第II部

プログラミング言語

10 プログラミング言語の概要

第 I 部で学んだ種々のデータ構造やアルゴリズムを用いて，実際にコンピュータシステムを構築するためには，それらを記述するプログラミング言語が必要である．本章では，近代的なプログラミング言語の役割，その構造，およびその基本原理を概観し，第 II 部の目標とその概要を説明する．

§10.1 プログラミング言語の役割

デジタルコンピュータは，機械語(machine language)とよばれる人工的な言語で書かれたプログラムを実行する機械である．コンピュータによる問題解決は，必要な情報をビット列で表現(コード化)し，そのコード化された情報を機械語プログラムで処理することによって行なわれる．複雑で大規模な問題でも，必要なすべての情報をコード化し，その情報を処理するプログラムを構築することによって，コンピュータで解決可能である．厳密に定義可能でかつ有限な処理手順を与えることができる問題であれば，原理的には，その処理手順を表現する機械語プログラムを構成可能である．

しかしながら，機械語自身は，算術演算やメモリーの書き換えなどを行なう単純な機械命令の集まりに過ぎない．図 10.1 の機械語列は自然数の階乗を計算するプログラムである．このような簡単な処理であれば，直接機械命令の列として書くことも可能であるが，複雑で大規模な処理手順を，人間が機械語を用いてプログラムすることは事実上不可能である．

```
.L102:
        movl    %eax, %ebx
        movl    $1, %eax
        movl    $1, %ecx
.L101:
        cmpl    %ebx, %ecx
        jg      .L100
        movl    %ecx, %edx
        decl    %eax
        imull   %edx, %eax
        incl    %ecx
        jmp     .L101
.L100:
        ret
```

図 **10.1**　自然数の階乗を計算する機械語プログラム

第 I 部で学んだとおり，複雑な問題を解決するためには，問題解決に必要な情報を系統的にコード化するための種々の高水準のデータ構造と，それを効率よく処理するアルゴリズムを設計する必要がある．実際にそれらをコンピュータで実行するためには，さらに，それらデータ構造やアルゴリズムを表現し，それらを機械語に翻訳し実行するための言語，すなわち**プログラミング言語**(programming language)が必要である．

情報の表現方法やその処理アルゴリズムの設計方法は，それらを記述するプログラミング言語に深く依存した作業である．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，アルゴリズムを実行する手段にとどまらず，ソフトウェアを構築する枠組みを与える重要なものである．近代的なプログラミング言語の主な役割は，複雑で大規模なソフトウェアを正しくしかも効率よく実現するための記述システムを提供することである．

ソフトウェア記述システムとしてプログラミング言語がもつべき性質の主なものは，以下の 3 つである．

(i) 高水準の記述能力

プログラミング言語は，プログラムの論理構造の本質的な部分のみを簡潔に記述できる，高水準の記述力を提供することが望ましい．例えば，前述の自然数の階乗を計算するプログラムは，その論理構造のみを，以下のように再帰的な関数として記述できることが望ましい．

```
fun factorial n = if n = 0 then 1
                  else n * (factorial(n - 1))
```

このプログラムは，数学における階乗の定義とほぼ同等の抽象的で高水準の定義である．

(ii) 厳密な意味の定義

信頼性ある大規模なソフトウェアシステムを構築するためには，それを実現するプログラムの意味が厳密に定義されていなければならない．したがってプログラミング言語は，その言語で記述可能なすべてのプログラムの意味，すなわちその動作を完全に規定する形式的体系でなければならない．さらにその意味の定義も，関数や再帰などの数学的に厳密で抽象的な概念を基礎とした，高水準のものである必要がある．

(iii) 効率的な実装

プログラミング言語は，高水準の記述システムを提供するばかりでなく，それによって記述されたプログラムをコンピュータで効率よく実行する言語処理系を提供する必要がある．このためには，プログラミング言語で記述できるすべてのプログラムを，その意味を実現する機械語に変換する方式が確立されていなければならない．

以上のような特徴をもつ高水準プログラミング言語を構築するためには，言語の構文構造および言語の意味を明確に定義し，言語をコンピュータ上で実現する実装方式を構築しなければならない．以下これら構成要素の概要を説明する．

§10.2 言語の構文構造の定義

実際に人が文字を用いてプログラムを記述するために，プログラムがテキストとしてどのような構造をもつのかを定義する必要がある．これがプログラミング

言語の**構文論**(統語論，シンタックス，syntax)である．近代的なプログラミング言語のシンタックスの定義は，文法構造の定義と型システムの定義の 2 段階で行なわれる．

プログラミング言語の文法構造の定義は，自然言語の文法構造の分析に用いられるチョムスキーの**生成文法**(generative grammar)の理論を基礎とする．自然言語と比較したプログラミング言語特有の性質は，曖昧さを許さないこと，および文法構造がコンピュータで解析可能であることの 2 点である．

プログラムをコンピュータで機械的に実行するためには，文字列として与えられたプログラムの文法構造が一意に決まる必要がある．例えば自然言語においては，

「青い夏の夕暮れ」

のような 2 つの文法構造がありうる曖昧な文も許容されるが，プログラミング言語ではこのような曖昧さは許容されない．プログラミング言語の文法構造は，さらに，コンピュータで解析可能な程度の複雑さである必要がある．現在の高水準プログラミング言語のほとんどは，これらの要求を満たす制限された**文脈自由文法**(context-free grammar)を用いて定義される．文脈自由文法は，前後の文脈に依存しない語彙の局所的な組合わせに関する規則によって，言語の構文構造を定義するものである．

しかしながら，文脈自由文法のみでプログラミング言語の構文構造を完全に規定することは困難である．文脈自由文法ではとらえられない構文構造の主なものに，名前の定義とその参照関係の整合性，およびデータ型の間の整合性があげられる．例えば `factorial(3)` は，この文以前に名前 `factorial` が定義されているときのみ，構文論的に正しい文とみなされる．このような名前の定義と参照に関する整合性は文脈依存の制約である．またもし `x` が文字列を意味する名前であり `factorial` が自然数を対象とする操作の名前の場合，`factorial(x)` のような文は，操作が要求する型と操作の対象が属する型との間の整合性がなく，意味をなさない．このような型に関する整合性の制約も，文脈自由文法では定義することができないものである．

従来のプログラミング言語では，これらの整合性制約を文法の付帯条件として日常言語で定義する場合が多かった．しかし，日常言語による定義は厳密さに欠ける場合が多く，また後に紹介する多相関数のような高度で洗練された機能が要求する型の整合性制約を，日常言語で完全に定義することは困難である．高度で安全なプログラミング言語を構築するためには，これら整合性制約も厳密でかつ系統的な形式的体系として定義する必要がある．この目的に最も適した体系は，論理学における自然演繹体系の一種である**型理論**(type theory)である．

高度で洗練された近代的プログラミング言語の構文論的な定義は，文脈自由文法に基づく構文構造の定義に，型理論による整合性制約を加えたものである．

§10.3 意味の定義

プログラミング言語の構文構造の定義は，どのような文字列が正しいプログラムであるかを規定するのみである．プログラミング言語を完全に定義するには，これら構文構造の正しいプログラムの意味，すなわちプログラムがどのような計算を表現するかを厳密に規定する必要がある．機械語プログラムの場合の意味は，それをコンピュータで実行したときのコンピュータの動作そのものであるが，高水準プログラミング言語で書かれたプログラムの場合は，より抽象的な意味の定義が望ましい．

プログラミング言語の意味を定義するためのアプローチにはいくつかあるが，その代表的なものは**表示的意味論**(denotational semantics)，**公理的意味論**(axiomatic semantics)，および**操作的意味論**(operational semantics)の 3 つである．

表示的意味論は，プログラムの表示する計算を，すでにその構造が分かっている数学的な対象と解釈する意味論である．例えば自然数の階乗を計算するプログラムの意味は，自然数から自然数への関数として与えられる．この表示的意味論は，プログラムが表現しうる可能な意味のすべてを含むような集合を構築し，プログラムの集合からこの意味の集合への写像を定義することによって与えられる．このとき使われる可能な意味の集合を**意味領域**(domain)とよぶ．この意味論は，プログラミング言語の構文構造とは独立に定義される意味論であり，論理学におけるモデル論に相当する．通常意味論といった場合は，この表示的意味論をさす．表示的意味論の成否は，抽象度が高くしかもプログラムが表示する計算の性質を正確に反映した意味領域をいかに構築するかによるところが大きい．

公理的意味論は，プログラムの間に成立すべき同値関係を定義することによって，プログラムの意味を定義する方法である．この意味論は，個々のプログラムの意味を直接定義するものではなく，プログラムの意味の定義が満たすべき制約を定めるものであるが，詳細な同値関係を定めることにより，プログラムの意味を定義することが可能である．例えば関数を表現するプログラムの意味は，その関数が対象とする各引数ついて，そのプログラムを適用した結果がどのような値(を表すプログラム)と等しいかを決定することにより，定義することができる．公理的意味論は，プログラミング言語の構文構造のみから決定される意味論であり，論理学における証明論に相当する．

以上の 2 つの意味論は論理学の枠組みに基づく抽象度の高い意味論であり，プログラムのもつ諸性質を数学的に分析する上で重要である．これら抽象的な意味論に加えて，プログラミング言語を実装するためには，プログラムをコンピュータでどのように実行すべきかを，より直接的に定義する意味論が必要である．これがプログラミング言語の操作的意味論である．表示的意味論が抽象的な計算のモデルであるのに対して，操作的意味論は計算の実行過程をも含んだより具体的な計算のモデルであり，プログラミング言語の実装の基礎として使用される．

§10.4 実装方式の構築

以上のような諸概念を用いて定義されたプログラミング言語を実現するためには，文字列としてのプログラムを認識しそれをコンピュータで実行可能な機械語コードに変換する処理系を実装しなければならない．プログラミング言語の処理系は以下のような部分から構成される．

(i) 文法構造の解析

プログラミング言語の最初の処理は，プログラマが書いた文字列を読み込み，プログラミング言語が定義する語彙(単語)単位に分割し，さらにその単語の列が表現する文法構造をプログラミング言語が定義する文脈自由文法にしたがって分析し，文法的に正しい文として認識することである．文字列を単語単位に分割する処理を**字句解析**(lexical analysis)とよび，単語の列の文法構造を認識する処理を**構文解析**(parsing)とよぶ．文字列としてのプログラムは字句解析処理により語彙の列に変換された後，構文解析処理によりプログラムの構文構造を表現した**構文木**(syntax tree)とよばれるデータ構造に変換さる．

(ii) 型チェック

プログラミング言語の文であるためには，文法的に正しいことに加えて，文脈依存の整合性制約を満たさなければならない．プログラミング言語処理系は，構文解析ののち型理論的な解析を行ない，与えられた文が意味あるプログラムであるか否かのチェックを行なう．この処理では，プログラム中に使用されるすべての名前(変数や関数名)の使われ方を記録し，種々の関数の適用の間に矛盾がないかをチェックし，さらに変数の型および変数が何を指すかといった参照関係を決定する．Minimal や後に紹介する ML などの近代的なプログラミング言語では，この処理は，**型付きラムダ計算**(typed lambda calculus)と呼ばれる数理言語を用いて行なわれる．型チェックの後，型付きラムダ式とよばれる簡潔な数理言語の要素に変換される．

(iii) 実行コードへのコンパイル

プログラミング言語処理系の最後の仕事は，整合性あるプログラムを，その意味を実現する機械語コードに変換する処理である．この変換を行なう処理を**コンパイル**(compilation)とよぶ．

プログラミング言語処理系の以上の構造を図 10.2 に示す．

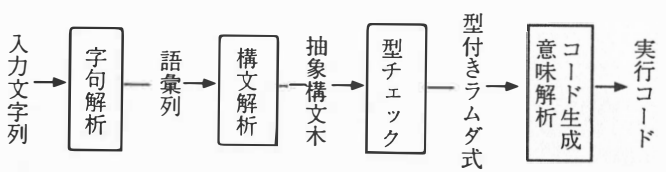


図 10.2 プログラミング言語処理系の構造

§10.5 理論の役割と第 II 部の構成

プログラミング言語を用いて書かれた，膨大なプログラム群からなるソフトウェアシステムが信頼性あるものであるためには，それを記述するプログラミング言語が堅牢な基礎の上に成り立っていなければならない．堅牢なプログラミング言語の実現のためには，プログラミング言語のもつ諸機能の相互関係の理論的分析や整合性の検証と厳密な意味の定義，およびそれに基づく系統的な実装方式の確立が必要である．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，望ましい機能をも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．

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 の定義は，誰が読んでも唯一の意味をも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きわめて高い信頼性を与えている．さらに ML は，第 15 章で詳しく解説する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，信頼性ある複雑なソフトウェアを効率よく開発するのに大いに役立つが，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もつ ML は，現時点では，最もよく設計された高水準の実用プログラミング言語の 1 つであり，またプログラミング言語の新しい機能などの研究の基礎ともなっている言語である．本書で使用した教育用プログラミング言語 Minimal も ML をそのモデルとしている．

第 II 部の目的は，ML を典型とする近代的プログラミング言語の動作原理および実装の方法を理解することである．特に，論理学との対応に重点をおき，プログラミング言語の諸概念を解説する．本シリーズの姉妹編の『コンピュータサイエンス入門 論理とプログラム意味論』の論理学と併せて学ぶならば，プログラミング言語の理論的基礎とコンピュータサイエンスのための論理学双方のより深い理解と展望が得られるであろう．

第 II 部の以降の展開はおよそ以下の通りである．まず第 11 章でプログラミング言語のシンタックスの定義および解析の概要を解説する．第 12 章において，型付きラムダ計算とよばれる数理言語を導入し，プログラミング言語の型チェックおよび意味の定義に関する理論を展開する．第 13 章では，プログラミング言語の種々の機能は型付きラムダ計算で表現できることを示す．第 14 章では，型付きラムダ計算の実装方式を解説する．最後に第 15 章において，最先端のプログラミング言語の機能である型推論と多相型の理論を紹介する．

§10.6 準備：集合に関する記法について

次章に移る前に第 II 部で使用する集合に関する記法を以下にまとめておく．

A と B を集合とする．A と B の**直積** $A \times B$ と**直和** $A + B$ はそれぞれ以下の集合である．

$$A \times B = \{(a,b) \mid a \in A,\ b \in B\}$$
$$A + B = \{1(a) \mid a \in A\} \cup \{2(b) \mid b \in B\}$$

A と B の関係は $A \times B$ の部分集合である．A と A の関係を A 上の(二項)関係という．R を A 上の関係とする．R が**反射的**であるとは，任意の $x \in A$ に対して $(x,x) \in R$ を満たすことをいう．R が**対称的**であるとは，任意の $x, y \in A$ に対してもし $(x,y) \in R$ なら $(y,x) \in R$ となることをいう．R が**推移的**であるとは，任意の $x, y, z \in A$ に対してもし $(x,y) \in R$ かつ $(y,z) \in R$ なら $(x,z) \in R$ となることをいう．A 上の**同値関係**とは，A 上の反射的，対称的かつ推移的な関係である．R の**反射的推移的閉包**とは R を含み反射的かつ推移的な関係の中で最小のものである．R の反射的推移的閉包を通常 R^* と書く．f が A から B への関数であるのは，任意の $x \in A$ について $(x,y) \in f$ となる $y \in B$ があり，任意の $x \in A$, $y \in B$, $z \in B$ について $(x,y) \in f$ かつ $(x,z) \in f$ なら $y = z$ となるときである．このとき，A を関数 f の定義域といい，$dom(f)$ と書く．f が A から B の関数であることを $f \in A \to B$ と書き，$(x,y) \in f$ を $f(x) = y$ または $f\,x = y$ と書く．X が集合の集合であるとき，X の要素すべての和集合を $\bigcup X$ と書く．

問 10.1 集合 A 上の関係 R, S に対して，RS を以下のように定める．

$$RS = \{(x,y) \mid \text{ある } z \in A \text{ があって } (x,z) \in R \text{ かつ } (z,y) \in S\}$$

R, S が関数なら RS も関数であることを示せ．

R^n を以下のように定義する．

$$R^0 = \{(x, x) \mid x \in A\}$$
$$R^{n+1} = R^n R \quad (n \geq 0)$$

R の反射的推移的閉包は $\bigcup\{R^n \mid n \geq 0\}$ と書けることを示せ．

11 シンタックスの定義と解析

形式的に見れば，言語はある決まった有限のシンボル集合(**アルファベット**)の要素の列の集合である．通常のプログラミング言語のアルファベットは，コンピュータのキーボードで直接入力可能な，英数字を中心とした文字と記号の集合である．プログラミング言語を定義するためには，まずこれらの記号のどのような並びが意味あるプログラムであるかに関する厳密な定義を与えなければならない．通常この定義はプログラミング言語を構成する**語彙**(lexicon)の定義，および**文法**(grammar)の定義の 2 段階で行なわれる．本章では，語彙と文法の定義および解析の概要を学ぶ．

§11.1 正規表現を用いた語彙の定義

プログラミング言語の語彙の定義は，変数名やキーワード，数値定数などの構成要素をどのように表記するかを規定するものであり，通常，**正規言語**(regular language)とよばれる数理言語を用いて行なわれる．

正規言語を厳密に定義するために文字列に関する記法を導入する．Σ を言語のアルファベットとする．Σ 上の**文字列**とは Σ の要素の有限な並びである．ただし，Σ 上の文字列集合には空文字列 ϵ が含まれるものとする．v と w を並べて作られる文字列を v と w の**連結**といい，vw と書く．$v = a_1 \cdots a_n$，$w = b_1 \cdots b_m$ の場合，$vw = a_1 \cdots a_n b_1 \cdots b_m$ である．ϵ はこの連結操作に関する単位元である．すなわち任意の文字列 v に対して $v\epsilon = \epsilon v = v$ である．A, B を Σ 上の文字列の集

合とし，以下の演算を定義する．

$$AB = \{w_1 w_2 \mid w_1 \in A,\ w_2 \in B\}$$
$$A^n = \begin{cases} \{\epsilon\} & n = 0 \\ AA^{n-1} & n > 0 \end{cases}$$
$$A^* = \bigcup \{A^n \mid n \geq 0\}$$

Σ 上の文字列全体の集合は Σ^* である．

Σ 上の正規言語の集合は，以下の規則によって定義される Σ^* の部分集合の集合である．

(i) 空集合 $\emptyset$ は正規言語である．
(ii) $\{\epsilon\}$ は正規言語である．
(iii) $a \in \Sigma$ なる a に対して，$\{a\}$ は正規言語である．
(iv) R が正規言語なら R^* は正規言語である．
(v) R_1 と R_2 が正規言語なら，$R_1 \cup R_2$ と $R_1 R_2$ はいずれも正規言語である．
(vi) 以上のいずれかを満たすもののみが Σ 上の正規言語である．

以上の定義は，定義される概念を定義自身の中で参照している．このような定義を**帰納的定義**とよぶ．帰納的定義はコンピュータサイエンスが対象とする種々の集合の定義として頻繁に使用される．

上の定義における規則(i)から規則(v)は，正規言語の集合に含まれるべき要素を条件として述べている．規則(vi)は，定義される正規言語の集合が，規則(i)から規則(v)を満たす集合の中で最小のものであることを要求している．このような帰納的定義で定義される集合は，空集合から始めて，規則で要求される要素を付け加えていった極限として求められる．

Σ^* の部分集合の集合に関する漸化式 $\mathcal{R}^n$ を以下のように定義する．

$$\mathcal{R}^0 = \emptyset$$
$$\mathcal{R}^{n+1} = \mathcal{R}^n \cup \{\emptyset\} \cup \{\{\epsilon\}\} \cup \{\{a\} \mid a \in \Sigma\} \cup \{R^* \mid R \in \mathcal{R}^n\}$$
$$\cup \{R_1 R_2 \mid R_1 \in \mathcal{R}^n, R_2 \in \mathcal{R}^n\} \cup \{R_1 \cup R_2 \mid R_1 \in \mathcal{R}^n, R_2 \in \mathcal{R}^n\}$$

この漸化式の右辺の $\mathcal{R}^n$ 以外の各要素は，それぞれ規則(i)から規則(v)に対応している．正規言語の集合は，空集合にこれらを加える操作を続けた極限である．すなわち正規言語の集合を $\mathcal{R}$ とすると，

$$\mathcal{R} = \bigcup \{\mathcal{R}^n \mid n \geq 0\}$$

と表せる．実際 $\mathcal{R}$ は，規則(i)から規則(vi)で表された条件をすべて満たす唯一の集合であることを確認できる．

問 11.1 上のように定義された $\mathcal{R}$ が以下の性質を満たすことを示せ．
(i) $\emptyset \in \mathcal{R}$
(ii) $\{\epsilon\} \in \mathcal{R}$
(iii) すべての $a \in \Sigma$ について $\{a\} \in \mathcal{R}$
(iv) $R \in \mathcal{R}$ なら $R^* \in \mathcal{R}$ である．
(v) $R_1, R_2 \in \mathcal{R}$ なら $R_1 R_2 \in \mathcal{R}$ かつ $R_1 \cup R_2 \in \mathcal{R}$ である．
さらに，もしある集合 A が(i)から(v)までの性質を満たせば，任意の n について $\mathcal{R}^n \subseteq A$ であることを示せ．これを用いて $\mathcal{R}$ は，(i)から(v)を満たす最小の集合であることを証明せよ．

正規言語は以上のような簡単な構造をもつため，**正規表現**(regular expression)とよばれる簡潔な表現で定義でき，コンピュータで効率よく認識することができる．正規表現を定義する上で正規言語に関する以下の性質が有用である．

補題 11.1
(i) R が正規言語なら任意の自然数 n に対して R^n も正規言語である．
(ii) R が正規言語なら $\bigcup \{R^n \mid n \geq 1\}$ も正規言語である．この集合を R^+ と書く．
(iii) R が正規言語なら $\Sigma^* \setminus R$ も正規言語である．
(iv) R_1 と R_2 が正規言語なら $R_1 \cap R_2$ も正規言語である． □

問 11.2 補題 11.1 の性質(i)と(ii)を示せ．性質(iii)を使い性質(iv)を示せ．

問 11.3 R と S が正規言語であるとする．$\epsilon \notin R$ なら等式 $X = S \cup RX$ を満たす X は唯一存在し，かつ正規言語であることを示せ．

正規表現 r とそれが表す正規言語 $[\![r]\!]$ を定義する．

(i) ϕ は正規表現であり，$[\![\phi]\!] = \emptyset$ である．

(ii) ϵ は正規表現であり，$[\![\epsilon]\!] = \{\epsilon\}$ である．

(iii) 任意の $a \in \Sigma$ に対して a は正規表現であり，$[\![a]\!] = \{a\}$ である．

(iv) $r, r_1, r_2, \ldots, r_n$ が正規表現なら r^n(n は任意の自然数)，$r*$，$r+$，$\hat{}r$，$r_1 \backslash r_2$，$r_1 \mid \cdots \mid r_n$，$r_1 \cdots r_n$ も正規表現であり，それぞれ以下の集合を表す．

$$
\begin{aligned}
[\![r^n]\!] &= [\![r]\!]^n \\
[\![r*]\!] &= [\![r]\!]^* \\
[\![r^+]\!] &= [\![r]\!]^+ \\
[\![\hat{}r]\!] &= \Sigma^* \setminus [\![r]\!] \\
[\![r_1 \setminus r_2]\!] &= [\![r_1]\!] \setminus [\![r_2]\!] \\
[\![r_1 \mid \cdots \mid r_n]\!] &= [\![r_1]\!] \cup \cdots \cup [\![r_n]\!] \\
[\![r_1 \cdots r_n]\!] &= [\![r_1]\!] \cdots [\![r_n]\!]
\end{aligned}
$$

正規言語の定義および補題 11.1 より，正規表現で定義される言語のクラスは正規言語の集合と一致することが確かめられる．

プログラミング言語のキーワードや定数表現などの語彙は，通常この正規表現を用いて定義される．

例 11.2 数を表す正規表現

$$
\begin{aligned}
\Sigma &= \{0,1,2,3,4,5,6,7,8,9,.,+,-,E\} \\
sign &= + \mid - \mid \epsilon \\
digit &= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
int &= sign\ digit^+ \\
number &= int \mid int\ E\ int \mid int\ .\ digit* \mid int\ .\ digit*\ E\ int
\end{aligned}
$$

例えば -1234，3.14，さらに指数部をもつ $1.09E20$ のような文字列が $[\![number]\!]$ に含まれる． □

§11.2 文脈自由文法を用いた文法構造の定義

プログラミング言語の文法構造の定義を完成させるためには，正規表現で定義された語彙をどのように組合わせたものが，プログラミング言語の文になりうるかを規定しなければならない．ほとんどのプログラミング言語では，この定義は文脈自由文法で行なわれる．文脈自由文法は，文として可能な語彙の組合わせをすべて生成する規則の集合を与えることによって言語を定義する．文脈自由文法 $G = (\boldsymbol{T}, \boldsymbol{N}, S, \boldsymbol{P})$ を以下の 4 つ組と定義する．

(i) **終端記号**(terminal symbol)の有限集合 $\boldsymbol{T}$

終端記号は言語を構成する語彙である．例えば自然言語の文法では，「教科書」などの単語や「.」や「,」などの文法記号が含まれる．通常のプログラミング言語では $\boldsymbol{T}$ の各要素は正規表現で定義される集合を表す．

(ii) **非終端記号**(non terminal symbol)の有限集合 $\boldsymbol{N}$

非終端記号は，文法構造の構成要素単位に対応する名前に相当する．例えば自然言語の文法では，「動詞句」や「名詞句」などである．

(iii) 開始記号 $S \in \boldsymbol{N}$

言語の「文」に対応する非終端記号であり，文法を生成規則と見た場合，生成の出発点となる記号である．すなわち S から導かれる文字列がこの言語の定義する文である．

(iv) 生成規則の有限集合 $\boldsymbol{P}$

生成規則は，非終端記号がどのような構造かを表す以下の形をした規則である．

$$N \to \alpha$$

ここで $N \in \boldsymbol{N}$，$\alpha \in (\boldsymbol{N} \cup \boldsymbol{T})^*$ である．同一の非終端記号に対する規則は複数存在してよい．自然言語における「文」の構造の定義には，例えば以下のような規則が含まれる．

$$\text{文} \to \text{名詞句 動詞句 .}$$

ここで '名詞句' と '動詞句' は非終端記号，'. ' は終端記号である．

文脈自由文法によって生成される文は，開始記号から始めて，生成規則を有限回使って導出できる終端記号列である．この関係を形式的に定義するために，$\alpha = \alpha_1 A \alpha_2$ かつ $A \to \gamma \in P$ であるとき，$\alpha \Longrightarrow \alpha_1 \gamma \alpha_2$ と書く．すなわち関係 $\alpha \Longrightarrow \beta$ は，α 中の 1 つの非終端記号を，その非終端記号を左辺とする生成規則

の右辺で置き換えて文字列 β が得られることを表す．この 2 項関係の反射的推移的閉包を $\overset{*}{\Longrightarrow}$ と書く．すなわち $\alpha \overset{*}{\Longrightarrow} \beta$ なら β は α から書き換え規則を何回か使って得られる記号列である．文脈自由文法 $G = (\boldsymbol{T}, \boldsymbol{N}, S, \boldsymbol{P})$ によって生成される文の集合 $L(G)$ を

$$L(G) = \{w \mid S \overset{*}{\Longrightarrow} w, w \in T^*\}$$

と定義する．

例 11.3 文脈自由文法 $G = (\boldsymbol{T}, \boldsymbol{N}, S, \boldsymbol{P})$ を以下のように定義する．

$$\begin{aligned}\boldsymbol{N} &= \{S\}\\ \boldsymbol{T} &= \{(,)\}\\ \boldsymbol{P} &= \{S \to (),\ S \to (S),\ S \to SS\}\end{aligned}$$

この文法で生成される文の集合 $L(G)$ は右括弧‘(’左括弧‘)’のみからなり，かつ括弧の対応のとれた文字列の集合である． □

正規言語に基づく語彙の簡潔な定義のために正規表現が使用されたように，文脈自由文法を簡潔に定義する記法として **BNF 記法**(BNF notation)が広く使われている．文脈自由文法 $G = (\boldsymbol{T}, \boldsymbol{N}, S, \boldsymbol{P})$ の中で，$\boldsymbol{T}$ は語彙の集合であり，すでに了解されているものとする．すると文法 G を決定するためには非終端記号に対する規則の集合を与えればよい．$\boldsymbol{N} = \{A_1, \ldots, A_n\}$ とし，$\{\alpha_1^i, \ldots, \alpha_m^i\} = \{\alpha \mid A_i \to \alpha \in \boldsymbol{P}\}$ とする．BNF 記法とは，これら規則を各非終端記号ごとに，以下のようにまとめて記述する記法である．

$$\begin{aligned}A_1 &::= \alpha_1^1 \mid \cdots \mid \alpha_{m_1}^1\\ &\vdots\\ A_n &::= \alpha_1^n \mid \cdots \mid \alpha_{m_n}^n\end{aligned}$$

この記法において，“|”は「または」の意味である．さらに便宜的に最初に書かれた非終端記号 A_1 が開始記号であると約束すると，文脈自由文法を簡潔に定義することができる．例えば例 11.3 で定義した文法は，以下の BNF 記法で表現できる．

$$K ::= () \mid (K) \mid KK$$

BNF 記法で用いられた非終端記号を**メタ変数**とよび，定義された言語の要素の代表として用いる．

§11.3 字句解析

プログラミング言語を実現するためには，ユーザが書いた記号列を解析し，その文法構造を認識しなければならない．構文構造の定義が正規表現による語彙の定義と文脈自由言語による文法構造の定義の 2 段階で行なわれたのに対応し，この解析処理も，記号列を語彙の単位に分解する**字句解析**(lexical analysis)処理と，文法構造を解析する**構文解析**(parsing)処理の 2 段階で行なわれる．

字句解析処理の基本原理は，任意の正規言語に対して，その言語を認識する有限状態機械とよばれるアルゴリズムが構成できることに基づいている．**有限状態機械**(finite state machine) $M = (\Sigma, Q, q_0, F, \delta)$ は，以下の 5 つの要素からなる組である．

(i) 入力記号集合 Σ

機械に対する入力記号の集合である．語彙を認識する機械の場合，入力記号の集合は言語のアルファベットである．

(ii) 機械の状態の有限集合 Q

Q の各要素は，機械が現在どのような状態にあるかを表す．語彙を認識する機械の場合，Q の各要素は，現在処理中の入力記号列が表現する可能性のある語彙の集合に対応する．

(iii) 機械の初期状態 q_0

機械がまだ一文字も処理していない状態を表す．

(iv) 最終状態集合 $F \subseteq Q$

機械が正常に停止するときの可能な状態の集合である．語彙を認識する機械の場合，入力文字列をすべて読み込んだとき F のいずれかの状態に達すれば，その文字列は正しい語彙であることを意味する．

(v) 状態遷移関数 δ

$Q \times \Sigma$ から Q への関数であり，つぎの記号を 1 つ読んだときの機械の動作

を定義する．

有限状態機械 $M=(\Sigma,Q,q_0,F,\delta)$ の状態遷移関数 δ を，以下の定義により，文字列の集合 Σ^* と状態集合 Q 上の関数 $\hat{\delta}$ に拡張する．

$$\hat{\delta}(\epsilon,q)=q$$
$$\hat{\delta}(aw,q)=\hat{\delta}(w,\delta(a,q)),\quad \text{ただし } a\in\Sigma,\ w\in\Sigma^*$$

問 11.4 任意の $w,v\in\Sigma^*$ に対して

$$\hat{\delta}(wv,q)=\hat{\delta}(v,\hat{\delta}(w,q))$$

であることを示せ．(ヒント：w の長さに関する帰納法を試みよ．)

有限状態機械 $M=(\Sigma,Q,q_0,F,\delta)$ の認識する言語 $L(M)$ を以下のように定める．

$$L(M)=\{w\mid \hat{\delta}(w,q_0)\in F\}$$

正規言語は有限状態機械によって認識することができる．以下の定理が知られている．

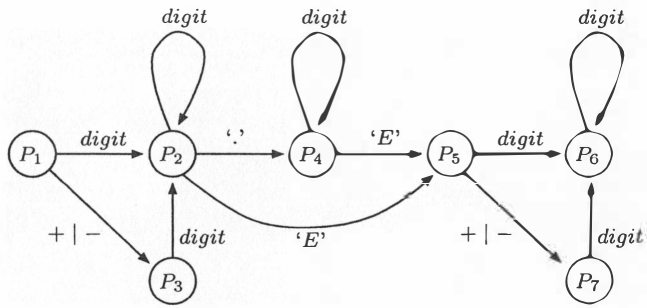
定理 11.4 Σ 上の任意の正規言語 R に対して，$L(M)=R$ となる有限状態機械 M が存在する． □

図 11.1 に以前定義した数を表す正規表現 *number* で表される正規言語を認識する有限状態機械の例を示す．ただし，最終状態に到達し得ない状態とその状態への遷移は省略してある．

問 11.5 図 11.1 で省略されている状態を補い，有限状態機械の厳密な定義を与えよ．

有限状態機械は，状態と入力にしたがってあらかじめ決められたつぎの状態に遷移することを繰り返す簡単な機械であり，メモリー上に表現された状態を状態遷移関数 δ の定義に従い更新するプログラムを書くことにより，コンピュータで容易に実現することができる．相互再帰的関数が使用できる言語であれば，各状態に対して，その状態からの遷移を表す関数を定義すればよい．状態が自然数で表されているとすると，状態 i に対する関数 f_i は以下のように定義できる．

$$f_i(x::L)=f_{\delta(i,x)}(L)$$
$$f_i(nil)=\mathit{if}\ i\in F\ \mathit{then}\ \mathit{Accept}\ \mathit{else}\ \mathit{Reject}$$



ただし S=P_1，$F=\{P_2,P_4,P_5,P_6\}$ である．

図 **11.1** 有限状態機械の例

ここで $x::L$ は入力の先頭文字が x で，それに続く文字列が L であることを表し，*nil* は入力の終了を表す．

問 11.6 例 11.2 で定義した *number* で表現される正規言語を認識する相互再帰的な関数を定義せよ．

さらに与えられた正規表現の定義から，それが表現する正規言語を認識する有限状態機械を自動的に構築するアルゴリズムが知られており，そのアルゴリズムを実現する字句解析処理生成システム lex が実用化されている．プログラミング言語の字句解析処理の多くは，以上の原理にもとづく lex システムを用いて構築されている．

