合の要素の**生成に関する帰納法**，あるいは要素の**構造に関する帰納法**と呼ぶ．以下にその簡単な例を示す．

命題 1.3. 2　任意の算術式は同数の右括弧と左括弧を含む.

証明　Aを与えられた算術式とし，Aに含まれる右括弧の個数と左括弧の個数をそれぞれ$RP(A)$，$LP(A)$とする．$RP(A) = LP(A)$であることをAの構造に関する帰納法で示す.

$A = 0$の場合．$RP(0) = 0 = LP(0)$より命題は成立する.

$A = succ(A)$の場合．$RP(succ(A)) = RP(A) + 1$かつ$LP(succ(A)) = LP(A)+1$である．しかるに，帰納法の仮定より，$RP(A) = LP(A)$である．よって$RP(succ(A)) = LP(succ(A))$である.

$A = plus(A_1, A_2)$の場合．$RP(plus(A_1, A_2)) = 1 + RP(A_1) + RP(A_2)$かつ$LP(plus(A_1, A_2)) = 1 + LP(A_1) + LP(A_2)$である．帰納法の仮定より，各$i$について$RP(A_i) = LP(A_i)$である．よって$RP(plus(A_1, A_2)) = LP(plus(A_1, A_2))$である.

$times(A_1, A_2)$の場合．上記と同様である．■

問 1.3. 2 (帰納的閉包に対する帰納原理)　帰納的な証明が有効であることの根拠は，以下の性質に基づく.

> Cを含む$Ind(C, F)$の部分集合YがFに関して閉じていれば，$Y = Ind(C, F)$である.

1. 上記性質を確認せよ.
2. 上記の性質を使い,（帰納的方法を使わずに）命題1.3.2を証明せよ.（ヒント：Aの部分集合で，右括弧と左括弧の数が同数である文字列の集合を考えよ.）
3. 上記性質は，再帰的に定義された集合の性質に関する帰納的な証明の正しさを保証していることを確認せよ.

1.3.2　言語に対する再帰的な関数定義と文法の曖昧さ

帰納的に定義された集合の要素の性質等の定義は，その生成に関して再帰的に行われる．Xを，CのFに関する帰納的閉包とし，Aを与えられた集合とする．XからAへの関数ϕに関する以下のような規則を，XからAへの**関数の再帰的な定義**と呼ぶ.

1. 各定数$c \in C$に対応する値$\overline{c}$を定義する．

$$\phi(c) = \overline{c}$$

2. 各$f^{r(n)} \in F$に対して関数$\overline{f^{r(n)}} \in A^n \to A$を定め，$f^{r(n)}(x_1, \ldots, x_n)$に対応する値を以下の規則によって定める．

$$\phi(f^{r(n)}(x_1, \ldots, x_n)) = \overline{f^{r(n)}}(\phi(x_1), \ldots, \phi(x_n))$$

例えば，算術式Aの中の括弧の数を計算する関数$paren$の再帰的な定義は，各算術式構成子に対応する関数を

$$\begin{aligned}
\overline{0} &= 0 \\
\overline{succ} &= \lambda x \in N.x + 2 \\
\overline{plus} &= \lambda(x, y) \in N \times N.x + y + 2 \\
\overline{times} &= \lambda(x, y) \in N \times N.x + y + 2
\end{aligned}$$

と定め，これらを使って以下のような規則を定義することによって行う．

$$\begin{aligned}
paren(0) &= 0 \\
paren(succ(A)) &= \overline{succ}(paren(A)) \\
paren(plus(A_1, A_2)) &= \overline{plus}(paren(A_1), paren(A_2)) \\
paren(times(A_1, A_2)) &= \overline{times}(paren(A_1), paren(A_2))
\end{aligned}$$

以上のような形の再帰的な規則の集合が，関数を定義しているかどうかは必ずしも自明ではない．実際，言語の文法によっては，再帰的な規則のみでは関数が定義できない場合がある．その例として，以下の文法を考えてみよう．

$$E ::= 0 \mid succ(E) \mid E\ plus\ E \mid E\ times\ E$$

この言語の各要素 E に自然数を対応させる関数 $\phi(E)$ を以下のように再帰的に定義しようとする．

$$\begin{aligned}\phi(0) &= 0\\ \phi(succ(E)) &= 1+\phi(E)\\ \phi(E_1\ plus\ E_2) &= \phi(E_1)+\phi(E_2)\\ \phi(E_1\ times\ E_2) &= \phi(E_1)\times\phi(E_2)\end{aligned}$$

しかし上式によれば，例えば

$$E_0 = succ(0)\ plus\ succ(succ(0))\ times\ succ(succ(0))$$

に対する値が以下のように二つ存在してしまう．

$$\begin{aligned}\phi(E_0) &= \phi(succ(0))+\phi(succ(succ(0))\ times\ succ(succ(0)))\\ &= 5\\ \phi(E_0) &= \phi(succ(0)\ plus\ succ(succ(0)))\times\phi(succ(succ(0)))\\ &= 6\end{aligned}$$

