

# 第1章 プログラミング言語の種類と概要

このテキストは、佐賀大学理工学部知能情報システム学科の選択科目「プログラミング言語論」のための講義資料です。本学科のカリキュラムにマッチするように書かれています。

## 1.1 プログラミング言語の役割

プログラミング言語の必要性は明らかと思いますが、確認しておきます。Cプログラムの代入文:

```
x = a*b+c;
```

をアセンブリ・プログラム風に記述すると、以下のようになります。

```
load r1 = [&a]      // a のアドレスからレジスタ r1 へ数値をロード
load r2 = [&b]      // b のアドレスからレジスタ r2 へ数値をロード
mult r3 = r1,r2     // r1 と r2 を乗算し、レジスタ r3 へ格納
load r4 = [&c]      // c のアドレスからレジスタ r4 へ数値をロード
add r5 = r3,r4      // r3 と r4 を加算し、レジスタ r5 へ格納
store [&x] = r5     // r5 の値を x のアドレスへストア
```

この二つを比べれば、Cプログラムを読む/書く労力が上のアセンブリ・プログラム片を読む/書く労力よりも小さいことは明らかです。Cプログラムの改造がアセンブリ・プログラムの改造よりも容易であることも明らかです。

プログラミング言語の重要性はアセンブリ・プログラミング（機械語プログラミング）を経験した者ならば誰でも身に染みて分かります。では、どのようなプログラミング言語を用いれば、より効率的なプログラム開発が可能になるのでしょうか。その答えは一概ではありません。どのような処理を行うプログラムを作るか、プログラムの実行速度を重視するのか読み易さを重視するのか、どのようなコンピュータの上で実行するのか、ユーザーインターフェースはどのようにするのか、等々に依存しています。実際、用途に応じて様々なプログラミング言語がこれまでに設計・開発されてきました。

この講義では、そのような様々なプログラミング言語の特徴を学びます。本学科で最初に学んだプログラミング言語 C++はそのひとつです。まず始めに C++を含む様々なプログラミング言語を機能・言語構造によって分類し、概観します。そして、C++とは異なる特徴を持つプログラミング言語 Lisp と Prolog を特に深く学び、プログラムやプログラミングに関する理解の幅を広げます。

表 1.1: 低水準プログラミング言語の分類

	分類	言語名	備考
低水準	機械語 (ネイティブ・コード)	—	プロセッサ毎に異なる
	アセンブリ言語	—	プロセッサ毎に異なる
	中間言語	P コード	Pascal の中間言語
		バイトコード	狭義には Java の中間言語

## 1.2 プログラミング言語の種類と歴史

世の中に存在する（存在した）プログラミング言語を鳥瞰しておくことは重要です。この節では代表的なプログラミング言語について簡単に紹介していきます。

プログラミング言語は高水準プログラミング言語（high-level programming language<sup>1</sup>）と低水準プログラミング言語（low-level programming language）に大別されます。高水準言語はハードウェア・アーキテクチャを意識させない言語構造を持ち、人手によるプログラミングのしやすさ、プログラムの読みやすさを指向したプログラミング言語です。これに対して低水準言語はハードウェア・アーキテクチャを反映した言語構造を持ち、特定のハードウェアで高速に動作する利点を持ちます。

以下では、まず低水準言語について簡単に述べ、その後に高水準言語の代表的なものをいくつか紹介します。

### 1.2.1 低水準プログラミング言語

#### 機械語

機械語（machine language）は、プロセッサが直接解読可能な2進数データ列（ビット列）です。現在、機械語を読むことも機械語でプログラムを作るとはほとんどあり得ません。機械語プログラムのことをネイティブ・コード（native code）と呼ぶことがありますが、その場合には、議論の文脈の中で、ある特定のプロセッサが暗黙に仮定していることが一般的です。

#### アセンブリ言語

アセンブリ言語（assembly language）は、機械語に1対1対応するプログラミング言語です。2進数データ列である機械語はそのままでは人間にとって非常に読みづらいため、キーワードなどを用

<sup>1</sup>しばしば「高級プログラミング言語」と訳されます。高級/低級という呼び方はあまり適切とは思えませんが、google 検索では「高級...」という訳語を用いているサイト数が「高水準...」の2倍近くあります。

1 いて少し読み易く表したものです。アセンブリ言語を機械語に変換する事をアSEMBル (assemble)  
2 するとも言い、その変換を行うプログラムをアSEMBラ (assembler) と呼びます<sup>2</sup>。

3 1952年にはMark I コンピュータ用アSEMBラ<sup>3</sup>が開発されており、これが世界初のアSEMBラ  
4 とも言われています。アSEMBラで得られた知見は、後の高水準言語コンパイラ<sup>4</sup>の開発につな  
5 がっていきます。

## 6 中間言語

7 中間言語 (intermediate language) は、低水準言語と高水準言語の中間の性質を持つような言語  
8 の総称です。低水準言語では機械依存が強すぎて可搬性<sup>5</sup>に乏しく、高水準言語では機械から離れ  
9 すぎて実行効率が悪いという二つの性質の中間的な性質を持ちます。Pascal のPコード (P-code)  
10 やJavaのバイトコード (Bytecode) が有名です。中間とは言うものの、どちらかと言えば低水準  
11 寄りのものが多いため、このテキストでは中間言語を低水準言語に含めました。

### 12 1.2.2 高水準プログラミング言語

13 高水準言語は手続き型プログラミング言語 (procedural programming language、以下、手続き  
14 型言語) と宣言型プログラミング言語 (declarative programming language、以下、宣言型言語)  
15 に大別されます (表 1.2 参照)。

16 手続き型言語とは手順の時系列によって処理内容を記述する種類の言語であって、本学科で学ん  
17 だC++は手続き型言語です。処理内容を順番に記述していくため、考え方が単純で平易なプログ  
18 ラミングが可能です。しかし、処理の順序を少し間違えただけで予想もしない結果になることが多  
19 く、複雑なプログラムの開発では注意力が要求されます。プログラムのデバッグでも手順の実行順  
20 序を丹念に追わねばならず、容易ではありません。

21 宣言型言語とは、処理内容をその数学的性質に着目して、あたかも数式を書くように記述する種  
22 類の言語です。処理内容を数学の公理のように表現可能であるときに、コンパクトで美しいプログ  
23 ラムを書くことができます。反面、明確な時間順序を含むような処理内容の記述には不向きです。  
24 処理の順序を考慮する必要がない<sup>6</sup>ため、プログラムのデバッグは比較的容易とされています。

25 宣言型言語はさらに関数型プログラミング言語 (functional programming language、以下、関  
26 数型言語) と論理型プログラミング言語 (logic programming language、以下、論理型言語) に分  
27 けることができます。関数型言語とは、関数を定義することをプログラミングと見なす言語です。

<sup>2</sup>三つの術語「アセンブリ (assembly)」、「アSEMBル (assemble)」、「アSEMBラ (assembler)」は似ています。品詞に注意して誤用のないように注意しましょう。

<sup>3</sup>当時、AUTOCODER と命名されました。“AUTOCODER” (自動コード化器) はその後、アSEMBラを指す一般名詞として使用されるようになっていきます。