§11.4 構文解析

プログラミング言語の構文構造の解析の第二段階は，字句解析処理の結果得られた語彙の列の構造を分析する構文解析である．この処理は，スタート記号から与えられた語彙列に至る生成規則の系列が存在しうるか否かをチェックし，存在すれば，その生成系列を表すデータ構造を出力する操作である．この分析は，与えられた文を生成する可能な生成系列の探索を必要とする．文脈自由文法で定義された言語の場合，この問題に関して以下の定理が知られている．

定理 11.5 任意の文脈自由文法 G に対して，その文法で生成された言語を認識

する仮想的な機械である非決定性機械プッシュダウンオートマトンを構成することができる．□

正規言語を認識する有限状態機械はコンピュータで容易に実現可能であったが，残念ながら，文脈自由言語を認識する非決定性機械プッシュダウンオートマトンをコンピュータで効率よく実現するのは困難である．有限状態機械がメモリーを使用しない電卓のような単純な機械であるのに対して，非決定性プッシュダウンオートマトンは，機械内部に書き換え可能なメモリをもち，さらに，メモリー状態と入力記号の組合わせに対して並行に処理を行なう，通常のコンピュータの能力を越える強力な機械である．この理由から，非決定性機械プッシュダウンオートマトンはプログラミング言語の構文解析のモデルとしてはあまり使用されることはないので，その詳しい定義は省略する．興味ある読者は数多く出版されている形式言語理論に関する教科書を参照されたい．

文脈自由言語の構文解釈問題を効率よく解決するコンピュータプログラムの開発は，プログラミング言語の研究の初期の重要な研究課題であったが，幸いこの問題には LR 構文解析法とよばれる，ほぼ満足いく解決法が知られている．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なうプログラムを自動生成するシステム yacc が実用化されている．yacc システムでは，文法規則に含まれる曖昧さを解消するための結合性に関する規則が記述でき，広い範囲の文法に対する構文解析処理を自動生成可能である．

結合性とは代数における演算子の結合則の類似から，生成規則の適用範囲を決める規則である．結合性の概念を説明するために，f と g を演算子として以下の文法を考える．

$$E ::= c \mid E\ f\ E \mid E\ g\ E$$

この文法は，2 つの演算子の引数の範囲に関して曖昧性がある．以下の説明では括弧を使って構文の構造を表すことにする．

f に対する規則と g に対する規則が競合する

$$E_1\ f\ E_2\ g\ E_3$$

のような記号列に対して，f に対する導出を先に行ない，$(E_1\ f\ E_2)\ g\ E_3$ の形の導出を生成することを要求する規則を「f は g より**結合力**が強い」という．逆に

g に対する導出を先に行ない，$E_1\ f\ (E_2\ g\ E_3)$ の形の導出を生成することを要求する規則を「g は f より結合力が強い」という．

さらに同一の導出規則が競合する

$$E_1\ f\ E_2\ f\ E_3$$

の形の記号列に対して，常に $(E_1\ f\ E_2)\ f\ E_3$ の形の構文木を作成することを要求する規則を，「f は**左結合**する」といい，逆に f を含む規則を右から優先して適用し，$E_1\ f\ (E_2\ f\ E_3)$ の形の構文木を生成することを要求する規則を，「f は**右結合**する」という．

この結合力の概念は，二項演算子以外にも適用することができ，曖昧さのない文法を簡潔に記述する上で有効である．

プログラミング言語のシンタックスの解析処理は，多くの場合，前述の lex と yacc の組合せによって生成される．これら 2 つの原理に関しては，多くの教科書が出版されているので，詳細はそれらに譲ることにする．

構文解析の結果，文字列としてのプログラムは**構文木**とよばれるデータ構造に変換される．構文木は，非終端記号を内部ノードとしてもち，終端記号を外部ノードとしてもつ木であり，文の導出系列を表現している．例えば，例 11.3 で定義した括弧の式の文法における導出

$$S \Longrightarrow SS \Longrightarrow S() \Longrightarrow (S)() \Longrightarrow (())()$$

は図 11.2 に示す構文木で表現される．

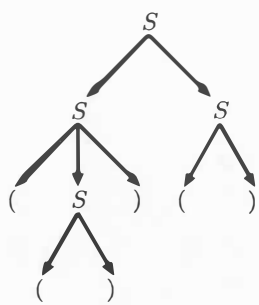


図 **11.2** 構文木の例

構文木を帰納的に定義するために，$\alpha \in \boldsymbol{N} \cup \boldsymbol{T}$ とし，α をルートノードとする木を α-木とよぶことにする．α-木の集合は，帰納的に以下のように与えられる．

(i) $\alpha \in \boldsymbol{T}$ なら α は α-木である．
(ii) $\boldsymbol{P}$ が規則 $N \to \alpha_1 \cdots \alpha_n$ を含み，各 T_{α_i} が α_i-木なら，N をルートノードとし，$T_{\alpha_1}, \cdots, T_{\alpha_n}$ を子とする木は N-木である．
(iii) 以上の規則によって生成される木のみが α-木である．

S を開始記号とする文法の構文木は S-木と定義できる．

この定義から明らかなように，構文木は，各非終端記号 N をデータ型とみなし，N の各生成規則に対してデータ構成子を定義することによって，簡単に表現できる．例えば BNF 文法 $K ::= () \mid (K) \mid KK$ で生成される構文木は Minimal の以下のデータ型で表現できる．

```
type K = OneParen | Paren of K | Apply of K * K ;
```

問 11.7 LR 構文解析アルゴリズムは，実用的なプログラミング言語の文法の構文解析に利用される強力な方法であるが，この方法により，ユーザが，与えられた文法から直接構文解析処理を書くのは困難である．

ユーザが直接プログラム可能な構文解析法に，再帰的降下法がある．この方式は，各非終端記号に対応した関数を再帰的に呼び出すことによって構文解析を行なうものである．この方法が可能である最も簡単な場合は，以下の文法のように，各生成規則が，その右辺に含まれる終端記号によって特定できる場合である．

PROGRAM ::= E ;
E ::= *id* | *int* | (fn *id* => E) | (E E) | (fix E)
| (E + E) | (E - E) | (E * E) | (E / E) | (E = E)
| (pair E E) | (fst E) | (snd E)
| (inl E) | (inr E) | (case E of 1(*id*) =>E, 2(*id*) => E)
| let *id* = E in E end

本問の目的は，Minimal または ML を使ってこの言語に対する字句解析および構文解析処理プログラムを作成することである．さらに次章以降，理論の展開にあわせた練習問題で，本問のプログラムを簡単な ML 系言語のコンパイラへと拡張することにする．

Minimal でも十分プログラミング可能であるが，ML に興味のある読者は，巻末にいくつかあげる ML に関する参考文献を参照し，ML でプログラミングしてみることを勧める．

(i) *id* を変数を表す文字列，*int* を例 11.2 で定義された整数を表す表現とする．*id*, *int* さらにその他の語彙('(', ')', '+', '-', '*', '/', '=', '=>', ',', ';', 'fn', 'fix', 'pair', 'fst', 'snd', 'inl', 'inr', 'case', 'of', 'let', 'end', その他の 1 文字)を表現するデータ型を定義せよ．ただし *id* は文字列を値としてもち，*int* は整数を値としてもつものとする．
(ii) 入力ストリームから一文字ずつ読み込む関数を定義し，この関数を使って，*id*, *int*, さらにその他の語彙を認識する関数を定義せよ．
(iii) 上記の構文の構文木を表現するデータ型を定義し，構文木をプリントする関数を定義せよ．ただし構文木データ型には，構文エラーを表す要素を含むものとする．
(iv) 上記文法を解析し，構文木を返す再帰的関数を定義せよ．
(v) 標準入力から文字を読み込み，構文解析を行ないデータ構造に変換し，プリント関数でプリントするプログラムを作成せよ．
(vi) ";" までの文字列を読み構文解析を行ない，結果をプリントすることを，入力が終了するまで繰り返すトップレベルの read_print 関数を書け．read_print はユーザと対話しながら，以下のような動作をする．

```
read_print();                (トップレベルプログラムの起動)
->(fn x => x);               (ユーザが式を入力)
(fn x => x)                  (プログラムが結果をプリント)
->((x (x,y)) z);
((x (x,y)) z)
->(x y z);
Syntax error.
->
```

ここで "->" は作成したプログラムの入力促進文字列(プロンプト)である．

12 型付きラムダ計算

本章では，プログラミング言語の理論的分析のための数理言語として使用される型付きラムダ計算を導入し，プログラムの型の解析および意味論の定義に関する理論を展開する．

§12.1 核言語としてのラムダ計算

プログラミング言語の理論を完成させるためには，前章で定義した構文構造の定義と解析に加えて，プログラムの各部分間の整合性制約を規定する型システム，プログラムの意味，およびその実現方式を与える必要がある．これらの理論を展開する上で，具体的な構文構造を使用するのは必ずしも適当ではない．その主な理由は，具体的なプログラミング言語の構文構造は，より基本的な構文の組合わせによって定義可能な冗長性を含んでいるためである．それら定義可能な構文は，プログラミング言語をより使いやすいものにするために導入されている構文であり，「**構文上の糖衣**(syntax sugar)」とよばれる．

適当な糖衣構文の導入は実用的なプログラミング言語にとって重要であるが，プログラミング言語の実装や理論的な分析にとっては本質的ではない．プログラミング言語の定義や理論的分析においては，他の要素によっては定義できない最小限の構文要素からなる**核言語**が分離できることが望ましい．この核言語の要件は，広い範囲のプログラミング言語の構文が表現可能であること，および厳密に定義された簡潔な言語であることである．この要件を満たす近代的言語の代表的な核

言語がラムダ計算(lambda calculus)である．ラムダ計算は，簡潔な構文であるにもかかわらず，およそすべての計算可能なプログラムを表現可能な汎用の表現力を有している．本章において，ラムダ計算の構文と基本的な操作を定義したのち，ラムダ計算上で，プログラミング言語の型の整合性制約の理論，およびプログラムのふるまいの定義に関する理論を展開する．

§12.2 ラムダ式の定義

加算無限個の変数集合 Var が与えられているとし，x を Var を代表するメタ変数とする．また，定数集合 $Const$ が与えられているとし，c を $Const$ を代表するメタ変数とする．ラムダ式の集合は以下の BNF 文法で与えられる．

$$\begin{aligned} M ::= {} & c \mid x \mid \lambda x.\, M \mid M\ M \mid (M, M) \mid M[1] \mid M[2] \\ & \mid 1(M) \mid 2(M) \mid (\mathbf{case}\ M\ \mathbf{of}\ 1(x) \Rightarrow M,\ 2(x) \Rightarrow M) \\ & \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid \mathbf{fix}(M) \end{aligned}$$

各構文の意味は以下のとおりである．

- $\lambda x.\, M$ は，x を受け取り(一般に x を含む)式 M の値を計算する，名前のない関数を表す．この構文を**ラムダ抽象**(lambda abstraction)と呼ぶ．
- $M_1\ M_2$ は，関数 M_1 を実引数 M_2 に適用する**関数適用**(function application)である．
- (M_1, M_2) は M_1 と M_2 の組を表す．
- $M[1]$, $M[2]$ はそれぞれ，組 M の第一要素および第二要素を取り出す射影演算である．
- $1(M)$ と $2(M)$ はそれぞれ，M を直和の左および右の要素への埋め込みを表す．
- $(\mathbf{case}\ M\ \mathbf{of}\ 1(x) \Rightarrow M_1,\ 2(y) \Rightarrow M_2)$ は，M が直和の左への埋め込みか右への埋め込みかで場合わけを行ない，左への埋め込みなら，埋め込まれた値に x と名前を付けプログラム M_1 を実行し，また M が右への埋め込みなら，埋め込まれた値に y という名前を付けプログラム M_2 を実行する構文である．
- $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ は，M_1 の表す値に x という名前を付け，M_2 において使用する構文である．このようにラムダ式の値に名前を付けることを名前の束縛という．この構文は意味論的には $(\lambda x.\, M_2)\, M_1$ と同等であるが，第 15 章での多相型システムの定義の中で重要な役割を果たす．しかしそれまでは $(\lambda x.\, M_2)\, M_1$ の別記法とみなすことにする．
- $\mathbf{fix}(M)$ は繰り返しや再帰的関数の定義に使用される構文である．

プログラミング言語の実装において，ラムダ式は構文解析の後に生成される中間言語としての役割を果たす．ここではラムダ式を文字列ではなく，その構文構造がすでに分かっている構文木として取り扱う．ただし，具体的なラムダ式を表記する場合は，以下の約束に従うものとし，さらに必要な場合は，括弧を用いて構文構造を明確にする．

(i) 関数適用は左結合する．
(ii) ラムダ抽象 $\lambda x.\, M$における M はできる限り大きくとる．
(iii) $M[1]$と $M[2]$は関数適用より結合力が強い．

この**約束のもとでは，以下のような対応となる．**

表記	構文構造
$M_1\ M_2\ \cdots\ M_n$	$(\cdots(M_1\ M_2)\ \cdots\ M_n)$
$\lambda x_1.\lambda x_2 \ldots \lambda x_n.M_1 \cdots M_n$	$(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M_1 \cdots M_n)\ldots))$
$M_1 M_2[1]$	$M_1(M_2[1])$

問 12.1 ラムダ式を文字列の集合とみた場合，上記 BNF 文法は曖昧である．曖昧の原因となっている文法規則をすべて指摘し，括弧を付けることによってそれら規則を変形し，曖昧さのない文法とせよ．

ラムダ抽象 $\lambda x.\, M$，場合分け構文 $(\mathbf{case}\ M\ \mathbf{of}\ 1(x) \Rightarrow M_1,\ 2(x) \Rightarrow M_2)$ および名前の束縛構文 $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ における変数 x は，いずれも，将来与えら

れる実引数の値を表す仮の名前であり，これら変数名自身は重要ではない．この事情は，例えば論理式 $\forall x.P(x)$ や積分式 $\int f(x)dx$ における x と同様である．このような変数を**束縛変数**(bound variable)とよぶ．名前の衝突が起こらない限りにおいて，式の中の束縛変数の名前を付け変えても，式の意味は変化しない．例えば，$\int f(x)dx$ が $\int f(y)dy$ と同一の意味をもつのと同様に，$\lambda x.x$ は $\lambda y.y$ と同じ意味である．

束縛されていない変数を**自由変数**(free variable)とよぶ．自由変数は，文脈によって決まる特定の値を表す変数であり，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図 12.1 のように定義される．

$$
\begin{aligned}
FV(c) &= \emptyset \\
FV(x) &= \{x\} \\
FV(\lambda x.M) &= FV(M) \setminus \{x\} \\
FV(M_1\ M_2) &= FV(M_1) \cup FV(M_2) \\
FV((M_1, M_2)) &= FV(M_1) \cup FV(M_2) \\
FV(M[1]) &= FV(M) \\
FV(M[2]) &= FV(M) \\
FV(1(M)) &= FV(M) \\
FV(2(M)) &= FV(M) \\
FV((\textbf{case}\ M_1\ \textbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3)) & \\
&= FV(M_1) \cup (FV(M_2) \setminus \{x\}) \cup (FV(M_3) \setminus \{y\}) \\
FV(\textbf{let}\ x = M_1\ \textbf{in}\ M_2) & \\
&= FV(M_1) \cup (FV(M_2) \setminus \{x\}) \\
FV(\textbf{fix}(M)) &= FV(M)
\end{aligned}
$$

図 12.1 ラムダ式の自由変数の定義

ラムダ式に対する基本的な操作は，ラムダ式の自由変数への代入，すなわちラムダ式に現れる自由変数を，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 が束縛変数を含まない場合は，式 M の構造に従い再帰的に置き換えを実行するだけでよい．例えば変数および関数適用に対する規則は以下のように与えられる．

$$
\begin{aligned}
[N/x]x &= N \\
[N/x]y &= y \quad (x \neq y) \\
[N/x](M_1\ M_2) &= ([N/x]M_1\ [N/x]M_2)
\end{aligned}
$$

問 12.2 束縛変数を含まない他の 7 つの場合の代入の定義を与えよ．

しかしラムダ抽象，場合分け構文および値束縛の 3 つ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**(free variable capture)とよぶ．これを防ぐために，自由変数が捕捉されるおそれがある場合は，その原因となっている束縛変数の名前の付け替えを行なう必要がある．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．

ラムダ抽象に対する変数への代入の定義は以下のように与えられる．

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

ラムダ抽象の最後の場合の 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．

問 12.3 ラムダ抽象の例を参考に，場合分け構文と値の束縛構文に対する変数の代入の定義を与えよ．

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同値関係を厳密に定義することができる．ラムダ抽象の場合の規則は以下のとおりである．

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

これを α **同値公理**とよぶ．他の束縛変数に対しても同様の同値規則を定義できる．

問 12.4 他の 2 つの α 同値公理を定義せよ．

これら公理によって生成される同値関係を α **同値関係**とよび，M_1 と M_2 が α 同値であるとき $M_1 \equiv M_2$ と書く．以降ラムダ式はこの同値関係の下で考え，α 同値なラムダ式は同じものとみなす．この下では束縛変数は自由に選んでよいから，束縛変数に関して以下のような便宜的な仮定をする．

仮定 12.1 束縛変数は互いに異なりまた自由変数とも異なる．さらに，この性質は，ラムダ式の変数への代入によっても保存される． □

すなわち，例えば $[\lambda x.\, xy/z]\lambda w.\, wz\ z$ のような代入を行なった後に，必ず束縛変数が重ならないように名前変えが行なわれ，$\lambda w.\, w(\lambda v.\, vy)\ (\lambda u.u\ y)$ のようなラムダ式が生成されるものと仮定する．この仮定の下では，ラムダ式の変数への代入は自由変数の捕捉を心配することなく行なうことができる．

問 12.5 先に定義したラムダ式の代入は，束縛変数に関する仮定を保存しない．すなわち M と N が同時に束縛変数に関する仮定を満たしていても，$[N/x]M$ が束縛変数に関する仮定を満たすとは限らない．この性質が成り立つようにラムダ式の変数への代入の規則を変更せよ．

§12.3 ラムダ式の型の解析

ラムダ式の構文構造のみでは，正しいプログラムの表現として十分ではない．例えば $(\lambda x.\, x)[1]$ や $(\lambda x.\, x1)(1, 2)$ のような式は文法上許されるが，不整合を含んでおり意味のないラムダ式である．このような不整合を検出し，プログラミング言語で定義可能なプログラムを，整合性あるものに限ることができれば，その言語で記述されたプログラムの信頼性を大幅に高めることができる．そのような整合

性制約を規定するのがラムダ計算の型システムである．型システムを構築するには，文脈自由文法より強力な記述力をもったシステムが必要である．そのようなシステムの典型を論理学にもとめることができる．論理学には，大きく分けて論理学の記述対象の性質を調べるモデル論と，論理学によって記述される証明の性質を調べる証明論がある．プログラミング言語の対象とするプログラムは，構文論的な対象であるため，その性質の記述には証明論的アプローチが適している．

(a) プログラムと論理学の関係

論理学は整合性を検証する一般的な枠組みであるが，通常の古典的な論理学には，コンピュータ上での実行にそぐわない概念が含まれており，プログラムの整合性記述システムと相入れない要素を含む．その代表的なものは，背理法や排中律(すなわち「P または P でない」を常に成り立つ規則として使用する推論方法)である．この推論を使えば，例えば「a^b が有理数であるような無理数 a, b が存在する」という命題を，以下のような議論によって示すことができる．

(i) $(\sqrt{2})^{\sqrt{2}}$ は無理数か有理数かのどちらかである．
(ii) もし $(\sqrt{2})^{\sqrt{2}}$ が有理数なら $a = \sqrt{2}, b = \sqrt{2}$ ととればよい．
(iii) もし $(\sqrt{2})^{\sqrt{2}}$ が無理数なら $a = (\sqrt{2})^{\sqrt{2}}, b = \sqrt{2}$ ととればよい．

問 12.6 上記の証明は排中律を使用したものであるが，排中律は背理法と等価である．上記の証明と等価な証明を背理法を使って記述せよ．

しかしながら，この証明から a, b の値を求めることはできない．このような証明は非構成的であるといわれる．このような推論を禁止し，「$P(a)$ が成り立つ a が存在する」という命題の証明には，$P(a)$ を成り立たせる a の値を実際に構成することを要求するように，論理学の法則に制限を加えて得られる論理学が，**直観主義的論理学**である．これは，命題論理学の場合，従来の古典論理学から上述の排中律(またはそれと同等の推論法則)を取り除いて得られる体系に相当する．

直観主義的論理学では，各論理演算子は以下のように解釈される．

- $A \supset B$(含意命題，A ならば B)：A の証明から B の証明を構成する有限の手続きが存在する．

・ $A \wedge B$(連言命題，A かつ B)：A と B の両方の証明が存在する．

・ $A \vee B$(選言命題，A または B)：A の証明か B の証明のいずれかと，それがどちらの証明であるかを示す情報が存在する．

命題をその命題の証明の集合と解釈すると，この論理演算子の解釈は，それぞれ以下のようなプログラミング言語の概念に対応する．

$$A \supset B \iff A \text{から} B \text{への関数型} A \to B$$
$$A \wedge B \iff A \text{と} B \text{の直積型} A \times B$$
$$A \vee B \iff A \text{と} B \text{の直和型} A + B$$

そこで論理演算子の直観主義的解釈のもとでは，命題論理学とプログラミング言語の間に以下のような対応関係を考えることができる．

$$\text{命題} \iff \text{データ型}$$
$$\text{証明} \iff \text{プログラム}$$
$$P \text{は命題} A \text{の証明} \iff P \text{は型} A \text{の値を計算するプログラム}$$

すなわち，命題をプログラムの型とみなせば，直観主義的論理学における証明は，その型の値を計算するプログラムとみなせる．この対応関係により，プログラミング言語に対して，論理法則に基づくプログラムの構成の正しさに関する推論システムを構築することができる．これがラムダ計算の型システムである．

論理学の証明システムはいくつか存在するが，その中でラムダ式の論理構造の記述にとくに適しているものは**自然演繹システム**である．本節では自然演繹システムの概要を説明した後，それにもとづきラムダ計算の型システムを定義する．論理学の証明システムとしてより広く使用されているヒルベルト流の公理系は，ラムダ式をコンパイルした後の実行コードの構造の記述とみなすことができる．この話題は，プログラミング言語の実装のモデルを学ぶ第 14 章で取り上げる．

(b) 自然演繹システム

自然演繹システムを含めた論理学の体系や，その証明論とモデル論の関係は，本シリーズの姉妹編の『コンピュータサイエンス入門 論理とプログラム意味論』で詳しく解説されているので，本書ではラムダ計算と関連の深い要素のみを概説する．

以下の BNF 文法で定義される命題論理式を考える．

$$A ::= b \mid \alpha \mid A \supset A \mid A \wedge A \mid A \vee A$$

ここで，b は与えられた命題定数，α を命題変数をあらわす．

命題を表記する際，$\supset$ は右結合し，$\vee$ と $\wedge$ は $\supset$ より結合力が強いと約束する．

論理学における証明論は，与えられた仮定からある命題が証明できるか否かを構文論的に決定するシステムである．Δ を論理式の重複を許した有限な集合(**重複集合**，multi set)とする．重複集合には同じ要素が複数存在するが，それらは区別されるものとする．通常の集合と同様に，Δ に A を加えて得られる集合を $\Delta \cup A$ と書き，A が重複集合 Δ の要素であることを $A \in \Delta$ と書く．仮定の集合 Δ から命題 A が証明できることを，$\Delta \rhd A$ と書くことにする．命題論理学の自然演繹システムは，この証明可能性を表す関係 $\Delta \rhd A$ を導く，以下の形をした規則の集合で与えられる．

$$\frac{\Delta_1 \rhd A_1 \quad \cdots \quad \Delta_n \rhd A_n}{\Delta \rhd A}$$

この規則は，$\Delta_1 \rhd A_1$ から $\Delta_n \rhd A_n$ のすべてが証明可能なら，$\Delta \rhd A$ が証明可能であることを表している．

本書では，純粋な命題論理学の証明規則に加えて，対象領域の性質に応じて導入される，非論理的公理(non logi cal axi om)を考える．Ax を与えられた非論理的公理の集合とする．Ax にはすべての命題定数が含まれると仮定する．

直観主義的命題論理学の規則の集合を図 12.2 に与える．規則 (taut) および (Ax) は，それぞれ Δ に仮定された命題および公理は証明なしで使用してよいことを表している．その他の規則は，各論理演算子 F について，その演算子を導出する (F:I) という名前の規則と，その演算子を使用する (F:E) という名前の規則からなる．例えば規則 ($\supset$:I) は，$A \supset B$ を証明するためには A を仮定に追加して B を証明すればよいことを表しており，規則 ($\supset$:E) は $A \supset B$ と A がともに証明可能なら B が証明可能であることを表している．

$$\text{(taut)}\ \ \Delta \cup A \rhd A \qquad (\text{Ax})\ \ \Delta \rhd a \quad (a \in Ax)$$

$$(\supset\text{:I})\ \frac{\Delta \cup A \rhd B}{\Delta \rhd A \supset B} \qquad (\supset\text{:E})\ \frac{\Delta \rhd A \supset B \quad \Delta \rhd A}{\Delta \rhd B}$$

$$(\wedge\text{:I})\ \frac{\Delta \rhd A \quad \Delta \rhd B}{\Delta \rhd A \wedge B} \qquad (\wedge\text{:E1})\ \frac{\Delta \rhd A \wedge B}{\Delta \rhd A}$$

$$(\wedge\text{:E2})\ \frac{\Delta \rhd A \wedge B}{\Delta \rhd B} \qquad (\vee\text{:I1})\ \frac{\Delta \rhd A}{\Delta \rhd A \vee B}$$

$$(\vee\text{:I2})\ \frac{\Delta \rhd A}{\Delta \rhd B \vee A} \qquad (\vee\text{:E})\ \frac{\Delta \rhd A \vee B \quad \Delta \cup A \rhd C \quad \Delta \cup B \rhd C}{\Delta \rhd C}$$

図 **12.2**　命題論理学の自然演繹証明システム $\mathcal{N}$

この証明システムを $\mathcal{N}$ とよび，$\Delta \rhd A$ がこの証明システムで証明可能であるとき，$\mathcal{N} \vdash \Delta \rhd A$ と書く．$\mathcal{N} \vdash \Delta \rhd A$ なら，A は Δ の論理的帰結であることを容易に確かめることができる．

問 12.7　各命題変数に真(true)か偽(false)を割り当てる関数を η とし，命題 A の η のもとでの真偽値 $[\![A]\!]\eta$ を以下のように定める．

$$[\![a]\!]\eta = true \quad (a \in Ax \text{ のとき})$$

$$[\![\alpha]\!]\eta = \eta(\alpha)$$

$$[\![A \supset B]\!]\eta = \begin{cases} true & [\![A]\!]\eta = false \text{ または } [\![B]\!]\eta = true \text{ のとき} \\ false & \text{上記以外} \end{cases}$$

$$[\![A \wedge B]\!]\eta = \begin{cases} true & [\![A]\!]\eta = true \text{ かつ } [\![B]\!]\eta = true \text{ のとき} \\ false & \text{上記以外} \end{cases}$$

$$[\![A \vee B]\!]\eta = \begin{cases} true & [\![A]\!]\eta = true \text{ または } [\![B]\!]\eta = true \text{ のとき} \\ false & \text{上記以外} \end{cases}$$

任意の η について，もしすべての $B \in \Delta$ について $[\![B]\!]\eta = true$ なら $[\![A]\!]\eta = true$ であるとき，$\Delta \models A$ と書く．$\mathcal{N} \vdash \Delta \rhd A$ なら，$\Delta \models A$ であることを示せ，

問 12.8　論理学において，常に真である式を恒真式とよぶ．問 12.7 の定義を使えば，論理式 A が恒真式であるのは，任意の η に対して $[\![A]\!]\eta = true$ となるとき，と定義できる．

古典的な命題論理学の場合，恒真式はすべて証明可能である．しかし，直観主義的命題論理学では恒真式であっても証明可能でない命題が存在する．その例をあげよ．

(c)　ラムダ計算の型システム

論理学の推論規則は論理的に整合性ある証明の合成方法を記述したものである．証明をプログラムととらえるならば，それらは，整合性あるプログラムを構成する方法の記述とみなすことができる．この見方に従い，ラムダ式の構造が整合性をもつか否かを検査する証明システムを定義することができる．これがラムダ計算の型システムである．

b を整数などの**基底型**(base type)，α を**型変数**(type variable)をそれぞれ代表するメタ変数とする．命題集合に対応して，型の集合をメタ変数 τ を用いて，以下の文法で定義する．

$$\tau ::= b \mid \alpha \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau$$

命題の表記同様，型を表記する際は $\to$ は右結合し，$\times$ と $+$ は $\to$ より結合力が強いと約束する．

ラムダ式に含まれる各定数 $c \in Const$ に対して唯一の型が定義されているものとし，c が型 τ をもつ定数であることを $c : \tau \in Const$ と書く．さらに，基底型 b に対しては，その型をもつ定数 $c : b \in Const$ があると仮定する．

論理学における証明で用いた仮定の集合 Δ は，プログラムの中で使用する変数の型に関する宣言に相当する．これを変数の**型環境**とよび，メタ変数 Γ で表す．Γ は形式的には変数の有限集合から型への関数である．

記法 $\Gamma\{x : \tau\}$，$\Gamma|_X$ および $\Gamma|_{\overline{x}}$ でそれぞれ，Γ を $x : \tau$ で拡張して得られる関数，Γ の定義域を変数の集合 X に制限した関数，および Γ の定義域を $dom(\Gamma) \setminus \{x\}$ に制限してえられる関数を表す．すなわち $\Gamma\{x : \tau\}(x) = \tau$ かつ，$x \neq y$ なる y に対して $\Gamma\{x : \tau\}(y) = \Gamma(y)$ であり，$dom(\Gamma|_X) = X$ かつ $x \in X$ なる y に対して $\Gamma|_X(y) = \Gamma(y)$ であり，$\Gamma|_{\overline{x}} = \Gamma|_{dom(\Gamma)\setminus\{x\}}$ である．

M が Γ のもとで型が τ をもつ型の正しいラムダ式であることを

$$\Gamma \rhd M : \tau$$

と書く．この式を**型判定**とよぶ．ラムダ式の型システムは型判定を導出する証明システムであり，その規則は，型と論理式の対応関係に従い自然演繹システムか

ら機械的に導かれる．図 12.3 に型システムの定義を示す．この証明システムを Λ とよび，Λ で型判定 $\Gamma \rhd M : \tau$ が導かれるとき，$\Lambda \vdash \Gamma \rhd M : \tau$ と書く．ただし，以下 Λ が前後の文脈から明らかな場合は，$\Lambda \vdash \Gamma \rhd M : \tau$ を単に $\Gamma \rhd M : \tau$ と書くことにする．

(const) $\Gamma \rhd c : \tau$ （定数 $c : \tau \in Const$ のとき）

(var) $\Gamma\{x : \tau\} \rhd x : \tau$ (abs) $\dfrac{\Gamma\{x : \tau_1\} \rhd M : \tau_2}{\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2}$

(app) $\dfrac{\Gamma \rhd M_1 : \tau_1 \to \tau_2 \quad \Gamma \rhd M_2 : \tau_1}{\Gamma \rhd M_1\ M_2 : \tau_2}$

(prod) $\dfrac{\Gamma \rhd M_1 : \tau_1 \quad \Gamma \rhd M_2 : \tau_2}{\Gamma \rhd (M_1, M_2) : \tau_1 \times \tau_2}$

(proj1) $\dfrac{\Gamma \rhd M : \tau_1 \times \tau_2}{\Gamma \rhd M[1] : \tau_1}$ (proj2) $\dfrac{\Gamma \rhd M : \tau_1 \times \tau_2}{\Gamma \rhd M[2] : \tau_2}$

(inj1) $\dfrac{\Gamma \rhd M : \tau_1}{\Gamma \rhd 1(M) : \tau_1 + \tau_2}$ (inj2) $\dfrac{\Gamma \rhd M : \tau_2}{\Gamma \rhd 2(M) : \tau_1 + \tau_2}$

(case) $\dfrac{\Gamma \rhd M : \tau_1 + \tau_2 \quad \Gamma\{x_1 : \tau_1\} \rhd M_1 : \tau_3 \quad \Gamma\{x_2 : \tau_2\} \rhd M_2 : \tau_3}{\Gamma \rhd (\mathbf{case}\ M\ \mathbf{of}\ 1(x_1) \Rightarrow M_1,\ 2(x_2) \Rightarrow M_2) : \tau_3}$

(fix) $\dfrac{\Gamma \rhd M : \tau \to \tau}{\Gamma \rhd \mathbf{fix}(M) : \tau}$ (let) $\dfrac{\Gamma \rhd M_1 : \tau_1 \quad \Gamma\{x : \tau_1\} \rhd M_2 : \tau_2}{\Gamma \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau_2}$

図 **12.3** ラムダ計算の型導出システム

(const) から (case) までの規則は，自然演繹システムにおける (taut) から (∨:E) までの規則に対応している．規則 (fix) は，各 τ に対して型 $(\tau \to \tau) \to \tau$ をもつ定数 $fix^{(\tau\to\tau)\to\tau}$ の存在を仮定するのと同値であり，型理論的には定数の推論規則に含まれる．また規則 (let) は，$\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ を $(\lambda x.M_2)\ M_1$ の別記法とみなせば，規則 (abs) と規則 (app) を組合せたものに他ならない．したがって，この型システムの定義は前節の自然演繹システムと同一であることがわかる．

定数のもつ型の集合 $\{\tau \mid c : \tau \in Const\}$ と命題論理学の非論理的公理の集合 Ax の間，および型変数の集合と命題変数の集合の間にそれぞれ 1 対 1 の対応関係があれば，型の集合と命題の集合は，前述の対応関係によって 1 対 1 に対応する．この仮定の下で，命題 A に対応する型を τ_A と書き，型 τ に対応する命題を

A_τ と書く．与えられた仮定のリスト Δ に対して，Γ_Δ を以下のように定義する．

$$\Gamma_\Delta = \{x_i : \tau_A \mid A \in \Delta,\ x_i \text{ はすべて異なる}\}$$

また与えられた型環境 Γ に対してその変数を消去し，型を対応する命題で置き換えて得られる型の重複集合を Δ_Γ と書く．Δ の中の命題の順序は同一視するから Δ_Γ は唯一定まる．

以下の定理は **Curry-Howard 同型関係**として知られている性質の 1 つである．

定理 12. 2 もし $\mathcal{N} \vdash \Delta \rhd A$ ならある M があって $\Lambda \vdash \Gamma_\Delta \rhd M : \tau_A$ である．逆に $\Lambda \vdash \Gamma \rhd M : \tau$ なら $\mathcal{N} \vdash \Delta_\Gamma \rhd A_\tau$ である． □