問題の原因は，言語を生成する文法の構造にある．上記の文法は，異なった文法規則の適用の結果が同一の文と成りうるような文法である．そのような文法は**曖昧**であるという．上記の再帰的な関数定義を注意深く見れば明らかなように，再帰的な関数定義は，帰納的に生成された集合の要素に対する定義ではなく，集合の要素の生成系列に対する定義となっている．したがって，再帰的な関数定義が常に意味を持つためには，生成系列の集合と一対一の関係にあるような言語，すなわち，曖昧性のない文法で生成された言語でなければならない．曖昧性のない言語の条件は，帰納的閉包 $X = Ind(C,F)$ に関する条件として，一般的に，以下のように与えられる．

1. 任意の相異なる $f^{r(n)}, g^{r(m)} \in F$ に対して，$f^{r(n)}(X) \cap g^{r(m)}(X) = \emptyset$ である．
2. 任意の $f^{r(n)} \in F$ について，$f^{r(n)}(X) \cap C = \emptyset$ である．
3. 任意の $f^{r(n)} \in F$ について，$f^{r(n)}|_{X^n}$ は単射関数である．

帰納的に生成された集合がこの性質を満たすとき，**自由に生成された集合**と呼ぶ．自由に生成された集合は，生成規則の適用の系列の集合と一対一に対応することを確かめることができる．上に定義した算術式の集合$L(E)$は上記の性質を満たさないので，自由に生成された集合ではない．例えば，$E\ plus\ E$および$E\ times\ E$を生成する規則に対応する関数をそれぞれf_{plus}およびf_{times}とすると，$E_0 \in f_{plus}(L(E), L(E)) \cap f_{times}(L(E), L(E))$となり，1. の条件を満たさない．上の例で見たように，このような集合に対しては，関数の再帰的な定義は，必ずしも唯一の関数を定義するとは限らない．しかし，自由に生成された集合に対しては，関数の再帰的な定義は，必ず唯一の関数を定義する．

定理 1.3. 1　Aを任意の集合とし，Xを，定数の集合Cおよび関数の集合Fによって自由に生成された集合とする．各定数cに対してAの要素$\bar{c}$が与えられ，各関数$f^{r(n)}$に対して関数$\overline{f^{r(n)}} \in A^n \to A$が与えられたとき，以下の条件を満たす関数$\phi \in X \to A$が唯一つ存在する．

1. 任意の定数cに対して

$$\phi(c) = \bar{c}$$

2. Xの要素$f^{r(n)}(x_1, \ldots, x_n)$について，

$$\phi(f^{r(n)}(x_1, \ldots, x_n)) = \overline{f^{r(n)}}(\phi(x_1), \ldots, \phi(x_n))$$

この関数ϕを，関数$\lambda c \in C.\bar{c}$の$\overline{f^{r(n)}}$に関する唯一の準同型拡張と呼ぶ．

証明　以前定義したXの生成系列X_iを使用して証明する．

以下の等式によって定義される関数の系列ϕ_iを考える．

$$\phi_0(c) = \bar{c}$$

$$\phi_{i+1}(x) = \begin{cases} \phi_i(x) & (x \in X_i) \\ \overline{f^{r(n)}}(\phi_i(x_1), \ldots, \phi_i(x_n)) & (x = f^{r(n)}(x_1, \ldots, x_n), x \notin X_i) \end{cases}$$

X_{i+1}の定義および自由に生成された集合の条件より，任意の$i > 0$，任意の$f^{r(n)}$について，$f(X_i^n \setminus X_{i-1}^n) \cap X_i = \emptyset$が成立することを示すことができ

る．ただし，$X_{-1} = \emptyset$とする．よって$x \in X_{i+1}$なら，$x \in X_i$か，またはある$f^{r(n)}$および$(x_1, \ldots, x_n) \in X_i^n \setminus X_{i-1}^n$があって$x = f^{r(n)}(x_1, \ldots, x_n)$と一意に書ける．この性質を使えば，$\bigcup_{0 \leq i} \phi_i$が$\bigcup_{0 \leq i} X_i$上の条件を満たす関数を定義していることを示すことができる．

またもし，条件を満たす関数が存在すれば，任意のiについて，その関数をX_iに制限したものは，$\bigcup_{0 \leq i} \phi_i$を$X_i$に制限したものと一致することを示すことができる．よって，条件を満たす関数が存在すれば，それは$\bigcup_{0 \leq i} \phi_i$に等しい．■

問 1.3.3 Xを定数集合Cと関数集合Fから自由に生成された集合，$X_0, \ldots, X_i, \ldots$を以前定義したXの生成系列とする．

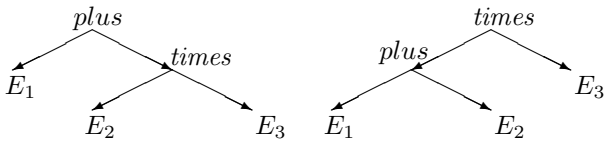
1. 任意の関数$f^{r(n)} \in F$，任意の$0 \leq i$について$f^{r(n)}(X_i^n \setminus X_{i-1}^n) \cap X_i = \emptyset$が成立することを示せ．ただし$X_{-1} = \emptyset$とする．この性質を使い，定理1.3.1の証明で定義した関数$\bigcup_{0 \leq i} \phi_i$が$\bigcup_{0 \leq i} X_i$上の条件を満たす関数であることを示せ．
2. 唯一の準同型拡張ϕをX_iに制限したものは，定理1.3.1の証明で定義した関数ϕ_iと一致することをiに関する帰納法で示せ．