<sup>4</sup>コンパイラを含む言語処理系については次章で解説します。

<sup>5</sup>言語処理系が様々なコンピュータへ容易に移植できること、プログラムが様々な種類のコンピュータで実行できることを可搬性が良いと言います。

<sup>6</sup>このような性質を一般に参照透明性 (Referential transparency) と呼びます。

表 1.2: 様々な高水準プログラミング言語の分類

	分類 1	分類 2	言語名	誕生 時期	利用状況 (私見による)
高水準	手続き型 (命令型)	手続き型 (命令型)	FORTTRAN FORTRAN77 Fortran90 ...	1950 年代	科学計算分野に利用されている
			COBOL	1950 年代	商業分野に利用されている
			ALGOL	1960 年代	歴史的使命を終えた
			Pascal	1970	ほとんど利用されていない
			C	1972	利用されている
			C++	1980 年代	使用されている
			Basic	1964 から	使用されている
			Ada	1980 年代	ほとんど利用されていない
			Java	1990 年代	最も利用されている
			Python	1990 年代	利用が拡大している
			Swift	2014	使用されている
	宣言型	関数型	LISP	1958	特定分野に使用されている
			ML	1974	特定分野に使用されている
		論理型	Prolog	1972	特定分野に使用されている

1 関数とは、入力パラメータ値が与えられて関数値が返るような、C++の関数と同等のものと考え  
2 て構いませんが、関数型言語では関数本体を手続き的に記述することがせず、関数本体は関数呼び  
3 出しだけから構成されています。関数値を求めることが関数型言語におけるプログラムの実行で  
4 す。論理型言語とは、論理式を定義することをプログラミングと見なす言語です。論理式を公理と  
5 して与え、その公理の下である定理の証明を行うことをプログラムの実行とみなします。なお、こ  
6 れだけの説明で関数型、論理型言語が理解できるはずありません。後ほど本編にて詳細に説明し  
7 ていきます。

8 代表的なプログラミング言語を表 1.2 にまとめました。これらについて以下にプログラム例を付  
9 けて簡単に解説します。

## 10 **FORTTRAN**

11 **FORTTRAN**（フォートランと読む）は、歴史上最初にコンパイラが作られた高水準プログラミ  
12 ング言語です<sup>7</sup>。**FORTTRAN**の名称は formula translator（数式変換器）に由来します。1954 年に  
13 考案され、1957 年に IBM704 上で実際に稼働する最初のコンパイラがリリースされました。誕生

<sup>7</sup>FORTTRAN 開発の中心メンバーであった John Backus は、FORTTRAN の開発その他の業績によって 1977 年に  
チューリング賞を受賞しました。

SUM = 0	SUM = 0
DO 1000 I = 1, 10	DO I = 1, 10
SUM = SUM+I	SUM = SUM+I
1000 CONTINUE	ENDDO
WRITE(6,*) SUM	WRITE(6,*) SUM
END	END
(a) FORTRAN 66	(b) Fortran 90

図 1.1: FORTRAN のプログラム例

1 から半世紀経った今もなお使用され続けています。寿命の長い理由のひとつは、当初から実行  
2 性能を重視する設計思想を貫き、科学技術計算に広く使用された点にあります。もうひとつの理  
3 由は、言語仕様が時代に合わせて少しずつ変更/拡張していき、それがユーザーに受け入れられた  
4 点にあります。代表的な仕様を列挙すると、FORTRAN I/II/III/IV, FORTRAN 66, FORTRAN  
5 77, Fortran 90, Fortran 2003 等となります<sup>8</sup>。最近では並列コンピュータでの利用を目的に HPF  
6 (High Performance Fortran) なども提案/開発されています。

7 図 1.1 に 1 から 10 までの数の和を求めるプログラムを FORTRAN 66 と Fortran 90 で記述し  
8 た例を示します。両者の違いに言語仕様の変遷を見ることができます。

9 FORTRAN に初めて接する人はプログラムに変数宣言のないことに驚くかもしれません。FOR-  
10 TRAN では I, J, K, L, M, N までの文字で始まる変数は整数型、それ以外の文字で始まる変数  
11 は実数型という暗黙の約束があるため、型宣言をしなくてもよいのです。この約束はプログラムの  
12 記述を短くし、一見すると便利のように思えます。しかしこの種の省力化はプログラマの思い込み  
13 や勘違い、ケアレスミスによるバグを作りこみやすいことが分かっており、最近のプログラミング  
14 言語では採用されていません。

## 15 COBOL

16 FORTRAN が第一世代の科学技術計算用プログラミング言語であるのに対して、COBOL (コボ  
17 ル) は第一世代の事務処理用プログラミング言語です。COBOL の名称は common business oriented  
18 language に由来します。1960 年に最初の言語仕様が策定され、同じ年に最初のコンパイラが開発  
19 されました。FORTRAN 同様に、時代と共に言語仕様を少しずつ改変しながら、現在まで使用さ  
20 れ続けている長寿の言語です。

21 図 1.2 はプログラム例です<sup>9</sup>。まず、宣言部が長いという特徴を持っています。これはプログラ  
22 ムの保守性を高めるためです。次に、私たちがよく知る形式の代入文がなく、英文のような代入文

<sup>8</sup>1977 年以前のは FORTRAN と記し、それ以降のものは Fortran と記します。

<sup>9</sup>このプログラムは 1985 年に策定された COBOL 第 3 次規格に基づきます。

```

IDENTIFICATION DIVISION.
PROGRAM-ID.      TEST.
ENVIRONMENT      DIVISION.
DATA             DIVISION.
WORKING-STORAGE SECTION.
01 WK-SUM        PIC 9(3)   COMP.
01 I             PIC 9      COMP.
PROCEDURE DIVISION.
    MOVE 0 TO WK-SUM.
    PERFORM VARYING I FROM 1 BY 1 UNTIL I > 10
        ADD I TO WK-SUM
    END-PERFORM.
    DISPLAY WK-SUM.

```

図 1.2: 数の和を求める COBOL のプログラム例

1 "MOVE 0 TO WK-SUM." や "ADD I TO WK-SUM." が使用されています。これはプログラムの可読  
 2 性を高めるためです。WK-SUM は WK から SUM を減ずる演算式ではなく、ひとつの変数を表します。  
 3 マイナス記号 - は COBOL では減算記号ではありません<sup>10</sup>。また、初期の COBOL の言語仕様  
 4 には浮動小数点データ型がありませんでした。逆に COBOL では任意桁数の整数型演算が可能で  
 5 す。これらの特徴は COBOL が事務処理用言語として設計されたためです。

## 6 ALGOL

7 ALGOL (アルゴル) は、FORTRAN、COBOL の開発に刺激を受けて、それまでのプログラ  
 8 ミングの諸概念を整理/明確化して作られた言語です。名前は algorithmic language に由来します。  
 9 国際的な協力の下に言語仕様が検討され、1958 年に最初の仕様が作られました。しかしそのとき  
 10 には既に FORTRAN と COBOL が広く普及しており、ALGOL が実際に使われることがありませ  
 11 んでした。しかし、言語解析技術など、その後のプログラミング言語の研究に与えた影響は小さ  
 12 くありません。アルゴリズムを説明するときには今でもよく「ALGOL-like (ALGOL ライクな)」  
 13 という言い回しが使われます。

14 図 1.3(a) に、1 から 10 までの数の和を求めるプログラムを ALGOL で記述した例を示します<sup>11</sup>。  
 15 文末のセミコロンや for に、現在の C/C++/java との共通点を見ることができます。

## 16 Pascal

17 Pascal (パスカル) は、ALGOL の仕様策定にも関わったスイス工科大のヴィルト (Wirth) 教  
 18 授が、効率的なプログラム開発とプログラム実行ができる言語を目指して開発した言語です。構造

<sup>10</sup>ただし、- の前後に空白などの区切り記号がある場合には減算記号とみなされます。

<sup>11</sup>このプログラムが正しいプログラムである保証はありません。言語仕様書やインターネット上のサンプル・プログラムを真似て作りました。

<pre> begin   integer i;   real sum;   sum := 0;   for i := 1 step 1 until 10 do     sum := sum + i;   Print Real(sum) end </pre> <p>(a) ALGOL</p>	<pre> program test(input, output) var i : integer,     sum : real; begin   sum := 0;   for i := 1 to 10 do     begin       sum := sum+i     end;     writeln(sum)   end. </pre> <p>(b) Pascal</p>
--	---