この性質は両者の証明システムの規則が 1 対 1 に対応していることからほぼ明らかであろう．

このような型判定に関する性質を厳密に証明するためには，多くの場合帰納法が用いられる．式や証明などの構文論的な対象は，それらを生成する規則を有限回使って得られたものである．そこで，これら構文論的対象の大きさを，その生成に使用された規則の回数と定義する．大きさは 1 以上であるから，この大きさに関して帰納法を使用することができる．ラムダ式の大きさに関する帰納法を，**式の構造に関する帰納法**，型判定の証明の大きさに関する帰納法を，**型判定の導出に関する帰納法**とよぶ．ラムダ式の型判定に関する多くの性質は，このいずれかによって証明可能な場合が多い．読者は以下の証明を吟味し，その方法を習得することを勧める．

補題 12. 3 もし $\Gamma \rhd M : \tau$ なら，$FV(M) \subseteq dom(\Gamma)$ である．

［証明］$\Gamma \rhd M : \tau$ の導出に関する帰納法によって証明する．$\Gamma \rhd M : \tau$ の導出の最後に使用された推論規則により，場合分けを行なう．

(const) の場合． $\Gamma \rhd c : \tau$ であり，$FV(c) = \emptyset \subseteq dom(\Gamma)$ である．

(var) の場合． $M \equiv x$ かつ $\Gamma = \Gamma'\{x : \tau\}$ であり，$FV(x) = \{x\} \subseteq dom(\Gamma'\{x : \tau\})$ である．

(abs) の場合． $M \equiv \lambda x.M_1$ かつ，ある τ_1, τ_2 があって $\tau = \tau_1 \to \tau_2$，$\Gamma\{x : \tau_1\} \rhd M_1 : \tau_2$ である．帰納法の仮定より，$FV(M_1) \subseteq dom(\Gamma\{x : \tau_1\})$ である．したがって $FV(\lambda x.M_1) = FV(M_1) \setminus \{x\} \subseteq dom(\Gamma)$ である．

(app) の場合． $M \equiv M_1\ M_2$ かつ，ある τ_1 があって $\Gamma \rhd M_1 : \tau_1 \to \tau$，$\Gamma \rhd M_2 : \tau_1$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$ かつ $FV(M_2) \subseteq dom(\Gamma)$ である．よって $FV(M_1\ M_2) \subseteq dom(\Gamma)$ である．

(prod) の場合． $M \equiv (M_1, M_2)$ かつ，ある τ_1, τ_2 があって $\tau = \tau_1 \times \tau_2$，$\Gamma \rhd M_1 : \tau_1$ かつ $\Gamma \rhd M_2 : \tau_1$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$ かつ $FV(M_1) \subseteq dom(\Gamma)$ である．よって $FV((M_1, M_2)) \subseteq dom(\Gamma)$ である．

(proj1) の場合． $M \equiv M_1[1]$ かつ，ある τ_2 があって $\Gamma \rhd M_1 : \tau \times \tau_2$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$ である．よって $FV(M_1[1]) = FV(M_1) \subseteq dom(\Gamma)$ である．(proj2) の場合も同様である．

(inj1) の場合． $M \equiv 1(M_1)$，ある τ_1, τ_2 があって $\tau = \tau_1 + \tau_2$ かつ $\Gamma \rhd M_1 : \tau_1$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$ である．よって $FV(1(M_1)) = FV(M_1) \subseteq dom(\Gamma)$ である．(inj2) の場合も同様である．

(case) の場合． $M \equiv (\mathbf{case}\ M_0\ \mathbf{of}\ 1(x_1) \Rightarrow M_1,\ 2(x_2) \Rightarrow M_2)$ かつ，ある τ_1, τ_2 があって $\Gamma \rhd M_0 : \tau_1 + \tau_2$，$\Gamma\{x_1 : \tau_1\} \rhd M_1 : \tau$，$\Gamma\{x_2 : \tau_2\} \rhd M_2 : \tau$ である．帰納法の仮定より $FV(M_0) \subseteq dom(\Gamma)$, $FV(M_1) \subseteq dom(\Gamma\{x_1 : \tau_1\})$, $FV(M_2) \subseteq dom(\Gamma\{x_2 : \tau_2\})$ である．よって

$$FV(M) = FV(M_0) \cup (FV(M_1) \setminus \{x_1\}) \cup (FV(M_2) \setminus \{x_2\}) \subseteq dom(\Gamma)$$

である．

(fix) の場合． $M \equiv \mathbf{fix}(M_1)$ かつ $\Gamma \rhd M_1 : \tau \to \tau$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$ である．よって $FV(\mathbf{fix}(M_1)) = FV(M_1) \subseteq dom(\Gamma)$ である．

(let) の場合． $M \equiv \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ かつ，ある τ_1 があって $\Gamma \rhd M_1 : \tau_1$ かつ $\Gamma\{x : \tau_1\} \rhd M_2 : \tau$ である．帰納法の仮定より $FV(M_1) \subseteq dom(\Gamma)$, $FV(M_2) \subseteq dom(\Gamma\{x : \tau_1\})$ である．よって

$$FV(\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) = FV(M_1) \cup (FV(M_2) \setminus \{x\}) \subseteq dom(\Gamma)$$

である． ∎

問 12.9 上記の証明にならい，定理 12.2 を証明せよ．

以下の 2 つの補題も同様に証明できる．

補題 12.4 もし $\Gamma \rhd M : \tau$ かつ $x \notin dom(\Gamma)$ なら $\Gamma\{x : \tau'\} \rhd M : \tau$ である． □

補題 12.5 もし $\Gamma \rhd M : \tau$ かつ $FV(M) \subseteq X$ なら $\Gamma|_X \rhd M : \tau$ である． □

$y \notin dom(\Gamma)$ のとき，Γ の領域の要素 x を y に替えて得られる関数を $[y/x]\Gamma$ と書く．厳密には以下のように定義される．

$$[y/x]\Gamma = \begin{cases} \Gamma & (x \notin dom(\Gamma) \text{ のとき}) \\ \Gamma|_{\overline{x}}\{y : \Gamma(x)\} & (x \in dom(\Gamma) \text{ のとき}) \end{cases}$$

変数の名前替えに関して，以下の性質が成り立つ．

補題 12.6 もし $\Gamma \rhd M : \tau$ かつ $y \notin dom(\Gamma)$ なら，$[y/x]\Gamma \rhd [y/x]M : \tau$ である．

［証明］式 M の構造に関する帰納法による．ラムダ抽象以外の場合は練習問題とする．

$\lambda x.M_1$ の場合． $x \notin dom(\Gamma)$ のときは $[y/x]\Gamma = \Gamma$，$[y/x](\lambda x.M_1) = \lambda x.M_1$ であり，成立する．$x \in dom(\Gamma)$ と仮定する．$x \notin FV(\lambda x.M_1)$ であるから，補題 12.5 より，$\Gamma|_{\overline{x}} \rhd \lambda x.M_1 : \tau$ である．よって，補題 12.4 より $\Gamma|_{\overline{x}}\{y : \Gamma(x)\} \rhd \lambda x.M_1 : \tau$，すなわち，$[y/x]\Gamma \rhd [y/x](\lambda x.M_1) : \tau$ である．

$\lambda z.M_1$，$z \neq x, z \neq y$ の場合． 型システムの定義より，ある τ_1, τ_2 があって $\tau = \tau_1 \to \tau_2$ かつ $\Gamma\{z : \tau_1\} \rhd M_1 : \tau_2$ である．$[y/x](\Gamma\{z : \tau_1\}) = ([y/x]\Gamma)\{z : \tau_1\}$ であるから，帰納法の仮定より $([y/x]\Gamma)\{z : \tau_1\} \rhd [y/x]M_1 : \tau_2$ である．規則 (abs) より $[y/x]\Gamma \rhd \lambda z.[y/x]M_1 : \tau_2$ である．また $[y/x](\lambda z.M_1) = \lambda z.[y/x]M_1$ である．

$\lambda y.M_1$，$x \neq y$ の場合． 型システムの定義より，ある τ_1, τ_2 があって $\tau = \tau_1 \to \tau_2$ かつ $\Gamma\{y : \tau_1\} \rhd M_1 : \tau_2$ である．w を新しい変数とする．帰納法の仮定より，$[w/y](\Gamma\{y : \tau_1\}) \rhd [w/y]M_1 : \tau_2$ である．$y \notin dom(\Gamma)$ であるから，$[w/y](\Gamma\{y : \tau_1\}) = \Gamma\{w : \tau_1\}$ であり，$\Gamma\{w : \tau_1\} \rhd [w/y]M_1 : \tau_2$ であ

る．$y \notin dom(\Gamma\{w:\tau_1\})$ であり，また，$[w/y]M_1$ の大きさは M_1 と等しいから，$[w/y]M_1$ に対しても帰納法の仮定が使えて，$[y/x](\Gamma\{w:\tau_1\}) \rhd [y/x]([w/y]M_1) : \tau_2$，すなわち $([y/x]\Gamma)\{w:\tau_1\} \rhd [y/x]([w/y]M_1) : \tau_2$ である．規則 (abs) より $[y/x]\Gamma \rhd \lambda w.[y/x]([w/y]M_1) : \tau_1 \to \tau_2$ である．ラムダ式の代入の定義より，$[y/x]\Gamma \rhd [y/x](\lambda y.M_1) : \tau_1 \to \tau_2$ である． ∎

問 12.10 補題 12.6 証明の残りの場合を示し，証明を完成せよ．

以上の性質を用いて，以下の重要な性質が証明できる．

補題 12.7（型に関する代入補題）もし $\Gamma\{x:\tau_1\} \rhd M_1 : \tau_2$ かつ $\Gamma \rhd M_2 : \tau_1$ なら，$\Gamma \rhd [M_2/x]M_1 : \tau_2$ である．

［証明］M_1 の構造に関する帰納法により証明する．以下，変数とラムダ抽象の場合を示す．

x の場合． 型システムの定義より $\tau_1 = \tau_2$ であるから，$\Gamma \rhd M_2 : \tau_2$，すなわち $\Gamma \rhd [M_2/x]x : \tau_2$．

y $(y \neq x)$ の場合． 補題 12.4 および 12.5より $\Gamma \rhd y : \tau_2$，すなわち $\Gamma \rhd [M_2/x]y : \tau_2$．

$\lambda x.M_0$ の場合．$x \notin FV(\lambda x.M)$ であるから，補題 12.5 より，$\Gamma \rhd \lambda x.M_0 : \tau_2$，すなわち $\Gamma \rhd [M_2/x](\lambda x.M_0) : \tau_2$ である．

$\lambda y.M_0$, $x \neq y$ かつ $y \notin FV(M_2)$ の場合． 型システムの定義より，ある τ_2^1, τ_2^2 があって，$\tau_2 = \tau_2^1 \to \tau_2^2$，$\Gamma\{x:\tau_1\}\{y:\tau_2^1\} \rhd M_0 : \tau_2^2$ である．$x \neq y$ であるから $\Gamma\{x:\tau_1\}\{y:\tau_2^1\} = \Gamma\{y:\tau_2^1\}\{x:\tau_1\}$．補題 12.4 および 12.5 より $\Gamma\{y:\tau_2^1\} \rhd M_2 : \tau_1$．よって帰納法の仮定より，$\Gamma\{y:\tau_2^1\} \rhd [M_2/x]M_0 : \tau_2^2$．規則 (abs) より，$\Gamma \rhd \lambda y.[M_2/x]M_0 : \tau_2$，すなわち $\Gamma \rhd [M_2/x](\lambda y.M_0) : \tau_2$ が成り立つ．

$\lambda y.M_0$, $x \neq y$, $y \in FV(M_2)$ の場合． 型システムの定義より，ある τ_2^1, τ_2^2 があって，$\tau_2 = \tau_2^1 \to \tau_2^2$，$\Gamma\{x:\tau_1\}\{y:\tau_2^1\} \rhd M_0 : \tau_2^2$ である．z をいずれの型判定にも現れない新しい変数とする．補題 12.6 より，$\Gamma|_{\overline{y}}\{x:\tau_1\}\{z:\tau_2^1\} \rhd [z/y]M_0 : \tau_2^2$．$z \neq x$ であるから，$\Gamma|_{\overline{y}}\{z:\tau_2^1\}\{x:\tau_1\} \rhd [z/y]M_0 : \tau_2^2$．補題 12.4 より $\Gamma\{z:\tau_2^1\}\{x:\tau_1\} \rhd [z/y]M_0 : \tau_2^2$．$z \notin FV(M_2)$ であるから，補題 12.4 および 12.5より，$\Gamma\{z:\tau_2^1\} \rhd M_2 : \tau_1$．また，式 $[z/y]M_0$ の大きさは M_0 と同一であるから，$[z/y]M_0$ に対する帰納法の仮定より，$\Gamma\{z:\tau_2^1\} \rhd [M_2/x]([z/y]M_0) : \tau_2^2$．規則 (abs) より $\Gamma \rhd \lambda z.[M_2/x]([z/y]M_0) : \tau_2$．式の代入の定義より，$\Gamma \rhd [M_2/x](\lambda y.M_0) : \tau_2$． ∎

問 12.11 補題 12.7 の証明の残りの場合を示し，証明を完成せよ．

(d) 束縛変数と変数の静的スコープ規則

ラムダ式は α 同値性のもとで考える．したがって，型システムが妥当であるためには，α 同値なラムダ式はすべて同一の型をもたねばならない．以下の命題はこの性質を保証するものである．

命題 12.8 $\Gamma \rhd \lambda x.M : \tau$ かつ $y \notin FV(M)$ なら $\Gamma \rhd \lambda y.[y/x]M : \tau$ である． □

さらに型システムの定義の形から，部分の型がすべて等しければ全体の型も等しいことを容易に確かめることができる．したがって以下の性質が言える．

命題 12.9 もし $\Gamma \rhd M : \tau$ かつ $M \equiv N$ なら $\Gamma \rhd N : \tau$ である． □

問 12.12 命題 12.8 と 12.9 を証明せよ．

命題 12.9 より，型に関しても α 同値なラムダ式は同一視してよいことが保証される．このことから，プログラミング言語のモデルである型付きラムダ式に対しても，以前定義した束縛変数に関する仮定(仮定 12.1)をしても構わないことがわかる．この性質に基づき，ラムダ式を α 同値性に関する同値類とみなし，束縛変数に関する仮定を満たすものを，その代表元として使用することがよく行なわれる．

α 同値なラムダ式は同一の意味をもつから，ラムダ計算の型システムをこのように定義しても，理論的には一般性を失うことはない．しかしながら，α 同値性を型システムの定義以前に仮定するシステムは，プログラミング言語の構文論的モデルとしては必ずしも妥当ではない．

ML のように(関数)定義を入れ子にできる言語を，**ブロック構造**(block structure)をもつ言語とよぶ．従来のプログラミング言語である C や PASCAL などを

含めたほとんどすべての高水準言語は，ブロック構造をもつ言語である．ブロック構造をもつ言語では，同一名の変数を複数回定義することができる．したがって結果として構成されるプログラムは，同一の束縛変数が複数現れることがあり，束縛変数の仮定を満たさない．

このような場合，同一名の変数の定義の有効範囲を重なりなく決める規則が必要となる．変数の有効範囲を，その変数の**スコープ**(scope)とよぶ．通常のプログラミング言語では，変数定義の有効範囲を，その構文構造から決まる有効範囲の中で同一の名前が再定義されている部分を除く範囲と定める．この規則を，変数の**静的スコープ規則**とよぶ．この静的スコープ規則においては，変数を囲む最も内側の定義が有効となる．

例えば以下の Minimal^0 の関数定義を見てみよう．

```
(fn x => fn y => (x,(fn (x,z) => x * z) (x,y))) 2 3
```

この例では x が 2 回定義され 3 回使用されているが，静的スコープ規則の下では最初と最後の使用に対しては最初の定義が有効であり，2 回目の使用に対しては 2 回目の定義が有効である．この静的スコープ規則は，プログラムを階層的に作成していく上で有効であり，ほとんどすべてのブロック構造をもつ言語で採用されている．

α 同値性を最初から仮定しない Λ の型システムの定義は，以上の変数の静的スコープ規則を正確に表現している．束縛変数を導入する規則における記法 $\Gamma\{x:\tau\}$ は，Γ にすでに x に対する仮定があれば，それを無効にし新たな仮定を追加する効果がある．これによって静的スコープ規則が実現される．例えば，上記のプログラムは以下のラムダ式で表現される．

$$(\lambda x.\lambda y.(x,(\lambda x.mul(x[1],x[2]))(x,y)))\ 2\ 3$$

この式に対する型の導出を考えればわかるとおり，2 つ目の λx の内側に現れる $x[1]$ と $x[2]$ の中の x は $int \times int$ の仮定の下で型チェックされ，内側の λx の定義に対応した変数として扱われている．

問 12.13 上記ラムダ式の型の導出を計算し，この性質を確認せよ．

しかしながら，型チェック済みのプログラムの意味などを論じる場合は，束縛変数の名前は重要ではない．そこでこれ以降の議論において，プログラムの束縛変数に関して，以前説明した仮定(仮定 12.1)をする．この仮定のもとでは，規則 (abs) やその他の束縛変数を導入する規則において導入される束縛変数は，自由変数の型を与える型環境 Γ に現れないから，これら規則における記法 $\Gamma\{x:\tau\}$ は，Γ に，$x \notin dom(\Gamma)$ なる変数 x に対する型宣言 $x:\tau$ を追加して得られる型環境としてよい．

§12.4 ラムダ計算の意味論

型理論的分析に基づくそのプログラムの型の整合性制約によって，プログラミング言語の構文構造は厳密に定義される．すなわちある Γ,τ があって $\Gamma \rhd M : \tau$ となる型の正しいラムダ式が，正しく書かれたプログラムである．プログラミング言語の定義を完成させるためには，このような型の正しいラムダ式の意味を定義する必要がある．

第 10 章で概説したとおり，広い意味でのラムダ計算の意味論には公理的意味論，表示的意味論および操作的意味論の 3 つがある．この中で操作的意味論はプログラミング言語の実装に深く関係しており，意図する実装のモデルによって種々の定義が可能である．そこで操作的意味論は第 14 章の実装方式のところで解説することとし，以下，本章では公理的意味論と表示的意味論の枠組の概要を解説する．

(a) β 簡約関係と公理的意味論

ラムダ計算は計算可能な関数のモデルである．数学におけるグラフとしての関数と違い，関数が値を計算する過程が，ラムダ式を別のラムダ式に変形する操作としてモデル化される．変形規則は，ラムダ計算の各式構成子の意図する意味にしたがって，図 12.4 のように与えられる．

(β) は関数の仮引数を実引数で置き換える，関数適用の自然な規則である．(fix) は，各種繰り返し構文や再帰関数を実現するための規則である．その他の規則はそれぞれ直積型および直和型データの操作を表現する自然な規則である．

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

$$(\beta)\ (\lambda x.M)\ N \Longrightarrow [N/x]M$$
$$(\text{fst})\ (M_1, M_2)[1] \Longrightarrow M_1$$
$$(\text{snd})\ (M_1, M_2)[2] \Longrightarrow M_2$$
$$(\text{case1})\ \mathbf{case}\ 1(M_1)\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3) \Longrightarrow [M_1/x]M_2$$
$$(\text{case2})\ \mathbf{case}\ 2(M_1)\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3) \Longrightarrow [M_1/y]M_3$$
$$(\text{fix})\ \mathbf{fix}(M) \Longrightarrow (M\ \mathbf{fix}(M))$$

図 **12.4**　ラムダ計算の変形規則

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．この関係を簡潔に表現するためにラムダ式の**文脈**を定義する．文脈とは，注目するラムダ式を囲むラムダ式のことであり，形式的にはラムダ式を埋め込むべき穴を含むラムダ式である．文脈は以下の BNF 文法で与えられる．

$$\begin{aligned} C ::= {} & [\ ] \mid (\lambda x.C) \mid (C\ M) \mid (M\ C) \mid (C, M) \mid (M, C) \mid C[1] \mid C[2] \\ & \mid 1(C) \mid 2(C) \mid (\mathbf{case}\ C\ \mathbf{of}\ 1(x) \Rightarrow M,\ 2(x) \Rightarrow M) \\ & \mid (\mathbf{case}\ M\ \mathbf{of}\ 1(x) \Rightarrow C,\ 2(x) \Rightarrow M) \\ & \mid (\mathbf{case}\ M\ \mathbf{of}\ 1(x) \Rightarrow M,\ 2(x) \Rightarrow C) \\ & \mid \mathbf{let}\ x = C\ \mathbf{in}\ M \mid \mathbf{let}\ x = M\ \mathbf{in}\ C \mid \mathbf{fix}(C) \end{aligned}$$

ここで $[\ ]$ はラムダ式が埋め込まれるべき穴を表す．文脈 C の穴にラムダ式 M を埋め込んで得られるラムダ式を $C[M]$ と書く．1 ステップ簡約関係を，この文脈を使い，以下のように定義する．

$$\frac{M \Longrightarrow N}{C[M] \longrightarrow C[N]}$$

ラムダ計算の表現する計算の基本である**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．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みなすことができる．

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

命題 12.10 任意のラムダ式 M について，$M \Downarrow N_1$ かつ $M \Downarrow N_2$ なら，$N_1 \equiv N_2$ である．□

この命題は，以下の定理の直接の帰結である．

定理 12.11（**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 $N_3 \equiv N_4$ 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．

問 12.14　命題 12.10 が定理 12.11 の直接の帰結であることを確かめよ．

ラムダ式の公理的意味論は，この簡約関係を用いて定義される．われわれの興味の対象は，型付けされたプログラムの意味である．そこで，型付きのラムダ式，すなわち型判定 $\Gamma \rhd M : \tau$ に関する公理的意味論を定義する．型付きラムダ式 $\Gamma \rhd M : \tau$ と $\Gamma \rhd N : \tau$ が等しいことを

$$\Gamma \rhd M = N : \tau$$

と書く．型付きラムダ計算の公理的意味論は，この形の式を導出する証明システムである．推論規則の集合を図 12.5 に与える．規則 (axiom) は変換可能なラムダ式が等しいことを表している．規則 (ref)，(sym)，(trans) は等式理論における反射律，対称律，推移律に対応する規則である．規則 (cong) は部分が等しい式は等しいことを表している．

(b) ラムダ式の表示的意味論

ラムダ式の表示的意味を定義するためには，ラムダ式が表示すべき値の集合である意味領域を定義する必要がある．われわれが定義したラムダ計算は型システ

$$(\text{axiom})\ \frac{\Gamma \rhd M_1 : \tau \quad \Gamma \rhd M_2 : \tau \quad M_1 \Longrightarrow M_2}{\Gamma \rhd M_1 = M_2 : \tau}$$

$$(\text{ref})\ \Gamma \rhd M = M : \tau$$

$$(\text{sym})\ \frac{\Gamma \rhd M_1 = M_2 : \tau}{\Gamma \rhd M_2 = M_1 : \tau}$$

$$(\text{trans})\ \frac{\Gamma \rhd M_1 = M_2 : \tau \quad \Gamma \rhd M_2 = M_3 : \tau}{\Gamma \rhd M_1 = M_3 : \tau}$$

$$(\text{cong})\ \frac{\Gamma \rhd M = N : \tau}{\Gamma \rhd C[M] = C[N] : \tau'}$$

図 **12.5**　ラムダ式の公理的意味論

ムをもつシステムであるから，そのためには各型 τ に対応する値の集合 A^τ を定義しなければならない．Λ の型の集合には型変数が含まれている．型変数 α は任意の型を代表する型であり，その意味領域 A^α は α の解釈に依存する．そこで，意味領域を，型変数を含まない型についてのみ定義し，型変数を含む型の領域はその中の型変数の解釈によって与えられる型の領域と考える．

Types を以下の文法で与えられる型変数を含まない型の集合とする．

$$\rho ::= b \mid \rho \to \rho \mid \rho \times \rho \mid \rho + \rho$$

この型を**基礎型**とよぶ．型の意味から，各基礎型は以下のような集合に対応すると考えられる．

$$\begin{aligned}
A^b &\ :\ \text{基底型 } b \text{ をもつ定数の集合}\\
A^{\rho_1 \to \rho_2} &\ :\ A^{\rho_1} \text{ から } A^{\rho_2} \text{ への関数の集合}\\
A^{\rho_1 \times \rho_2} &\ :\ A^{\rho_1} \text{ と } A^{\rho_2} \text{ の直積集合 } A^{\rho_1} \times A^{\rho_2}\\
A^{\rho_1 + \rho_2} &\ :\ A^{\rho_1} \text{ と } A^{\rho_2} \text{ の直和集合 } A^{\rho_1} + A^{\rho_2}
\end{aligned}$$

しかし残念ながら，このような単純な集合論的構造のみでは，Λ の式の意味を解釈するための意味領域は構成できない．その主な理由は，このようにして構成された集合では $\mathbf{fix}(M)$ を解釈できないことである．公理的意味論から，$\mathbf{fix}(M) = M(\mathbf{fix}(M))$が型 $\rho \to \rho$ をもつ任意の式 M に対して成立しなければならない．もし M が関数であるなら，$\mathbf{fix}(M)$ はその関数を適用しても変化しない点，すなわち関数 M の不動点である．したがって，この条件は，領域 $A^{\rho \to \rho}$ に属する関数はすべて不動点をもつことを要求している．しかしながら集合論的に定義可能な関数は，一般に不動点をもつとは限らない．例えば与えられた自然数に 1 を加える関数などは，通常の集合論的解釈のもとでは不動点はもたない．

Λ の意味領域を構築するためには，$A^{\rho \to \rho}$ を，この型をもちうるすべてのラムダ式の意味を含むほど大きく，かつこの領域に属する関数がすべて不動点をもつような集合に制限する必要がある．このような条件を満たす Λ の意味領域を構成する問題は，それ自身興味深い重要な問題であるが，このテーマは本シリーズの姉妹編の『コンピュータサイエンス入門 論理とプログラム意味論』で詳しく扱われているので，本章では意味領域の構築は行なわず，意味領域の満たすべき条件と，それに基づき導かれる意味の定義の性質の概要を述べるにとどめる．

Λ の意味領域を以下の性質を満たす型でインデックスされた集合の集合

$$\mathcal{A} = \{A^\rho \mid \rho \in \mathit{Types}\}$$

と定義する．

(i) $A^{\rho_1 \to \rho_2}$ は A^{ρ_1} から A^{ρ_2} への関数の集合である．

(ii) 任意の ρ_1 と ρ_2 に対して 3 つの関数

$$\begin{aligned}
\mathit{Prod}^{\rho_1,\rho_2} &\in A^{\rho_1} \times A^{\rho_2} \to A^{\rho_1 \times \rho_2}\\
\mathit{Proj}_1^{\rho_1,\rho_2} &\in A^{\rho_1 \times \rho_2} \to A^{\rho_1}\\
\mathit{Proj}_2^{\rho_1,\rho_2} &\in A^{\rho_1 \times \rho_2} \to A^{\rho_2}
\end{aligned}$$

が存在して，任意の $x \in A^{\rho_1}$, $y \in A^{\rho_2}$ に対して以下の等式が成立する

$$\begin{aligned}
\mathit{Proj}_1^{\rho_1,\rho_2}(\mathit{Prod}^{\rho_1,\rho_2}(x,y)) &= x\\
\mathit{Proj}_2^{\rho_1,\rho_2}(\mathit{Prod}^{\rho_1,\rho_2}(x,y)) &= y
\end{aligned}$$

(iii) 任意の ρ_1, ρ_2, ρ_3 に対して以下の 3 つの関数が存在して

$$\begin{aligned}
\mathit{Case}^{\rho_1,\rho_2,\rho_3} &\in A^{\rho_1+\rho_2} \times A^{\rho_1 \to \rho_3} \times A^{\rho_2 \to \rho_3} \to A^{\rho_3}\\
\mathit{Inj}_1^{\rho_1,\rho_2} &\in A^{\rho_1} \to A^{\rho_1+\rho_2}\\
\mathit{Inj}_2^{\rho_1,\rho_2} &\in A^{\rho_2} \to A^{\rho_1+\rho_2}
\end{aligned}$$

任意の $x \in A^{\rho_1}$, $y \in A^{\rho_2}$, $f \in A^{\rho_1 \to \rho_3}$, $g \in A^{\rho_2 \to \rho_3}$ に対して以下の等式が成立する

$$Case^{\rho_1,\rho_2,\rho_3}\ (Inj_1^{\rho_1,\rho_2}(x), f, g) = f(x)$$
$$Case^{\rho_1,\rho_2,\rho_3}\ (Inj_2^{\rho_1,\rho_2}(y), f, g) = g(y)$$

(iv) 任意の ρ に対して，不動点演算子

$$Fix^{\rho} \in A^{\rho \to \rho} \to A^{\rho}$$

が存在して任意の $f \in A^{\rho \to \rho}$ に対して以下の等式が成立する

$$Fix^{\rho}(f) = f\ (Fix^{\rho}(f))$$

(v) 任意の $c : \tau \in Const$ に対して $\bar{c} \in A^{\tau}$ が存在する．

つぎに意味領域 $\mathcal{A}$ のもとでの Λ の式の意味の満たすべき条件を定義する．最も基本的な条件は，型 τ をもつ式の意味は領域 A^{τ} の要素となることである．型 τ は一般に型変数を含んでいるので，τ に対応する領域は τ の型変数のとる値に依存する．型変数の解釈関数 ϕ を型変数の集合から $Types$ への関数とする．ϕ の下での τ の意味 $[\![\tau]\!]\phi$ を，τ の中の型変数 α を $\phi(\alpha)$ で置き換えて得られる基礎型と定義する．以下の議論では ϕ は 1 つ固定されているものとし，$[\![\tau]\!]\phi$ を単に τ と書くことにする．

式は自由変数を含んでいるので，式の意味はその式に含まれる自由変数に代入される値に依存する．$\mathcal{A}$ を意味領域としたとき，自由変数に $\mathcal{A}$ の要素を対応させる $\mathcal{A}$ **環境** η を，変数の集合 Var の部分集合から $\bigcup\{A^{\rho} \mid \rho \in Types\}$ への関数とする．$\mathcal{A}$ 環境 η が条件，

$$dom(\eta) = dom(\Gamma) \text{ かつ，任意の } x \in dom(\eta) \text{ について } \eta(x) \in A^{\Gamma(x)}$$

を満たすとき η は Γ を満たすといい，$\eta \models \Gamma$ と書く．$\Gamma \rhd M : \tau$ を任意の型判定，η を $\eta \models \Gamma$ なる $\mathcal{A}$ 環境とする．意味領域 $\mathcal{A}$ における η のもとでの Λ の型判定 $\Gamma \rhd M : \tau$ の意味を $\mathcal{A}[\![\Gamma \rhd M : \tau]\!]\eta$ と書く．型判定の意味は，型判定の導出に関して再帰的に図 12.6 のように与えられる．

$$\begin{aligned}
\mathcal{A}[\![\Gamma \rhd c : \tau]\!]\eta &= \bar{c} \\
\mathcal{A}[\![\Gamma \rhd x : \tau]\!]\eta &= \eta(x) \\
\mathcal{A}[\![\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2]\!]\eta &= v \in A^{\tau_1} \text{ に } [\![\Gamma\{x:\tau_1\} \rhd M : \tau_2]\!]\eta\{x:v\} \\
&\qquad \text{を対応させる関数} \\
\mathcal{A}[\![\Gamma \rhd M_1\ M_2 : \tau]\!]\eta &= \mathcal{A}[\![\Gamma \rhd M_1 : \tau_1 \to \tau]\!]\eta(\mathcal{A}[\![\Gamma \rhd M_2 : \tau_1]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd (M_1, M_2) : \tau_1 \times \tau_2]\!]\eta &= Prod^{\tau_1,\tau_2}(\mathcal{A}[\![\Gamma \rhd M_1 : \tau_1]\!]\eta, \\
&\qquad \mathcal{A}[\![\Gamma \rhd M_2 : \tau_2]\!]\eta\ ) \\
\mathcal{A}[\![\Gamma \rhd M[1] : \tau_1]\!]\eta &= Proj_1^{\tau_1,\tau_2}(\mathcal{A}[\![\Gamma \rhd M : \tau_1 \times \tau_2]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd M[2] : \tau_2]\!]\eta &= Proj_2^{\tau_1,\tau_2}(\mathcal{A}[\![\Gamma \rhd M : \tau_1 \times \tau_2]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd 1(M) : \tau_1 + \tau_2]\!]\eta &= Inj_1^{\tau_1,\tau_2}(\mathcal{A}[\![\Gamma \rhd M : \tau_1]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd 2(M) : \tau_1 + \tau_2]\!]\eta &= Inj_2^{\tau_1,\tau_2}(\mathcal{A}[\![\Gamma \rhd M : \tau_2]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd (\mathbf{case}\ M_1\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3) : \tau_3]\!]\eta & \\
&= Case^{\tau_1,\tau_2,\tau_3} \\
&\qquad (\mathcal{A}[\![\Gamma \rhd M_1 : \tau_1 + \tau_2]\!]\eta, \\
&\qquad \mathcal{A}[\![\Gamma \rhd \lambda x.M_2 : \tau_1 \to \tau_3]\!]\eta, \\
&\qquad \mathcal{A}[\![\Gamma \rhd \lambda y.M_3 : \tau_2 \to \tau_3]\!]\eta) \\
\mathcal{A}[\![\Gamma \rhd \mathbf{fix}(M) : \tau]\!]\eta &= Fix^{\tau}(\mathcal{A}[\![\Gamma \rhd M : \tau \to \tau]\!]\eta)
\end{aligned}$$

図 12.6 ラムダ計算の表示的意味論の定義

この意味の定義が常に可能であるとは限らない．それは関数型が必要な関数を含んでいるとは限らないためである．$\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2$ の形の任意の型判定に対してその意味に対応する関数が領域 $A^{\tau_1 \to \tau_2}$ に存在すると，意味の定義に関する上記の等式は，型判定の導出に関する再帰的な定義となる．このとき，意味領域 $\mathcal{A}$ を Λ のモデルとよぶ．

この意味の定義は型判定の導出に対するものであるが，同一の型判定に対する導出の意味は等しいことを示すことができる．したがって，型判定 $\Gamma \rhd M : \tau$ と Γ を満たす $\mathcal{A}$ 環境の任意の組に対して唯一の意味が定義される．

(c) 公理的意味論と表示的意味論の関係

表示的意味論は，式の意味を数学的な対象として直接定義する意味論であり，論理学におけるモデル論に対応する．これに対して公理的意味論は，言語の文法構造に従って定義された構文論的意味論であり，論理学における証明論に対応する．論理学における証明論と意味論の関係同様，公理的意味論と表示的意味論の間に