XをCとFによって自由に生成された集合とする．上記の定理より，Xの要素xの大きさ$|x|$を，xを生成するために使われた生成規則の適用回数と定義できる．したがって，自由に生成された集合に対しては，要素の大きさに関する帰納法を用いることができる．

BNF文法で定義された任意の言語も，文法によって生成される文字列の集合と捉えるのではなく，文字列の生成に使われる文法規則の系列そのものと解釈すれば，自由に生成された集合となる．この見方に従えば，例えば前に挙げた文法

$$E ::= 0 \mid succ(E) \mid E\ plus\ E \mid E\ times\ E$$

で生成される言語の要素は$E_1\ plus\ E_2\ times\ E_3$のような文字列ではなく，以下のような木となる．



文の生成構造を表わすこのような木を，**構文木**と呼ぶ．本書では，プログラミング言語を，この生成規則の適用の系列を表現している構文木の集合とする．この見方に従えば，任意のBNF文法は，自由に生成された集合の定義と見なすことができ，したがって定理1.3.1によって，言語の文法に関する再帰的な規則は，必ず唯一の関数を定義することが保証される．この見方は，プログラミング言語の理論的な研究のみならず，プログラミング言語の処理系の構築などにおいても採用されている．すなわち，プログラミング言語の処理系は，まず，与えられた文字列から，その文字列の生成規則の系列に対応する構文木を生成し，この構文木を処理の対象とする．

文字列から構文木を生成する処理は，プログラミング言語の**構文解析**と呼ばれる．構文解析もプログラミング言語の重要な一部であるが，それに対してはLR構文解析を代表とする確立した技術が存在し，また数多くの良書が出版されているので，本書では，構文解析は扱わず，構文解析はすでに済んでおり，プログラムとして構文木そのものが与えられているものと見なす．オートマトンや文脈自由文法の理論を含む形式言語理論の基礎を学習した者は，以下の問を試みることによって，LR構文解析のおよそを理解することができるであろう．

問 1.3.4（LR構文解析の原理）　現在使われているプログラミング言語の構文解析法の基礎は，Knuth[30]によって提案されたLR構文解析の理論である．この理論に基づき，与えられた文法から，実際にその文法に従って構文解析を行うプログラムを自動生成するシステムが実用化されている．本問は，LR構文解析の原理に関するものであり，形式言語理論の基礎知識を必要とする．

BNF文法は，形式言語理論における文脈自由文法に対応する．文脈自由文法の特殊な場合である正規言語は，有限オートマトンによって効率よく構文解析を行うことができるが，一般の文脈自由文法の構文解析を行う効率よいアルゴリズムは知ら

れていない．KnuthによるLR構文解析法の中心をなすアイデアは，オートマトンによる正規文法の構文解析法を繰り返し使うことによって，より広範囲の文脈自由文法の構文解析を効率よく行う，というものである．

$G = (V, T, P, S)$ を与えられた文脈自由文法とする．ここで V は非終端記号の集合，T は終端記号の集合，P は文法規則の集合，S はスタート記号である．$\alpha \in (V \cup U)^*$ の中の非終端記号に文法規則を適用し β に変換することを，α から β の導出という．特に，α の中の最右端の非終端記号を置き換える導出を最右導出と呼ぶ．S から導出できる $V \cup T$ のシンボル列を文形式と呼び，さらにその導出がすべて最右導出であるとき右文形式と呼ぶ．本問では，導出を最右導出に限ることにし，α から β が最右導出されることを $\alpha \Longrightarrow \beta$ と書く．

文字列 α に対してある $A \in T, \beta \in (V \cup T)^*, w \in T^*$ があって，

$$S \overset{*}{\Longrightarrow} \beta A w \Longrightarrow \beta \alpha w$$

の形の導出が存在するとき，α を文形式 $\beta\alpha w$ のハンドルと呼ぶ．文形式の先頭から始まり，あるハンドルの終わり以内に収まる部分文字列を，その文形式の活性文字列（viable prefix）と呼ぶ．与えられた文法 G の活性文字列の集合 $VP(G)$ は以下のように与えられる．

$$VP(G) = \{\alpha | S \overset{*}{\Longrightarrow} \beta A w \Longrightarrow \beta\gamma w, \alpha \text{ は } \beta\gamma \text{ の先頭から始まる部分文字列}\}$$

LR構文解析は，以下の性質に基づく．

命題 1.3.3 (Knuth)　任意の文脈自由文法 G に対して，$VP(G)$ は正規言語である．

活性文字列受理オートマトン $M = (Q, \Sigma, \delta, q_0, F)$ を，以下のように定義できる．

- $\Sigma = V \cup T$,
- $Q = \{A \to \alpha \cdot \beta | A \to \alpha\beta \in P\} \cup \{S' \to \cdot S\}$
  ここで S' は V にない新しい記号,
- $F = Q$.