図 1.3: ALGOL/Pascal のプログラム例

- 1 的なプログラム開発が可能でありながら、機能を欲張らず、プログラムの重要概念がコンパクトに  
 2 まとめられている点が特徴です。情報学科での教育用言語にも適しており、1970 年代初めから 80  
 3 年代終わりまで幅広く使用されました。言語処理系の普及を容易にするために、中間言語 P-code  
 4 を導入し、P-code を解釈実行するインタプリタ (2.2 節参照) さえ用意すれば処理系が移植できる  
 5 という手法を取りました。Pascal という名称は、数学者パスカルに由来します。
- 6 図 1.3(b) に、1 から 10 までの数の和を求めるプログラムを Pascal で記述した例を示します。
- 7 Pascal の構文の特徴は、変数宣言 (たとえば "i : integer") において変数名 ("i") の後にデー  
 8 タ型名 ("integer") が記述される点にあります。この並び順は、コンパイル処理の中の意味解析  
 9 を少しだけ難しくする例として有名です<sup>12</sup>。

## 10 C

- 11 AT&T のベル研究所 (Bell laboratories) で開発された C (シー) は、UNIX を記述した言語と  
 12 して有名です。Pascal などと比較してエレガントな言語とはいいがたいのですが、OS をほとんど  
 13 この言語だけで記述できるなど応用範囲が広く、急速に普及しました。現在、ほとんどのプログラ  
 14 ムの開発において C はアセンブリ言語に取って替わったと言ってよいでしょう。1972 年に最初の  
 15 処理系が開発されました。C を普及させた要因のひとつはカーニハン (Brian Kernighan) とリッ  
 16 チー (Dennis Ritchie) が 1978 年に書いた C マニュアル<sup>13</sup> の貢献が大きいとも言われています。  
 17 この当時の C を俗に「カーニハン&リッチーの C (K&R C)」と呼びます。C の仕様策定と標準化  
 18 は、1983 年に ANSI<sup>14</sup> に受け継がれ、これが今日 ANSI C と呼ばれているものです。

<sup>12</sup>コンパイル処理の詳細は後期選択科目「コンパイラ」において学びます。

<sup>13</sup>B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall(1978). Second edition (1988).

<sup>14</sup>the American National Standards Institute

<pre>int main(){     int i;     float sum = 0;     for(i = 1; i &lt;= 10; i++)         sum += i;     printf("%f\n",sum); }</pre> <p>(a) C</p>	<pre>int main(){     float sum = 0;     for(int i = 1; i&lt;= 10; i++)         sum += i;     cout &lt;&lt; sum &lt;&lt; endl; }</pre> <p>(b) C++</p>
---	--

図 1.4: C/C++のプログラム例

- 1 図 1.4(a) に、1 から 10 までの数の和を求めるプログラムを C で記述した例を示します。
- 2 混乱を避けるため、これ以降は C のことを「C 言語」と呼びます。
- 3 **C++**
- 4 C++（シープラスプラス）は、1980 年前半にベル研究所で開発された拡張 C 言語です。C 言
- 5 語の構文規則を洗練し、機能を整理した言語でもあります。最も大きな特徴は C 言語にオブジェ
- 6 クト指向を導入した点にあります。C++の仕様は ANSI C にも影響を与えました。
- 7 図 1.4(b) に、1 から 10 までの数の和を求めるプログラムを C++で記述した例を示します。た
- 8 だしこの例題では C 言語と C++の違いが際立ちません。

## 9 Basic

- 10 Basic（ベーシック）<sup>15</sup> は、コンピュータがきわめて高価であった時代に数 K バイトから数 10K
- 11 バイトのメモリ容量に収まるように作られた言語です。最初の処理系は 1964 年に開発されていま
- 12 すが、その後、1980 年代末までパーソナルコンピュータ用言語として普及しました。機能は限定
- 13 され、実行速度は遅いのですが、覚えやすいという特長から主に初心者用として普及してきまし
- 14 た。しかし最近では Visual Basic のように高機能な言語も作られています。
- 15 図 1.5 に、1 から 10 までの数の和を求めるプログラムを Basic で記述した例を示します。なお、
- 16 Basic の言語仕様には多くの改良版、方言があり、図 1.5 が典型的な Basic のプログラムという訳
- 17 ではありません。

<sup>15</sup>歴史的には全て大文字の BASIC という名称がオリジナルの言語名称ですが、現在はほとんど全ての派生言語が Basic という名称を使っています。



```

10 sum = 0
20 for i = 1 to 10
30     sum = sum+i
40 next
50 print sum
60 end

```

図 1.5: Basic のプログラム例

```

with Ada.Text_IO; use Ada.Text_IO;
procedure main is
    sum: Float;
begin -- of main
    sum := 0;
    for i in 1..10 loop
        sum := sum+1;
    end loop;
    put(sum);
end main;

```

図 1.6: Ada のプログラム例

## 1 Ada

Ada (エイダ) は、国防総省が組み込みシステム向けの信頼性・保守性（つまり兵器の信頼性・保守性）に優れたプログラミング言語の開発を目指して米国言語仕様を国際入札によって選定した言語です。1977 年から検討が始まり、1983 年に言語仕様が策定されました。Ada にはプログラム抽象化などのその当時の先進的な概念が導入されました。しかし言語仕様が複雑で大き過ぎたためコンパイラの開発に手間取り、大企業や軍事などの特別な環境以外では普及が進まず、そうこうしている間に多くのプログラマに忘れ去られつつあります。言語の名称は、歴史上最初のプログラマと言われる Ada 夫人に由来します。

図 1.6 に、1 から 10 までの数の和を求めるプログラムを Ada で記述した例を示します。プログラムの雰囲気が ALGOL や Pascal に似ています。C プログラムなどと比べると、読み易そうな印象を受けます。

## 12 Java

Java (ジャバ) は、1990 年初めに家電組み込み用言語として開発が進められましたが、折しもインターネットの拡大期にあったため、その目的をネットワーク対応にシフトし、大きな注目を集めるようになりました。1995 年に最初の処理系が開発され、その後も改良が続けられています。

```

class Test {
    public static void main(String[] args){
        float sum = 0;
        for(int i = 1; i <= 10; i++){
            sum += i;
            System.out.println(sum);
        }
    }
}

```

図 1.7: Java のプログラム例

1 Java の構文は C++ に類似しており、C/C++ プログラマには馴染みやすかった点も普及の一因と  
 2 思われます。様々な興味深い特徴を持ちますが、その中でも、中間言語（バイトコード）を設定  
 3 し、プラットフォーム非依存<sup>16</sup>である点は大きな特徴です。現在、最も普及している言語のひと  
 4 つです。

5 図 1.7 に、1 から 10 までの数の和を求めるプログラムを Java で記述した例を示します。プログ  
 6 ラムが C や C++ に似ていることに気づくでしょう（図 1.4 参照）。

## 7 Python

8 Python（パイソン）は、Java とほぼ同時期にオランダで開発され、その後、改良が継続されて  
 9 きた言語です。オブジェクト指向は当初から組み込まれていますが、その後、関数型などの機能も  
 10 取り込み、また多数の有用なライブラリ（たとえば機械学習のライブラリ）が開発されています。  
 11 プログラムサイズが小さくなるように言語が設計されており、結果、プログラムの可読性が高く、  
 12 開発効率が良いことから急速に利用が広がっています。