は**健全性**と**完全性**の 2 つの関係が考えられる．

証明システムが表示的意味論に関して健全であるとは，証明可能な命題が，すべてのモデルで真であることを意味する．Λ の公理的意味論はプログラムの当然満たすべき制約であり，したがって表示的意味論に関して健全でなければならない．これに対して，証明システムが完全であるとは，すべてのモデルで真である任意の命題が証明可能であることを意味する．

$\mathcal{A}$ を Λ のモデルとする．Γ を満たす任意の $\mathcal{A}$ 環境 η について，

$$\mathcal{A}[\![\Gamma \rhd M : \tau]\!] \neq \mathcal{A}[\![\Gamma \rhd N : \tau]\!]\,\eta$$

が成り立つとき，

$$\mathcal{A} \models \Gamma \rhd M = N : \tau$$

と書く．これが任意のモデル $\mathcal{A}$ に対して成立するとき，等式 $\Gamma \rhd M = N : \tau$ は**モデル論的に恒真**であるといい，

$$\models \Gamma \rhd M = N : \tau$$

と書く．公理的意味論の健全性は，$\vdash \Gamma \rhd M = N : \tau$ なら $\models \Gamma \rhd M = N : \tau$ であることを意味し，公理的意味論の完全性は，$\models \Gamma \rhd M = N : \tau$ なら $\vdash \Gamma \rhd M = N : \tau$ であることを意味する．

完全性は，公理的意味論がプログラムの意図するすべての性質を表現していることを意味する強い性質であり，一般に成立しない場合が多い．Λ の公理的意味論も表示的意味論に関して完全ではない．

問 12.15 $\models \Gamma \rhd M = N : \tau$ であるが $\vdash \Gamma \rhd M = N : \tau$ ではない例をあげよ．

以下 Λ の公理的意味論の健全性を証明する．そのためには，すべての公理が恒真であることを示す必要がある．

以下の補題は，公理 (β) の恒真性を示す上で有用である．

補題 12. 12(意味に関する代入補題) $\eta \models \Gamma$ を満たす任意の $\mathcal{A}$ 環境 η に対して

$$\mathcal{A}[\![\Gamma \rhd [N/x]M : \tau_2]\!] \neq \mathcal{A}[\![\Gamma\{x : \tau_1\} \rhd M : \tau_2]\!]\eta\{x : \mathcal{A}[\![\Gamma \rhd N : \tau_1]\!]\eta\}$$

が成立する． □

この補題は，補題 12.7 と同様 M の構造に関する帰納法で証明できる．

問 12.16 補題 12.7 の証明を参考に補題 12.12 を証明せよ．ただし，§12.2 で説明した束縛変数に関する約束を仮定してよい．

定理 12. 13(公理的意味論の健全性) もし $\vdash \Gamma \rhd M = N : \tau$ なら，$\models \Gamma \rhd M = N : \tau$ である．

［証明］公理的意味論の各公理の両辺の意味が等しく，かつ各推論規則が意味の同値性を保存することを示すことにより証明する．等式理論の公理および推論規則は自明である．以下，公理 (β) と (fix) について証明する．他の公理は練習問題とする．

(β) の場合．$\Lambda \vdash \Gamma \rhd (\lambda x.M)N : \tau$ と仮定する．型システムの定義よりある τ_1，τ_2 があって $\Gamma\{x : \tau_1\} \rhd M : \tau_2$ かつ $\Gamma \rhd N : \tau_1$ である．$\mathcal{A}$ を任意のモデル，η を Γ を満たす任意の $\mathcal{A}$ 環境とし，$\mathcal{A}[\![\Gamma \rhd N : \tau_1]\!]\eta = d$，$\mathcal{A}[\![\Gamma \rhd \lambda x.M : \tau_1 \to \tau_2]\!]\eta = f$ とすると以下の等式が成り立つ．

$$\begin{aligned}\mathcal{A}[\![\Gamma \rhd (\lambda x.M)N : \tau_2]\!]\eta &= f(d)\\ &= \mathcal{A}[\![\Gamma\{x : \tau_1\} \rhd M : \tau_2]\!]\eta\{x : d\}\\ &= \mathcal{A}[\![\Gamma \rhd [N/x]M : \tau_2]\!]\,\eta \text{（補題 12.12 より）}\end{aligned}$$

以上より，$\models \Gamma \rhd (\lambda x : \tau_1\ .M)N = [N/x]M : \tau_2$ が成り立つ．

(fix) の場合．$\Lambda \vdash \Gamma \rhd \mathbf{fix}(M) : \tau$ と仮定する．型システムの定義より $\Lambda \vdash \Gamma \rhd M : \tau \to \tau$ である．$\mathcal{A}$ を任意のモデル，η を Γ を満たす任意の $\mathcal{A}$ 環境とし，$\mathcal{A}[\![\Gamma \rhd M : \tau \to \tau]\!]\eta = f$ とおく．以下の等式が成立する．

$$\begin{aligned}\mathcal{A}[\![\Gamma \rhd \mathbf{fix}(M) : \tau]\!] &\neq Fix^{\tau}(f) \quad \text{（意味の定義より）}\\ &= f(Fix^{\tau}(f)) \quad (Fix^{\tau} \text{ に関する仮定より})\\ &= \mathcal{A}[\![\Gamma \rhd M(\mathbf{fix}(M)) : \tau]\!]\,\eta \text{（意味の定義より）}\end{aligned}$$

よって $\models \Gamma \rhd \mathbf{fix}(M) = M(\mathbf{fix}(M)) : \tau$ である． ∎

問 12.17 他のすべての公理も健全であることを示し，定理 12.13 を完成せよ．

§12.5 型システムとラムダ式の意味との関連

ラムダ計算の型システムは，ラムダ式の構文構造のみによって定義された論理システムである．このような型システムを**静的型システム**とよぶ．静的型システムは，プログラムを実行することなく，そのプログラムを実行した結果の型に関する性質を演繹するシステムである．例えばラムダ式

$$(\lambda f.\lambda x.f\ (f\ x))\ (\lambda x.x)\ 1$$

に対して型システムは(任意の型環境のもとで) int 型を演繹する．プログラムの型の静的な演繹システムが意味をもつためには，それが正しくなければならない．すなわち，型システムが静的に演繹した型が，実際にプログラムを実行して得られる型と一致していなければならない．上の例でいえば，実際にこのラムダ式をプログラムとして実行した結果は int 型のデータであることを保証するものでなければならない．この性質を保証するのが型システムの健全性である．

公理的意味論の場合は，その定義の中に型が同一であることが含まれており，自明の性質である．表示的意味論に対する型システムの健全性は，以下の性質であり，やはり表示的意味論の定義よりほぼ明らかである．

命題 12.14 もし $\Gamma \rhd M : \tau$ なら任意の Λ のモデル $\mathcal{A}$ と $\eta \models \Gamma$ なる $\mathcal{A}$ 環境 η に対して $\mathcal{A}[\![\Gamma \rhd M : \tau]\!]\eta \in A^{\tau}$ である． □

型システムの健全性は，主に型と無関係に定義されたラムダ式の意味論に関して論じられる．その代表は以下に論ずる簡約関係と，第 14 章で展開する操作的意味論である．

ラムダ式の簡約関係は，ラムダ式の表現するふるまいを直接構文論的に定義しているものであり，ラムダ式の意味の定義とみなすことが可能である．簡約関係は，型システムと無関係にすべての型なしラムダ式の集合に対して定義される関係である．通常のプログラミング言語の実装においても，プログラムの動作は，型の制約とは無関係にプログラムの構造のみによって決定されるのが普通である．このような意味論に対する型システムの健全性は，型システムの正しさを保証する重要な性質となる．簡約意味論の場合，型システムの健全性は以下の型保存定理で示される性質である．

定理 12.15（型保存定理） もし $\Gamma \rhd M : \tau$ かつ $M \xrightarrow{*} N$ なら $\Gamma \rhd N : \tau$ である． □

この定理の証明のために，各簡約公理が型を保存することを示す．

補題 12.16

(i) もし $\Gamma \rhd (\lambda x.M)N : \tau$ なら $\Gamma \rhd [N/x]M : \tau$ である．

(ii) もし $\Gamma \rhd (M_1, M_2)[1] : \tau$ なら $\Gamma \rhd M_1 : \tau$ である．

(iii) もし $\Gamma \rhd (M_1, M_2)[2] : \tau$ なら $\Gamma \rhd M_2 : \tau$ である．

(iv) もし $\Gamma \rhd (\mathbf{case}\ 1(M)\ \mathbf{of}\ 1(x) \Rightarrow M_1,\ 2(y) \Rightarrow M_2) : \tau$ なら $\Gamma \rhd [N/x]M_1 : \tau$ である．

(v) もし $\Gamma \rhd (\mathbf{case}\ 2(M)\ \mathbf{of}\ 1(x) \Rightarrow M_1,\ 2(y) \Rightarrow M_2) : \tau$ なら $\Gamma \rhd [N/y]M_2 : \tau$ である．

(vi) もし $\Gamma \rhd \mathbf{fix}(M) : \tau$ なら $\Gamma \rhd M(\mathbf{fix}(M)) : \tau$ である． □

この結果から，1 ステップ簡約関係が型を保存することをラムダ式の文脈に関する帰納法によって示すことができ，定理 12.15 を簡約の長さに関する帰納法で示すことができる．

問 12.18

(i) 補題 12.16 を証明せよ．

(ii) 補題 12.16 を使い定理 12.15 を証明せよ．

以上の型保存定理によって，型の同一性を公理的意味論の定義から除くことも可能である．すなわち，規則 (axiom) を

$$(\text{axiom})\quad \frac{\Lambda \vdash \Gamma \rhd M_1 : \tau \quad M_1 \Longrightarrow M_2}{\Gamma \rhd M_1 = M_2 : \tau}$$

と変更し，公理的意味論の定義をこの規則および以前定義したその他の規則によって $\Gamma \rhd M_1 = M_2 : \tau$ の形の式を($\Lambda \vdash \Gamma \rhd M_1 : \tau$ と $\Lambda \vdash \Gamma \rhd M_2 : \tau$ を仮定せず)導出する証明システムと定義することが可能である．この場合，$\Gamma \rhd M_1 = M_2 : \tau$

が証明可能であれば $\Lambda \vdash \Gamma \rhd M_1 : \tau$ かつ $\Lambda \vdash \Gamma \rhd M_2 : \tau$ であることを証明することができる．しかしながら，簡約関係の逆関係は型を保存するとは限らないから，公理的意味論の規則 (axiom) を

$$(\text{axiom})\ \frac{\Lambda \vdash \Gamma \rhd M_1 : \tau \quad M_2 \Longrightarrow M_1}{\Gamma \rhd M_1 = M_2 : \tau}$$

と変更して得られるシステムは意味をなさない．

問 12.19 $\Lambda \vdash \Gamma \rhd M_1 : \tau$ かつ $M_2 \xrightarrow{*} M_1$ であっても $\Lambda \vdash \Gamma \rhd M_2 : \tau$ とならない例をあげよ．

簡約関係の逆関係も型を保存するようなラムダ式の集合を考える．この集合は型付け可能なラムダ式の集合 $\{M \mid$ ある Γ と τ に対して $\Lambda \vdash \Gamma \rhd M : \tau\}$ の部分集合である．M が満たすべき十分条件を M の構文論的性質として与えよ．

以上証明した型保存定理は，主にプログラミング言語の観点から論じられる性質であるが，以前学んだ型付きラムダ計算と自然演繹証明システムの関係によれば，型を保存したラムダ式の簡約は同一の命題の証明の変換に対応するはずである．これが証明の標準化である．

各論理接続子 F について，(F:I) 規則で F を導入した後 (F: E) 規則で F を取り除くような証明は冗長である．各論理接続子に関して，このような冗長性を取り除く証明の簡約化変換が定義できる．$\Delta \rhd A$ の証明を $\overset{\Pi}{(\Delta \rhd A)}$ と書くことにする．

以下の証明を考える．

$$\dfrac{\dfrac{\begin{array}{c}\Delta \cup A \rhd A \\ \vdots \\ \Delta \cup A \rhd B\end{array}}{\Delta \rhd A \supset B} \quad \overset{\Pi}{(\Delta \rhd A)}}{\Delta \rhd B}\ (\supset\text{:E})$$

この証明では，Δ にさらに A を仮定して B を証明し，そこから $A \supset B$ を証明し，それとは別に Δ から A を証明し，最終的に Δ から B を証明している．しかし Δ から A が証明可能であれば，A を仮定する代わりに A の証明を直接使用すればよい．この考え方に従い，$\Delta \cup A \rhd B$ の証明に現れる $\Delta \cup A \rhd A$ の形の証明を，$\Delta \rhd A$ の証明で置き換え，仮定の集合から A を取り除くと ($\supset$:E) 規則を

使用しない以下の証明に変形できる．

$$\begin{array}{c}\overset{\Pi}{(\Delta \rhd A)} \\ \vdots \\ \Delta \rhd B\end{array}$$

以上の変換は，A の証明 Π を A の仮定を使用しているところに代入していることに他ならず，β 簡約に対応している．**fix** を除く他の簡約規則も，証明を単純化する変換に対応している．

問 12.20 ($\wedge$:E) 規則と ($\vee$:E) 規則で終る証明に対して上記と同様の分析を行ない，それらに対して (fst)，(snd) および (case1)，(case2) に対応する証明の簡略変換が可能であることを示せ．

13 種々の構文構造の表現

以上定義したラムダ計算は，プログラミング言語で使用されている幅広いデータ構造や制御構造を表現可能であり，ML などの近代的なプログラミング言語の定義や実装のための核言語として広く利用されている．種々の具体的な構文は，構文解析の後，その意図する意味を実現するラムダ式に変換される．本章では，近代的プログラミング言語の機能を概説した後，それらをラムダ式に変換する方法を学ぶ．

§13.1 近代的プログラミング言語の特徴

第 10 章で解説したとおり，近代的プログラミング言語の典型を ML 言語に求めることができる．第 I 部で使用した Minimal は ML をモデルに設計されている．以下，Minimal の文法を使って ML 系言語の主な特徴を概説する．

（i）第一級のデータとしての高階の関数

近代的な言語の最も大きな特徴は，関数の定義と利用に関する柔軟な機構を提供していることである．以下の構文は，相互に依存する再帰的関数 $f_1, \ldots, f_n$ を定義する．

```
fun f1 p1 = M1
and f2 p2 = M2
  ⋮
```

```
and f_n p_n = M_n
```

このような名前付きの関数定義に加え，関数を直接式として定義することができる．

```
fn p => M
```

この式は引数 p を受け取り M の値を返す関数である．

これらの構文で定義された関数は整数などの通常のデータ構造同様，自由に値として使用することができる．すなわち関数を引数として別の関数に渡したり，データ構造の中に格納したりといった操作を自由に行なうことができる．以下は，高階の関数を用いて集合操作を実現している例である．

```
val emptySet = (fn x => false);
fun member (x,S) = S x;
fun insert (x,S) = (fn y => x = y or S y);
fun union (S1,S2) = (fn x => S1 x or S2 x);
fun intersection (S1,S2) = (fn x => S1 x & S2 x);
fun difference (S1,S2) = (fn x => (S1 x) & not (S2 x));
```

ここで `x = y` は `x` と `y` が同じ値か否かのテスト，`&` と `or` はそれぞれ論理積および論理和を表す．

(ii) データ型の定義とパターン・マッチング

ユーザは，リストや木構造などの種々のデータ型を，以下の形の型宣言を通じて容易に定義することができる．

```
type tid = l_1 of τ_1 | ··· | l_n of τ_n ;
type (α_1,...,α_n) tid = l_1 of τ_1 | ··· | l_n of τ_n ;
```

2 番目の定義の $\alpha_1,\ldots,\alpha_n$ は型パラメータであり，実際のテキストでは `'a` や `'b` などのように表記する．例えば以前定義した整数を要素とするリスト構造は，型パラメータを使用することによって，以下のような汎用のリスト構造に一般化可能である．

```
type 'a list = Nil | Cons of 'a * 'a list ;
```


定義されたデータは，パターン・マッチングを通じて，簡単に利用することができる．例えばリストの総和を計算する関数は以下のように定義できる．

```
fun length x = case x of
                  Nil =>0
                | Cons(h,t) => 1 + (length t)
               end
```

(iii) 多相型と型推論

ML の最も大きな特徴はその多相型システムにある．ML は強く型付けられた言語であり，プログラムが行なうすべての操作の型の整合性は，コンパイル時に完全にチェックされ，コンパイルされたプログラムは，未定義の操作を実行してシステムを停止させるようなことは起こらないことが保証される．さらにこの型の整合性のチェックは，型宣言を含まないプログラムから，そのプログラムがもつ最も一般的な型を自動的に推論することによって行なわれる．例えば，上で定義した `length` 関数に対して，型システムは以下の型を自動的に推論する．

```
length :  'a list -> int
```

この型は，`length` が任意の要素型をもつリストを受け取り整数を返す関数であることを示している．この多相型推論により，繁雑な型宣言をすることなく，汎用性ある関数を定義し安全に使用することができる．

以上の特徴は，ML の他，今日の多くの関数型プログラミング言語に採り入れられている．

Standard ML などの現実のプログラミング言語は，上記の機能に加えて，実用上有用なポインター型や例外処理などの手続き型言語の機能が加えられている．Minimal も，アルゴリズム記述の容易さのために，手続き型言語の機能である代入可能な変数や逐次実行構文などが導入されている．これら手続き型言語の機能のラムダ計算での表現は本書の範囲を越えるので，Minimal に以下の制限を加えて手続き型言語の機能を除いて得られる言語を，ラムダ計算で表現する方法を解説する．

(i) 代入可能な変数の宣言(var x = e)を禁止する.

(ii) ブロック構文(begin ⋯ end)から逐次実行の要素を取り除き，この構文の本体を，複数の宣言文の後に一つの式がある構文に制限する.

(iii) Minimal のパターン・マッチングは，プログラミングの便宜のために，すべての場合を尽くさない場合分け構文を許している．このような不完全な場合分け構文では，マッチングが失敗し，実行時に例外が発生することがある．例外の発生も手続き型言語の一種であるので，例外が起こる可能性を禁止するために，パターンを以下の文法で与えられるものに限定する.

pat ::= prodpat | l (prodpat)
prodpat ::= id | (prodpatlist)
prodpatlist ::= prodpat | prodpat , prodpatlist

ここで l は typ e構文で定義されたデータ構成子名である.

以上のように制限された Minimal は，近代的な関数型言語の中核とみなすことができる．以下，この制限付きの Minimal を Minimal^0 とよび，Minimal^0 の各構文 E のラムダ式での表現を $\overline{E}$ と書き，E を $\overline{E}$ に対応させる規則を与え，その正当性を論じる.

§13.2 N 個の組

Minimal^0 の unit 型および式 () に対応して，ラムダ計算においても，ただ 1 つの要素からなる基底型 $unit$ と，$unit$ 型の唯一の要素を表す定数 () が定義されているものとする．これを利用して Minimal^0 の組型をラムダ式に翻訳する.

組型は直積に対応する．集合論的には n-次の直積 $A_1 \times \cdots \times A_n$ は 2 次の直積の繰り返し $A_1 \times (A_2 \times \cdots (A_n \times \{()\}) \cdots)$ と同型である．この洞察に基づき，n 個の組を表す式 $(E_1, \ldots, E_n)$ および n 個の組の i 番目の射影 $E[i]$ をそれぞれ以下のように変換する.

$$\overline{(E_1, \ldots, E_n)} = (\overline{E_1}, (\ldots, (\overline{E_n}, ())) \ldots)$$
$$\overline{E[i]} = \overline{E}[2] \ldots [2][1] \qquad \text{(右辺は } i-1 \text{ 個の } [2] \text{ を含む)}$$

この変換に対して以下の簡約関係が成り立つ.

$$\overline{(E_1, \ldots, E_n)[i]} \xrightarrow{*} \overline{E_i}$$

よって，この変換は，n-次の組型の意図するふるまいを実現しているといえる.

この翻訳は型に関しても正しいことを，以下のようにして確かめることができる．n-次の組型に関する型付け規則は以下のように与えられる.

$$(\mathrm{prod}_n) \quad \frac{\Gamma \rhd M_1 : \tau_1 \quad \cdots \quad \Gamma \rhd M_n : \tau_n}{\Gamma \rhd (M_1, \ldots, M_n) : \tau_1 \times \cdots \times \tau_n}$$

$$(\mathrm{proj}_i) \quad \frac{\Gamma \rhd M : \tau_1 \times \cdots \times \tau_n}{\Gamma \rhd M[i] : \tau_i} \quad (1 \le i \le n)$$

$\Gamma \rhd M : \tau$ が上記 2 つの規則を含む型システムで証明可能であることを $ML \vdash \Gamma \rhd M : \tau$ と書く．ラムダ式の翻訳に加えて，n-次の組型の翻訳を以下のように定める.

$$\overline{\tau_1 \times \cdots \times \tau_n} = \overline{\tau_1} \times (\overline{\tau_2} \times (\cdots (\overline{\tau_n} \times unit) \cdots)))$$

この翻訳を型環境 Γ の中に現れる各型に施して得られる型環境を $\overline{\Gamma}$ と書く．以下の性質を，M の構造に関する帰納法で示すことができる.

命題 13.1 $ML \vdash \Gamma \rhd M : \tau$ なら $\Lambda \vdash \overline{\Gamma} \rhd \overline{M} : \overline{\tau}$ である. □

§13.3 ユーザ定義のデータ型と場合分け構文

Minimal^0 のデータ型の宣言は，宣言される型が要素の型に現れる再帰的な定義が可能であり，さらに，型パラメータをもつ型が宣言できる．これら 2 つの要素の取り扱いは第 15 章で説明することとし，本節では，型パラメータがなくかつ再帰的でない，以下のような型宣言のみを考える.

type T = l_1 of τ_1 | ⋯ | l_n of τ_n ;

ここで各要素の型 τ_i に T は現れない.

Minimal^0 の文法では引数型の宣言 of τ がない場合もある．その場合は of unit が宣言されているとみなす.

この型の要素は，型 τ_i をもつ式を M とすると $l_i(M)$ の形をした値である．さらに l_i は，定義された型 T の要素を区別するためのユーザにとっての名前であり，名前自身は本質的ではない．すると上記の宣言で定義される型 T は以下の n-次の直和型と見なすことができる．

$$\overline{T} = \overline{\tau_1} + \cdots + \overline{\tau_n}$$

集合論的には n-次の直和 $A_1 + \cdots + A_n$ は 2 次の直和の繰り返し $A_1 + (A_2 + \cdots (A_n + \emptyset) \cdots)$ と同型である．またユーザ定義のデータ型は常に型宣言の後に使用されるから，ユーザが定めたラベル l_i が i 番目の直和の要素であることはあらかじめ分かっている．そこで，$l_i(E)$ は以下のように変換できることがわかる．

$$\overline{l_i(E)} = 2(\cdots 2(1(\overline{E}))\cdots) \qquad (\text{右辺は } i-1 \text{ 個の } 2(\_) \text{を含む})$$

ユーザ定義のデータ型の場合分け構文の変換は以下のように与えられる．

$$\begin{aligned}
&\overline{\texttt{case } E_0 \texttt{ of } l_1(x_1)\texttt{=>}E_1\texttt{, ..., } l_n(x_n)\texttt{=>}E_n \texttt{ end}} \\
&= \textbf{case } \overline{E_0} \textbf{ of} \\
&\qquad 1(x_1) \Rightarrow \overline{E_1}, \\
&\qquad 2(X_1) \Rightarrow \textbf{case } X_1 \textbf{ of} \\
&\qquad\qquad 1(x_2) \Rightarrow \overline{E_2}, \\
&\qquad\qquad \cdots \\
&\qquad\qquad\qquad \textbf{case } X_{n-1} \textbf{ of} \\
&\qquad\qquad\qquad\qquad 1(x_n) \Rightarrow \overline{E_n}, \\
&\qquad\qquad\qquad\qquad 2(X_n) \Rightarrow \textbf{fix}(\lambda x.x)
\end{aligned}$$

ここで $X_1, \ldots, X_n$ は新しい変数である．また最後の $\textbf{fix}(\lambda x.x)$ は任意の型をもつラムダ式である．

この変換に対して以下の簡約関係が成り立つ．

$$\overline{\texttt{case } l_i(E_0) \texttt{ of } l_1(x_1)\texttt{=>}E_1\texttt{,...,}l_n(x_n)\texttt{=>}E_n \texttt{ end}} \xrightarrow{*} [\overline{E_0}/x_i]\overline{E_i}$$

よって以上の変換はユーザ定義のデータ型の意図するふるまいを実現しているといえる．

以上の変換も型を保存することを示すことができる．Minimal^0 において，

$$\texttt{type } T \texttt{ = } l_1 \texttt{ of } \tau_1 \texttt{ | } \cdots \texttt{ } l_n \texttt{ of } \tau_n \texttt{ ;}$$

が宣言されているときのユーザ定義のデータ型に関する型付け規則を以下のように定める．

$$(\text{data}) \quad \frac{\Gamma \rhd M : \tau_i}{\Gamma \rhd l_i(M) : T}$$

$$(\text{case}) \quad \frac{\Gamma \rhd M : T \quad \Gamma\{x_i : \tau_i\} \rhd M_i : \tau \ (1 \le i \le n)}{\Gamma \rhd \texttt{case } M \texttt{ of } l_1(x_1) \texttt{ => } M_1\texttt{,... } l_n(x_n) \texttt{ => } M_n \texttt{ end} : \tau}$$

上記 2 つの規則を加えた型システムで証明可能であることを $ML \vdash \Gamma \rhd M : \tau$ と書く．

型宣言で定義された型を翻訳するために，Λ は空集合に対応する基底型 $void$ を含むと仮定する．ただし $void$ 型の定数は存在しないものとする．上記の型宣言の翻訳を n-次の直和型を使って，以下のように定義する．

$$\overline{T} = \overline{\tau_1 + \cdots + \tau_n} = \overline{\tau_1} + (\overline{\tau_2} + (\cdots (\overline{\tau_n} + void) \cdots))$$

上記の翻訳はこの 2 つの型付けを保存する．すなわち $ML \vdash \Gamma \rhd M : \tau$ なら $\Lambda \vdash \overline{\Gamma} \rhd \overline{M} : \overline{\tau}$ であることを示せる．

問 13.1 $\Lambda \vdash \overline{\Gamma} \rhd \overline{M} : \overline{\tau_i}$ なら $\Lambda \vdash \overline{\Gamma} \rhd \overline{l_i(M)} : \overline{T}$ であることを示せ．同様に，$\Lambda \vdash \overline{\Gamma} \rhd \overline{M} : \overline{T}$ かつ各 $1 \le i \le n$ に対して $\Lambda \vdash \overline{\Gamma}\{x_i : \overline{\tau_i}\} \rhd \overline{M_i} : \overline{\tau}$ なら

$$\Lambda \vdash \overline{\Gamma} \rhd \overline{\texttt{case } M \texttt{ of } l_1(x_1) \texttt{ => } M_1\texttt{,... } l_n(x_n) \texttt{ => } M_n \texttt{ end}} : \overline{\tau}$$

であることを示せ．

以上を用いて $ML \vdash \Gamma \rhd M : \tau$ なら $\Lambda \vdash \overline{\Gamma} \rhd \overline{M} : \overline{\tau}$ であることを示せ．

問 13.2 真理値は条件式とともに使われ以下のようなふるまいをする．

$$\texttt{if true then } E_1 \texttt{ else } E_2 \xrightarrow{*} E_1$$
$$\texttt{if false then } E_1 \texttt{ else } E_2 \xrightarrow{*} E_2$$

真理値は 2 つの定数のみを含むデータ型であるから，Minimal^0 の文法で

```
type bool = true | false ;
```

と定義されたデータとみなすことができる．この考えに従い，`true`，`false` および条件式 `if` E_0 `then` E_1 `else` E_2 の翻訳を与え，それが上記のふるまいをすることを確かめよ．

§13.4 ブロック構文

Λ において値の束縛構文 **let** $x = M_1$ **in** M_2 は $(\lambda x.M_2)\ M_1$ の別記法として取り扱ってきたが，以降の翻訳では，基本構文として使用することにする．第 15 章で解説するとおり，この構文は多相型言語においては上記の翻訳より柔軟な型付けが可能な基本構文である．

Minimal^0 のブロック構文 `begin` D_1; $\cdots$; D_n; E `end` を考える．ここで D_i は値または関数宣言，E は式である．翻訳を容易にするために，Minimal^0 に値の束縛構文 `let` D `in` E `end` を追加し，このブロック構文を以下の形の入れ子になった値の束縛構文に変換する．

$$\texttt{let}\ D_1\ \texttt{in}\ \cdots\ \texttt{let}\ D_n\ \texttt{in}\ E\ \texttt{end}\ \cdots\ \texttt{end}$$

ブロック構文の翻訳を完成させるためには，各宣言 D の種類に応じて，値の宣言構文 `let` D `in` E `end` の翻訳を与えればよい．

単純な値の宣言文は値束縛に直接翻訳可能である．

$$\overline{\text{let val } x = E_1 \text{ in } E_2 \text{ end}} = \textbf{let } x = \overline{E_1} \textbf{ in } \overline{E_2}$$

つぎに関数宣言を考える．まず予備的準備として複数の引数をもつ関数定義

`fun` $f\ x_1\ \cdots\ x_n$ `=` e

を

`fun` $f\ x_1$ `=` `fn` x_2 `=>` ... `fn` x_n `=>` e

に変換する．この変換の後，関数は唯一の引数をもつ．

以下の関数定義を考える．

`let fun` $f\ x$ `=` E_1 `in` E_2 `end`

この定義が再帰を含まなければ，すなわち E_1 が f を含まなければ，この定義は，関数を表すラムダ式に f という名前を付けたものであるから，以下のラムダ式に翻訳される．

$$\textbf{let } f = \lambda x.\overline{E_1} \textbf{ in } \overline{E_2}$$

しかし，関数の本体 E_1 の中で，この定義によって定義される関数 f が使用される再帰的な定義の場合は，上記の翻訳では $\lambda x.\overline{E_1}$ に中に現れる f が未定義となってしまう．

再帰的定義は，定義される関数と同一のふるまいをする関数を変数 f として関数定義の中で使用することが可能な定義である．そこで，上記の定義を，関数 f が満たすべき等式

$$f = \lambda x.\overline{E_1}$$

と考えると，再帰を含む関数は，以下の等式を満たすラムダ式 M で表現できる．

$$M = \lambda x.[M/f]\overline{E_1}$$

以下のラムダ式が，簡約関係が定める公理的意味論において，この等式を満たす．

$$\textbf{fix}(\lambda f.\lambda x.\overline{E_1})$$

実際，以下の簡約関係がある．

$$\begin{aligned}\textbf{fix}(\lambda f.\lambda x.\overline{E_1}) &\xrightarrow{*} (\lambda f.\lambda x.\overline{E_1})\ (\textbf{fix}(\lambda f.\lambda x.\overline{E_1})) \\ &\xrightarrow{*} \lambda x.[\textbf{fix}(\lambda f.\lambda x.\overline{E_1})/f]\overline{E_1}\end{aligned}$$

そこで，再帰的関数の翻訳を以下のように与えることができる．

$$\overline{\texttt{let fun } f\ x = E_1 \texttt{ in } E_2 \texttt{ end}} = \textbf{let } f = \textbf{fix}(\lambda f.\lambda x.\overline{E_1}) \textbf{ in } \overline{E_2}$$

つぎに，最も一般的な相互再帰的な関数定義の翻訳を考える．

$$\begin{aligned}RECFUN \equiv\ &\texttt{let fun } f_1\ x_1 = E_1 \\ &\vdots \\ &\texttt{and } f_n\ x_n = E_n \\ &\texttt{in } E_0 \texttt{ end}\end{aligned}$$

ここで各 E_i は関数名 $f_1,\ldots,f_n$ を含む式である．この定義は n 個の再帰的関数の生成と $f_1,\ldots,f_n$ の名前の定義の組合せである．相互再帰的な関数定義は以下の等式を同時に満たすものと考えられる．

$$f_1 = \lambda x_1.\overline{E_1}$$
$$\vdots$$
$$f_n = \lambda x_n.\overline{E_n}$$

この等式システムを，関数の組に対する等式と考えることにより，以下のような等式とみなせる．

$$(f_1,\ldots,f_n) = (\lambda x_1.\overline{E_1},\ldots,\lambda x_n.\overline{E_n})$$

ただし等号は各要素ごとになりたつものとする．この等式を満たす組を表すラムダ式を M と置くと，M は以下の等式を満たすラムダ式のはずである．

$$M = (\lambda x_1.[\,M[\,1]/f_1,\ldots,M[n]/f_n]\overline{E_1},\ldots,\lambda x_n.[\,M[\,]/f_1,\ldots,M[n]/f_n]\overline{E_n})$$

1 つの再帰的関数定義の議論を一般化することにより，この性質を満たすラムダ式を以下のように与えることができる．

$$\mathbf{fix}(\lambda F.(\lambda f_1\ldots\lambda f_n.(\lambda x_1.\overline{E_1},\ldots,\lambda x_n.\overline{E_n}))\ F[\,1]\cdots\ F[n])$$

問 13.3 上記のラムダ式を M とし，$f_1 \equiv M[1],\ldots,f_n \equiv M[n]$ とおくと，$f_1,\ldots,f_n$ はラムダ式の簡約関係が導出する同値関係に関して，上記等式を満たすことを確認せよ．

以上から，一般的な相互再帰的関数定義のラムダ式への翻訳は，n-次の組を使って以下のように与えられる．

$$\begin{aligned}&\overline{RECFUN}\\ &= \ \mathbf{let}\ F = \mathbf{fix}\ \lambda F.(\lambda f_1\ldots\lambda f_n.(\lambda x_1.\overline{E_1},\ldots,\lambda x_n.\overline{E_n}))\ F[\,1]\cdots\ F[n])\ \mathbf{in}\\ &\quad\ \mathbf{let}\ f_1 = F[\,1]\mathbf{in}\\ &\qquad\vdots\\ &\quad\ \mathbf{let}\ f_n = F[n]\ \mathbf{in}\ \overline{E_0}\end{aligned}$$

前に定義した n-次の組の翻訳をさらに施すことによって，Λ の式に翻訳される．

以上の変換も型を保存することを確かめることができる．型環境を以下のように定める

$$\Gamma_0 = \Gamma\{f_1 : \tau_1^1 \to \tau_2^1,\ldots,f_n : \tau_1^n \to \tau_2^n\}$$
$$\Gamma_i = \Gamma\{f_1 : \tau_1^1 \to \tau_2^1,\ldots,f_n : \tau_1^n \to \tau_2^n, x_i : \tau_1^i\}\quad (1 \le i \le n)$$