状態遷移関数 δ を定義し，上記性質を証明せよ．上記のオートマトンを，ハンドルを右端に含む極大活性文字列のみを受理するように変更せよ．

LR構文解析は，活性文字列受理オートマトンにより，与えられた文形式 $\alpha = \beta\gamma\delta$ の極大の活性文字列 $\beta\gamma$ を検出し，その末尾にあるハンドル γ を，対応する生成規則 $A \Longrightarrow \gamma$ の左辺 A で置き換え，α を $\beta A \delta$ に変換する操作を繰り返すことによって行う．ただし，次回以降の活性文字列探索時も，文形式の A より前の部分 β は共通であるから，この部分のオートマトンの状態遷移をスタックに記憶しておき，オートマトンが同一文字列を繰り返し辿る無駄を省くことができる．この技術により，LR

構文解析は，実用上ほぼオートマトンによる正規言語の構文解析と同程度の効率で構文解析を実行できるのである．

LR 構文解析は，与えられた文の最右導出を逆に辿ることに相当する．与えられた文字列の構文木を決定する導出の系列は，上記の構文解析における文字列の置き換えで使われた文法規則の系列によって決定される．

以下，メタ変数を複数含む階層的な BNF 文法を使用する．とくに，既存の集合は，特定のメタ変数定数で表わす．例えば，自然数を既存の集合とし，その要素をメタ変数を n で表わせば，算術式の集合を表わす BNF 文法は，

$$A ::= n \mid plus(A, A) \mid times(A, A)$$

と書ける．

1.4　型無しラムダ計算

本節では，プログラミング言語が表現する計算のモデルである型無しラムダ計算の概要，および，そのプログラミング言語との関係を概説する．型無しラムダ計算についての詳しい性質は，文献 [1, 62, 60] などのラムダ計算に基づく計算論の教科書を参照されたい．

1.4.1　型無しラムダ計算の定義

Var を可算無限個の変数の集合とし，メタ変数 x で代表する．型無しラムダ計算のラムダ式の集合は，以下の BNF 文法で与えられる．

$$M ::= x \mid (\lambda x.M) \mid (M\ M)$$

$\lambda x.M$ は**ラムダ抽象**（lambda abstraction）と呼ばれ，直観的には，x を受け取り，（一般に x を含む）式 M の値を計算する名前の無い関数を表わす．$M_1\ M_2$ は関数 M_1 を実引き数 M_2 に適用する**ラムダ適用**（lambda application）を表わす．

以下はラムダ式の例である．

- $(x\ y)$

- $(\lambda x.(\lambda y.x))$
- $(\lambda x.(\lambda y.(\lambda z.((x\ z)(y\ z)))))$
- $((\lambda x.(x\ x))\ (\lambda x.(x\ x)))$
- $(\lambda x.((x\ y)(\lambda y.y)))$

ラムダ式を表記する際，以下の約束に従って括弧をできる限り省略する．

1. $\lambda x.M$ における M はできる限り大きく取る．
2. 関数適用は左結合する．

この約束に従い，$(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M)\ldots))$ は $\lambda x_1.\lambda x_2 \ldots \lambda x_n.M$ と書き，$(\cdots(M_1\ M_2)\ \cdots\ M_n)$ は $M_1\ M_2\ \cdots\ M_n$ と書く．

問 1.4.1　上記の例の５番目のラムダ式は，$\lambda x.x\ y\ \lambda y.y$ と書ける．他の例のラムダ式についても括弧を省略して表記せよ．

ラムダ抽象 $\lambda x.M$ における x は，関数の仮引き数を表わすにすぎず，x の名前自身は重要ではない．したがって例えば，$\lambda x.x$ は $\lambda y.y$ と同じ意味である．このような変数を**束縛変数**と呼ぶ．束縛変数は，種々の取りうる値とその結果の依存関係を示すために導入された，仮の名前である．例えば，論理式 $\forall x.P(x)$ における x や，積分式 $\int f(x)dx$ における x も同様の働きをする束縛変数である．束縛されていない変数を**自由変数**と呼ぶ．自由変数は，文脈によって決まる特定の値を表わす変数であり，勝手に名前を替えることは許されない．ラムダ式 M に含まれる自由変数の集合 $FV(M)$ は，式の生成に関して再帰的に以下のように定義される．

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(M_1\ M_2) &= FV(M_1) \cup FV(M_2) \\
FV(\lambda x.M) &= FV(M) \setminus \{x\}
\end{aligned}
$$

ラムダ式に対する最も基本的な操作は，ラムダ式の変数への代入，すなわちラムダ式に現われる自由変数を，与えられた別のラムダ式で置き換える操作である．ラムダ式 M の中の自由変数 x をすべて N で置き換えて得られるラムダ式を，$[N/x]M$ と書く．$[N/x]M$ の定義は，M がラムダ抽象以外の場

合はほぼ自明であるが，M がラムダ抽象の場合は注意が必要である．例えば，以下の置き換えを考えてみよう．

$$[(y\ z)/x](\lambda y.x\ y)$$

$\lambda y.x\ y$ の中の x を単純に $(y\ z)$ で置き換えると $\lambda y.(y\ z)\ y$ となる．この置き換えの結果，$(y\ z)$ の中の自由変数 y が束縛変数に変わってしまい，y の意味が変わってしまっている．この現象を**自由変数の捕捉**と呼ぶ．これを防ぐために，自由変数が捕捉される恐れがある場合は，その原因となっている束縛変数の名前の付け替えを行う必要がある．上記の例では，変数 y の捕捉を防ぐために，今までに使用されていない名前 w を選び，$\lambda y.x\ y$ の束縛変数 y の名前を付け替え，$\lambda w.x\ w$ とした後，x の置き換えを行い，$\lambda w.(y\ z)\ w$ を得る．束縛変数は，仮引き数であることを示す仮の名前であり，名前を付け替えても意味が変わらないから，以上の処理によって，式の意味を変えずに，自由変数の捕捉を防ぐことができる．