13 図 1.8 に、1 から 10 までの数の和を求めるプログラムを Python で記述した例を示します。

14 まず気づくことは文末にセミコロン ; がありません。セミコロンは C 言語以降のほとんどのプ  
 15 ログラミング言語に採用されている記法ですが、それがないとプログラミング言語として成り立た  
 16 ない訳ではありません<sup>17</sup>。

17 次に、変数 `sum` にデータ型の宣言がないことに注意してください。Python は動的データ型付け  
 18 (dynamic data typing) の言語のため、変数にはどんなデータも自由に代入可能です。1 行目の右  
 19 辺 `float(0)` は `float` 型の 0 です。よってこの場合、`sum` には `float` 型のデータが代入されます。  
 20 もし `sum = 0` と書くならば `sum` には `int` 型のデータが代入されます。

<sup>16</sup>プラットフォームとは、コンピュータの機種（またはアーキテクチャ）および OS の種類のことです。機種や OS に  
 依らず成り立つ性質をプラットフォーム非依存と言います。しばしば OS 非依存とも言います。

<sup>17</sup>技術的な言い方をすれば、セミコロンが無いからといって、必ずしも構文構造が曖昧（あいまい）になり、プログラムの  
 構造判定ができなくなる訳ではありません。

```

sum = float(0)
for i in range(1,11):
    sum += i
print(sum)

```

図 1.8: Python のプログラム例

2 行目の for 文の `in rang()` を用いる書き方は最近のプログラミング言語の流行りの書き方です。

Python のブロック構造は各行のインデントの深さで決まるルールになっています。よって、図 1.8 のプログラムを

```

5 sum = float(0)
6 for i in range(1,11):
7     sum += i
8     print(sum)

```

と書くと、`print(sum)` はループの中で 10 回繰り返されることになります。このルールはプログラマにプログラムの書き方を強制することになるため、「誰が書いても同じ形のプログラムになりやすい」＝「プログラムが読みやすい」という利点があると考えられています。

## 12 Swift

最新のプログラミング言語の例として、Apple 社が 2014 年に発表し、直後からユーザーに提供されているプログラミング言語 Swift（スイフト）を紹介します。

プログラミング言語の仕様に関する研究は 1990 年代に成熟期に入り、それ以降に新たに提唱・開発されたプログラミング言語は数えるほどしかありません。しかし、言語の仕様に関する研究は着実に進んでおり、Swift は、Apple 社のキャッチコピーをそのまま使えば「モダン、安全、高速、インタラクティブ」を標榜する言語として開発されています。技術的には、新しいデータ型の導入、データ型のチェックの強化を行っています。

図 1.9 は、Swift によるプログラム例です。

1 行目の変数宣言では、変数名：データ型名 となっています。これは、Pascal/Ada と同じ順番です。これに対して、ALGOL/C/C++/Java では、データ型名 変数名 という順番です。この順番に優劣はありませんが、言語仕様の歴史的な流れとして興味深く感じられます。

2 行目の for 文の書き方は、Ada に類似しており、また Python とも似ており、当世風です。単純な繰り返しの記述法としては C/C++/Java の `for(;;)` よりも読みやすいと言えるでしょう。

Swift では、3 行目を `sum += i` と書くことは許されません。何故ならば、この式の左辺の変数 `sum` は浮動小数点数型であり、右辺に現れる 変数 `i` は整数型です。両辺でデータ型が異なるため、

```

var sum : Float = 0
for i : Int in 1...10 {
    sum += Float(i)
}
print(sum)

```

図 1.9: Swift のプログラム例

1 注意が必要です。C/C++/Java では右辺で計算された整数値を言語処理系が浮動小数点数値へ自  
 2 動的に（暗黙に）型変換し、代入を実行しますが、Swift ではこのような暗黙の型変換は許されず、  
 3 プログラマが明示的に型変換を記述しなければなりません。プログラマにとっては面倒な作業です  
 4 が、Swift では暗黙の型付け、型変換は往々にしてバグを生むと考えています。これは Python と  
 5 は対極にあるスタイルです。一般に、Swift の方法を静的型付けと呼び、python のそれを動的型付  
 6 けと呼びます。それぞれ一長一短があり、前者は安全でバグの予防を目的とし、後者は柔軟なプロ  
 7 グラミングを目的としています。

8 ここまで紹介したプログラミング言語はいずれも手続き型プログラミング言語であって、プログ  
 9 ラムの実行ではプログラム中に書かれた実行文を上から順に実行していきます。

10 これに対して以下に紹介する言語は宣言型言語であり、実行順序はプログラム中の記述順に依存  
 11 しません。プログラムの内容は数学的に記述されるため、実行されるべきもの（関数や論理式）が  
 12 数学的な性質に基づき、適切なタイミングで実行されていく性質を持ちます。

### 13 LISP

14 LISP（リスプ）は、1958年にMITのマッカーシー（McCarthy）によって発明された言語であ  
 15 り、名前は List Processing（リスト処理）に由来しています。歴史上最初の関数型言語であり、高  
 16 水準言語としても FORTRAN に次ぐ歴史を持ちます。様々な言語拡張を経て、Common Lisp とい  
 17 う新たな言語としていまだに使用され続けています。以下、単に Lisp と書いた場合には Common  
 18 Lisp を指します。リスト処理の内容を含めて Lisp の詳細は後章で述べますが、高い表現力と柔軟  
 19 性を持つ言語として、人工知能的な処理などの非定型的な処理に適しています。

20 図 1.10(a) に、1 から  $n$  までの数の和を求める関数を Lisp インタプリタ上に定義し、1 から 10  
 21 までの数の和を同じくインタプリタ上で求める例を示します。なお、宣言型言語は for ループのよ  
 22 うな繰り返し処理に不向きなので、プログラムは一般に再帰を用いて記述します。

<pre>&gt; (defun sum (n)       (if (= n 1)           1           (+ n (sum (- n 1))))) &gt; (sum 10)</pre> <p>(a) Lisp</p>	<pre># let rec sum n =       if n = 1       then 1       else n + sum (n - 1) ;; # sum 10 ;;</pre> <p>(b) ML</p>
--	--

図 1.10: Lisp/ML のプログラム例

```
sum(1,1) :- !.
sum(N,X) :- N1 is N-1, sum(N1,X1), X is N+X1.

?- sum(10,X).
```

図 1.11: Prolog のプログラム例

## 1 ML

ML (エムエル) は、1974 年に定理証明システムの一部として開発された関数型言語であり、強力なデータ型推論機能があることで有名です。人工知能的な処理に由来する点では LISP と類似しています。LISP がやや特異な構文規則を採用している（たとえば  $n-1$  を  $(- n 1)$  と書かねばならない）のに対して、ML は初心者にも取っ付き易い言語になっています。パターンマッチングなどの、プログラミング言語としての機能も充実しています。

図 1.10(b) に、図 1.10(a) と同様の例を、ML の方言のひとつである Caml<sup>18</sup> について示します。

## 8 Prolog

Prolog (プロログ) は、1970 年代前半にフランスで開発された論理型プログラミング言語です。論理式のある特殊な形式を用いて、限定された論理的推論を行うことができます。そして、その推論過程をプログラムの実行と見なします。複数の解を求めることができる、逆計算ができるなどの、これまでのプログラミング言語（プログラミングパラダイム）に見られない表現力と柔軟な機能を有します。Lisp や ML 同様に人工知能的な処理に適しているのは、宣言型言語の共通する特徴でもあります。詳細はこの授業の後半で述べます。

図 1.11 に、1 から  $n$  までの数の和を求める述語の定義と、1 から 10 までの数の和を Prolog インタプリタ上で求める例を示します。

<sup>18</sup>Caml はフランスで開発された ML の処理系です。詳しくは <http://caml.inria.fr/>などを参照。

### 1.3 プログラミング言語の比較

ここまで代表的なプログラミング言語を簡単に眺めてきました。

全ての言語について和計算を行うプログラム例を示しました。結果、どのプログラムも似た形になっていることに気づくはずです。実際、多くの手続き型言語は共通する概念を持っています。思いつくままに挙げてみると...