再帰的な関数定義 $RECFUN$ の型付け規則は，以下のように与えることができる．

$$(\mathrm{rec})\ \frac{\Gamma_i \rhd E_i : \tau_2^i\ (1 \le i \le n)\quad \Gamma_0 \rhd E_0 : \tau}{\Gamma \rhd RECFUN : \tau}$$

上に与えた $RECFUN$ の翻訳はこの型付けを保存することを示すことができる．

問 13.4 各 $1 \le i \le n$ に対して $\Lambda \vdash \overline{\Gamma_i} \rhd \overline{E_i} : \overline{\tau_2^i}$ が成り立ち $\Lambda \vdash \overline{\Gamma_0} \rhd \overline{E} : \tau$ が成り立つなら，$\Lambda \vdash \overline{\Gamma} \rhd \overline{RECFUN} : \tau$ が成り立つことを示せ．

§13.5 パターン・マッチング

パターン・マッチングは，組型や再帰的関数などこれまでに取り扱った構文構造とちがい，すでに存在するデータ構造を別のより基本的な構造に翻訳するというよりは，組の要 素の取り出しや場合分けを関数定義に融合する略記法と考えるのが妥当である．したがって，これから定義する翻訳を，パターン・マッチング構文の意味そのものと考える．

Minimal^0 に含まれるパターン・マッチングを含む式を順次系統的に変形し，最終的に純粋なラムダ式に変換する手順を定義する．

（i）組パターンとデータ型パターンの分離

パターンは type 文で定義されたラベルをもつデータ型パターンか組パターンのいずれかである．一般のパターン・マッチングを，これら 2 種類のパターンに分解する．すなわち，以下の変形を行なう．

```
case e of
    l1(arg1) => e1
    ⋮
```

```
    | l_n(arg_n) => e_n
end
```

を以下のように変形する．

```
case ē of
      l(x_1) => (fn arg_1 => e_1) x_1
      ⋮
    | l_n(x_n) => (fn arg_n => e_n) x_n
end
```

以上の変形を可能な限り繰り返せば，Minimal^0 の式に含まれるパターンは組パターンのみのはずである．

（ii）組パターンの除去

組をパターンを引数にもつ関数定義

```
fun f (arg_1,...,arg_n) = e_1
```

は以下のように変形できる．

```
fun f x = (fn arg_1 => ... fn arg_n => e_1) x[1] ... x[n]
```

名前のない関数式も同様に変換する．すなわち

```
fn (arg_1,...,arg_n) => e_1
```

は

```
fn x => (fn arg_1 => ... fn arg_n => e_1) x[1] ... x[n]
```

に変換する．以上の変換を繰り返し行なうことにより，すべての明示的なパターンを変換することが可能である．

問 13.5 以上の翻訳方式を組合せれば，Minimal^0 を Λ に翻訳可能である．付録を参考に，Minimal^0 の文法を定義し，適当な括弧を導入することにより，曖昧さのない文法とせよ．

type 文で定義される型宣言は再帰的ではなく，かつ型パラメータを含まないと仮定し，Minimal^0 を Λ の文法に変換する再帰的アルゴリズムを定義せよ．

このアルゴリズムを用いて，Minimal^0 構文を入力しラムダ式に変換しその結果をプリントするプログラムを以下の手順で作成せよ．

（i）Minimal^0 の構文木を表すデータ型を定義し，そのプリント関数を書け．

（ii）Minimal^0 の構文木とユーザ定義のデータ型の情報を受け取り，問 11.7 で定義した Λ の構文木を表すデータ型に変換する関数を定義せよ．（ヒント：翻訳に必要なユーザ定義のデータ型の情報は，データ型の生成に使用されるラベルが，どの整数値に対応するかである．翻訳関数はこの対応関係を保持し，type 文に対しては，この宣言で定義されるラベルに対する情報を追加する処理を行なえばよい．）

（iii）(ii)で定義した関数を用いて，問 11.7 で作成した read_print 関数を，Minimal^0 の構文を入力し，Minimal^0 の構文と，その翻訳結果である Λ の構文の両方をプリントするように変更せよ．

14 プログラム実行のモデル

ラムダ計算は少数の構文要素からなる抽象度の高い構文であるが，仮引数名を用いた関数定義機構など，ほぼプログラミング言語の文法構造をそのまま含んでおり，計算機による効率的な実行のモデルとはなっていない．プログラミング言語の理論的基礎を完成させるためには，ラムダ式で表現されたプログラムを計算機で実行するための系統的な方法を確立しなければならない．これがラムダ計算の操作的意味論である．本章ではその代表的なものを 3 つ紹介する．

§14.1 コンビネータへのコンパイル

ラムダ式は，その中の変数を別のラムダ式で置き換える β 簡約関係によって計算を表現する．ラムダ式の表す計算を実現する最も直接的な方法は，この β 簡約操作そのものをコンピュータ上で実現することであるが，このような素朴な方法では，プログラムの効率的な実行系は得られない．ラムダ式の実行系を構築する上での課題は，ラムダ式の変数への代入に基づく関数定義を，計算機で効率よく実行できるように翻訳することである．

β 簡約に基づく関数定義機構そのものをコンピュータで効率よく実装するのは困難であるが，個々の関数を実現するのは容易である．そこでもし，ラムダ抽象によって定義される無限に多様な関数が，いくつかの少数の関数の組合わせによって実現可能なら，その少数の関数のみを効率よく実行するプログラムを用意することによって，ラムダ式の実行系が構築可能である．

この考え方にしたがいラムダ計算の操作的意味論を実現できる．これが，ラムダ式のコンビネータへの変換である．コンビネータとは，実行環境に依存せずに完全にそのふるまいが定義された関数のことであり，ラムダ計算では，自由変数を含まないラムダ抽象式に対応する．ラムダ式のコンビネータへの変換が可能であるためには，ラムダ計算で定義可能な任意の関数を表現可能な，少数のコンビネータ集合が存在しなければならない．このヒントを命題論理学に求めることができる．

(a) ヒルベルトシステム

型付きラムダ計算は自然演繹システムに対応することを学んだが，論理学の分野では，伝統的に，ヒルベルトの公理的な論理学が広く使用されている．

自然演繹システムの場合と同様，仮定の集合 Δ から命題 A が証明可能であることを $\Delta \rhd A$ と書く．**ヒルベルトシステム**は，少数の公理と以下の論理法則のみからなる証明システムである．

$$\frac{\Delta \rhd A \supset B \quad \Delta \rhd A}{\Delta \rhd B}$$

この推論規則は，自然演繹システムにおける (⊃:E) 規則であるが，古典的な形式論理学では三段論法の一種を表す **Modus Ponens** とよばれる推論規則である．以前同様，与えられた非論理的公理の集合 Ax を含む体系を考える．直観主義的命題論理学に対するヒルベルトシステムの公理集合を図 14.1 に与える．これら各公理は，それぞれ，その公理に現れるメタ変数 A, B, C を任意に置き換えて得られるすべての公理を代表するものである．

(K) $A \supset B \supset A$　　(S) $(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$
(P) $A \supset B \supset A \wedge B$　　(F) $A \wedge B \supset A$　　(N) $A \wedge B \supset B$
(L) $A \supset A \vee B$　　(R) $B \supset A \vee B$
(A) $A \vee B \supset (A \supset C) \supset (B \supset C) \supset C$　　(Ax) a $(a \in Ax)$

図 **14.1**　ヒルベルトシステムの公理集合

これら公理集合と Modus Ponens 規則からなる証明系を $\mathcal{H}$ とよび，命題 A が仮定の集合 Δ と公理から Modus Ponens 規則のみを使って導かれるとき，$\mathcal{H} \vdash \Delta \rhd A$ と書く．

問 14.1　Ax のメンバーである非論理的公理以外のヒルベルトシステムの公理は，すべて恒真式であることを確かめよ．この性質より，$\mathcal{H} \vdash \Delta \rhd A$ なら $\Delta \models A$ であることを示せ．

ヒルベルトシステムも，自然演繹システムと同等の証明能力をもつことが示される．その証明で中心的な役割を果たすものが，以下の演繹定理である．

定理 14.1（演繹定理）　$\mathcal{H} \vdash \Delta \cup A \rhd B$ なら $\mathcal{H} \vdash \Delta \rhd A \supset B$ である．

［証明］$\mathcal{H} \vdash \Delta \cup A \rhd B$ に関する機械的な帰納法で証明可能であるが，ここではより簡潔な証明を与える．

もし $\Delta \cup A \rhd B$ が A に関する仮定 $\Delta \cup A \rhd A$ を使用せず証明可能な場合は，その証明に含まれる各仮定の重複集合から A を取り除いて得られる証明は，$\Delta \rhd B$ の正しい証明である．この証明を $\overset{\Pi}{(\Delta \rhd B)}$ とすると以下の証明が存在する．

$$\frac{\dfrac{}{\Delta \rhd B \supset A \supset B}\,(\mathrm{K}) \quad \overset{\Pi}{(\Delta \rhd B)}}{\Delta \rhd A \supset B}$$

$\Delta \cup A \rhd B$ の証明が A に関する仮定 $\Delta \cup A \rhd A$ を使用する場合について，以下 $\Delta \cup A \rhd B$ の証明の構造に関する帰納法で示す．

$B \equiv A$ の場合．以下の証明が存在する．

$$\Pi \equiv \frac{\dfrac{\dfrac{}{\Delta \rhd (A \supset (C \supset A) \supset A) \supset (A \supset C \supset A) \supset A \supset A}\,(\mathrm{S}) \quad \dfrac{}{\Delta \rhd A \supset (C \supset A) \supset A}\,(\mathrm{K})}{\Delta \rhd (A \supset C \supset A) \supset A \supset A} \quad \dfrac{\overset{\Pi}{(\Delta \rhd (A \supset C \supset A) \supset A \supset A)} \quad \dfrac{}{\Delta \rhd A \supset C \supset A}\,(\mathrm{K})}{\Delta \rhd A \supset A}$$

B が Modus Ponens の帰結の場合．ある C に対して $\mathcal{H} \vdash \Delta \cup A \rhd C \supset B$ かつ $\mathcal{H} \vdash \Delta \cup A \rhd C$ である．帰納法の仮定より，$\mathcal{H} \vdash \Delta \rhd A \supset C \supset B$ かつ $\mathcal{H} \vdash \Delta \rhd A \supset C$ である．これらの証明をそれぞれ Π_1，Π_2 とすると以下の証明が構築できる．

$$\frac{\dfrac{\dfrac{}{\Delta \rhd (A \supset C \supset B) \supset (A \supset C) \supset A \supset B}\,(\mathrm{S}) \quad \overset{\Pi_1}{(\Delta \rhd A \supset C \supset B)}}{\Delta \rhd (A \supset C) \supset A \supset B} \quad \overset{\Pi_2}{(\Delta \rhd A \supset C)}}{\Delta \rhd A \supset B}$$

この定理を用いて，ヒルベルトシステムと自然演繹システムが証明能力において等価であることを証明する．

定理 14.2 $\mathcal{H} \vdash \Delta \rhd A$ であれば $\mathcal{N} \vdash \Delta \rhd A$ である．逆に $\mathcal{N} \vdash \Delta \rhd A$ であれば $\mathcal{H} \vdash \Delta \rhd A$ である．

［証明］ヒルベルトシステムの Modus Ponens 規則は自然演繹システムの推論規則でもあるから，前半の命題は，ヒルベルトシステムの各公理が自然演繹システムで証明可能であることを示せばよい．これは練習問題とし，以下は後半の命題を，$\mathcal{N} \vdash \Delta \rhd A$ の証明の構造に関する帰納法で証明する．証明の最後に使用された規則により場合分けを行なう．

(taut) または (ax) の場合． $A \in \Delta$ または $A \in Ax$ であり，$\mathcal{H} \vdash \Delta \rhd A$ である．

($\supset$:I) の場合． 証明システムの定義より，ある A_1, A_2 があって，$A = A_1 \supset A_2$ かつ $\mathcal{N} \vdash \Delta \cup A_1 \rhd A_2$ である．帰納法の仮定より $\mathcal{H} \vdash \Delta \cup A_1 \rhd A_2$ である．演繹定理(定理 14.1)より $\mathcal{H} \vdash \Delta \rhd A_1 \supset A_2$ である．

($\supset$:E) の場合． 証明システムの定義より，ある A_1 があって，$\mathcal{N} \vdash \Delta \rhd A_1 \supset A$ かつ $\mathcal{N} \vdash \Delta \rhd A_1$ である．帰納法の仮定より $\mathcal{H} \vdash \Delta \rhd A_1 \supset A$ かつ $\mathcal{H} \vdash \Delta \rhd A_1$ である．規則 Modus Ponens より $\mathcal{H} \vdash \Delta \rhd A$ である．

($\wedge$:I) の場合． 証明システムの定義より，ある A_1, A_2 があって，$A = A_1 \wedge A_2$，$\mathcal{N} \vdash \Delta \rhd A_1$ かつ $\mathcal{N} \vdash \Delta \rhd A_2$ である．帰納法の仮定より $\mathcal{H} \vdash \Delta \rhd A_1$ かつ $\mathcal{H} \vdash \Delta \rhd A_2$ である．これらの証明を Π_1, Π_2 とすると以下の証明が構成できる．

$$\dfrac{\dfrac{}{\Delta \rhd A_1 \supset A_2 \supset A_1 \wedge A_2}\ (\mathrm{P}) \quad \begin{matrix}\Pi_1\\(\Delta \rhd A_1)\end{matrix}}{\Delta \rhd A_2 \supset A_1 \wedge A_2} \quad \begin{matrix}\Pi_2\\(\Delta \rhd A_2)\end{matrix}}{\Delta \rhd A_1 \wedge A_2}$$

($\wedge$:Ei) の場合． 証明システムの定義より，ある A_1, A_2 があって，$A = A_i (i = 1$ または $i = 2)$かつ $\mathcal{N} \vdash \Delta \rhd A_1 \wedge A_2$ である．帰納法の仮定より $\mathcal{H} \vdash \Delta \rhd A_1 \wedge A_2$ である．この証明を Π とすると以下の証明が構成できる．

$$\dfrac{\dfrac{}{\Delta \rhd A_1 \wedge A_2 \supset A_i}\ (\mathrm{F} \text{ または } \mathrm{N}) \quad \begin{matrix}\Pi\\(\Delta \rhd A_1 \wedge A_2)\end{matrix}}{\Delta \rhd A_i}$$

($\vee$:Ii) の場合． 証明システムの定義より，ある A_1, A_2 があって，$A = A_1 \vee A_2$ かつ $\mathcal{N} \vdash \Delta \rhd A_i (i = 1$ または $i = 2)$である．帰納法の仮定より $\mathcal{H} \vdash \Delta \rhd A_i$ である．この証明を Π とすると以下の証明が構成できる．

$$\dfrac{\dfrac{}{\Delta \rhd A_i \supset A_1 \vee A_2}\ (\mathrm{L} \text{ または } \mathrm{R}) \quad \begin{matrix}\Pi\\(\Delta \rhd A_i)\end{matrix}}{\Delta \rhd A_1 \vee A_2}$$

($\vee$:E) の場合． 証明システムの定義より，ある A_1, A_2 があって，$\mathcal{N} \vdash \Delta \rhd A_1 \vee A_2$，$\mathcal{N} \vdash \Delta \cup A_1 \rhd A$ かつ $\mathcal{N} \vdash \Delta \cup A_2 \rhd A$ である．帰納法の仮定より $\mathcal{H} \vdash \Delta \rhd A_1 \vee A_2$，$\mathcal{H} \vdash \Delta \cup A_1 \rhd A$ かつ $\mathcal{H} \vdash \Delta \cup A_2 \rhd A$ である．演繹定理より $\mathcal{H} \vdash \Delta \rhd A_1 \supset A$ かつ $\mathcal{H} \vdash \Delta \rhd A_2 \supset A$ である．これらの証明を Π_1, Π_2, Π_3 とすると以下の証明が構成できる．

$$\Pi_4 = \dfrac{\dfrac{}{\Delta \rhd (A_1 \vee A_2) \supset (A_1 \supset A) \supset (A_2 \supset A) \supset A}\ (\mathrm{A}) \quad \begin{matrix}\Pi_1\\(\Delta \rhd A_1 \vee A_2)\end{matrix}}{\Delta \rhd (A_1 \supset A) \supset (A_2 \supset A) \supset A}$$

$$\Pi_5 = \dfrac{\dfrac{\begin{matrix}\Pi_4\\(\Delta \rhd (A_1 \supset A) \supset (A_2 \supset A) \supset A)\end{matrix} \quad \begin{matrix}\Pi_2\\(\Delta \rhd A_1 \supset A)\end{matrix}}{\Delta \rhd (A_2 \supset A) \supset A} \quad \dfrac{\begin{matrix}\Pi_5\\((A_2 \supset A) \supset A)\end{matrix} \quad \begin{matrix}\Pi_3\\(\Delta \rhd A_2 \supset A)\end{matrix}}{\Delta \rhd A}}{\Delta \rhd A}$$

問 14.2 以下の各命題を自然演繹システムで証明し，さらにその証明を定理 14.2 を使ってヒルベルトシステムの証明に変形せよ．

(i) $A \supset A$

(ii) $A \supset B \supset A$

(iii) $A \wedge B \supset B \wedge A$

(iv) $A \vee B \supset B \vee A$

(b) ラムダ式のコンビネータへの翻訳

ヒルベルトシステムの Modus Ponens 規則は，自然演繹システムの場合と同様，関数適用に対応する．するとヒルベルトシステムは，名前による関数定義機構をもたず定数関数のみからなる，一種の関数型言語系とみることができる．これがコンビネータ論理である．以下の対応関係が成立する．

ヒルベルトシステム	$\Longleftrightarrow$	コンビネータ論理
Modus Ponens	$\Longleftrightarrow$	関数適用
公理	$\Longleftrightarrow$	コンビネータ

ヒルベルトシステムの 9 つの公理に対するコンビネータを導入する．さらに $\mathbf{fix}(M)$ に対応したコンビネータを追加する．各公理の自然演繹システムにおける証明を構築すると，それらは表 14.1 に示す自由変数をもたないラムダ式に対応することを確認できる*1．

表 14.1 ヒルベルトの公理とコンビネータの対応

$\mathcal{H}$ の公理	コンビネータと対応するラムダ式
(S)	$\mathbf{S}$ $\lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z)$
(K)	$\mathbf{K}$ $\lambda x.\lambda y.x$
(P)	$\mathbf{P}$ $\lambda x.\lambda y.(x,y)$
(F)	$\mathbf{F}$ $\lambda x.x[1]$
(N)	$\mathbf{N}$ $\lambda x.x[2]$
(L)	$\mathbf{L}$ $\lambda x.1(x)$
(R)	$\mathbf{R}$ $\lambda x.2(x)$
(A)	$\mathbf{A}$ $\lambda x.\lambda y.\lambda z.(\mathbf{case}\ x\ \mathbf{of}\ 1(x_1) \Rightarrow y\ x_1,\ 2(x_2) \Rightarrow z\ x_2)$
(Ax)	c c $(c : \tau_a \in \mathit{Const})$
	$\mathbf{X}$ $\lambda x.\mathbf{fix}(x)$

問 14.3 ヒルベルトシステムの各公理を自然演繹システムで証明し，その証明に対応するラムダ式を求め，表 14.1 の関係を確認せよ．

A を表 14.1 のコンビネータのいずれかとし，コンビネータ理論における式を以下の文法で定義する．

$$C ::= x \mid c \mid A \mid C\ C$$

ただし x と c はそれぞれ，Λ の変数集合および定数集合と同じ集合を代表するメタ変数とする．C の中の自由変数の集合を $FV(C)$ と書く．コンビネータ式は束縛変数を含まないため，$FV(C)$ は，単に C に含まれる変数の集合である．後に定義するコンビネータ式の同値関係と区別するために，C_1 と C_2 が同一であることを $C_1 \equiv C_2$ と書く．

*1 コンビネータ名 $\mathbf{S}$ および $\mathbf{K}$ は歴史的な由来をもつ．それ以外の名前は，対応するラムダ式の型付け規則にちなんで導入した．

コンビネータ式の型システムは，各コンビネータに対して，対応するラムダ式の型を与えることによって定義できる．変数の型環境 Γ のもとで，コンビネータ式 C が型 τ をもつことを $\Gamma \rhd C : \tau$ と書く．型付け規則を図 14.2 に与える．ここで各コンビネータ定数の型は，可能なすべての型をとるものとする．$\Gamma \rhd C : \tau$ がこの型システムで導出可能であることを $\mathcal{C} \vdash \Gamma \rhd C : \tau$ と書く．

(S) $\Gamma \rhd \mathbf{S} : (\tau_1 \to \tau_2 \to \tau_3) \to (\tau_1 \to \tau_2) \to \tau_1 \to \tau_3$

(K) $\Gamma \rhd \mathbf{K} : \tau_1 \to \tau_2 \to \tau_1$

(P) $\Gamma \rhd \mathbf{P} : \tau_1 \to \tau_2 \to \tau_1 \times \tau_2$

(F) $\Gamma \rhd \mathbf{F} : \tau_1 \times \tau_2 \to \tau_1$

(N) $\Gamma \rhd \mathbf{N} : \tau_1 \times \tau_2 \to \tau_2$

(L) $\Gamma \rhd \mathbf{L} : \tau_1 \to \tau_1 + \tau_2$

(R) $\Gamma \rhd \mathbf{R} : \tau_1 \to \tau_2 + \tau_1$

(A) $\Gamma \rhd \mathbf{A} : \tau_1 + \tau_2 \to (\tau_1 \to \tau_3) \to (\tau_2 \to \tau_3) \to \tau_3$

(X) $\Gamma \rhd \mathbf{X} : (\tau \to \tau) \to \tau$

(const) $\Gamma \rhd c : \tau$ （定数 c が型 τ をもつとき）

(var) $\Gamma\{x : \tau\} \rhd x : \tau$

(app) $$\frac{\Delta \rhd C_1 : \tau_1 \to \tau_2 \quad \Delta \rhd C_2 : \tau_1}{\Delta \rhd C_1\ C_2 : \tau_2}$$

図 14.2 コンビネータ式の型システム

各コンビネータについて，対応する閉じたラムダ式の簡約と一致するように，図 14.3 に示す簡約公理を定義する．これらの公理は，コンビネータ式に対する簡約関係を定める．この関係を $C \xrightarrow{*} C$ と書く．

問 14.4 関係 $C \xrightarrow{*} C'$ の厳密な定義を与えよ．

コンビネータ式集合はこの簡約関係のもとで 1 つの計算系を構成する．しかもこの計算系は変数による関数の生成といった複雑な操作が含まれていないため，各コンビネータの動作を実装することで容易に実現することができる．

問 14.5 コンビネータの簡約関係が型を保存することを示せ．すなわち $\mathcal{C} \vdash \Gamma \rhd C : \tau$ かつ $C \xrightarrow{*} C'$ なら $\mathcal{C} \vdash \Gamma \rhd C' : \tau$ であることを示せ．さらにこの関係がヒルベルトシステムの証明の簡約化となっていることを確かめよ．

$$
\begin{aligned}
\mathbf{S}\ M_1\ M_2\ M_3 &\Longrightarrow (M_1\ M_3)\ (M_2\ M_3)\\
\mathbf{K}\ M_1\ M_2 &\Longrightarrow M_1\\
\mathbf{F}\ (\mathbf{P}\ M_1\ M_2) &\Longrightarrow M_1\\
\mathbf{N}\ (\mathbf{P}\ M_1\ M_2) &\Longrightarrow M_2\\
\mathbf{A}\ (\mathbf{L}\ M_1)\ M_2\ M_3 &\Longrightarrow M_2\ M_1\\
\mathbf{A}\ (\mathbf{R}\ M_1)\ M_2\ M_3 &\Longrightarrow M_3\ M_1\\
\mathbf{X}\ M &\Longrightarrow M\ (\mathbf{X}\ M)
\end{aligned}
$$

図 14.3　コンビネータの簡約公理

前節で示した論理学における自然演繹システムとヒルベルトシステムとの対応関係を参考に，型付きラムダ計算を，以上定義された型付きコンビネータ計算系に翻訳することができる．

まず名前による関数の定義が，コンビネータ論理で模倣可能であることを示す．一般に変数 x を含むコンビネータ式 C に対して，$\lambda^* x.C$ を以下のように定義する．

$$
\begin{aligned}
\lambda^* x.C &= \mathbf{K}\ C\ \ (x \notin FV(C)\ \text{の場合}\ )\\
\lambda^* x.x &= \mathbf{SKK}\\
\lambda^* x.(C_1 C_2) &= \mathbf{S}(\lambda^* x.C_1)(\lambda^* x.C_2)
\end{aligned}
$$

$\lambda^* x.C$ は，一般に x を含む式 C を，x を引数として受け取り C を計算する効果をもつ(x を含まない)関数に変換する操作であり，ラムダ計算のラムダ抽象に対応する．上記の変換は，演繹定理の証明に現れる証明図の構成と一致する．実際，以下の性質が成り立つ．

定理 14.3　もし $\mathcal{C} \vdash \Gamma\{x : \tau_1\} \rhd C : \tau_2$ なら $\mathcal{C} \vdash \Gamma \rhd \lambda^* x.C : \tau_1 \to \tau_2$ である．　□

問 14.6　演繹定理を参考に定理 14.3 を証明せよ．

さらに以下の定理によって，上記変換が，ラムダ抽象式と同様にふるまう関数を実現していることが示される．

定理 14.4　$(\lambda^* x.C_1)\ C_2 \xrightarrow{*} [C_2/x]C_1$

［証明］$x \notin FV(C_1)$ の場合は，以下の等式簡約による．

$$
\text{左辺} \equiv \mathbf{K}\ C_1\ C_2 \xrightarrow{*} C_1 \equiv \text{右辺}
$$

$x \in FV(C_1)$ の場合を C_1 に関する帰納法で示す．

x の場合．$(\lambda^* x.C_1)C_2 \equiv \mathbf{SKK}\ C_2 \xrightarrow{*} C_2 \equiv [C_2/x]C_1$ である．

$C_1^1\ C_1^2$ の場合．以下の等式による．

$$
\begin{aligned}
(\lambda^* x.(C_1^1\ C_1^2))\ C_2 &\equiv \mathbf{S}\ (\lambda^* x.C_1^1)\ (\lambda^* x.C_1^2)\ C_2\\
&\xrightarrow{*} ((\lambda^* x.C_1^1)\ C_2)\ ((\lambda^* x.C_1^2)\ C_2)\\
&\xrightarrow{*} [C_2/x]C_1^1\ [C_2/x]C_1^2\ \ (\text{帰納法の仮定より})\\
&\equiv [C_2/x](C_1^1\ C_1^2)
\end{aligned}
$$

∎

ラムダ式 M のコンビネータ式への翻訳 $\overline{M}$ は以下のように与えられる．

$$
\begin{aligned}
\overline{x} &= x\\
\overline{c} &= c\\
\overline{\lambda x.M} &= \lambda^* x.\overline{M}\\
\overline{M_1\ M_2} &= \overline{M_1}\ \overline{M_2}\\
\overline{(M_1, M_2)} &= \mathbf{P}\ \overline{M_1}\ \overline{M_2}\\
\overline{M[1]} &= \mathbf{F}\ \overline{M}\\
\overline{M[2]} &= \mathbf{N}\ \overline{M}\\
\overline{1(M)} &= \mathbf{L}\ \overline{M}\\
\overline{2(M)} &= \mathbf{R}\ \overline{M}\\
\overline{(\mathbf{case}\ M_1\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3)} &= \mathbf{A}\ \overline{M_1}\ (\lambda^* x.\overline{M_2})\ (\lambda^* y.\overline{M_3})\\
\overline{\mathbf{fix}(M)} &= \mathbf{X}\ \overline{M}
\end{aligned}
$$

以下に翻訳の例を示す．

$$
\begin{aligned}
\overline{x\ y} &= xy\\
\overline{\lambda x.x} &= \mathbf{SKK}\\
\overline{\lambda x.(x, x)} &= \mathbf{S(S(KP)(SKK))(SKK)}\\
\overline{(\lambda x.(x, x))((\lambda x.x)1)} &= \mathbf{S(S(KP)(SKK))(SKK)(SKK1)}
\end{aligned}
$$

問 14.7 以下の簡約を確かめよ．

- $\overline{(\lambda x.\, x)\lambda x.\, x} \xrightarrow{*} \mathbf{SKK}$
- $\overline{(\lambda x.(x,x))\ 1} \xrightarrow{*} \mathbf{P}\ 1\ 1$

上記の問から期待されるように，ラムダ式のコンビネータへの翻訳は型を保存することが示せる．

定理 14.5 もし $\Lambda \vdash \Gamma \rhd M : \tau$ なら $\mathcal{C} \vdash \Gamma \rhd \overline{M} : \tau$ である．

［証明］$\Lambda \vdash \Gamma \rhd M : \tau$ の導出に関する帰納法で示す．最後に使用された規則により場合分けを行なう．以下，いくつかの場合を示し残りは練習問題とする．

(var), (const) および (app) の場合． 両者の型付け規則は同一であり，成立する．

(abs) の場合． ある M_1, τ_1, τ_2 があって，$\tau = \tau_1 \to \tau_2$，$M \equiv \lambda x.\, M_1$ かつ $\Lambda \vdash \Gamma\{x : \tau_1\} \rhd M_1 : \tau_2$ である．帰納法の仮定より $\mathcal{C} \vdash \Gamma\{x : \tau_1\} \rhd \overline{M_1} : \tau_2$ である．定理 14.3 より $\mathcal{C} \vdash \Gamma \rhd \lambda^* x.\, \overline{M_1} : \tau_1 \to \tau_2$ である．一方 $\overline{\lambda x.M_1} = \lambda^* x.\overline{M_1}$ である．

(pair) の場合． ある M_1, M_2, τ_1, τ_2 があって，$\tau = \tau_1 \times \tau_2$，$M \equiv (M_1, M_2)$，$\Lambda \vdash \Gamma \rhd M_1 : \tau_1$ かつ $\Lambda \vdash \Gamma \rhd M_2 : \tau_2$ である．帰納法の仮定より $\mathcal{C} \vdash \Gamma \rhd \overline{M_1} : \tau_1$ かつ $\mathcal{C} \vdash \Gamma \rhd \overline{M_2} : \tau_2$ である．規則 (app) を二回使用すると $\mathcal{C} \vdash \Gamma \rhd \mathbf{P}\ \overline{M_1}\ \overline{M_2} : \tau_1 \times \tau_2$ を得る．一方 $\overline{(M_1, M_2)} = \mathbf{P}\ \overline{M_1}\ \overline{M_2}$ である．

(proj) の場合． ある M_1, τ_1 があって，$M \equiv M_1[1]$ かつ $\Lambda \vdash \Gamma \rhd M_1 : \tau \times \tau_1$ である．帰納法の仮定より $\mathcal{C} \vdash \Gamma \rhd \overline{M_1} : \tau \times \tau_1$ である．規則 (app) より $\mathcal{C} \vdash \Gamma \rhd \mathbf{F}\ \overline{M_1} : \tau$ を得る．一方 $\overline{M_1[1]} = \mathbf{F}\ \overline{M_1}$ である．(snd) の場合も同様に示せる． ∎

問 14.8 他のすべての場合を示し，定理 14.5 の証明を完成せよ．

一般に操作的意味論は，効率よい実行のための機構を含むため，公理的意味論や表示的意味論と完全に一致することはまれである．しかし，コンビネータへの変換によって実現される操作的意味論の場合は，ラムダ計算の公理的意味論とほぼ一致することを示すことができる．ここでは型のない式に関する公理的意味論

を考えるが，以下の議論は，型付き式の公理的意味論に対してもそのまま適用可能である．(各自確かめよ)．

関数の同一性に関する以下の推論規則を考える．

$$(\mathrm{ext}^0)\ \frac{\text{任意の } Z \text{ に対して } X\ Z = Y\ Z}{X = Y}$$

この規則は，すべての引数に対して同一のふるまいをする関数は等しいことを表しており，関数を表現する計算系にとって自然な規則である．この規則は無限に多くの前提から結論を導く規則であり，そのままでは扱いにくいが，以下のように変えても一般性を失わない．

$$(\mathrm{ext})\ \frac{X\ x = Y\ x \text{ ただし } x \text{ は } x \notin FV(X) \cup FV(Y) \text{ なる変数}}{X = Y}$$

この推論規則をラムダ式の公理的意味論に加えて得られる同値関係を $\Lambda^{ext} \vdash M_1 = M_2$ と書く．すなわち，$\Lambda^{ext} \vdash M_1 = M_2$ は，Λ の公理的意味論から型情報を取り除いて得られるシステムに規則 (ext) を加えて得られるシステムである．同様にコンビネータ論理の簡約関係から導出される公理的意味論に (ext) 規則を加えて得られる同値関係を $CL^{ext} \vdash C_1 = C_2$ と書く．

問 14.9 関係 $CL^{ext} \vdash C_1 = C_2$ を厳密に定義せよ．

問 14.10 コンビネータ論理の簡約関係から導出される公理的意味論に (ext^0) 規則を加えて得られる同値関係を，$CL^{ext^0} \vdash C_1 = C_2$ と書くと，

$$CL^{ext^0} \vdash C_1 = C_2 \Longleftrightarrow CL^{ext} \vdash C_1 = C_2$$

であることを示せ．

コンビネータ式は，各コンビネータを対応するラムダ式に置き換えることによってラムダ式に変換できる．この変換を $\overline{C}$ と書く．簡単な計算により $CL^{ext} \vdash C_1 = C_2$ なら $\Lambda^{ext} \vdash \overline{C_1} = \overline{C_2}$ であることがわかる．

問 14.11 $CL^{ext} \vdash C_1 = C_2$ なら $\Lambda^{ext} \vdash \overline{C_1} = \overline{C_2}$ であることを示せ．

ラムダ式からコンビネータ式への変換に関しても同様の性質が成り立つことを示す．そのためにいくつかの補題を証明する．

補題 14.6 $x \neq y$ かつ $x \notin FV(C_2)$ のとき $[C_2/y]\lambda^*x.C_1 \equiv \lambda^*x.[C_2/y]C_1$ である．

［証明］$x \notin FV(C_1)$ の場合は以下の等式より成立する．

$$\begin{aligned}\lambda^*x.[C_2/y]C_1 &\equiv \mathbf{K}\ ([C_2/y]C_1)\\ &\equiv [C_2/y]\ (\mathbf{K}\ C_1)\\ &\equiv [C_2/y]\ \lambda^*x.C_1\end{aligned}$$