以上の処理を含んだ $[N/x]M$ の定義は以下のように与えられる．

$$\begin{aligned}
[N/x]x &= N \\
[N/x]y &= y \quad (x \neq y) \\
[N/x](M_1\ M_2) &= ([N/x]M_1\ [N/x]M_2) \\
[N/x](\lambda y.M) &= \begin{cases}
\lambda x.M & (x = y) \\
\lambda y.[N/x]M & (x \neq y, y \notin FV(N)) \\
\lambda z.[N/x]([z/y]M) & (x \neq y, y \in FV(N)\text{かつ} \\
 & \quad z \notin FV(M), z \notin FV(N))
\end{cases}
\end{aligned}$$

ラムダ抽象の最後のケースで導入される z は，自由変数の捕捉を防ぐために導入された新しい束縛変数である．新しい束縛変数は，任意の，しかし一定の仕方で選ぶものとする．変数は可算無限個存在するから，必要な変数 z は常に存在し，ラムダ式の代入は，任意のラムダ式に対して定義される．

代入の定義を使って，束縛変数の名前の付け替えによって得られるラムダ

式間の関係を以下のように定義する．

(α)　　$\lambda x.M = \lambda y.[y/x]M \quad (y \notin FV(M))$

この規則によって生成される同値関係をα**同値関係**と呼ぶ．ラムダ式は，この同値関係の下で考える．以降$M = N$と書いた場合，MとNは，α同値関係によって等しいことを表わすものとする．すなわち，$M = N$は，Nが，規則(α)をMの一部に0回以上適用して得られることを表わす．

ラムダ式に対する基本的な操作は，以下のβ**簡約公理**で表わされる．

(β)　　$(\lambda x.M)\ N \Longrightarrow [N/x]M$

ラムダ式Mの一部に上記の規則を適用してラムダ式Nが得られるとき，MはNに**１ステップで簡約**されるといい，

$$M \longrightarrow N$$

と書くことにする．ラムダ計算の表現する計算の基本である（β）**簡約関係**は，この１ステップ簡約関係の反射的推移的閉包$M \xrightarrow{*} N$である．この関係は，Mに0回以上の１ステップ簡約を施してNが得られることを表わす．$M \longrightarrow N$となるNが存在しないとき，Mは**正規形**（normal form）であるという．$M \xrightarrow{*} N$でかつNが正規形であるとき，$M \Downarrow N$と書く．さらに，与えられたMに対して$M \Downarrow N$となるNが存在するとき，$M \Downarrow$と書く．正規形のラムダ式は，これ以上計算する余地のない式であり，プログラムの最終結果の表現と見なすことができる．

関係$M \Downarrow N$が，ラムダ計算におけるプログラムの行う計算のモデルである．以下の定理が，この見方の妥当性を保証している．

命題 1.4.1　任意のラムダ式Mについて，$M \Downarrow N_1$ かつ$M \Downarrow N_2$なら，$N_1 = N_2$である．

この命題は，以下の定理の直接の帰結である．

定理 1.4.1 (合流性)　任意のラムダ式Mについて，もし$M \xrightarrow{*} N_1$かつ$M \xrightarrow{*} N_2$なら，$N_1 \xrightarrow{*} N_3$，$N_2 \xrightarrow{*} N_4$かつ$N_3 = N_4$となるN_3, N_4が存

在する．

この定理の証明には，種々の準備が必要であり，ここでは省略する．興味のある読者は，前に挙げたラムダ計算に関する教科書を参照．なかでも文献[62]で与えられている証明が，最も簡潔で分かりやすい証明と思われる．

問 1.4. 2　命題1.4.1が定理1.4.1の直接の帰結であることを確かめよ．

1.4.2　汎用な計算モデルとしての型無しラムダ計算

ラムダ計算は，以上のような簡単な構造を持つシステムではあるが，その表現力は極めて高く，汎用の計算のモデルとして十分に強力なシステムである．ラムダ計算の表現力に関する厳密な分析は，計算モデルに関する他の著書に譲り，ここでは，ラムダ計算の表現力を具体的に示すにとどめる．

まえがきで述べたように，情報処理システムの基本は，人間が情報を言葉で表現するように，情報を，有限なシンボル列（すなわち形式言語における文）で表現し，その有限なシンボル表現を処理することである．情報を有限なシンボル列で表現することを**コード化**といい，表現されたものをコードという．ラムダ計算が，プログラムの表現する計算のモデルたりうるためには，プログラムが使用するすべての情報を，ラムダ式にコード化可能でなければならない．以下，種々の代表的なデータ構造がラムダ式にコード化可能であることを示す．基本的な考え方は，データ構造が表現すべき「ふるまい」そのものを関数として表現することである．

論理値

論理値は真（$true$）と偽（$false$）の二つの異なる値である．プログラミング言語におけるそのふるまいは真偽の判定であり，以下のような関数定義によって特徴付けられる．

$$F(b) = \begin{cases} x & (b\text{が}true\text{のとき}) \\ y & (b\text{が}false\text{のとき}) \end{cases}$$

論理値bは，上記のようなふるまいそのものと考えることにより，任意のxとyが与えられると$F(b)$の値を返すような関数として表現可能である．こ

の考えに従い，論理値bの型無しラムダ計算におけるコード$\overline{b}$を

$$\overline{true} = \lambda x.\lambda y.x$$
$$\overline{false} = \lambda x.\lambda y.y$$