条件分岐 ... 処理内容を選択する。

繰り返し ... 同一の処理を一定回数繰り返す。

ブロック構造 ... 処理単位を階層化する。

手続き呼び出し/関数呼び出し ... 処理内容を抽象化/共通化する。

配列 ... 大量のデータの並び。

構造型/レコード型 ... 複数データをまとめて階層化する。

入出力 ... データの入出力を行う。

これらの実現方法は言語によって異なる場合があります。しかし、ほとんどの手続き型言語ではこれら概念（機能）をサポートしており、類似したプログラミングが可能です。よって、ひとつの手続き型言語を一定レベルまで掘り下げてマスターすると、その他の手続き型言語は簡単にマスターできるものです。問題は、最初の言語を十分にマスターできているか否かです。C++を中途半端にしかマスターしていない<sup>19</sup> ままに他の言語を勉強するのはあまり効率的ではありません。

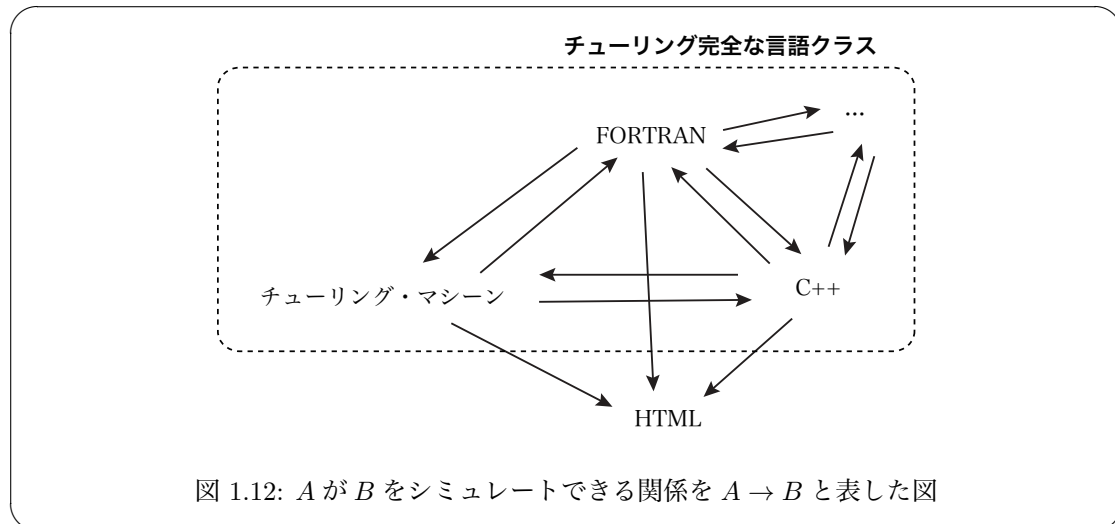
さて、上のような理由で、この授業では手続き型言語については新たに学ぶことをしません。この授業では、普段あまり接する機会のない非手続き型言語の Lisp と Prolog を通して、手続き型とは全く異なる発想のプログラミング言語を学んでいきます。

### 1.4 チューリング完全

前節ではプログラミング言語を構成する諸概念が多くの言語で共通であることを述べました。それらが共通になってしまうことは、ひとつの数学的必然性 — そのプログラミング言語がチューリング完全 (Turing complete) でなければならない — ことに起因します。

チューリング完全とは、そのプログラミング言語（広義には計算システム）がチューリング・マシーンと等価な計算能力を持つことを言います。「チューリング・マシーンと等価」とは、簡単に言えば、チューリング・マシーンをシミュレートできることです。もしチューリング完全ならば任

<sup>19</sup>ただし C++の悩ましいところは、C++は言語仕様が相当に大きく少しマニアックなところがあるので、C++を完璧にマスターしようなどと考えるのもあまり得策とは言えません。



1 意の計算可能な問題は、そのプログラミング言語で計算できることとなり、十分な計算能力を持つこ  
2 との保証となります。

3 チューリング・マシンを思い出しましょう。チューリング・マシンは無限長のテープを持っ  
4 ています。配列を用いればテープをシミュレートできます。尤もコンピュータのメモリは有限です  
5 から無限長の配列は用意できませんが、メモリが無限であるならば、理論上は無限長の配列も用  
6 意できると考えましょう。チューリング・マシンの有限状態部はプログラミング言語に条件分  
7 岐、繰り返しの機能があればシミュレートできます。よって世の中に知られているほとんど全ての  
8 プログラミング言語はチューリング完全です。一般にチューリング完全でないものはプログラミ  
9 ング言語と呼びません。しかし実際にはチューリング完全でない言語：Web 文書作成用の HTML  
10 (HyperText Markup Language) やデータベース言語 SQL (Structured Query Language) でも  
11 プログラミング言語と呼ばれることがあります。

12 図 1.12 にチューリング・マシンと様々なプログラミング言語のシミュレート関係を図示しま  
13 した。たとえば FORTRAN はチューリング・マシンをシミュレートできます。逆に、全ての計  
14 算問題はチューリング・マシンで計算できますから、チューリング・マシンは FORTRAN の  
15 全てのプログラムの動作をシミュレートできます。よって、両者の間には両方向の矢印が張られ  
16 ます。C++ とチューリング・マシンも同様です。C++ はチューリング完全ですから、C++ は  
17 FORTRAN の全てのプログラムの動作をシミュレートできます。逆も成り立ちます。よって C++  
18 と FORTRAN の間には両方向の矢印が張られます。このようにしてほとんど全てのプログラミン  
19 グ言語とチューリング・マシンの間には両方向の矢印が張られます。しかし、HTML へは片方  
20 向の矢印しか張られません。HTML はチューリング完全ではないからです。

表 1.3: 今、話題のプログラミング言語のランキング (TIOBE Programming Community index による)

#	言語名	#	言語名	#	言語名
1	Java	18	Go	35	Rust
2	C	19	Delphi/Object Pascal	36	Ada
3	Python	20	Visual Basic	37	LabVIEW
4	C++	21	SAS	38	Logo
5	Visual Basic .NET	22	PL/SQL	39	Kotlin
6	C#	23	Dart	40	Ladder Logic
7	JavaScript	24	D	41	Bash
8	PHP	25	Scratch	42	Julia
9	SQL	26	COBOL	43	Haskell
10	Objective-C	27	Fortran	44	Hack
11	MATLAB	28	Scala	45	PowerShell
12	Assembly language	29	Lua	46	Awk
13	Perl	30	Transact-SQL	47	ML
14	R	31	ABAP	48	PL/I
15	Ruby	32	Lisp	49	Erlang
16	Groovy	33	Prolog	50	RPG
17	Swift	34	Scheme	...	...