$x \in FV(C_1)$ の場合，C_1 の構造に関する帰納法で証明する．

x の場合． 両辺とも $\mathbf{S}\ \mathbf{K}\ \mathbf{K}$ である．

$C_1^1\ C_1^2$ の場合． 以下の等式より成立する．

$$\begin{aligned}\lambda^*x.[C_2/y]\ (C_1^1 C_1^2) &\equiv \lambda^*x.\ ([C_2/y]C_1^1\ [C_2/y]C_1^2)\\ &\equiv \mathbf{S}\ (\lambda^*x.[C_2/y]\ C_1^1)(\lambda^*x.[C_2/y]C_1^2)\\ &\equiv \mathbf{S}\ ([C_2/y]\lambda^*x.C_1^1)\ ([C_2/y]\lambda^*x.C_1^2)\ \text{（帰納法の仮定）}\\ &\equiv [C_2/y]\ \lambda^*x.(C_1^1\ C_1^2)\end{aligned}$$

∎

補題 14.7 $CL^{ext} \vdash C_1 = C_2$ なら $CL^{ext} \vdash \lambda^*x.C_1 = \lambda^*x.C_2$ である．

［証明］同値関係の導出に関する帰納法により，$CL^{ext} \vdash C_1 = C_2$ なら $CL^{ext} \vdash [C/x]C_1 = [C/x]C_2$ が容易に示せる．

y を与えられた式の中に現れない変数とすると以下の等式が成り立つ．

$$\begin{aligned}CL^{ext} \vdash (\lambda^*x.\ C_1)y &= [y/x]C_1 \quad \text{（定理 14.4）}\\ &= [y/x]C_2 \quad \text{（仮定）}\\ &= (\lambda^*x.C_2)y \quad \text{（定理 14.4）}\end{aligned}$$

よって規則 (ext) より $CL^{ext} \vdash \lambda^*x.C_1 = \lambda^*x.C_2$ である． ∎

補題 14.8 $CL^{ext} \vdash \overline{[M_1/x]M_2} = [\overline{M_1}/x]\overline{M_2}$

［証明］M_2 の構造に関する帰納法による．以下定数，変数，および関数に関する場合のみを示す．

c または $y(\neq x)$ の場合． 両辺とも c または y であり，成立する．

x の場合． 両辺とも $\overline{M_1}$ であり，成立する．

$\lambda y.M_2^1$ の場合． 以下の等式より成立する．

$$\begin{aligned}CL^{ext} \vdash \overline{[M_1/x]M_2} &\equiv \overline{\lambda y.[M_1/x]M_2^1}\\ &\equiv \lambda^*y.\overline{[M_1/x]\ M_2^1}\\ &= \lambda^*y.[\overline{M_1}/x]\ \overline{M_2^1}\ \text{（帰納法の仮定，補題 14.7）}\\ &\equiv [\overline{M_1}/x](\lambda^*y.\overline{M_2^1})\ \text{（補題 14.6）}\\ &\equiv [\overline{M_1}/x]\overline{M_2}\end{aligned}$$

$M_2^1\ M_2^2$ の場合． 以下の等式より成立する．

$$\begin{aligned}CL^{ext} \vdash \overline{[M_1/x]\ M_2} &\equiv \overline{[M_1/x]M_2^1\ [M_1/x]M_2^2}\\ &\equiv \overline{[M_1/x]M_2^1}\ \overline{[M_1/x]M_2^2}\\ &= [\overline{M_1}/x]\overline{M_2^1}\ [\overline{M_1}/x]\ \overline{M_2^2}\ \text{（帰納法の仮定）}\\ &\equiv [\overline{M_1}/x]\ \overline{M_2}\end{aligned}$$

∎

問 14.12 補題 14.8 のその他の場合をすべて行ない，証明を完成せよ．

以上の準備の下で以下の定理が証明できる．

定理 14.9 $\Lambda^{ext} \vdash M_1 = M_2$ なら $CL^{ext} \vdash \overline{M_1} = \overline{M_2}$ である． □

問 14.13 定理 14.9 を以下の手順で証明せよ．

(i) ラムダ式の各簡約公理 $M_1 \Longrightarrow M_2$ に対して $CL^{ext} \vdash \overline{M_1} = \overline{M_2}$ が成立することを示せ．

(ii) ラムダ式の文脈の定義に関する帰納法により $CL^{ext} \vdash \overline{M_1} = \overline{M_2}$ なら $CL^{ext} \vdash \overline{C[M_1]} = \overline{C[M_2]}$ であることを示せ．

(iii) ある $x \notin FV(M_1) \cup FV(M_2)$ に対して $CL^{ext} \vdash \overline{M_1\ x} = \overline{M_2\ x}$ なら，$CL^{ext} \vdash \overline{M_1} = \overline{M_2}$ であることを示せ．

(iv) 公理的意味論の規則 (sym) と (trans) に関しても同様の性質を示し，証明を完成せよ．

(c) コンビネータ式の実行

コンビネータの組合せに翻訳されたラムダ式をコンピュータで実行するためには，**S**，**K** を始めとする各コンビネータを，それらの簡約公理を実現する操作と

して実現し，それらの組合せからなるプログラムを，適当な**評価戦略**にしたがって実行すればよい．

コンビネータ簡約は，ラムダ計算の簡約関係同様非決定的であり，同一のコンビネータ式に対して複数の簡約系列が存在する．評価戦略は，複数ある簡約の可能性から，一定の仕方でつぎに実行する操作を選び出す規則である．コンビネータによるラムダ式の実行で通常採用される評価戦略は，最も外側にある最も左側の実行可能な操作を実行する，**最外最左簡約**とよばれる戦略である．図 14.4 に実行例を示す．

$$
\begin{aligned}
&\overline{(\lambda x.(x,x))((\lambda x.x)1)}\\
&\quad\equiv\quad \mathbf{S(S(KP)(SKK))(SKK)(SKK1)}\\
&\quad\longrightarrow\quad \mathbf{S(KP)(SKK)(SKK1)(SKK(SKK1))}\\
&\quad\longrightarrow\quad \mathbf{KP(SKK1)(SKK(SKK1))(SKK(SKK1))}\\
&\quad\longrightarrow\quad \mathbf{P(SKK(SKK1))(SKK(SKK1))}\\
&\quad\longrightarrow\quad \mathbf{P(K(SKK1)(K(SKK1)))(SKK(SKK1))}\\
&\quad\longrightarrow\quad \mathbf{P(SKK1)(SKK(SKK1))} \longrightarrow \mathbf{P(K1(K1))(SKK(SKK1))}\\
&\quad\longrightarrow\quad \mathbf{P1(SKK(SKK1))} \longrightarrow \mathbf{P1(K(SKK1)(K(SKK1)))}\\
&\quad\longrightarrow\quad \mathbf{P1(SKK1)} \longrightarrow \mathbf{P1(K1(K1))} \longrightarrow \mathbf{P\ 1\ 1}\\
&\quad\equiv\quad \overline{(1,1)}
\end{aligned}
$$

図 **14. 4** コンビネータによるラムダ式の実行例

この例を注意深く見ればわかるように，引数である $(\lambda x.x)1$ がコピーされ，この簡約が複数回行なわれている．このように，この戦略は関数の引数の簡約が複数回行なわれるという無駄が生じる可能性があるが，再帰的関数を含むプログラムに対して，最終結果に至る簡約系列があれば，必ずその結果に至る最も強力な戦略である．

問 14. 14 問 11.7で作成した構文解析プログラムを使って，ラムダ式をコンビネータに翻訳し実行する処理系を以下の手順で作成せよ．

（i）問 11.7 で定義したラムダ式データ型をコンビネータ式に翻訳する関数を定義せよ．

（ii）コンビネータ式を表現するデータ型，およびそのプリント関数を定義せよ．ただし本節で定義したコンビネータに加え，数値の比較および算術演算を行なうコンビネータ EQ, ADD, SUB, MUL, DIV を含むものとする．

（iii）与えられたコンビネータ式が変換可能か否かを判定する関数，および最外最左の簡約を 1 回行なう関数をそれぞれ定義し，これらを用いて，可能な限り最外最左簡約を実行する関数を定義せよ．

（iv）以下の処理を行なうトップレベルプログラム `read_eval_print` を定義せよ．

(a) 問 11.7 で定義した構文解釈関数によりラムダ式を入力し，入力したラムダ式データをプリントする．

(b) ラムダ式をコンビネータ式に変換し，変換したコンビネータ式をプリントする．

(c) コンビネータ式に対して最外最左簡約を実行し，結果をプリントする．

このプログラムは以下のような動作をする．

```
->((fn x => x) (fn x => x));
Source expr:  ((fn x=>x) (fn x=>x))
Compiled to:  S K K (S K K)
Reduced to:  S K K
```

（v）以下の Minimal^0 プログラムに対応するラムダ式を入力し結果が 5050 となることを確かめよ．

```
let fun factorial n = if n = 0 then 1
                      else n * (factorial (n - 1))
in factorial 10 end ;
```

ヒルベルトシステムの公理集合は，定理 14.2 に示されるように，自然演繹システムに関して完全であり，新たな公理を追加しても論理学の証明能力が強くなることはない．このことはコンビネータ論理の基本コンビネータが，ラムダ式を翻訳する上で十分であることに対応する．しかしながら，新たな基本コンビネータの導入により，より簡潔で効率よい翻訳が可能になることがある．以下のコンビネータを考える．

コンビネータと対応するラムダ式

(I) **I** $\lambda x.x$

(B) **B** $\lambda x.\lambda y.\lambda z.x\ (y\ z)$

(C) **C** $\lambda x.\lambda y.\lambda z.x\ z\ y$

これらは以下の簡約公理を満たす．

$$\begin{aligned}
\mathbf{I}\,C &\Longrightarrow C \\
\mathbf{B}\,C_1\,C_2\,C_3 &\Longrightarrow C_1\,(C_2\,C_3) \\
\mathbf{C}\,C_1\,C_2\,C_3 &\Longrightarrow C_1\,C_3\,C_2
\end{aligned}$$

問 14.15

（i）コンビネータへの型付け規則と，対応するヒルベルトシステムの公理を与えよ．

（ii）簡約公理が型を保存することを示せ．

（iii）以下の等式が成り立つことを示せ．

$$\begin{aligned}
CL^{ext} &\vdash \mathbf{S}\,\mathbf{K}\,\mathbf{K} = \mathbf{I} \\
CL^{ext} &\vdash \mathbf{S}\,(\mathbf{K}\,C_1)\,(\mathbf{K}\,C_2) = \mathbf{K}\,(C_1\,C_2) \\
CL^{ext} &\vdash \mathbf{S}\,(\mathbf{K}\,C)\,\mathbf{I} = C \\
CL^{ext} &\vdash \mathbf{S}\,(\mathbf{K}\,C_1)\,C_2 = \mathbf{B}\,C_1\,C_2 \\
CL^{ext} &\vdash \mathbf{S}\,C_1\,(\mathbf{K}\,C_2) = \mathbf{C}\,C_1\,C_2
\end{aligned}$$

ラムダ抽象式に対応するコンビネータ生成の定義 $\lambda^*x.C$ に，問 14.15 の結果得られる等式を以下のように統合することによって，より簡潔で効率よいコンビネータへの変換を実現できる．

$$\lambda^*x.C = \mathbf{K}\,C \quad (x \notin FV(C)\ \text{の場合})$$

$$\lambda^*x.x = \mathbf{I}$$

$$\lambda^*x.(C_1C_2) = \begin{cases}
\mathbf{K}\,(C_1'\,C_2') & (\lambda^*x.C_1 = \mathbf{K}\,C_1', \\
 & \quad \lambda^*x.C_2 = \mathbf{K}\,C_2'\ \text{の場合}) \\
C_1' & (\lambda^*x.C_1 = \mathbf{K}\,C_1', \\
 & \quad \lambda^*x.C_2 = \mathbf{I}\ \text{の場合}) \\
\mathbf{B}\,C_1'\,(\lambda^*x.C_2) & (\lambda^*x.C_1 = \mathbf{K}\,C_1'\ \text{の場合}) \\
\mathbf{C}\,(\lambda^*x.C_1)\,C_2' & (\lambda^*x.C_2 = \mathbf{K}\,C_2'\ \text{の場合}) \\
\mathbf{S}\,(\lambda^*x.C_1)\,(\lambda^*x.C_2) & (\text{上記以外の場合})
\end{cases}$$

問 14.16 問 14.14 で作成したプログラムにこの最適化処理を加え，階乗を計算するプログラムを入力し，結果を比較せよ．

この問題を実際に解いてみればわかるとおり，上記の冗長なコンビネータによって，ラムダ式をより簡潔で効率よいコンビネータに変換できる．

§14.2 自然意味論

β 変換を直接実行することなくラムダ計算を機械的に実現するもう 1 つの代表的な方法は，表示的意味論の考え方に従い，環境を使って変数の束縛を実現する自然意味論である．自然意味論の考え方を理解するため，表示的意味論における変数 x および関数適用 $(\lambda x.M)\ N$ の意味の定義を思い出してみよう．型情報を省略すると，以下の等式で表現される．

$$\begin{aligned}
[\![x]\!]\eta &= \eta(x) \\
[\![(\lambda x.M)\ N]\!]\eta &= [\![M]\!]\eta\{x : [\![N]\!]\eta\}
\end{aligned}$$

この意味定義は，以下のような評価手順を示唆している．

（i）ラムダ式 M の値の計算は，M に含まれる自由変数の値を与える環境の下で，M の値を計算することによって行なう．この考えのもとでは，β 変換の際の変数への代入は，変数の値を環境から取り出すことに対応する．

（ii）関数適用 $(\lambda x.M)\ N$ の実行は，まず引数 N の値 v を求め，仮引数 x と値 v の対応を加えた環境の下で関数本体 M を評価することによって行なう．

（iii）上記の効果を達成するために，$\lambda x.M$ の評価では，M の計算は行なわず，関数が適用されたときの評価のために，現在の環境と $\lambda x.M$ の組を保存する．

最外最左評価戦略と比較した大きな特徴は，関数の適用の前にその引数を評価することである．この評価戦略は**適用順評価**(appl icat iveorder eval uat ion)とよぶ通常の汎用プログラミング言語で広く使用されている評価方法である．この評価戦略では，引数の評価は一回しか行なわれないため，プログラムのより効率的な実行が可能である．

しかしながら，この評価戦略のみでは $\mathbf{fix}(M)$ の評価がうまく実現できない．$\mathbf{fix}(M)$ の型の制約から M は関数型をもつはずであり，したがってその評価の結

果は，$\lambda f.\, M'$ の形のラムダ式と環境 Eの組のはずである．そこで，簡約 $\mathbf{fix}(M) \longrightarrow M(\mathbf{fix}(M))$ に対応する適用順評価は，E を f の値で拡張した環境の下で M' を評価することになるが，f のとるべき値は，この $\mathbf{fix}(M)$ そのものを評価した結果の値のはずであり，あらかじめ用意できない．この困難を解決するために，$\mathbf{fix}(M)$ の構文を，$\mathtt{fun}\ f\ x = M$ の形をした再帰的関数定義の表現である $\mathbf{fix}(\lambda f.\lambda x.\, M)$ の形に制限し，f の値の束縛を，この関数が適用される時点まで遅延することにする．以下 $\mathbf{fix}(\lambda f.\lambda x.\, M)$ を $\mu.\, f.\, x.\, M$ と書く．

以上の方針に従い，「環境 E のもとでラムダ式 M が結果 v を計算する」という性質を表す以下の形の評価関係を定めたものが，自然意味論である．

$$E \vdash M \Downarrow v$$

環境 E は変数の集合から値の集合への関数である．v は以下の文法で定義される値の集合を表す．

$$v ::= c \mid cls(E, \lambda x.\, M) \mid rec(E, \mu.f.x.\, M) \mid (v, v) \mid 1(v) \mid 2(v) \mid wrong$$

$cls(E, \lambda x.\, M)$ は，ラムダ抽象 $\lambda x.\, M$とそれが定義されたときの環境 E の組を保存するデータ構造であり，関数閉包(funct ioncl osure)とよばれる．$rec(E, \mu.\, f.\, xM)$ は，再帰的関数定義式 $\mu.\, f.\, xM$ とそれが定義されたときの環境 E の組を保存するデータ構造であり，再帰的関数閉包とよばれる．この再帰的関数閉包が引数 v に適用されると，f に $rec(E, \mu.\, f.\, xM)$ を対応させる束縛と x に引数 v を対応させる 2 つの束縛が環境 E に加えられ，M が実行される．$wrong$ は，実行時のエラーを表す．

評価関係の定義を図 14.5 に与える．ただし，実行時にエラーとなる推論規則は省略してある．完全な評価関係は，以下のいずれかの場合に $wrong$ を返す規則を追加して得られるものである．

(i) 部分式が，規則が要求する形とは違う値を返す場合．
(ii) 部分式のいずれかが $wrong$ となる場合．

自然意味論に関しても型の健全性を示すことができる．そのために，まず値のもつ型を定義する．値 v が型 τ をもつことを $\models v : \tau$ と書き，実行時環境 E が型

$$E \vdash c \Downarrow c \qquad E\{x : v\} \vdash x \Downarrow v \qquad E \vdash \lambda x.M \Downarrow cls(E, \lambda x.M)$$

$$E \vdash \mu.f.\, x.\, M \Downarrow rec(E, \mu.\, fx.M)$$

$$\frac{E \vdash M_1 \Downarrow cls(E_1, \lambda x.M_1') \quad E \vdash M_2 \Downarrow v_1 \quad E_1\{x : v_1\} \vdash M_1' \Downarrow v_2}{E \vdash M_1\ M_2 \Downarrow v_2}$$

$$\frac{\begin{array}{c} E \vdash M_1 \Downarrow rec(E_1, \mu.f.\, xM_1') \\ E \vdash M_2 \Downarrow v_1 \\ E_1\{f : rec(E_1, \mu.f.\, xM_1'), x : v_1\} \vdash M_1' \Downarrow v_2 \end{array}}{E \vdash M_1\ M_2 \Downarrow v_2}$$

$$\frac{E \vdash M_1 \Downarrow v_1 \quad E \vdash M_2 \Downarrow v_2}{E \vdash (M_1, M_2) \Downarrow (v_1, v_2)} \qquad \frac{E \vdash M \Downarrow (v_1, v_2)}{E \vdash M[1] \Downarrow v_1}$$

$$\frac{E \vdash M \Downarrow (v_1, v_2)}{E \vdash M[2] \Downarrow v_2} \qquad \frac{E \vdash M \Downarrow v}{E \vdash 1(M) \Downarrow 1(v)} \qquad \frac{E \vdash M \Downarrow v}{E \vdash 2(M) \Downarrow 2(v)}$$

$$\frac{E \vdash M_1 \Downarrow 1(v_1) \quad E\{x : v_1\} \vdash M_2 \Downarrow v_2}{E \vdash (\mathbf{case}\ M_1\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3) \Downarrow v_2}$$

$$\frac{E \vdash M_1 \Downarrow 2(v_1) \quad E\{y : v_1\} \vdash M_3 \Downarrow v_2}{E \vdash (\mathbf{case}\ M_1\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3) \Downarrow v_2}$$

図 **14.5** 自然意味論の評価関係

環境 Γ を満たすことを $\models E : \Gamma$ と書く．この 2 つの関係は，相互再帰的に以下のように与えられる．

$$\begin{aligned}
\models c : \tau &\iff c : \tau \in Const \text{ のとき} \\
\models cls(E, \lambda x.\, M) : \tau_1 \to \tau_2 &\iff \text{ある } \Gamma \text{ に対して} \models E : \Gamma \text{かつ} \\
&\qquad \Gamma \rhd \lambda x.\, M : \tau_1 \to \tau_2 \\
\models rec(E, \mu.f.\, xM) : \tau_1 \to \tau_2 &\iff \text{ある } \Gamma \text{ に対して} \models E : \Gamma \text{かつ} \\
&\qquad \Gamma \rhd \mu.\, f.\, x.\, M : \tau_1 \to \tau_2 \\
\models (v_1, v_2) : \tau_1 \times \tau_2 &\iff \models v_1 : \tau_1, \models v_2 : \tau_2 \\
\models 1(v_1) : \tau_1 + \tau_2 &\iff \models v_1 : \tau_1 \\
\models 2(v_1) : \tau_1 + \tau_2 &\iff \models v_1 : \tau_2 \\
\models E : \Gamma &\iff x \in dom(E) \text{ なるすべての } x \text{ に対して} \\
&\qquad \models E(x) : \Gamma(x)
\end{aligned}$$

以下の定理が成り立つ．

定理 14.10 もし $\Gamma \rhd M : \tau$, $\models E : \Gamma$, かつ $E \vdash M \Downarrow v$ なら，$\models v : \tau$ である．

［証明］計算 $E \vdash M \Downarrow v$ の長さを，この関係を導出するために適用した規則の回数とし，この定理を $E \vdash M \Downarrow v$ の長さに関する帰納法で証明する．M の構造で場合分けを行なう．以下いくつかの場合を示し，その他は練習問題とする．

$\lambda x.M_1$ の場合． $v = cls(E, \lambda x.M_1)$ であり，$\Gamma \rhd \lambda x.M_1 : \tau$ であり，よって値の型の定義によって要求される Γ を，この Γ ととれば $\models cls(E, \lambda x.M_1) : \tau$ である．

$M_1\ M_2$ の場合． ある τ_1 があって，$\Gamma \rhd M_1 : \tau_1 \to \tau$ かつ $\Gamma \rhd M_2 : \tau_1$ である．計算の定義より，ある v_1 があって $E \vdash M_1 \Downarrow v_1$ である．帰納法の仮定より $\models v_1 : \tau_1 \to \tau$ である．値の定義より $v_1 = cls(E_1, \lambda x M_1')$ かまたは $v_1 = rec(E_1, \mu.f.x.M_1')$ である．$v_1 = cls(E_1, \lambda x.M_1')$ と仮定する．値の型の定義より，ある Γ' があって $\models E_1 : \Gamma'$ かつ $\Gamma' \rhd \lambda x.M_1' : \tau_1 \to \tau$ である．計算の定義より $E \vdash M_2 \Downarrow v_2$．帰納法の仮定より $\models v_2 : \tau_1$．$v_2 \neq wrong$ であるから，$E_1\{x : v_2\} \vdash M_1' \Downarrow v$．定義より，$\models E_1\{x : v_2\} : \Gamma'\{x : \tau_1\}$．よって帰納法の仮定を $E_1\{x : v_2\} \vdash M_1' \Downarrow v$ に適用すると $\models v : \tau$ を得る．また，$v_1 = rec(E_1, \mu.f.x.M_1')$ と仮定する．値の型の定義より，ある Γ' があって $\models E_1 : \Gamma'$ かつ $\Gamma' \rhd \mu.f.x.M_1' : \tau_1 \to \tau$ である．計算の定義より $E \vdash M_2 \Downarrow v_2$．帰納法の仮定より $\models v_2 : \tau_1$．$v_2 \neq wrong$ であるから，$E_1\{f : v_1\}\{x : v_2\} \vdash M_1' \Downarrow v$．定義より，$\models E_1\{f : v_1\}\{x : v_2\} : \Gamma'\{f : \tau_1 \to \tau\}\{x : \tau_1\}$．よって帰納法の仮定を $E_1\{f : v_1\}\{x : v_1\} \vdash M_1' \Downarrow v$ に適用すると $\models v : \tau$ である． ∎

問 14.17 定理 14.10 の証明のその他の場合を示し，証明を完成せよ．

以上の自然意味論の評価関係 $E \vdash M \Downarrow v$ は，適用順評価アルゴリズムの再帰的な記述とみなすことができる．すなわち，評価関係の定義から，

$$eval(E, M) = v \Longleftrightarrow E \vdash M \Downarrow v$$

なる性質をもつ評価アルゴリズム *eval* が直接導かれる．図 14.6 にその定義の一部を与える．

$$eval(E, c) = c$$

$$eval(E, x) = E(x)$$

$$eval(E, \lambda x.M) = cls(E, \lambda x.M)$$

$$eval(E, \mu.fx.M) = rec(E, \mu.f.xM)$$

$$\begin{aligned} eval(E, M_1\ M_2) = \text{let } & v_1 = eval(E, M_1) \\ & v_2 = eval(E, M_1) \\ \text{in case } & v_1 \text{ of} \\ & cls(E_1, \lambda x.M_1') \Rightarrow eval(E_1\{x : v_2\}, M_1'), \\ & rec(E_1, \mu.f.xM_1') \Rightarrow eval(E_1\{f : v_1\}\{x : v_2\}, M_1') \end{aligned}$$

$$eval(E, (M_1, M_2)) = (eval(E, M_1), eval(E, M_2))$$

図 **14.6** ラムダ式の評価関数の定義の一部

問 14.18 評価関数 *eval* の定義を完成し，さらに，以下の手順で，問 14. 14 で作成したラムダ式の評価プログラムを変更することにより，ラムダ式のインタープリタを作成せよ．

（i）v で表される値を表すデータ型と，そのプリント関数を定義せよ．ただし関数閉包は単に `closure` とプリントするものとする．

（ii）コンビネータへの翻訳と実行の部分を，*eval* 関数を実行する関数に置き換え，ラムダ式のインタープリタを完成せよ．

§14.3 SECD 機械へのコンパイル

自然意味論の評価関数 *eval* は，再帰関数が使用可能な既存のプログラミング言語によって容易に実現できる．このように，高水準プログラミング言語を別の高水準プログラミング言語で解釈し実行する方式を，インタープリタ(解釈プログラム)方式とよぶ．インタープリタは，既存の言語を使い簡単に実現できるという利点があるが，プログラムを効率よく実行するためには，別のプログラミング言語で解釈するのではなく，プログラムを機械語コードに変換するコンパイラを構築する必要がある．自然意味論で実現される意味をコンピュータで実現するためには，自然意味論の定義に使用されている諸概念の中で，コンピュータの機能で直接実現できないものを，機械語に翻訳する方法を確立する必要がある．

自然意味論の定義の中で，コンピュータの機能に対応していない主な要素は，評

価関数の再帰的な適用機構である．再帰的な関数適用を実装するためには，再帰的処理に入る前に，現在の処理の情報を記録し，再帰的処理が終了した後再開する必要がある．これは，再開のための制御情報と途中まで得られた計算結果をスタックに記憶することによって実現できる．この考えに従い自然意味論を実現するモデルが **SECD 機械**である．SECD 機械は現在のコンピュータの細部を抽象したものとみなすことができる．このようなモデルを，プログラミング言語のための**抽象機械**(abstract machine)とよぶ．

SECD 機械は，スタック S，環境 E，コード列 C およびダンプ D の 4 つの構成要素からなる．スタック S は，これまでに得られた計算の途中結果の値 V を保持するメモリー領域であり，最も最近得られた計算結果がリストの先頭要素となるように管理される．S の構造は以下の BNF 文法で与えられる．

$$
\begin{aligned}
S &::= nil \mid V::S \\
V &::= c \mid Cls(E,x,C) \mid Rec(E,f,x,C) \mid (V,V) \mid 1(V) \mid 2(V) \mid Wrong
\end{aligned}
$$

nil は空のスタックを表し，$V::S$ は，S の先頭に V を付け加えて得られるスタックを表す．値の構造は，関数閉包と再帰的関数閉包の中にラムダ式ではなくコード列が入ることを除けば，自然意味論における値と同様である．

環境 E は変数と値の対応関係を保持する環境であり，自然意味論で使用した環境と同様の構造をもつ．

コード列 C は，機械の動作を指示する命令コード I のリストであり，以下の文法で与えられる構造をもつ．

$$
\begin{aligned}
C &::= nil \mid I::C \\
I &::= Const(c) \mid Acc(x) \mid MakeCls(x,C) \mid MakeRec(f,x,C) \mid App \\
&\quad \mid Return \mid Pair \mid Fst \mid Snd \mid Inl \mid Inr \mid Case((x,C),(x,C))
\end{aligned}
$$

ダンプ D は，再帰処理の後再開するための制御情報を保持するスタックである．計算を再開するためには，現在の環境と再開後のコード列が分かればよい．ダンプ D はこの組を保持する以下の構造をもつスタックである．

$$D ::= nil \mid (C,E)::D$$

SECD 機械は，C の先頭にあるコードを SECD で定まる機械の状態に対して実行し，状態を更新することを繰り返す機械であり，その形式的な定義は以下の形の状態遷移関係によって与えられる．

$$(S,E,C,D) \longrightarrow (S',E',C',D')$$

状態遷移関係を図 14.7 に与える．ただし，この規則に存在しない機械状態に対しては，機械は $Wrong$ を出力し停止するものとする．すなわち，図に定める規則がいずれも該当しない場合には，

$$(S,E,C,D) \longrightarrow (Wrong::S,E,nil,D)$$

なるの遷移が起こるものとする．

$$
\begin{aligned}
(S,E,Const(c)::C,D) &\longrightarrow (c::S,E,C,D) \\
(S,E\{x:V\},Acc(x)::C,D) &\longrightarrow (V::S,E\{x:V\},C,D) \\
(S,E,MakeCls(x,C_0)::C,D) &\longrightarrow (Cls(E,x,C_0)::S,E,C,D) \\
(S,E,MakeRec(f,x,C_0)::C,D) &\longrightarrow (Rec(E,f,x,C_0)::S,E,C,D) \\
(V::Cls(E_0,x,C_0)::S,E,App::C,D) &\longrightarrow (S,E_0\{x:V\},C_0,(C,E)::D) \\
(V::Rec(E_0,f,x,C_0)::S,E,App::C,D) &\longrightarrow (S,E_0\{f:Rec(E_0,f,x,C_0)\}\{x:V\}, \\
&\qquad C_0,(C,E)::D) \\
(S,E,Return::C,(C_0,E_0)::D) &\longrightarrow (S,E_0,C_0,D) \\
(V_1::V_2::S,E,Pair::C,D) &\longrightarrow ((V_2,V_1)::S,E,C,D) \\
((V_1,V_2)::S,E,Fst::C,D) &\longrightarrow (V_1::S,E,C,D) \\
((V_1,V_2)::S,E,Snd::C,D) &\longrightarrow (V_2::S,E,C,D) \\
(V::S,E,Inl::C,D) &\longrightarrow (1(V)::S,E,C,D) \\
(V::S,E,Inr::C,D) &\longrightarrow (2(V)::S,E,C,D) \\
(1(V)::S,E,Case((x,C_1),(y,C_2))::C,D) &\longrightarrow (S,E\{x:V\},C_1,(C,E)::D) \\
(2(V)::S,E,Case((x,C_1),(y,C_2))::C,D) &\longrightarrow (S,E\{y:V\},C_2,(C,E)::D)
\end{aligned}
$$

図 14.7 SECD 機械の実行規則

各状態の組 (S,E,C,D) に対して高々 1 つの規則しか定義されていないため，この規則は決定的な計算の表現となっている．したがって各コードに対して，その動作を実現する機械語列を用意すれば，容易にコンピュータで実現可能である．

この SECD 機械を使って Λ の自然意味論を実現するためには，ラムダ式を SECD 機械コードに翻訳する方法を構築しなければならない．翻訳の主な仕事は，

木構造をなすラムダ式を一次元のコード列に変換することである．命令コード I のみからなるコード列を $[I]$ と書き，コード列 C_1 と C_2 を連結して得られるコード列を $C_1@C_2$ と書く．翻訳アルゴリズムを図 14.8 に与える．

$$
\begin{aligned}
[\![c]\!] &= [Const(c)] \\
[\![x]\!] &= [Acc(x)] \\
[\![\lambda x.M]\!] &= [MakeCls(x, [\![M]\!]@[Return])] \\
[\![\mu f \lambda x.M]\!] &= [MakeRec(f, x, [\![M]\!]@[Return])] \\
[\![M_1\ M_2]\!] &= [\![M_1]\!]@[\![M_2]\!]@[App] \\
[\![(M_1, M_2)]\!] &= [\![M_1]\!]@[\![M_2]\!]@[Pair] \\
[\![M[1]]\!] &= [\![M]\!]@[Fst] \\
[\![M[2]]\!] &= [\![M]\!]@[Snd] \\
[\![1(M)]\!] &= [\![M]\!]@[Inl] \\
[\![2(M)]\!] &= [\![M]\!]@[Inr] \\
[\![(\textbf{case}\ M_0\ \textbf{of}\ 1(x) \Rightarrow M_1,\ 2(y) \Rightarrow M_2)]\!] & \\
&= [\![M_0]\!]@[Case((x, [\![M_1]\!]@[Return]), (y, [\![M_2]\!]@[Return]))]
\end{aligned}
$$

図 **14.8**　ラムダ式の SECD コードへの翻訳アルゴリズム

ラムダ式 M を翻訳し，そのコードを環境 E の下で SECD 機械で実行した結果が V であることを，$Eval(E, M) = V$ と書く．すなわち，

$$Eval(E, M) = V \Longleftrightarrow (nil, E, [\![M]\!], nil) \xrightarrow{*} ([V], E, nil, nil)$$

である．この関係は，自然意味論で定義された操作的意味を忠実に実現するものであることを証明できる．そのために，自然意味論における値 v と環境 E の SECD 機械における表現 $\overline{v}$ と $\overline{E}$ をそれぞれ以下のように定義する．

$$
\begin{aligned}
\overline{c} &= c \\
\overline{cls(E, \lambda x.M)} &= Cls(\overline{E}, x, \overline{M}@[Return]) \\
\overline{rec(E, \mu.f.x.M)} &= Rec(\overline{E}, f, x, \overline{M}@[Return]) \\
\overline{(v_1, v_2)} &= (\overline{v_1}, \overline{v_2}) \\
\overline{1(v)} &= 1(\overline{v}) \\
\overline{2(v)} &= 2(\overline{v}) \\
\overline{wrong} &= Wrong \\
\overline{E} &= \{x : \overline{E(x)} \mid x \in dom(E)\}
\end{aligned}
$$

自然意味論と SECD 機械の実行には以下の関係がある．

定理 14.11　もし $E \vdash M \Downarrow v$ かつ $v \neq Wrong$ なら $(S, \overline{E}, [\![M]\!]@C, D) \xrightarrow{*} (\overline{v}::S, \overline{E}, C, D)$ である．

［証明］$E \vdash M \Downarrow v$ の長さに関する帰納法で証明する．M の構造により場合分けを行なう．

c および x の場合．　定義より明らか．

$\lambda x.M$ の場合．　$v = Cls(E, x, M)$ であり，$\overline{v} = Cls(\overline{E}, x, [\![M]\!]@[Return])$ である．また $[\![\lambda x.M]\!] = [MakeCls(x, [\![M]\!]@[Return])]$ である．一方

$$
\begin{aligned}
&(S, \overline{E}, MakeCls(x, [\![M]\!]@[Return])::C, D) \\
&\xrightarrow{*} \quad (Cls(\overline{E}, x, [\![M]\!]@[Return])::S, \overline{E}, C, D)
\end{aligned}
$$