と定義することができる．条件判定式は以下のように表現できる．

$$Cond = \lambda b\lambda A.\lambda B.b\ A\ B$$

このコード化は，任意のAとBについて

$$Cond\ \overline{true}\ A\ B \xrightarrow{*} A$$
$$Cond\ \overline{false}\ A\ B \xrightarrow{*} B$$

を満たすから，確かに論理値としてふるまうことが確認できる．

問 1.4.3 抽象的には，論理値は単に二つの要素からなる集合である．論理値のコード化を一般化し，n個の要素の集合$X = \{x_1, \ldots, x_n\}$の各要素，および，Xの要素xの種類を判定し，もしx_iなら処理A_iに分岐する演算$Switch$を表わすラムダ式を定義せよ．

自然数

自然数は，抽象的には,「零」と「次の自然数」という概念から生成されるものであり，プログラミング言語におけるそのふるまいは，以下のような自然数上の関数定義によって特徴付けられる．

$$F(n) = \begin{cases} z & (n\text{が零のとき}) \\ f(F(n_0)) & (n\text{が}n_0\text{の次の自然数のとき}) \end{cases}$$

ここでfとzは，関数Fを特徴付ける与えられた関数および定数である．自然数を上記のようなふるまいそのものと考えると，自然数nは，任意のfとzが与えられると$F(n)$の値を返すような関数として表現可能である．この考えに従い，自然数nの型無しラムダ計算におけるコード$\overline{n}$を

$$\overline{0} = \lambda f.\lambda z.z$$
$$\overline{1} = \lambda f.\lambda z.f\ z$$

$$\overline{2} = \lambda f.\lambda z.f(f\ z)$$
$$\vdots$$
$$\overline{n} = \lambda f.\lambda z.f^n\ z$$
$$\vdots$$

と定義することができる．ここで，$f^n\ z$ は，$\underbrace{(f(f\cdots(f}_{n\text{ 個の }f}\ z)\cdots))$ の略記法とする．

このコード化によって，自然数を使ったプログラムをラムダ計算で表現できることが確かめられる．例えば，次の自然数を求める関数 $Succ$ は，コードの構造を考えれば

$$Succ = \lambda n.\lambda f.\lambda z.f(n\ f\ z)$$

と定義できる．

問 1.4.4　$Succ\ \overline{n} \Downarrow \overline{n+1}$ であることを確かめよ．

与えられた自然数 m に n を加える演算 f_n は

$$f_n(m) = \begin{cases} n & (m\text{ が零のとき}) \\ f_n(m_0)+1 & (m\text{ が }m_0\text{ の次の自然数のとき}) \end{cases}$$

と表わせることを考えると，加法演算は以下のように表現できることがわかる．

$$Add = \lambda m.\lambda n.m\ Succ\ n$$

同様にして $m \times n$，n^m をそれぞれ計算する関数 Mul，Exp および与えられた自然数が零か否かをテストする関数 $IsZero$ も，その意味を考えることによって，以下のように定義できる．

$$Mul = \lambda m.\lambda n.m\ (Add\ n)\ \overline{0}$$
$$Exp = \lambda m.\lambda n.m\ (Mul\ n)\ \overline{1}$$
$$IsZero = \lambda n.n\ (\lambda x.\overline{false})\ \overline{true}$$

問 1.4.5 任意の m, n について

$$\begin{aligned} Add\ \overline{m}\ \overline{n} &\Downarrow \overline{m+n} \\ Mul\ \overline{m}\ \overline{n} &\Downarrow \overline{m \times n} \\ Exp\ \overline{m}\ \overline{n} &\Downarrow \overline{n^m} \\ IsZero\ \overline{0} &\Downarrow \overline{true} \\ IsZero\ \overline{m} &\Downarrow \overline{false}\ \ (m > 0) \end{aligned}$$

となることを示せ.

少々複雑になるが，減算 *Sub* や他の自然数演算も定義可能であり，さらに後に説明する再帰的関数の定義機構と組み合わせれば，自然数上の計算可能なすべての関数を，ラムダ式で表現できることが示せる.

問 1.4.6 上で与えた *Exp* の定義は，*Mul* を使って定義されている．この定義は理解しやすいが，計算の効率を考えると，最適な定義とはいえない．自然数のコードの構造を考えれば，n^m を計算するラムダ式は，$m \geq 1$ に対しては，*Exp* 以外にも，より効率のよい以下のコードが存在する.

$$Exp2 = \lambda m.\lambda n.m\ n$$

1. $m \geq 1$ なら，$Exp2\ \overline{m}\ \overline{n} \Downarrow \overline{n^m}$ となることを示せ.
2. 実際に $Exp\ \overline{2}\ \overline{3}$ および $Exp2\ \overline{2}\ \overline{3}$ を計算し，両者を比較せよ.
3. *Exp2* の例を参考にして，可算，乗算関数 *Add2*, *Mul2* を，それぞれ *Succ*, *Add* を使用せずに直接定義し，上記同様に *Add*, *Mul* との比較を行え.

n 個の要素の組

プログラムでよく使用される n 個の要素からなる組は，各 i 番目の要素を取り出す射影操作 $Proj_i^n$ が定義されたデータ構造である．したがってそのふるまいに注目すれば，組は，射影関数 $Proj_i^n$ を受け取り，組の中の i 番目の要素を返す関数と考えることができる．この考え方に従い，組構成関数 $Prod^n$ および射影関数 $Proj_i^n$ を，以下のようにコード化可能である.