## 1.5 補足

### 1.5.1 高水準プログラミング言語の状況

オランダの TIOBE というソフトウェア会社が毎月、その時点で話題になっているプログラミング言語のランキングを

[http://www.tiobe.com/tiobe\\_index](http://www.tiobe.com/tiobe_index)

に発表しています。ここでいう「話題」(英語の原文では popularity) とは、主にインターネットの検索に基づいたランキングです。詳細は上のページを参照してもらうこととし、2019 年 3 月のランキングは表 1.3 の通りです。過去、java、C、C++ は常に上位 3 位以内にいますが、評価値は徐々に下がってきており、今年は C++ が 3 位から陥落し、代わりに Python が入りました。

上に書いたように、このランキングは話題性によるランキングですから、その言語が利用されている頻度や言語の優秀性とは直接関係しないことに注意してください。また、このランキングは上動は激しく、信頼性に欠けるとの指摘もあります。ひとつの参考資料として理解してください。



## 第2章 プログラミング言語処理系

様々なプログラミング言語を学ぶに当たり、それら言語のプログラムを実際に実行する言語処理系についても知識を整理しておくことが有益です。

プログラミング言語のプログラムを実際に実行する方法は、コンパイラを用いる方法とインタプリタを用いる方法の二種類<sup>1</sup>に大別されます。

### 2.1 コンパイラ

コンパイラ (compiler) は、あるプログラミング言語  $L_S$  で書かれたプログラムを別のプログラミング言語  $L_T$  で書かれたプログラムへ自動変換するソフトウェアです。通常、変換元の言語 (source language)  $L_S$  は高水準プログラミング言語であり、変換先の言語 (target language)  $L_T$  は低水準言語 (多くの場合は機械語) です。変換前の  $L_S$  のプログラムをソース・プログラム、原始プログラム (source program)、ソース・コード、原始コード (source code) などと呼びます。変換後の  $L_T$  のプログラムをオブジェクト・プログラム、目的プログラム (object program)、オブジェクト・コード、目的コード (object code) などと呼びます<sup>2</sup>。

なお、オブジェクト・コードはそのままでは実行できないことは注意しておきます。たとえばソース・プログラム中で三角関数  $\sin()$  を呼んでいる場合、オブジェクト・コードにもその関数呼び出しは存在しますが、 $\sin()$  の値を計算するプログラムはオブジェクト・コードには含まれません。そのため、リンカ (linker) を使ってシステムが用意する三角関数プログラムと結合し、実行可能コード (executable code) を作る必要があるのです。以下では、コンパイラにリンカも加えて、広い意味でコンパイラと呼ぶ場合があります。

コンパイラを用いたプログラム開発では、ソース・プログラムを作成した後にいちいちコンパイル (compile) せねばならないという手間が掛かることが欠点です。しかし、一旦実行可能コードが完成した後はプログラムが高速に実行できるという利点があります。高速性を重視する大規模科学技術計算などでは重要です。

<sup>1</sup>前者をコンパイル方式またはコンパイラ方式と呼びます。後者をインタプリタ方式またはインタプリタ方式と呼びます。google 検索のヒット数で術語の使用頻度を調べてみると、コンパイル方式 581 件、コンパイラ方式 267 件、インタプリタ方式 4 件、インタプリタ方式 1620 件 (2005 年 4 月) となりました。インタプリタ方式という呼び方がほとんどされていないのは語感のせいと思われます。

<sup>2</sup>google 検索のヒット数で術語の使用頻度を調べてみると、ソース・プログラム 42400 件、原始プログラム 1150 件、ソース・コード (あるいはソースコード) 698000 件、原始コード 370 件、オブジェクト・プログラム 1900 件、目的プログラム 1920 件、オブジェクト・コード 6990 件、目的コード 1010 件でした。

## 2.2 インタプリタ

コンパイラがソース・プログラムを実行可能コードに変換するソフトウェアであるのに対して、インタプリタ (interpreter) はソース・プログラムを逐次解析 (解釈) しつつ実行するソフトウェアです。

インタプリタを用いる場合、ソース・プログラムをいちいち実行可能コードへ変換する必要はなく、プログラムを作ったその場でプログラムの実行ができます。あるいはプログラムを作りながら実行できます。そのため、プログラマはシステムと対話的にプログラムを作成することができ、応答性に優れた開発環境を得られるという利点があります。しかし、実行の度にプログラムを解析しなければならないため、実行速度が遅いという欠点もあります。

「プログラムを作りながら実行できる」という特性を活かすには、プログラムの意味内容が局所的に定まり、広範囲の解析を必要としない単純な構造の言語が適しています。あるいはプログラムの実行環境が動的に変化するため、コンパイルしても実行性能が期待できないような言語が適しています。この授業で詳しく学ぶ宣言型言語 Lisp、Prolog の言語処理系はインタプリタで実装されます。前章で取り上げなかった Pearl や Ruby などのスクリプト言語もインタプリタまたは次に述べるコンパイラ・インタプリタ方式で実装されています。

## 2.3 コンパイラ・インタプリタ

中間言語を介した実行方法です。コンパイラがまずソース・プログラムを中間言語のプログラムに変換します。その中間言語プログラムをインタプリタが逐次実行します。この場合、コンパイラとインタプリタの両方を開発せねばなりません。しかし、中間言語プログラムが一旦生成されたならば、インタプリタが動く全ての環境でプログラムを実行可能であるため、可搬性に優れた方式です。実際、Pascal ではこの方式で様々なコンピュータに Pascal コンパイラが移植されました。Java では中間コードがネットワークを介して様々なコンピュータで実行できます。

可搬性以外のこの方式の特徴は、コンパイラによる実行方式とインタプリタによる実行方式の間と考えてよいでしょう。

## 2.4 ジャストインタイム・コンパイラ

中間コードをインタプリタ内でネイティブ・コードにコンパイルして実行する方式、あるいはそのコンパイラをジャストインタイム・コンパイラ (Just-In-time compiler) または **JIT** コンパイラなどと呼びます。Java のそれが有名です。Java の場合、新しい実行コードが呼び出される度に JIT コンパイラが起動します。コード中にループがあるような場合には中間コードをそのまま実行するよりも効果的です。

1    なお、実行速度を除けば、ユーザからは単に中間コードがインタプリタによって実行されている  
 2    としか見えないため、JIT コンパイラを用いても前節のコンパイラ・インタプリタの可搬性が損な  
 3    われることはありません。JIT コンパイラはインタプリタの実行速度を加速させるための仕組みの  
 4    ひとつと考えればよいでしょう。プログラム実行中にコンパイラを起動すると、プログラム本体の  
 5    実行は一時中断されます。プログラム実行の不自然な挙動<sup>3</sup>を避けるためにはコンパイラが長時間  
 6    動作することは好ましくなく、コンパイルに長時間を費やすことはできません。そのため実行コー  
 7    ドの質では妥協せざるを得ません。

## 8    2.5    補足

### 9    2.5.1    コンパイラ=編集器?

10    本来、コンパイラを「コンパイラ」と呼ぶのはちょっと変です。と言うのも、「コンパイラ」はも  
 11    ともと英語の compiler<sup>4</sup>であり、それをそのまま邦訳すると、「編集器」、「編集者」を意味します。  
 12    確かにコンパイラの仕事のひとつにプログラムの編集作業がありますが、ここまで見てきたように  
 13    コンパイラの本来の仕事はプログラム変換です。何故、プログラム変換器がコンパイラ（編集器）  
 14    と呼ばれるようになったのでしょうか。

15    実は歴史的には、「コンパイラ」はもともとサブルーチンを編集するプログラム、すなわちリン  
 16    カのことでした。ところが、FORTRAN コンパイラが開発される直前の 1954 年頃に “algebraic  
 17    compiler” という造語がアセンブラを少し高機能にしたプログラム変換と編集を行うプログラムの  
 18    名称として広く使用されるようになります。そして言葉の意味が少しずつずれていき、「コンパイ  
 19    ラ」の意味が現在のそれに移ってしまいました。