である．

$\mu.f.x.M$ の場合．　$\lambda x.M$ の場合と同様．

$M_1\ M_2$ の場合．　$v \neq Wrong$ であるから，評価関係の定義より

$$
\begin{aligned}
&E \vdash M_1 \Downarrow cls(E_1, \lambda x.M_1'), \\
&E \vdash M_2 \Downarrow v_1, \text{かつ} \\
&E_1\{x : v_1\} \vdash M_1' \Downarrow v
\end{aligned}
$$

であるか，または

$$
\begin{aligned}
&E \vdash M_1 \Downarrow rec(E_1, \mu.f.x.M_1'), \\
&E \vdash M_2 \Downarrow v_1, \text{かつ} \\
&E_1\{f : rec(E_1, \mu.f.x.M_1')\}\{x : v_1\} \vdash M_1' \Downarrow v
\end{aligned}
$$

である．以下は $E \vdash M_1 \Downarrow cls(E_1, x, M_1')$ の場合のみ示し，他の場合は練習問題とする．

$[\![M_1\ M_2]\!] = [\![M_1]\!]@[\![M_2]\!]@[App]$ である．帰納法の仮定および値の対応の定義より，

$$
\begin{aligned}
&(S, \overline{E}, [\![M_1]\!]@[\![M_2]\!]@[App]@C, D) \\
&\xrightarrow{*} \quad (Cls(\overline{E_1}, x, [\![M_1']\!]@[Return])::S, \overline{E}, [\![M_2]\!]@[App]@C, D)
\end{aligned}
$$

である．M_2 の評価に対する帰納法の仮定より，

$$(Cls(\overline{E_1}, x, [\![M_1']\!]@[Return])::S, \overline{E}, [\![M_2]\!]@[App]@C, D)$$
$$\xrightarrow{*} \quad (\overline{v_1}::Cls(\overline{E_1}, x, [\![M_1']\!]@[Return])::S, \overline{E}, App::C, D)$$

である．SECD 機械の動作の定義より

$$(\overline{v_1}::Cls(\overline{E_1}, x, [\![M_1']\!]@[Return])::S, \overline{E}, App::C, D)$$
$$\longrightarrow \quad (S, \overline{E_1\{x : \overline{v_1}\}}, [\![M_1']\!]@[Return], (C, \overline{E})::D)$$

一方，値の対応の定義より，$\overline{E_1\{x : v_1\}} = \overline{E_1}\{x : \overline{v_1}\}$ であるから M_1' の評価に対しても帰納法の仮定が使えて，

$$(S, \overline{E_1}\{x : \overline{v_1}\}, [\![M_1']\!]@[Return], (C, \overline{E})::D)$$
$$\xrightarrow{*} \quad (\overline{v}::S, \overline{E_1}\{x : \overline{v_1}\}, [Return], (C, \overline{E})::D)$$

である．SECD 機械の動作の定義より

$$(\overline{v}::S, \overline{E_1}\{x : \overline{v_1}\}, [Return], (C, \overline{E})::D) \longrightarrow (\overline{v}::S, \overline{E}, C, D)$$

である．以上より，

$$(S, \overline{E}, [\![M_1]\!]@[\![M_2]\!]@[App]@C, D) \xrightarrow{*} (\overline{v}::S, \overline{E}, C, D)$$

である．

(M_1, M_2) の場合．$v \neq wrong$ であるから，評価関係の定義よりある v_1, v_2 があって，$E \vdash M_1 \Downarrow v_1$，$E \vdash M_2 \Downarrow v_2$ かつ $v = (v_1, v_2)$ である．一方 $[\![(M_1, M_2)]\!]C = [\![M_1]\!]@[\![M_2]\!]@[Pair]@C$ である．帰納法の仮定より，

$$(S, \overline{E}, [\![M_1]\!]@[\![M_2]\!]@[Pair]@C, D) \longrightarrow (\overline{v_1}::S, \overline{E}, [\![M_2]\!]@[Pair]@C, D)$$

である．M_2 に対する帰納法の仮定より，

$$(\overline{v_1}::S, \overline{E}, [\![M_2]\!]@[Pair]@C, D) \longrightarrow (\overline{v_2}::\overline{v_1}::S, \overline{E}, Pair::C, D)$$

SECD 機械の動作の定義より，

$$(\overline{v_2}::\overline{v_1}::C, \overline{E}, Pair::C, D) \longrightarrow ((\overline{v_1}, \overline{v_2})::S, \overline{E}, C, D)$$

以上より，

$$(S, \overline{E}, [\![M_1]\!]@[\![M_2]\!]@[Pair]@C, D) \xrightarrow{*} (\overline{v}::S, \overline{E}, C, D)$$

である． ∎

問 14.19 M_1 M_2 で $E \vdash M_1 \Downarrow rec(E, f, x, M_1')$ となる場合を含むその他のすべての場合を証明し，上記定理を完成せよ．

この定理から直ちに以下の関係が導かれる．

系 14.12 もし $eval(E, M) = v$ なら，$Eval(\overline{E}, [\![M]\!]) = \overline{v}$ である． □

問 14.20 問 14.14 以下の手順で，問 14.14 で作成したラムダ式の評価プログラムを変更することにより，ラムダ式の SECD 機械へのコンパイラを作成せよ．

（i）SECD 機械コードを表すデータ型と，そのプリント関数を定義せよ．

（ii）コンビネータへの翻訳関数を，SECD 機械コードへの翻訳関数に置き換え，ラムダ式のコンパイラを完成せよ．

15 型の自動推論と型の多相性

これまで学んできたラムダ計算は，型付きのシステムであるが，ラムダ式自体は型の指定のない式である．これはプログラミング言語にとって 2 つの重要な意味がある．第一点はユーザが繁雑な型の指定を一切書かなくてよいこと，またそのためには，システムが型を推論しなければならないことである．第二点は，型の指定がないため複数の型をもつ汎用性のあるプログラムが書けることである．本章ではこの 2 つの原理を学ぶ．

§15.1 型情報の推論

ラムダ計算の型システムは，プログラムが型をもつ条件を論理法則の形で定義したにすぎない．型システムをプログラミング言語で実際に利用するには，与えられたプログラムの型を決定するアルゴリズムが必要である．PASCAL や C などの従来の型付き言語では，この操作を簡単にするために，プログラムが使用する変数の型をあらかじめ宣言することを要求している．そのような言語では，例えばリストの長さを計算するプログラムは，以下のような型宣言付きプログラムとなる．

```
fun length (x:int list) =
    case x of Nil => 0
       | Cons(h:int,t:int list) => 1 + length t
    end ;
```

このような型宣言付きのプログラムの型の整合性チェックは容易である．

問 **15.1** ラムダ式の文法を以下のような型宣言を含むように変更する．

$$M ::= c \mid x \mid \lambda x:\tau.M \mid (M\ M) \mid (M,M) \mid M[1] \mid M[2]$$
$$\mid (1(M):\tau) \mid (2(M):\tau) \mid (\mathbf{case}\ M\ \mathbf{of}\ 1(x:\tau) \Rightarrow M,\ 2(x:\tau) \Rightarrow M)$$
$$\mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid \mathbf{fix}(M)$$

(i) この文法に対する型付け規則を定義せよ．

(ii) ラムダ式は与えられた型環境の下で高々 1 つの型しかもたないことを示せ．

(iii) ラムダ式 M と型環境 Γ を受け取って，もし M が Γ の下で型をもてばその型を返し，型をもたなければエラーを報告するアルゴリズムを定義せよ．

しかしながら，型宣言は多くの場合自明であり繁雑である．われわれがモデルとするプログラミング言語 ML では，型宣言を必要としない．例えば上記のプログラムは単に

```
fun length x = case x of Nil => 0
             | Cons(h,t) => 1 + length t
```

と書ける．型宣言を必要としない言語で型チェックを行なうためには，式の型を推論することが可能でなければならない．

型推論問題を理解するために，以下の簡単なラムダ式のもちうる型を考えてみよう．

$$\lambda f.\lambda x.\, f\,(f\ x)$$

この関数は，自由変数を含まず 2 つの引数をとる関数なので，もし型の制約を満たすならば，空の型環境のもとで $\tau_1 \to \tau_2 \to \tau_3$ の形をした型をもつはずである．さらに τ_1 は f の型のはずであり，τ_2 は x の型のはずである．すると $f\ x$ が型の制約を満たすために，f の型である τ_1 は x の型 τ_2 を引数の型とする関数型でなければならない．さらに $f\,(f\,x)$ の関数適用を考えると，f の引数の型は $f\ x$ の結果の型と一致しなければならない．以上の制約をすべて満たす型は $(\tau \to \tau) \to \tau \to \tau$ であり，逆にこの形の任意の型はこのラムダ式の型であることがわかる．

型推論問題を解くためには，このような推論を自動的に行なうアルゴリズムを構築する必要がある．さらに自由変数を含むラムダ式の型を推論するためには，

自由変数の型付けを行なう型環境も推論しなければならない．最も一般的な型推論問題は，以下のように表現できる．

与えられたラムダ式 M に対して，M のもちうる型判定の集合

$$\{\Gamma \rhd M : \tau \mid \Lambda \vdash \Gamma \rhd M : \tau\}$$

を決定せよ．

この問題が解ければ当然型チェック問題やラムダ式が与えられた型をもつか，といった問題にも容易に答えることが可能である．以下 Λ の型推論問題を考える．ただし以前同様，名前の束縛構文 $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ は $(\lambda x.\, M_2)\, M_1$ の略記法とみなす．

(a) 主要な型判定と型の単一化

上記の例からわかるように，ラムダ式は一般に無数に多くの型判定をもちうる．したがって，型推論問題を解決するためには，これらすべての型判定を推論する必要がある．この問題が解決可能であるかは，型システムの表現力の強さに依存する．幸い Λ の型システムに対しては，与えられたラムダ式のもち得るすべての型判定を代表する，主要な型判定を推論することによって解決可能である．上記の例の $\lambda f.\lambda x.\, f\,(f\ x)$ の場合，

$$\emptyset \rhd \lambda f.\lambda x.f\ (f\ x) : (\alpha \to \alpha) \to \alpha \to \alpha$$

が主要な型判定であり，その他の型判定は，この型判定の型変数 α に適当な型を代入し，型環境に適当な仮定を加えることによって得ることができる．

主要な型判定は，型判定に含まれる型変数への代入を通じて定義される．X が，型または型を含む構造のとき，X に含まれる型変数の集合を $FTV(X)$ と書く．型変数の代入 S を型変数の有限集合から型への関数とする．型変数 $\alpha_1, \ldots, \alpha_n$ をそれぞれ $\tau_1, \ldots, \tau_n$ にうつす代入を $[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ と書く．型の代入 S は，$\alpha \notin dom(S)$ なる α に対して $S(\alpha) = \alpha$ とみなすことにより，型変数全体への関数 S^+ に拡張される．さらにこのように拡張された代入は，型の構造に関して再

帰的に拡張することによって以下のように，型全体の関数 $\hat{S}$ に拡張される．

$$\begin{aligned}\hat{S}(\alpha) &= S^{+}(\alpha)\\ \hat{S}(b) &= b\\ \hat{S}(\tau_1 \to \tau_2) &= \hat{S}(\tau_1) \to \hat{S}(\tau_2)\\ \hat{S}(\tau_1 \times \tau_2) &= \hat{S}(\tau_1) \times \hat{S}(\tau_2)\\ \hat{S}(\tau_1 + \tau_2) &= \hat{S}(\tau_1) + \hat{S}(\tau_2)\end{aligned}$$

与えられた S に対して $\hat{S}$ は一意に定まる．S_1, S_2 を型の代入とする．S_1 と S_2 の合成 $S_1 \circ S_2$ を $S_1 \circ S_2(\alpha) = \hat{S}_1(S_2^{+}(\alpha))$ と定義する．さらに，この合成演算は右結合すると約束し，$S_1 \circ (S_2 \circ S_3)$ を括弧を省略して $S_1 \circ S_2 \circ S_3$ と書くことにする．以降 S とその型への拡張 $\hat{S}$ を同一視し，$\hat{S}$ を単に S と書くことにする．

ある型の代入 S があって，$\tau_2 = S(\tau_1)$ であるとき，τ_2 は τ_1 の**例**(instance)であるという．また型判定 $\Gamma \rhd M : \tau$ と $\Gamma' \rhd M : \tau'$ に対して，ある S があって $S(\Gamma) \subseteq \Gamma'$ かつ $S(\tau) = \tau'$ となるとき，$\Gamma' \rhd M : \tau'$ は $\Gamma \rhd M : \tau$ の例であるといい，また $\Gamma \rhd M : \tau$ は $\Gamma' \rhd M : \tau'$ **より一般的**であるという．S を任意の型の代入とすると，簡単な帰納法により，もし $\Lambda \vdash \Gamma \rhd M : \tau$ なら $\Lambda \vdash S(\Gamma) \rhd M : S(\tau)$ であることがわかる．この性質と補題 12.4 より，以下の性質が成立する．

補題 15.1 $\Lambda \vdash \Gamma \rhd M : \tau$ かつ $\Gamma' \rhd M : \tau'$ が $\Gamma \rhd M : \tau$ の例なら $\Lambda \vdash \Gamma' \rhd M : \tau'$ である． □

したがって，導出可能な型判定 $\Lambda \vdash \Gamma \rhd M : \tau$ が，M に対して導出可能なすべての型判定より一般的であるなら，$\Gamma \rhd M : \tau$ は，M のもちうる型判定の集合を代表することになる．このような型判定を**主要な型判定**(principal typing)という．

Λ に対しては，主要な型判定を計算するアルゴリズムが存在し，したがって，型推論問題が可解であることを証明することができる．その方針は，与えられた型なしラムダ式 M の部分式のもつ型判定の間に成立すべき条件を，型の間の等式として表現し，その等式を満たす最も一般的な解を求めることである．

例えば上記の考察より，$\lambda f.\lambda x.\, f(f\,x)$ は $\emptyset \rhd \lambda f.\lambda x.\, f(f\,x) : (\tau \to \tau) \to \tau \to \tau$ の形の型判定をもつはずである．この型判定に関する分析を系統的に行なうと，およそ以下のようになる．まず，求める型判定を，型変数を用いて $\emptyset \rhd \lambda f.\lambda x.\, f(f\,x) : \alpha$ と置く．この型判定の導出には，この式に含まれる部分式 $f\,(f\,x)$，$(f\,x)$，f，および x に関する導出が含まれているはずである．それらをさらに以下のように置く．

(15.1) $\{f : \alpha_f, x : \alpha_x\} \rhd f\,(f\,x) : \alpha_1$

(15.2) $\{f : \alpha_f, x : \alpha_x\} \rhd (f\,x) : \alpha_2$

(15.3) $\{f : \alpha_f, x : \alpha_x\} \rhd f : \alpha_f$

(15.4) $\{f : \alpha_f, x : \alpha_x\} \rhd x : \alpha_x$

与えられた式の型判定は，型判定(15.1)から型付け規則 (abs) によって得られたもののはずである．さらに，型判定(15.1),(15.2)はそれぞれ，型判定(15.3)と(15.2)，および型判定(15.3)と(15.4)から型付け規則 (app)によって得られたもののはずである．したがって，型付け規則 (abs) と (app) の形を考えると，型の間に以下の等式が成立するはずである．

$$\begin{aligned}\alpha &= \alpha_f \to \alpha_x \to \alpha_1\\ \alpha_f &= \alpha_2 \to \alpha_1\\ \alpha_f &= \alpha_x \to \alpha_2\end{aligned}$$

この等式集合が，注目している型判定 $\emptyset \rhd \lambda f.\lambda x.\, f(f\,x) : \alpha$ の満たすべき条件である．そこで，上記等式を型変数間の方程式と考え，その最も一般的な解を求めれば，それが $\lambda f.\lambda x.\, f(f\,x)$ のもつ型判定の集合を表現しているはずである．この例の場合，最も一般的な解は以下のような型判定に対応する．

$$\emptyset \rhd \lambda f.\lambda x.\, f(f\,x) : (\alpha_1 \to \alpha_1) \to \alpha_1 \to \alpha_1$$

以上の洞察から，型推論問題は，型の間の等式集合の最も一般的な解を求めるアルゴリズムがあれば，計算可能と期待できる．これが**単一化アルゴリズム**(unification algorithm)である．

E を型の組の集合とする．代入 S が，任意の $(\tau_1, \tau_2) \in E$ について $S(\tau_1) = S(\tau_2)$ を満たすとき，E の単一化(unifier)という．S_1 と S_2 を E の単一化とする．S_1 が S_2 より一般的な単一化であるのは，ある代入 S_3 が存在して $S_2 = S_3 \circ S_1$ となるときである．

定理 15.2（単一化アルゴリズム）与えられた任意の型等式の集合 E に対して，もし E が単一化をもてば E の最も一般的な単一化(most general unifier, mgu)を返し，もし E が単一化をもたなければエラーを報告するアルゴリズム $\mathcal{U}$ が存在する． □

この定理の証明が与える単一化アルゴリズム $\mathcal{U}$ は，型推論ばかりでなく，計算機科学の数多くの分野で使用されている重要なアルゴリズムである．この定理の証明には種々の方法があるが，その中で最も簡潔でエレガントと思われる，等式系の変形規則を用いた定義とその正しさの証明を紹介する．

型の代入 S は $\{(\alpha, \tau) \mid S(\alpha) = \tau\}$ なる集合と同型であるから，以下のアルゴリズムの定義ではこの集合表現により，型の代入を表現することにする．型の等式の集合 E と S の組の変形規則の集合を図 1 51 のように与える．ただし図の規則の定義において，型の組 (τ_1, τ_2) の順序は重要ではないものとする．

(u-i) $(E \cup \{(\tau, \tau)\}, S) \Longrightarrow (E, S)$

(u-ii) $(E \cup \{(\alpha, \tau)\}, S)$
$\Longrightarrow ([\tau/\alpha](E) \setminus \{(\alpha, \tau)\} \cup [\tau/\alpha](S))$ (ただし $\alpha \notin FTV(\tau)$ のとき)

(u-iii) $(E \cup \{(\tau_1^1 \to \tau_1^2, \tau_2^1 \to \tau_2^2)\}, S) \Longrightarrow (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$

(u-iv) $(E \cup \{(\tau_1^1 \times \tau_1^2, \tau_2^1 \times \tau_2^2)\}, S) \Longrightarrow (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$

(u-v) $(E \cup \{(\tau_1^1 + \tau_1^2, \tau_2^1 + \tau_2^2)\}, S) \Longrightarrow (E \cup \{(\tau_1^1, \tau_2^1), (\tau_1^2, \tau_2^2)\}, S)$

図 **15.1** 単一化アルゴリズムの変形規則

関係 $\overset{*}{\Longrightarrow}$ を関係 $\Longrightarrow$ の反射的推移的閉包とする．この関係を使い，アルゴリズム $\mathcal{U}$ を以下の関数として定義する．

$$\mathcal{U}(E) = \begin{cases} S & ((E, \emptyset) \overset{*}{\Longrightarrow} (\emptyset, S) \text{ のとき}) \\ \mathit{failure} & (\text{上記以外}) \end{cases}$$

このアルゴリズムが単一化アルゴリズムであることを示す．

まず，各変形規則が単一化の集合を保存すること，すなわち，5 つの変形規則がいずれも以下の性質を満たすことを示す．

> $(E_1, S_1) \Longrightarrow (E_2, S_2)$ なら，$(E_1 \cup S_1)$ の単一化の集合と $(E_2 \cup S_2)$ の単一化の集合は一致する．

任意の S は，同一の型の組 $\{(\tau, \tau)\}$ の単一化であるから，規則 (u-i) について上記の性質は成立する．任意の代入 S について，もし S が $S(\tau) = S(\alpha)$ を満たせば，任意の τ' について $S([\tau/\alpha]\tau') = S(\tau')$ であることを τ' に関する帰納法で容易に示すことができる．よって，S が $E \cup \{(\alpha, \tau)\}$ の単一化であることと S が $[\tau/\alpha](E) \cup \{(\alpha, \tau)\}$ の単一化であることは同値である．よって，規則 (u-ii) についても上記の性質は成立する．また，任意の S について，

$$S(\tau_1^1 \to \tau_1^2) = S(\tau_2^1 \to \tau_2^2) \iff S(\tau_1^1) = S(\tau_2^1) \text{ かつ } S(\tau_1^2) = S(\tau_2^2)$$

であるから，規則 (u-iii) についても上記の性質は成立する．規則 (u-iv) と規則 (u-v) とに関しても同様である．

この性質より，もし $\mathcal{U}(E) = S$ なら，$(E, \emptyset) \overset{*}{\Longrightarrow} (\emptyset, S)$ であるから，S の最も一般的な単一化は E の最も一般的な単一化である．一方，S は，常に，$\alpha_i \notin FTV(\tau_i)$，かつ $\alpha_i \neq \alpha_j (i \neq j)$ を満たす $\{(\alpha_1, \tau_1), \ldots, (\alpha_n, \tau_n)\}$ の形をした集合であることを容易に示すことができる．したがって，S の最も一般的な単一化は S 自身である．よって，$\mathcal{U}(E) = S$ なら S は E の最も一般的な単一化である．

つぎにアルゴリズムが失敗を報告する場合を考える．$\mathcal{U}(E) = \mathit{failure}$ と仮定する．アルゴリズムの定義より，$(E, \emptyset) \overset{*}{\Longrightarrow} (E', S)$，かつ $(E', S) \not\Longrightarrow (E'', S')$ である $E' \neq \emptyset$ があるはずである．一方，変形規則の定義から，明らかに，$(E', S) \not\Longrightarrow (E'', S')$ であれば，E' は単一化をもたない．したがって $E' \cup S$ も単一化をもたない．変形規則は単一化集合を保存するから，E も単一化をもち得ない．

以上より，$\mathcal{U}$ は，もし停止すれば，その結果は定理 1 52 の条件を満たすことが示された．上記の手続きが正しいアルゴリズムであることを示すためには，さらに，すべての入力に対して停止することを示さなければならない．この性質は，再帰的に定義されたアルゴリズムの場合は，多くの場合自明である．しかし，$\mathcal{U}$ の定義の場合は必ずしも明らかではない．一般にアルゴリズムの停止性を示すた

めには，アルゴリズムが操作するデータの複雑さを表す量を定義し，アルゴリズムが行なう操作が，常にこの量を減少させることを示せばよい．このような量をアルゴリズムの停止測度とよぶ．

半順序集合が $m > m_1 > m_2 > \cdots$ となる無限の減少系列をもたないとき**整礎**であるという．停止測度は，整礎な集合の要素であればよい．アルゴリズムの停止性の証明のポイントは，アルゴリズムの性質からそのような停止測度を見つけ出すことである．

型の等式集合 E の中の型変数の数と，E に含まれる型の大きさの総和の組を $Measure(E)$ と書く．このような組の集合に対して以下の辞書式順序を考える．

$$(a,b) < (a',b') \iff a < a' \text{ または } (a = a' \text{ かつ } b < b')$$

組 $(0,0)$ は，この順序関係に関して最小元となっているから，可能な組の集合は整礎な集合である．

各変形規則は，以上定義した $Measure(E)$ を減少させる．すなわち，各変形規則に対して，もし $(E_1,S_1) \Longrightarrow (E_2,S_2)$ なら $Measure(E_1) > Measure(E_2)$ が成立する．規則 (u-i)，(u-iii)，(u-iv) および (u-v) の場合は，$Measure(E_1) = (a,b)$，$Measure(E_2) = (a',b')$ のとき，$a \geq a'$ かつ $b > b'$ である．規則 (u-ii) の場合は，$Measure(E_1) = (a,b)$，$Measure(E_2) = (a',b')$ のとき，$a > a'$ である．いずれの場合も，$Measure(E_1) > Measure(E_2)$ が成立する．よって $Measure(E)$ は $\mathcal{U}$ の停止測度であり，$\mathcal{U}$ はすべての入力に対して必ず停止することが示された．以上で定理 15.2 が証明された．図 15.2 に単一化の計算例を示す．

(b) 型推論アルゴリズムとその性質

単一化アルゴリズムの存在は，等式集合で表現された型の制約を満たす解が存在すれば最も一般的な解が存在し，さらにその解は型変数への代入として計算できることを示している．以下に定義する型推論アルゴリズム PTS は，与えられたラムダ式の部分式の型が満たすべき条件を型の等式集合として表し，その制約を満たす最も一般的な解を，単一化アルゴリズムを用いて求めるアルゴリズムである．PTS は式 M を受け取り，もし M が型をもてば，その最も一般的な型判定 $\Gamma \rhd M : \tau$ を返し，もし M が型をもたなければエラーを報告する．$\{\Gamma_1, \ldots, \Gamma_n\}$ を型環境の

$$
\begin{aligned}
&(\{(\alpha, \alpha_f \to \alpha_x \to \alpha_1), (\alpha_f, \alpha_x \to \alpha_2), (\alpha_f, \alpha_2 \to \alpha_1)\}, \emptyset)\\
\Longrightarrow\ &(\{(\alpha_f, \alpha_x \to \alpha_2), (\alpha_f, \alpha_2 \to \alpha_1)\}, \{(\alpha, \alpha_f \to \alpha_x \to \alpha_1)\})\\
\Longrightarrow\ &(\{(\alpha_x \to \alpha_2, \alpha_2 \to \alpha_1)\},\\
&\quad \{(\alpha_f, \alpha_x \to \alpha_2), (\alpha, (\alpha_x \to \alpha_2) \to \alpha_x \to \alpha_1)\})\\
\Longrightarrow\ &(\{(\alpha_x, \alpha_2), (\alpha_2, \alpha_1)\}, \{(\alpha_f, \alpha_x \to \alpha_2), (\alpha, (\alpha_x \to \alpha_2) \to \alpha_x \to \alpha_1)\})\\
\Longrightarrow\ &(\{(\alpha_2, \alpha_1)\}, \{(\alpha_x, \alpha_2), (\alpha_f, \alpha_2 \to \alpha_2), (\alpha, (\alpha_2 \to \alpha_2) \to \alpha_2 \to \alpha_1)\})\\
\Longrightarrow\ &(\{\}, \{(\alpha_2, \alpha_1), (\alpha_x, \alpha_1), (\alpha_f, \alpha_1 \to \alpha_1), (\alpha, (\alpha_1 \to \alpha_1) \to \alpha_1 \to \alpha_1)\})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{U}(\{(\alpha, \alpha_f \to \alpha_x \to \alpha_1), (\alpha_f, \alpha_x \to \alpha_2), (\alpha_f, \alpha_2 \to \alpha_1)\})\\
&= [\alpha_1/\alpha_2, \alpha_1/\alpha_x, \alpha_1 \to \alpha_1/\alpha_f, (\alpha_1 \to \alpha_1) \to \alpha_1 \to \alpha_1)/\alpha]
\end{aligned}
$$

図 **15.2** 単一化計算例

集合とするとき，集合 $\{(\Gamma_i(x), \Gamma_j(x)) \mid x \in dom(\Gamma_i) \cap dom(\Gamma_j), 1 \leq i < j \leq n\}$ を $matches(\{\Gamma_1, \ldots, \Gamma_2\})$ と書く．型推論アルゴリズムを図 15.3 に与える．

型推論アルゴリズムの満たすべき性質に，**健全性**と**完全性**がある．型推論アルゴリズムが健全であるとは，アルゴリズムが推論した型判定は導出可能であるという性質であり，すべての型推論アルゴリズムが満たすべき基本的性質である．型推論アルゴリズムが完全であるとは，そのアルゴリズムが，式のもつすべての型判定を推論できるという性質であり，その十分条件は，アルゴリズムが推論する型判定が，式のもつすべての型判定より一般的であることである．すべての式に対して主要な型判定が存在するシステムでは，型推論アルゴリズムが，常に主要な型判定を推論すれば，そのアルゴリズムは健全かつ完全である．実際に PTS は健全かつ完全な型推論アルゴリズムである．まず健全性を示す．

定理 15.3（PTS アルゴリズムの健全性） もし $PTS(M) = (\Gamma, \tau)$ なら，$\Lambda \vdash \Gamma \rhd M : \tau$ である．

［証明］ M を任意の式とし，$PTS(M) = (\Gamma, \tau)$ と仮定する．証明は M の構造に関する帰納法による．以下，変数，ラムダ抽象，関数適用の場合を示し，その他の場合は練習問題とする．

x の場合． $PTS(x) = (\{x : \alpha\}, \alpha)$ であり，$\Lambda \vdash \{x : \alpha\} \rhd x : \alpha$ である．

$\lambda x.\, M_1$ の場合． アルゴリズムの定義より，$PTS(M_1) = (\Gamma_1, \tau_1)$ なる Γ_1, τ_1

$$
\begin{aligned}
&PTS(c) = (\emptyset, \tau)\ (c \text{ の型が } \tau \text{ の場合})\\
&PTS(x) = (\{x:\alpha\}, \alpha)\ (\alpha \text{ fresh})\\
&PTS(\lambda x.M_1) = \text{let } (\Gamma_1,\tau_1) = PTS(M_1)\\
&\qquad\qquad \text{in if } x \in dom(\Gamma_1) \text{ then } (\Gamma_1|_{\overline{x}}, \Gamma_1(x) \to \tau_1)\\
&\qquad\qquad\quad \text{else } (\Gamma_1, \alpha \to \tau_1)\ (\alpha \text{ fresh})\\
&PTS(M_1\ M_2) = \text{let } (\Gamma_1,\tau_1) = PTS(M_1)\\
&\qquad\qquad (\Gamma_2,\tau_2) = PTS(M_2)\\
&\qquad\qquad S = \mathcal{U}(matches(\{\Gamma_1,\Gamma_2\}) \cup \{(\tau_1, \tau_2 \to \alpha)\})\ (\alpha \text{ fresh})\\
&\qquad\quad \text{in } (S(\Gamma_1) \cup S(\Gamma_2), S(\alpha))\\
&PTS((M_1,M_2)) = \text{let } (\Gamma_i,\tau_i) = PTS(M_i)\ (i \in \{1,2\})\\
&\qquad\qquad S = \mathcal{U}(matches(\{\Gamma_1,\Gamma_2\}))\\
&\qquad\quad \text{in } (S(\Gamma_1 \cup \Gamma_2), S(\tau_1) \times S(\tau_2))\\
&PTS(M[i]) = \text{let } (\Gamma,\tau) = PTS(M)\\
&\qquad\qquad S = \mathcal{U}(\{(\alpha_1 \times \alpha_2, \tau)\})\ (\alpha_1\alpha_2 \text{ fresh})\\
&\qquad\quad \text{in } (S(\Gamma), S(\alpha_i))\\
&PTS(1(M)) = \text{let } (\Gamma,\tau) = PTS(M) \text{ in } (\Gamma, \tau + \alpha)\ (\alpha \text{ fresh})\\
&PTS(2(M)) = \text{let } (\Gamma,\tau) = PTS(M) \text{ in } (\Gamma, \alpha + \tau)\ (\alpha \text{ fresh})\\
&PTS((\textbf{case } M_1 \textbf{ of } 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3)) =\\
&\quad \text{let } (\Gamma_i,\tau_i) = PTS(M_i)\ (i \in \{1,2,3\})\\
&\qquad S = \mathcal{U}(matches(\{\Gamma_1, \Gamma_2|_{\overline{x}}, \Gamma_3|_{\overline{y}}\}) \cup matches(\{\Gamma_2, \{(x,\alpha_1)\}\})\\
&\qquad\qquad \cup matches(\{\Gamma_3, \{(y,\alpha_2)\}\}) \cup \{(\tau_1, \alpha_1+\alpha_2), (\alpha_3,\tau_2), (\alpha_3,\tau_3)\})\\
&\quad \text{in } (S(\Gamma_1 \cup \Gamma_2), S(\tau_1) \times S(\tau_2))\ (\alpha_1,\alpha_2,\alpha_3 \text{ fresh})\\
&PTS(\textbf{fix}(M)) = \text{let } (\Gamma,\tau) = PTS(M)\\
&\qquad\qquad S = \mathcal{U}(\{(\tau, \alpha \to \alpha)\})\\
&\qquad\quad \text{in } (S(\Gamma), S(\alpha))\ (\alpha \text{ fresh})
\end{aligned}
$$

図 **15.3**　型付きラムダ計算の型推論アルゴリズム

が存在する．$x \in dom(\Gamma_1)$ と仮定する．アルゴリズムの定義より，$x \notin dom(\Gamma)$，$\Gamma_1 = \Gamma \cup \{x:\tau_2\}$，かつ $\tau = \tau_2 \to \tau_1$ である．帰納法の仮定より $\Lambda \vdash \Gamma\{x:\tau_2\} \rhd M_1 : \tau_1$ である．規則 (abs) より，$\Lambda \vdash \Gamma \rhd \lambda x.M_1 : \tau_2 \to \tau_1$ である．

つぎに $x \notin dom(\Gamma')$ と仮定する．アルゴリズムの定義より，$PTS(M_1) = (\Gamma_1,\tau_1)$，$\Gamma = \Gamma_1$，かつ $\tau = \alpha \to \tau_1$ である．帰納法の仮定より $\Lambda \vdash \Gamma \rhd M_1 : \tau_1$ である．補題 12.4 および 12.5 より，$\Lambda \vdash \Gamma\{x:\alpha\} \rhd M_1 : \tau_1$ である．よって規則 (abs) より，$\Gamma \rhd \lambda x.M_1 : \alpha \to \tau_1$ である．

$M_1\ M_2$ の場合．アルゴリズムの定義より，$PTS(M_1) = (\Gamma_1,\tau_1)$，$PTS(M_2) = (\Gamma_2,\tau_2)$，$S = \mathcal{U}(matches(\{(\Gamma_1,\Gamma_2)\}) \cup \{(\tau_2 \to \alpha, \tau_1)\})$，$\Gamma = S(\Gamma_1 \cup \Gamma_2)$，かつ $\tau = S(\alpha)$ である．帰納法の仮定および定理 12.7 より，$\Gamma \rhd M_1 : S(\tau_2) \to S(\alpha)$ かつ $\Gamma \rhd M_2 : S(\tau_2)$ が成り立つ．規則 (app) より，$\Gamma \rhd M_1\ M_2 : \tau$ が成り立つ．∎