$$\begin{aligned} Prod^n &= \lambda x_1 \ldots . \lambda x_n . \lambda P.P\ x_1\ \cdots\ x_n \\ Proj_i^n &= \lambda x_1 \ldots . \lambda x_n . x_i \end{aligned}$$

このコード化が組の表現になっていることは，以下の性質により確認できる．

$$(Prod^n\ M_1\ \cdots\ M_n)\ Proj_i^n \stackrel{*}{\longrightarrow} M_i$$

再帰的関数定義

再帰的な関数定義は，複雑なプログラムを書く上で必須の機構である．再帰的関数定義の文法を$fun\ f\ x = M$と書くことにすると，例えば自然数の階乗を求める関数は通常，再帰的に以下のように定義される．

$$fun\ \ fact\ \ n\ \ =\ \ if\ \ iszero(n)\ \ then\ \ 1\ \ else\ \ n * (fact\ \ (n-1))$$

この定義は，定義しようとしている関数$fact$が関数本体の中に現われているから，再帰的な定義である．もし$fact$がすでに定義されていると仮定すれば，この定義の右辺は，これまでに定義したラムダ式へのコード化を用いて，以下のラムダ式で表現可能である．

$$Cond\ (IsZero\ n)\ \overline{1}\ (Mul\ n\ (fact\ (Sub\ n\ \overline{1})))$$

$fact$は，この式の仮引き数nをラムダ抽象して得られる関数と等しいはずである．以下，本項での説明のために，二つの型無しラムダ式NとMの意味が等しいことを$M =_\beta N$と書くことにする．($=_\beta$は，以降の本書の議論で使用することは無いので，その厳密な定義は省略する.）$fact$を，ラムダ式を代表するメタ変数と考えると，$fact$の再帰的定義は，以下の等式を満たすラムダ式と解釈できる．

$$fact =_\beta \lambda n.Cond\ (IsZero\ n)\ \overline{1}\ (Mul\ n\ (fact\ (Sub\ n\ \overline{1})))$$

以上の洞察から，再帰的な関数定義$fun\ f\ x\ =\ M$は，変数fを含むラムダ式$\lambda x.M$が与えられたとき，fを未知数とする等式$f\ =_\beta\ \lambda x.M$を満たすラムダ式Nを求めるような操作と解釈できる．$F = \lambda f.\lambda x.M$とおくと，$F\ N \longrightarrow [N/f]M$であるから$F\ N$と$[N/f]M$は同一の意味を持つと見なせる．したがって，求めるラムダ式Nは，

$$F\ N =_\beta N$$

を満たすラムダ式と考えることができる．N は F を作用させても変化しない式であるから，F の**不動点**と呼ばれる．すると，与えられたラムダ式 F から F の不動点を計算する演算が，それ自身ラムダ式として定義できれば，任意の再帰的関数がラムダ式として定義可能となるはずである．そのような演算子を**不動点演算子**と呼ぶ．Y が不動点演算子であれば，$(Y\ (\lambda f.\lambda x.M))$ は

$$\begin{aligned} Y\ (\lambda f.\lambda x.M)) &=_\beta (\lambda f.\lambda x.M)(Y\ (\lambda f.\lambda x.M)) \\ &=_\beta [(Y\ (\lambda f.\lambda x.M))/f]\lambda x.M \end{aligned}$$

を満たし，したがって，再帰的関数定義が表現する等式

$$f =_\beta \lambda x.M$$

の解の条件を満たす．実際に，不動点演算子が幾つか知られている．以下は Turing の不動点演算子と呼ばれるものである．

$$Y_{Turing} = (\lambda z.\lambda x.x\ (z\ z\ x))\ (\lambda z.\lambda x.x\ (z\ z\ x))$$

問 1.4.7　ラムダ計算における関数は，グラフとしての関数ではなく，入力から出力を計算する手続きの表現である．この観点から，再帰的関数定義 $fun\ f\ x = M$ は，等式

$$f =_\beta \lambda x.M$$

ではなく，関数定義の本体に現われる関数名 f を，その定義で置き換える以下の書き換え規則と考えるのがより妥当である．

$$f \longrightarrow \lambda x.M$$

Y が

$$Y\ M \longrightarrow M(Y\ M)$$

を満たせば，上記の書き換え規則を実現することを確かめよ．

Y_{Turing} は，任意の M に対して $Y_{Turing}\ M \longrightarrow M(Y_{Turing}\ M)$ を満たすことを示せ．

この演算子の存在により，再帰的な関数定義は，ラムダ計算の中で定義可能であることが示される．例えば，再帰的に定義された関数 $fact$ は以下の

ラムダ式で表現できる．

$$fact = Y_{Turing}\ (\lambda fact.\lambda n.Cond\ (IsZero\ n)\ \overline{1}\ (Mul\ n\ (fact\ (Sub\ n\ \overline{1}))))$$

問 1.4.8　$fact\ \overline{3} \Downarrow \overline{6}$となることを確かめよ．

以上のような表現力を持つラムダ計算は，チューリング機械や部分帰納的関数と同等の表現力を有していることが証明されており，およそ有限の記述が可能で機械的に実行できる計算をすべて表現することができると見なされている．さらにラムダ計算は具体的な式を扱うシステムであり，プログラミング言語との親和性も高く，プログラミング言語の原理を分析するための基礎として最も適したモデルといえる．