### 20   2.5.2    コンパイル処理を実感してみる

21    上に「コンパイラは、あるプログラミング言語  $L_S$  で書かれたプログラムを別のプログラミング  
 22    言語  $L_T$  で書かれたプログラムへ自動変換するソフトウェアです」と書いたが、なかなか実感でき  
 23    ないかもしれません。そこで、C++コンパイラ — プログラミング言語 C++ で書かれたプログ  
 24    ラムをアセンブリ言語で書かれたプログラムへ自動変換するソフトウェア — で実際にアセンブリ  
 25    コードを出力してみます。

26    ここでは UNIX 環境でそれを試します。まず、以下の C++プログラムを `test.cpp` というファ  
 27    イル名で作成しておきます。

<sup>3</sup>たとえば出力処理が突然止まったりする現象が起これと、ユーザは不安になったり、いらついたりする原因になります。

<sup>4</sup>最近、「コンピレーション CD」という言葉を耳にしますが、そのコンピレーションは原語では compilation であり、コンパイラと同じく、compile（編集する）という動詞の派生語です。音楽関係では、コンパイラと言えば、文字通り、編集者を指します。ちなみにプログラミング言語の分野では、compilation をコンパイレーションと発音することが多いようです。

```

1  #include <iostream>
2  using namespace std;
3
4  int main(){
5  float x = 3.0, y = 4.0, z;
6      z = 7.0*x+y;
7  cout << z << endl;
8  }

```

9 そして UNIX 環境で以下のコマンドでプログラムをコンパイル&リンクします。

```
10 > c++ test.cpp
```

11 そうすると、a.out という実行可能コードが生成されますから、これを実行すると

```
12 > ./a.out
13 25

```

14 と計算結果が出力されます。

15 さて、コンパイル時にオプション `-S` をつけると、コンパイルされた後のアセンブリコードが  
16 ファイル `test.s` に生成されます。つまり、

```
17 > c++ -S test.cpp
```

18 そこで、`test.s` の中を先頭から少し見てみると、たとえば以下のようなものです。

```

19 .section __TEXT,__text,regular,pure_instructions
20 .build_version macos, 10, 14 sdk_version 10, 14
21 .section __TEXT,__literal8,8byte_literals
22 .p2align 3                ## -- Begin function main
23 LCPI0_0:
24 .quad 4619567317775286272    ## double 7
25 .section __TEXT,__literal4,4byte_literals
26 .p2align 2
27 LCPI0_1:
28 .long 1082130432            ## float 4
29 LCPI0_2:
30 .long 1077936128            ## float 3
31 .section __TEXT,__text,regular,pure_instructions
32 .globl _main
33 .p2align 4, 0x90
34 _main:                    ## @main
35 .cfi_startproc
36 ## %bb.0:
37 pushq %rbp
38 .cfi_def_cfa_offset 16
39 .cfi_offset %rbp, -16
40 movq %rsp, %rbp
41 .cfi_def_cfa_register %rbp
42 subq $32, %rsp
43 movq __ZNSt3__14coutE@GOTPCREL(%rip), %rdi

```

```

1  movsd LCPI0_0(%rip), %xmm0    ## xmm0 = mem[0],zero
2  movss LCPI0_1(%rip), %xmm1    ## xmm1 = mem[0],zero,zero,zero
3  movss LCPI0_2(%rip), %xmm2    ## xmm2 = mem[0],zero,zero,zero
4  movss %xmm2, -4(%rbp)
5  movss %xmm1, -8(%rbp)
6  movss -4(%rbp), %xmm1        ## xmm1 = mem[0],zero,zero,zero
7  cvtss2sd %xmm1, %xmm1
8  mulsd %xmm1, %xmm0
9  movss -8(%rbp), %xmm1        ## xmm1 = mem[0],zero,zero,zero
10 cvtss2sd %xmm1, %xmm1
11 addsd %xmm1, %xmm0
12 cvtsd2ss %xmm0, %xmm0
13 movss %xmm0, -12(%rbp)
14 movss -12(%rbp), %xmm0        ## xmm0 = mem[0],zero,zero,zero
15 callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEEElsEf
16 movq %rax, %rdi
17 leaq __ZNSt3__1L4endlIcNS_11char_traitsIcEEEEERNS_13basic_ostreamIT_T0_EES7_(%rip), %rsi
18 callq __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEEElsEPFRS3_S4_E
19 xorl %ecx, %ecx
20 movq %rax, -24(%rbp)          ## 8-byte Spill
21 movl %ecx, %eax
22 addq $32, %rsp
23 popq %rbp
24 retq
25 .cfi_endproc
26
27                                     ## -- End function
28 .p2align 4, 0x90                ## -- Begin function __ZNSt3__113basic_ostreamIcNS_11char_traitsIcEEEElsEPFRS3_S4_E: ## @_ZNSt3__113basic_ostreamIcNS_1
29 ...
30 後略

```

アセンブリコードの中に、元のソースプログラムの中に現れていた 7、3、4、`iostream` などの数値、記号が含まれています。また、アセンブリ命令 `mulsd` は乗算命令、`addsd` は加算命令です。


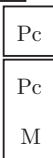
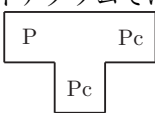

コンパイラとはソースプログラムをこのようなアセンブリコードに変換するソフトウェアなのです。その変換を行うためには、2 年生科目「形式言語とオートマトン」で学んだ内容をさらに深めたアルゴリズムを用いることになります。それについてこの授業ではこれ以上触れませんが、コンパイラの技法は初期のコンピュータサイエンスの金字塔のひとつと見なされています。

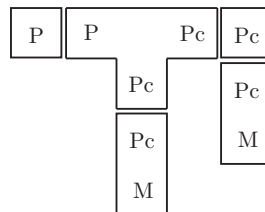
### 2.5.3 T ダイアグラム

プログラムの実行方法と言語処理系の関係を図式で表現するものとして **T** ダイアグラム (T 図式、T 記法ともいう) が有名です<sup>5</sup>。これを用いて Pascal と P コードの関係を表してみます。

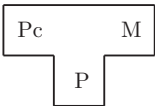
この図式では P コード (以下、 $P_c$  と表します) で書かれたプログラムを  $P_c$  と表します。このプログラムを実行させるには P コードインタプリタが必要ですが、 $P_c$  を入力とし、M (以下、

<sup>5</sup> “T” は tombstone (墓石) に由来します。

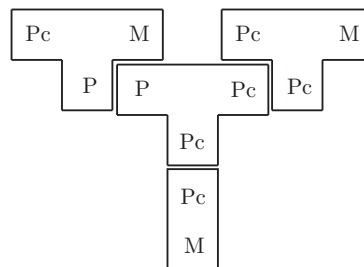
- 1 機械語の意味) の上で動作するインタプリタを  と表します。そして、上の Pc で書かれたプ  
 2 ログラムがインタプリタの上で実行される様子を  と表します。  
 3 さて、T ダイアグラムでは Pascal のプログラムを Pc のプログラムへ変換する、Pc で書かれた  
 4 コンパイラを  と表します。このコンパイラを用いて、Pascal で書かれたプログラ  
 5 ム  (以下、P は Pascal の意味) を Pc で書かれたプログラムへ変換し、そのプログラムを  
 6 インタプリタで実行する状況を T ダイアグラムでは以下の図式で表現します。