問 15.2　その他の場合を示し，定理 15.3 の証明を完成せよ．

定理 15.4（*PTS* アルゴリズムの完全性）　もし M が型判定 $\Gamma' \rhd M : \tau'$ をもてば，$PTS(M) = (\Gamma,\tau)$ かつ $\Gamma' \rhd M : \tau'$ は $\Gamma \rhd M : \tau$ の例である．

［証明］証明は M の構造に関する帰納法による．以下，変数，ラムダ抽象，関数適用の場合のみを示し，その他の場合は練習問題とする．

x の場合．$\Gamma \rhd x : \tau$ と仮定すると $\Gamma(x) = \tau$ である．したがって，$\Gamma \rhd x : \tau$ は $\{x:\alpha\} \rhd x : \alpha$ の例であり，成立する．

$\lambda x.M_1$ の場合．$\Gamma \rhd \lambda x.M_1 : \tau_1' \to \tau_2'$ と仮定する．型システムの定義により，$\Gamma\{x:\tau_1'\} \rhd M_1 : \tau_2'$ である．帰納法の仮定より，$PTS(M_1) = (\Gamma_1,\tau_1)$，ある S_0 が存在し，$S_0(\Gamma_1) \subseteq \Gamma\{x:\tau_1'\}$ かつ $S_0(\tau_1) = \tau_2'$ である．$x \in dom(\Gamma_1)$ の場合を考える．$S_0(\Gamma_1(x)) = \tau_1'$ である．アルゴリズムの定義により，$PTS(\lambda x.M_1) = (\Gamma_1|_{\overline{x}}, \Gamma_1(x) \to \tau_1)$．一方，$S_0(\Gamma_1|_{\overline{x}}) \subseteq \Gamma$ かつ $S_0(\Gamma_1(x) \to \tau_1) = \tau_1' \to \tau_2'$ である．したがって，$\Gamma \rhd \lambda x.M_1 : \tau_1' \to \tau_2'$ は $\Gamma_1|_{\overline{x}} \rhd \lambda x.M_1 : \Gamma_1(x) \to \tau_1$ の例である．つぎに，$x \notin dom(\Gamma_1)$ と仮定する．$S_0(\Gamma_1) \subseteq \Gamma$ である．アルゴリズムの定義により，$PTS(\lambda x.M_1) = (\Gamma_1, \alpha \to \tau_1)$．ここで α は新しい型変数であるから，$S_0(\Gamma_1) = [\tau_1'/\alpha] \circ S_0(\Gamma_1) \subseteq \Gamma$ かつ $[\tau_1'/\alpha] \circ S_0(\alpha \to \tau_1) = \tau_1' \to \tau_2'$ である．したがって，$\Gamma \rhd \lambda x.M_1 : \tau_1' \to \tau_2'$ は $\Gamma_1 \rhd \lambda x.M_1 : \alpha \to \tau_1$ の例である．

$(M_1\ M_2)$ の場合．$\Gamma \rhd M_1\ M_2 : \tau'$ と仮定する．型システムの定義により，ある τ_1' があって $\Gamma \rhd M_1 : \tau_1' \to \tau'$ かつ $\Gamma \rhd M_2 : \tau_1'$ である．M_1 に対する帰納法の仮定より，$PTS(M_1) = (\Gamma_1,\tau_1)$ かつ，ある S_1 が存在し，$S_1(\Gamma_1) \subseteq \Gamma$，$S_1(\tau_1) = \tau_1' \to \tau'$ である．M_2 に対する帰納法の仮定より，$PTS(M_2) = (\Gamma_2,\tau_2)$ かつ，ある S_2 が存在し，$S_2(\Gamma_2) \subseteq \Gamma$，$S_2(\tau_2) = \tau_1'$ である．一般性を失うことなく，$dom(S_1) = FTV(\Gamma_1) \cup FTV(\tau_1)$ および $dom(S_2) = FTV(\Gamma_2) \cup FTV(\tau_2)$ と仮定してよい．またアルゴリズムの性質より，$(FTV(\Gamma_1) \cup FTV(\tau_1)) \cap (FTV(\Gamma_2) \cup$

$FTV(\tau_2)) = \emptyset$ であることを容易に確かめることができる．よって $S_1 \cup S_2$ は型変数の代入である．α を新しい型変数とし，$S_3 = (S_1 \cup S_2 \cup [\tau'/\alpha])$ とすると，すべての $x \in dom(\Gamma_1) \cap dom(\Gamma_2)$ について $S_3(\Gamma_1(x)) = S_3(\Gamma_2(x)) = \Gamma(x)$ であり，また $S_3(\tau_1) = \tau_1' \to \tau'$，かつ $S_3(\tau_2 \to \alpha) = \tau_1' \to \tau'$ が成り立つ．以上より，S_3 は $matches(\{\Gamma_1, \Gamma_2\}) \cup \{(\tau_1, \tau_2 \to \alpha)\}$ の単一化である．よって，定理 15. 2より $\mathcal{U}(matches(\{\Gamma_1, \Gamma_2\}) \cup \{(\tau_1, \tau_2 \to \alpha)\})$ は成功し，S_3 より一般的な型の代入 S を返す．アルゴリズムの定義より $PTS(M_1\ M_2) = (S(\Gamma_1 \cup \Gamma_2), S(\alpha))$ である．S は S_3 より一般的であるから，ある S_4 が存在して $S_4(S(\Gamma_1 \cup \Gamma_2)) = S_3(\Gamma_1 \cup \Gamma_2) \subseteq \Gamma$ かつ $S_4(S(\alpha)) = S_3(\alpha) = \tau'$ である．よって $\Gamma \rhd M_1\ M_2 : \tau'$ は $S(\Gamma_1 \cup \Gamma_2) \rhd M_1\ M_2 : S(\alpha)$ の例である． ∎

問 **15.3** その他の場合を示し，定理 15. 4 の証明を完成せよ．

図 15. 4に $\lambda f.\lambda x.\ f(f\ x)$ に対するアルゴリズム PTS の動作を示す．字下げと記号 "|" は，再帰的呼び出しの範囲を表す．

$$
\begin{array}{l}
PTS(\lambda f.\lambda x.f\ (f\ x)) \\
\quad | PTS(\lambda x.f\ (f\ x)) \\
\quad |\quad | PTS(f\ (f\ x)) \\
\quad |\quad |\quad | PTS(f) = (\{f : \alpha_1\}, \alpha_1) \\
\quad |\quad |\quad | PTS(f\ x) \\
\quad |\quad |\quad |\quad | PTS(f) = (\{f : \alpha_2\}, \alpha_2) \\
\quad |\quad |\quad |\quad | PTS(x) = (\{x : \alpha_3\}, \alpha_3) \\
\quad |\quad |\quad |\quad | \mathcal{U}(\{(\alpha_2, \alpha_3 \to \alpha_4)\}) = [\alpha_3 \to \alpha_4/\alpha_2] \\
\quad |\quad |\quad |\quad = (\{f : \alpha_3 \to \alpha_4, x : \alpha_3\}, \alpha_4) \\
\quad |\quad |\quad | \mathcal{U}(\{(\alpha_1, \alpha_3 \to \alpha_4), (\alpha_1, \alpha_4 \to \alpha_5)\}) \\
\quad |\quad |\quad |\quad = [\alpha_3 \to \alpha_3/\alpha_1, \alpha_3/\alpha_4, \alpha_3/\alpha_5] \\
\quad |\quad = (\{f : \alpha_3 \to \alpha_3, x : \alpha_3\}, \alpha_3) \\
\quad | = (\{f : \alpha_3 \to \alpha_3\}, \alpha_3 \to \alpha_3) \\
= (\emptyset, (\alpha_3 \to \alpha_3) \to \alpha_3 \to \alpha_3)
\end{array}
$$

図 **15.4**　型推論の例

この型推論機構によって，型宣言のないプログラムの型チェックが可能となる．

問 **15.4** 以下の手順でラムダ式の型推論システムを実装せよ．

（i）型を表すデータ型とそのプリント関数を定義せよ．

（ii）型の単一化を行なう関数を定義せよ．ただしこの関数は型のリストのリストを引数として受け取り，各要素リストについて，要素リストの要素をすべて単一化する代入を返すものとする．

（iii）問 14.14 で定義したラムダ式の型を推論する関数を定義し，ソースプログラムのプリントの後，推論した型情報をプリントする処理を加えよ．

§15.2 多相関数の表現

プログラムは，一般に種々の型のデータに適応可能な，汎用性をもっている．例えば，恒等関数 $\lambda x.\ x$ は任意の型のデータに適用可能である．プログラムのもつこの汎用性をプログラムの**多相性**とよび，多相性をもつ関数を**多相関数**とよぶ．

型推論に基づく型の整合性のチェックのもう 1 つの利点は，この多相型が推論可能であることである．例えば $\lambda x.\ x$ に対しては，以下の型判定が推論される．

$$\emptyset \rhd \lambda x.\ x\colon\ \alpha \to \alpha$$

この型は，$\lambda x.\ x$ のもちうる最も一般的な型判定である．$\lambda x.\ x$ の多相性は，この最も一般的な型判定によって正確に表現されている．すなわち，$\lambda x.\ x$は，$\alpha \to \alpha$ の中の型変数 α を置き換えて得られる $\tau \to \tau$ の形の任意の型の関数として使用可能な多相関数である．例えば $((\lambda x.\ x)1, (\lambda x.\ x)true)$ において，α はそれぞれ int および $bool$ に自動的に置き換えられ，$int \to int$ および $bool \to bool$ の型として使用される．このように，プログラムは一般に多相性をもち，さらに，前節で定義した型推論アルゴリズムはその多相性を推論する能力をもつ．

(a) 多相関数の定義機構

しかしながら，これまでに定義してきたラムダ計算の型システムには，この多相性を利用する機構が存在しない．例えば $(\lambda id.(id\ 1, id\ true))\ \lambda x.\ x$ のようなプログラムを書いたとしても，関数 $\lambda id.(id\ 1, id\ true)$ の定義時点で，id は何らかの 1 つの型をもたねばならないから，型エラーとなってしまう．

Λ の型システムでは，$(\lambda id.(id\ 1, id\ true))(\lambda x.\ x)$のようなプログラムは型付けできないが，$\beta$ 簡約の結果得られる $((\lambda x.\ x)1, (\lambda x.\ x)true)$ は型付け可能である．

さらに，この式に 2 回現れる $\lambda x.x$ は同一であるから，この部分の型推論は $\lambda x.x$ の型推論の結果を，この式が使用される文脈に適合させたものであり，前節の結果から，それは $\lambda x.x$ の主要な型判定の例のはずである．そこで $(id\ 1, id\ true)$ に現れる id を，ラムダ抽象の変数としてではなく，$(\lambda x.x)$ に付けられた名前として取り扱えば，$((\lambda x.x)1, (\lambda x.x)true)$ と同様の型付けが可能と考えられる．この考えのもとに，多相性を利用する機構を言語の中に組み込む機構が多相的 let 束縛とよばれる以下の構文である．

$$\textbf{let}\ x = M_1\ \textbf{in}\ M_2$$

これまでは，この構文を $(\lambda x.M_2)\ M_1$ の別記法として取り扱ってきた．しかしながら，この構文で導入された変数 x は，一般のラムダ抽象の変数と違い，式 M_1 につけられた名前であるから，より詳しい型の情報を得ることができる．このことを利用して，この構文の型推論を，おおよそ以下のように行なうことによって，多相関数を実現できる．

（i）M_1 の最も一般的な型を計算し，それを x の型として記録する．
（ii）M_2 の中の x については，その型は上記ステップで記録された型が推論されたものとして，推論を実行する．

例えば，式 $\textbf{let}\ id = \lambda x.x\ \textbf{in}\ (id\ 1, id\ true)$ では，id が使われるたびに，使用される文脈に適した $\lambda x.x$ の型が推論され，$((\lambda x.x)1, (\lambda x.x)true)$ と同様の型付けが可能となる．

(b) let の型付け規則

Λ の型システムに上記の多相型 let 構文を導入するために，この構文の型付け規則を与えなければならない．もっとも簡単な方法は，以下の型付け規則を追加することである．

$$\text{(let1)}\ \frac{\Gamma \rhd M_1 : \tau_0 \quad \Gamma \rhd [M_1/x]M_2 : \tau}{\Gamma \rhd \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau}$$

ここで τ_0 は，結果の型と関係のない任意の型でよい．この規則により，M_2 の中の各々の x の出現に対して，M_1 と同一の型推論がなされる．M_1 は，一般に複数の型をもちうるから，上記の規則は，$(\lambda x.M_2)M_1$ では表現できない M_2 の多相性を表現している．Λ に，上記の規則を加えて得られるシステムを Λ^{let1} とし，$\Gamma \rhd M : \tau$ が Λ^{let1} で導出可能であることを $\Lambda^{let1} \vdash \Gamma \rhd M : \tau$ と書く．

M_1 の型の計算はすべて同一であるから，上記の規則をそのまま実行するのは明らかに無駄である．前節の結果から，M_1 のもちうる型判定は，その主要な型の例であるので，M_1 の主要な型判定を一度だけ計算し記録し，それを保存し，M_2 の中で x が使用されるたびに，その記録されている主要な型判定のコピーを作り使用すれば，この無駄を回避することができる．さらに，主要な型判定の中で Γ の部分は共通である．そこで，M_1 の型 τ と，その型に含まれかつ Γ に含まれない型変数を記録し，x が使用されるたびに，τ の中の Γ に含まれない型変数を新しい型変数に変えて使用すればよいことがわかる．例えば，型環境が $\Gamma\{y:\alpha_1\}$ で，M_1 が $\lambda x.(x,y)$ の場合，$\Gamma\{y:\alpha_1\} \rhd \lambda x.(x,y) : \alpha_2 \to \alpha_2 \times \alpha_1$ であり，α_2 は型環境に現れない変数であるから，x が使われるたびにその型を $\alpha' \to \alpha' \times \alpha_1$ として使用すれば，M_1 の主要な型を毎回計算し直すのと同一の効果を達成できる．この機構を実現するのが以下に定義する多相型システムである．

多相型を以下のように定義する．

$$\sigma ::= \forall(\alpha_1 \cdots \alpha_n).\tau$$

$\forall(\alpha_1 \cdots \alpha_n).\tau$ は，τ の中の型変数 $\alpha_1, \ldots, \alpha_n$ が型を導出された環境に依存せず，したがって任意の型で換えて使用してよいことを表す．$FTV(\tau) \setminus FTV(\Gamma) = \{\alpha_1 \cdots \alpha_n\}$ のとき $Cls(\Gamma, \tau) = \forall(\alpha_1 \cdots \alpha_n).\tau$ と定める．また $\sigma = \forall(\alpha_1 \cdots \alpha_n).\tau_0$ かつある $\tau_1, \ldots, \tau_n$ に対して $\tau = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]\tau_0$ となるとき τ を σ の**多相型の例**とよび，$\tau < \sigma$ と書く．すると，上で述べた多相型 let 構文の型チェックは，以下の規則で形式的に表現できる．

$$\text{(let2)}\ \frac{\Gamma \rhd M_1 : \tau_0 \quad \Gamma\{x : Cls(\Gamma, \tau_0)\} \rhd M_2 : \tau}{\Gamma \rhd \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau}$$

$$\text{(var2)}\ \ \Gamma\{x:\sigma\} \rhd x : \tau \quad \text{if } \tau < \sigma$$

この 2 つの規則を Λ の推論規則に加えて得られる型システムを Λ^{let2} とし，$\Gamma \rhd M : \tau$ が Λ^{let2} で導出可能であるとき，$\Lambda^{let2} \vdash \Gamma \rhd M : \tau$ と書く．

Λ^{let1} と Λ^{let2} は以下の意味で等価である．

定理 15.5 Γ が多相型を含まない型環境とする．もし $\Lambda^{let2} \vdash \Gamma \rhd M : \tau$ なら $\Lambda^{let1} \vdash \Gamma \rhd M : \tau$ であり，また逆も成立する． □

この証明は通常の式の構造に関する帰納法ではうまく証明できない．困難の原因は Λ^{let1} での $\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ に対する型の導出に，$[M_1/x]M_2$ の導出が使われていることにある．$[M_1/x]M_2$ の大きさを単純に計算すると，$\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ の大きさよりも一般的に大きく，したがってこの導出に対して，式の大きさに関する帰納法が適用できない．そこで，let を含む式の複雑さの尺度である，let-次数 $LD(M)$ を定義する．直観的には，$LD(M)$ は M に含まれる let 式の数であるが，let 式は入れ子になっているので，正しく数えあげるには意味論の定義と同様の手法が必要である．let-次数環境 δ を変数の部分集合から自然数への関数とし，自由変数の let-次数が δ で与えられた場合の式 M の let-次数 $ld(\delta, M)$ を，M に関して帰納的に以下のように定める．

$$
\begin{aligned}
ld(\delta, x) &= \delta(x) \\
ld(\delta, c) &= 0 \\
ld(\delta, \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) &= ld(\delta, M_1) + ld(\delta\{x : ld(\delta, M_1)\}, M_2) + 1 \\
ld(\delta, C(M_1, \ldots, M_n)) &= ld(\delta, M_1) + \cdots + ld(\delta, M_n)
\end{aligned}
$$

ここで，$C(M_1, \ldots, M_n)$ は上記で定義された以外の式構成子である．以下の代入補題が成り立つ．

補題 15.6 もし $ld(\delta, M) \leq m$ かつ $ld(\delta\{x : m\}, N) = n$ なら $ld(\delta, [M/x]\ N) \leq n$ である．

［証明］N の構造に関する帰納法による．以下，変数および let 文の場合をのみを示す．その他の場合はいずれも帰納法の仮定から直接帰結する．

x の場合．$[M/x]\ N = M$ かつ $n = m$ であり成立する．

$y \neq x$ の場合．$[M/x]\ N = y$ であり，$ld(\delta\{x : m\}, y) = ld(\delta, y) = n$ であり成立する．

$\mathbf{let}\ y = N_1\ \mathbf{in}\ N_2$ の場合．$ld(\delta\{x : m\}, N_1) = n_1$，$ld(\delta, [M/x]N_1) = n_1'$，とする．またもし $n \leq n'$ なら $ld(\delta\{x : n\},\ M) \leq ld(\delta\{x : n'\}, M)$ であることを簡単な帰納法で確認できる．よって以下が成立する．

$$
\begin{aligned}
& ld(\delta\{x : m\}, \mathbf{let}\ y = N_1\ \mathbf{in}\ N_2) \\
&= n_1 + ld(\delta\{x : m\}\{y : n_1\}, N_2) + 1 && (ld\ \text{の定義}) \\
&\geq n_1' + ld(\delta\{x : m\}\{y : n_1\}, N_2) + 1 && (\text{帰納法の仮定}) \\
&\geq n_1' + ld(\delta\{y : n_1'\}, [M/x]N_2) + 1 && (\text{帰納法の仮定，}ld\ \text{の性質}) \\
&= ld(\delta, \mathbf{let}\ y = [M/x]\ N_1\ \mathbf{in}\ [M/x]\ N_2) && (ld\ \text{の定義}) \\
&= ld(\delta, [M/x](\mathbf{let}\ y = N_1\ \mathbf{in}\ N_2)) && \blacksquare
\end{aligned}
$$

A と B を整礎な半順序集合とする．M の let-次数を $LD(M) = ld(\emptyset, M)$ と定義し，M の複雑さを，辞書式順序で順序付けられた $LD(M)$ と M の大きさの組とする．この複雑さに関する帰納法によって定理 15.5 を証明する．let 構文以外の両者の型システムは同一であるから，let 構文以外は，帰納法の仮定から直接導かれる．そこで $M \equiv \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2$ に関して証明する．

$\Lambda^{let1} \vdash \Gamma \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau$ と仮定する．型付け規則より，ある τ_0 があって，$\Lambda^{let1} \vdash \Gamma \rhd M_1 : \tau_0$ であり，$\Lambda^{let1} \vdash \Gamma \rhd [M_1/x]M_2 : \tau$ である．

Λ^{let1} の型システムでは，Λ と同様，すべての型付け可能な式は主要な型判定が存在することを M の複雑さに関する簡単な帰納法で示すことができる．さらに，簡単な議論により，任意の M と Γ に対して，もし M が Γ のもとで型判定をもてば，そのどれより一般的な型判定が存在することを示せる．このような型判定を，M の Γ の下での主要な型判定とよぶことにする．

そこで，M_1 の Γ の下での主要な型判定を $\Lambda^{let1} \vdash \Gamma \rhd M_1 : \tau_1$ とする．帰納法の仮定より $\Lambda^{let2} \vdash \Gamma \rhd M_1 : \tau_1$ かつ $\Lambda^{let2} \vdash \Gamma \rhd [M_1/x]M_2 : \tau$ である．後者の型判定の導出 D を考える．型システムの定義よりこの中で出てくる M_1 の導出は $\Gamma \subseteq \Gamma'$ なる Γ' に対する $\Lambda^{let2} \vdash \Gamma' \rhd M_1 : \tau_1'$ の形の型判定の導出である．定義より，$\Lambda^{let2} \vdash \Gamma' \rhd M_1 : \tau_1$ は M_1 の Γ' に関する主要な型判定であるから，$\tau_1' \leq Cls(\Gamma', \tau_1)$ である．よって導出 D の中に現れる M_1 の導出を公理 $\Gamma'\{x : Cls(\Gamma', \tau)\} \rhd x : \tau_1'$ で置き換え，さらに導出の中の各々の型環境に仮定 $x : Cls(\Gamma', \tau_1)$ を加えて得られる導出は Λ^{let2} の導出である．よって $\Lambda^{let2} \vdash \Gamma'\{x : Cls(\Gamma', \tau_1)\} \rhd M_2 : \tau$ である．よって規則 (let2) より $\Lambda^{let2} \vdash \Gamma' \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau$ である．

逆に $\Lambda^{let2} \vdash \Gamma \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau$ と仮定する．型付け規則よりある τ_1 があって，$\Lambda^{let2} \vdash \Gamma \rhd M_1 : \tau_1$ かつ $\Lambda^{let2} \vdash \Gamma\{x : Cls(\Gamma, \tau_1)\} \rhd M_2 : \tau$ である．後者の導出に現れる任意の x の型を τ_1' とする．型システムの定義より $\tau_1' \leq Cls(\Gamma, \tau_1)$ である．よって M_2 の導出の中の各公理 $\Gamma'\{x : Cls(\Gamma, \tau_1)\} \rhd x : \tau_1'$ を $\Gamma' \rhd M_1 : \tau_1'$ で置き換え，x の出現を M_1 で置き換え，さらに各型環境から $x : Cls(\Gamma, \tau_1)$ を取り除くと Λ^{let2} における $\Gamma \rhd [M_1/x]M_2 : \tau$ の導出が得られる．この導出に対して帰納法の仮定を適用すると $\Lambda^{let1} \vdash \Gamma \rhd [M_1/x]M_2 : \tau$ である．また帰納法の仮定より $\Lambda^{let1} \vdash \Gamma \rhd M_1 : \tau_1$ である．よって $\Lambda^{let1} \vdash \Gamma \rhd \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : \tau$ である．これで定理 15.5 は証明された．

(c) 多相型言語の型推論

ML を始めとする近代的なプログラミング言語で採用されている型推論アルゴリズムは，Λ の型推論アルゴリズムに多相型 let 式の推論機構を加えて拡張したものである．そのもっとも簡単な定義は，Λ^{let1} の型システムを直接実現するものである．これは Λ の型推論アルゴリズムに以下の処理を付け加えるだけで実現可能である．

$$PTS(\mathbf{let}\ x = M_1\ \mathbf{in}\ M_2) = \mathrm{let}\ (\Gamma_1, \tau_1) = PTS(M_1)\ \mathrm{in}\ PTS([M_1/x]M_2)$$

このアルゴリズムは明らかに Λ^{let1} に関して健全かつ完全である．

以上のアルゴリズムは規則 (let1) に基づく型推論アルゴリズムであるが，より効率的で実際に広く使用されている型推論アルゴリズムは，型環境を毎回推論する冗長さを取り除き，多相関数の推論には (let1) 規則と等価な規則 (let2) および (var2) を基礎とした推論を行なうものである．型環境 Γ に多相型を含んでよいものとし，型推論アルゴリズムを，式 M に加えて M の中の変数の型付けをすべて含む型環境 Γ を受け取り型変数の代入 S と型 τ を返すアルゴリズムとして定義する．アルゴリズムの一部を図 15.5 に示す．

このアルゴリズムは $PTS(M)$ の拡張である．すなわち，もし $\Gamma = \{x : \alpha \mid x \in FV(M), \text{各 } \alpha \text{ は相異なる}\}$ なる Γ に対して

$$\mathcal{W}(\Gamma, M) = (S, \tau)$$

$$\mathcal{W}(\Gamma, x) = \text{ if } \Gamma(x) = \tau \text{ then } (\emptyset, \tau) \text{ else let } \forall(\alpha_1 \ldots \alpha_n).\tau = \Gamma(x) \text{ in } (\emptyset, [\alpha_1'/\alpha_1, \ldots, \alpha_n'/\alpha_n](\tau))$$

$$\mathcal{W}(\Gamma, \lambda x.M_1) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma\{x : \alpha\}, M_1)\ \mathrm{in}\ (S_1, S_1(\alpha) \to \tau_1)$$

$$\mathcal{W}(\Gamma, M_1\ M_2) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1);\ (S_2, \tau_2) = \mathcal{W}(S_1(\Gamma), M_2);\ S_3 = \mathcal{U}(\{(S_2(\tau_1), \tau_2 \to \alpha)\})\ \mathrm{in}\ (S_3 \circ S_2 \circ S_1, S_3(\alpha))$$

$$\mathcal{W}(\Gamma, let\ x = M_1\ in\ M_2) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1);\ \sigma = Cls(S_1(\Gamma), \tau_1);\ (S_2, \tau_2) = \mathcal{W}(S_1(\Gamma)\{x : \sigma\}, M_2)\ \mathrm{in}\ (S_2 \circ S_1, \tau_2)$$

$$\mathcal{W}(\Gamma, (M_1, M_2)) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1);\ (S_2, \tau_2) = \mathcal{W}(S_1(\Gamma), M_2)\ \mathrm{in}\ (S_2 \circ S_1, S_2(\tau_1) \times \tau_2)$$

$$\mathcal{W}(\Gamma, M_1[i]) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1);\ S_2 = \mathcal{U}(\{(\tau_1, \alpha_1 \times \alpha_2)\})\ \mathrm{in}\ (S_2 \circ S_1, S_2(\alpha_i))\ (i = 1 \text{ or } i = 2)$$

$$\mathcal{W}(\Gamma, 1(M_1)) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1)\ \mathrm{in}\ (S_1, \tau_1 + \alpha))$$

$$\mathcal{W}(\Gamma, 2(M_1)) = \mathrm{let}\ (S_1, \tau_1) = \mathcal{W}(\Gamma, M_1)\ \mathrm{in}\ (S_1, \alpha + \tau_1))$$

$$\mathcal{W}(\Gamma, (\mathbf{case}\ M_1\ \mathbf{of}\ 1(x) \Rightarrow M_2,\ 2(y) \Rightarrow M_3)) = \mathrm{let}\ (S_1, \tau_0) = \mathcal{W}(\Gamma, M_1);\ S_2 = \mathcal{U}(\{(\tau_0, \alpha_1 + \alpha_2)\});\ (S_3, \tau_1) = \mathcal{W}(S_2 \circ S_1(\Gamma\{x : \alpha_1\}), M_2);\ (S_4, \tau_2) = \mathcal{W}(S_3 \circ S_2 \circ S_1(\Gamma\{x : \alpha_2\}), M_3);\ S_5 = \mathcal{U}(\{(S_4(\tau_1), \tau_2)\})\ \mathrm{in}\ (S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1, S_5(\tau_1))$$

図 15.5 多相型型推論アルゴリズム

なら $S(\Gamma) \rhd M : \tau$ は $PTS(M)$ の結果と一致する．

(d) ML の型推論システム

ML をはじめとする近代的な高水準プログラミング言語で採用されている型推論アルゴリズムは，以上定義した $\mathcal{W}$ アルゴリズムに，再帰的データ型に関する処理を加えたものである．

ML 系言語では再帰的データ型は，ユーザ定義の型宣言を通じて定義される．前に説明したとおり，型パラメータがない以下の形の型宣言は，

```
type T = l1 of τ1 | ··· ln of τn
```

直和型 $\tau_1 + \cdots + \tau_n$ に，型 T という名前を付けたものと理解できる．この型宣言の各要素型に型 T が現れる場合は，以下の等式を満たす型が定義され，それに T という名前を付けたものと理解できる．

$$T = \tau_1 + \cdots + \tau_n$$

$\mathbf{fix}(M)$ の公理的意味論の場合同様，この等式を満たす型は，型 T を引数とする関数 $\lambda T.\tau_1 + \cdots + \tau_n$ の不動点と定義できる．このような型を再帰的データ型とよび，

$$\mu T.\tau_1 + \cdots + \tau_n$$

と表記する．

一般に，再帰的データ型 $\mu T.\tau$ は，等式 $\mu T.\tau = [\mu T.\tau/T]\tau$ を満たす型である．型の集合を，無限の大きさをもつ型を許すように拡張すれば，型宣言で定義される関数の不動点が存在し，したがって，再帰的データ型をも含む型体系が可能である．例えば，宣言

```
type ilist = Nil of unit | Cons of int * ilist
```

で定義される整数のリスト型は，関数 $\lambda t.unit + (int \times t)$ の不動点として与えられる．実際，型の集合が無限の大きさの型を含めば，以下のような無限の型は確かに，この関数の不動点である．

$$unit + (int \times (unit + int \times (unit + \cdot\ \cdot\cdot$$

ML の型宣言はさらに，上記の再帰的型宣言に加えて，以下のような型パラメータを許す．

```
type (α1,...,αn)T = l1 of τ1 | ··· ln of τn
```

この型宣言によって定義されるものは，それら型パラメータ $\alpha_1,\ldots,\alpha_n$ を種々の型に置きかえることによって，種々の型を生成する型関数である．ML では，このような型関数に，型 $(\tau_1,\ldots,\tau_n)$ を適用して得られる型を $(\tau_1,\ldots,\tau_n)T$ と書く．この宣言も，一般的には要素の型 τ_i に T を含むことを許す．したがって，再帰的

関数定義の意味論の議論と同様の議論により，この宣言によって定義される型関数は，以下の関数の不動点と理解できる．

$$\lambda T.\lambda(\alpha_1\ ,\ldots,\alpha_n)\,\tau_1 + \cdots + \tau_n$$

本節で解説した型推論アルゴリズムによって，以上のような再帰的型を直接推論することは不可能である．この困難を解決するために，ML では，$(\tau_1\ ,\ldots,\tau_n)T$ で表示される型を，式 $(\tau_1\ ,\ldots,\tau_n)T$ そのものとして扱い，さらに型宣言において，以下の型の決まった関数も同時に宣言されたとみなす．

$$l_1 = \lambda x.1\,(x)\colon \forall(\alpha_1\ ,\ldots,\alpha_n).(\tau_1 \to(\alpha_1,\ldots,\alpha_n)T)$$
$$\vdots$$
$$l_n = \lambda x.n(x) : \forall(\alpha_1\ ,\ldots,\alpha_n).(\tau_n \to (\alpha_1\ ,\ldots,\alpha_n)T)$$

さらに，場合分け構文を型チェックするために，$(\tau_1,\ldots,\tau_n)T$ の型をもつ式 M が，場合分け構文の第一引数に現れた場合，その型は，対応する型宣言の左辺に現れる型変数 $\alpha_1\ ,\ldots,\alpha_n$ を，それぞれ $\tau_1\ ,\ldots,\tau_n$ で置き換えて得られる型として扱う．

例えば，

```
type 'a list = Nil of unit | Cons of 'a * 'a list
```

の宣言の後，`Nil` と `Cons` はそれぞれ以下の型の関数として扱われる．

```
Nil = unit -> 'a list
Cons = 'a * 'a list -> 'a list
```

したがって，例えば `Cons(2,(Cons(1,Nil())))` は *int list* 型として型チェックされ，さらに，$unit + int \times intlist$ をもつ式として場合分け構文で使用される．

問 15.5 問 15.4 で作成した型推論システムを以下の手順で改良することによって多相型関数の型推論を加えよ．

（i）型を表すデータ型とそのプリント関数に多相型の表現を加えよ．

（ii）式の定義に **let** $x = M_1$ **in** M_2 構文を追加し，問 15.4 で作成した型推論関数を $\mathcal{W}$ アルゴリズムを実現する関数に入れ換えよ．

(iii) 以下の手順で，型宣言処理を加えよ．

(a) 型を表すデータ構造に，$(\alpha_1,\ldots,\alpha_n)\ T$ の形のユーザ定義データ型を追加せよ．

(b) 型宣言の左辺を表現するデータ構造を定義せよ．

(c) 型宣言文を読み込んで，それをユーザ定義のデータ型と，型宣言の右辺を表現するデータ型の組に変換する処理を定義せよ．

(d) 型推論関数を，型宣言で定義された名前とその定義の対応を維持する環境を引数とする関数に変更し，この環境を参照することによって，データ型の生成および場合分け構文の翻訳と型チェックを行なう処理を追加せよ．

このようにして出来上がったラムダ式のコンパイラは ML などが基礎とする原理に基づき，以下のような動作をするはずである．

```
-> type 'a list = Nil | Cons of 'a * 'a list;
type 'a list = Cons of 'a * 'a list | Nil
Nil :  'a list
Cons :  'a * 'a list -> 'a list
-> fun length Nil = 0
     | length (Cons(x,y)) = 1 + (length y);
fun length = fn :  'a list -> int
-> val L = Cons(1,Cons(2,Cons(3,Nil)));
val L = Cons (1,Cons (2,Cons (3,Nil))) :  int list
-> length L;
3 :  int
```


A Minimal 利用者マニュアル

本書では Minimal をプログラミング言語の例として使っている．読者が本書を読みながら，プログラムを実際に実行して内容を確認したり，自分でプログラムを書いてみたりできるように，Minimal の処理系をインターネット上の以下の場所で公開している．

`http://www.kurims.kyoto-u.ac.jp/~cs/csnyumon/`

本付録では，Minimal を利用するための基礎知識を収録した．内容は以下の通りである．

(i) 入力ループの使い方	Minimal の基本的な使用法
(ii) 値，実行と制御構造	言語で扱う概念の細かい定義
(iii) 構文	Minimal が使う形式文法
(iv) 組込関数	提供されている関数のリスト

§A.1 入力ループの使い方

Minimal は他の多くのプログラミング言語と違い，プログラムをコンパイルして，実行するのではない．Minimal では，入力ループを使い，処理系と対話しながら，値や関数を定義したり，式を計算・実行したりしながらプログラムを作っていく．プログラムを単位とするよりも，この方が容易に関数やアルゴリズムの