1.4.3　ラムダ計算に基づくプログラミング言語のモデル

ラムダ計算は，以上見てきたように十分に強力な表現力を持つが，前節で定義した型無しラムダ計算は，プログラミング言語の持つべきプログラムの記述システムとしては必ずしもふさわしいものとはいえない．例えば，前節では，ラムダ式$\lambda x.\lambda y.y$が,（1）自然数の零,（2）論理値$false$,（3）二つの要素からなる組からその二番目の要素を取り出す演算の，三つのまったく異なったもののコードとして使用されることを示した．このように，ラムダ式のみでは，それが表現する機能は不明確であり，大規模なプログラムに対応する長大なラムダ式の，プログラムとしての意味を読み取ることはほとんど不可能といってよい．これは，ちょうど，von Neumann計算機のメモリ上の膨大なビット列として表現されたプログラムが，何を意味しているか読み取れないことに類似する．

ソフトウェア記述システムとしてのプログラミング言語は，単なる実行可能なコードの記述にとどまらず，コードが表現する計算の意図する性質やその構造をも表現するものでなければならない．そのようなプログラミング言語の数学的モデルにふさわしいシステムは，以上説明してきた（型無し）ラムダ計算に，**型**の概念を導入した**型付きラムダ計算**である．ラムダ式の型とはラムダ式が表現する計算の性質の記述である．ラムダ式に型を与えることにより，ラムダ式の使い方に制約が加えられ，ラムダ式で書かれたプログ

ラムの意味およびその構造が明確になり，大規模なシステムを安全にかつ効率よく記述することが可能になる．本書の目的は，種々の型付きラムダ計算の性質の分析を通じて，プログラミング言語の基本原理を学習し，プログラミング言語の設計や分析の基礎を習得することである．

最も基本的な型の概念は，整数集合や文字列集合等の同一の性質を持つ値の集合である．これらの型を**基底型** と呼ぶ．プログラミング言語の基本的なモデルは，これら基底型を関数計算系であるラムダ計算に埋め込んだシステムである．計算のどのような性質を型として導入するかによって，種々の表現力の違う型付きラムダ計算が存在する．最も単純なシステムは，基底型と高階の（すなわち値として扱うことの可能な）関数の型を含んだ，単純な型付きラムダ計算である．第2 章では，単純な型付きラムダ計算を定義し，その種々の性質を詳しく分析する．この基本的なシステムに，種々のデータを定義し利用するための再帰的データ型や，汎用性のあるプログラムを定義可能にする多相型，型の自動推論機構等を導入することによって，高水準プログラミング言語のモデルを作ることができる．第3 章では，型付きラムダ計算の基本的な枠組みの中で直接定義可能な以下の機能を解説する．

1. レコード型やバリアント型などの構造を持つデータ構成子.
2. リストや木構造等を定義するための再帰的データ型.
3. 再帰的関数定義.
4. ユーザ定義のデータ型とパターンマッチング.
5. 参照型と継続計算.

以上の拡張によって得られるプログラミング言語のモデルは，高階の関数やユーザ定義のデータ型，継続計算等の高度な機能を含んでいるものの，型付けの基本的な戦略は，PASCAL などの伝統的なプログラミング言語の型システムとほぼ同一である．これら伝統的な型付きプログラミング言語は，静的な型チェックがなされるためプログラムの信頼性が高いものの，LISP や Smalltalk などの型無し言語に比べると，多くの型宣言を必要とするためプログラミングが面倒であり，また型の制約のため汎用の手続きが書けず柔軟性に欠けるという欠点がある．

幸い，これら二つの欠点は，プログラミング言語の理論的研究によって，ほぼ克服することが可能になっている．Hindley と Milner によって確立された型推論の理論は，型宣言を省略したプログラムテキストから，そのプログラムが持つ最も一般的な型を自動的に推論することを可能にした．また Girard および Reynolds によって提案された多相型の理論は，プログラムの持つ汎用性を型として表現することを可能にした．これら二つを取り入れ，単純な型付きラムダ計算を拡張するならば，上に述べた単純な型システムの弱点を克服した型付きプログラミング言語のモデルが構築可能である．第 4 章では，単純な型付きラムダ計算の型推論の原理を解説する．続いて第 5 章で，多相型を含んだラムダ計算を解説し，さらに型推論システムと多相型を統合した型システムの原理を解説する．以上の拡張は，すべて，まえがきで紹介した Standard ML に取り入れられている．5.6 節では，プログラミング言語 Standard ML を紹介する．

以上のラムダ計算の型システムに基づくプログラミング言語のモデルの理論とは独立に，オブジェクト指向プログラミングの考え方に基づくプログラミング言語がいくつか提案され，実装されている．現在のところオブジェクト指向プログラミングの基礎理論が確立しているとはいい難いが，オブジェクト指向プログラミング言語の実装実験を通じ種々の一般性ある有用な概念が提案され，プログラミング言語の基礎理論の研究にも大きな影響を与えた．そのなかで実用上重要と思われる成果に，サブタイプ (subtype) および多相型レコード計算に関する理論がある．第 6 章ではこれら二つの理論を紹介する．