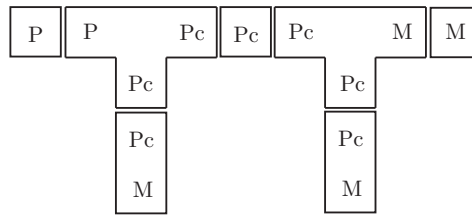
- 7  
 8 既に述べたように、Pascal が普及していく過程では、Pascal を利用したい各大学では自大学のコ  
 9 ンピュータで動く P コードインタプリタを作りました。次に、P コードで書かれたコンパイラを  
 10 普及元から手に入れました。そうすると上のダイアグラムに従って、Pascal プログラムを実行で  
 11 きるようになる訳です。

- 12 ここでクイズです。Pc のプログラムを M のプログラムへ変換する、Pascal で書かれたコンパイ  
 13 ラ  を開発し終えたと仮定しましょう。このコンパイラを用いて、Pascal プログラ  
 14 ムを機械語として実行するダイアグラムを考えてみなさい。

- 15 答えは以下の通りです。まず、以下の図式に従い、Pc のプログラムを M のプログラムへ変換す  
 16 る、Pc で書かれたコンパイラを求めます。

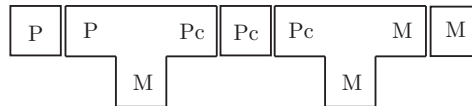


- 17  
 18 得られたコンパイラを用いて、左端の Pascal プログラムは右端の機械語プログラムへ変換され、  
 19 直接実行可能になります。



1

2 もうひとつクイズです。さらに一工夫すれば、以下のようなダイアグラムも実行可能です。どう  
3 すればよいでしょうか。



4

#### 5 2.5.4 プログラム意味論

6 プログラムの意味とは何でしょうか。

7 たとえば、以下の二つのプログラム:

```
8 int f(int n){
9     int sum = 0;
10    for(int i = 1; i <= n; i++) sum += i;
11    return sum;
12 }
```

13 と

```
14 int f(int n){
15     int sum = 0;
16     for(int k = 1; k <= n; k++) sum += k;
17     return sum;
18 }
```

19 は同じ意味を持ちます。その理由を問われた場合、「変数  $i$  と  $k$  は名前は違うけれども同じ働きを  
20 するから」などと答えることでしょう。次に、「同じ働きとはどういう意味なのか」と問われたな  
21 らば、たぶんプログラムの実行の様子を図に書いて、任意の  $n$  について、プログラムが同じ動作を  
22 行うことを示すでしょう。次に、上のプログラムと以下のプログラム:

```
23 int f(int n){
24     if(n <= 0) return 0;
25     else return f(n-1)+n;
26 }
```

27 の等価性を問われたならば、これには少し苦勞しますが、たぶん集合の概念を使って説明を試みる  
28 でしょう。

1 私たちはプログラムの中で変数名が本質的でないことを知っています。しかし変数名をデタラメ  
2 に用いてはいけないことも知っています。また全く異なる形であっても同じ計算を行うプログラム  
3 が存在することも知っています。そのような知識/理解を「何となく知っている」とか「約束事と  
4 して知っている」というのではなく、数学的に厳密に形式化したものがプログラム意味論 (theory  
5 of program semantics) と呼ばれています (ものすごくアバウトな言い方ですが)。意味論が構築  
6 されると、二つのプログラムが等価であるか否かを数学的に議論できるようになるため、あるプロ  
7 グラムから別のプログラムへの等価変換に数学的なお墨付きを与えることが可能になります。

8 意味論には様々あり、一番分かりやすいのが操作的意味論 (operational semantics) と呼ばれるも  
9 のです。これは抽象機械による実行として意味を定式化するものですが、この方法では実行順序や  
10 状態変化という概念を払拭できないため、数学的操作に馴染みません。表示の意味論 (denotational  
11 semantics) では、意味を集合や関数と捉えます。この方法では実行順序などが捨象されているた  
12 め、理論的な扱いがクリアになります。表示の意味論は関数型や論理型言語には馴染みやすいので  
13 すが、手続き型言語について表示の意味論を構築することは容易ではありません。

#### 14 2.5.5 静的/動的

15 しばしば静的、動的という言葉を目にします。たとえば静的意味/動的意味、静的変数/動的変  
16 数、静的束縛/動的束縛、静的スコープ/動的スコープなどなどです。何となく分かっているようで案  
17 外知らない言葉ではないでしょうか。

18 プログラミングの用語としての静的 (static) は、「コンパイル時に確定できる」性質のことを言  
19 います。逆に動的 (dynamic) は「コンパイル時には確定せず、実行時に確定する」性質のことを  
20 言います。

21 ひとつのソース・プログラムが与えられたとき、そのプログラムあるいはプログラムの中の構成  
22 要素は様々な性質を持ちます。その性質の中には、プログラムがどのような挙動を示そうとも不変  
23 な性質と、プログラムの挙動によって様々に変化しうる性質が存在します。前者はプログラムの挙  
24 動に依存しないため、コンパイラのフロントエンドにおいて比較的容易に判定できる性質です。た  
25 とえば「変数のデータ型が何か」、「変数のデータ実体はいつ生成されるのか」、「関数の本体はどこ  
26 にあるのか」などについて、コンパイル時に確定できるならば、それら変数、データ、関数は静的  
27 であると言えます。またそれらの性質の総体を静的意味 (static semantics) と呼びます。つまりプ  
28 ログラムの意味の中でコンパイル時に定まる部分を静的意味と呼ぶ訳です。コンパイル時に定まら  
29 ない意味を動的意味 (dynamic semantics) と呼びます。プログラムの多くの性質が静的であるよ  
30 うに設計されたプログラミング言語を静的であると言えます。C 言語は静的です。これに対して、  
31 たとえば関数型言語 Lisp のような、変数のデータ型さえもが動的にしか決まらないような言語は  
32 動的であると言えます。



## 1 第3章 C/C++を用いた再帰呼び出し

2 この章ではC/C++の再帰関数について復習し、後のLisp、Prologのための準備を行います。既  
3 に「プログラミング概論I, II」において学んだ内容ですが、再帰の考え方はこの講義の中核の概念  
4 ですから、ここで再確認しておきます。

### 5 3.1 再帰関数の基礎:階乗計算

6 自然数の1から $n$ までの積 $1 \times 2 \times \dots \times n$ を $n$ の階乗 (factorial) と呼び、 $n!$ で表します。これ  
7 を計算するプログラムを考えてみましょう。

#### 8 3.1.1 for 文による計算

9 階乗の計算をC関数で書くと図3.1の通り<sup>1</sup>です。この関数を以下のC++のmain関数から使  
10 用すれば、6!の計算が実行できます<sup>2</sup>。

```
11 int main(){  
12     cout << fact(6) << endl;  
13 }
```

14 考察 上のプログラムの実行の様子を図や表を用いて表しなさい。

#### 15 3.1.2 再帰による計算

階乗は、以下のように、漸化式を用いて定義することもできます。

$$f_1 = 1,$$

$$f_n = n \times f_{n-1}, \quad (n \geq 2)$$

あるいは

$$f(1) = 1,$$

$$f(n) = n \times f(n-1), \quad (n \geq 2)$$

<sup>1</sup>C/C++の初歩的知識の確認ですが、式" $i++$ "は、" $i = i+1$ "と等価です。式" $result *= i$ "は、" $result = result*i$ "と等価です。

<sup>2</sup>プログラム中の" $<< endl$ "は改行 (newline) を出力するための記述です。" $<< \backslash n$ "でも同様の改行は可能ですが、C++らしいプログラムには前者が望ましいでしょう。

```
int fact(int n){
    int result = 1;
    for(int i = 2; i <= n; i++) result *= i;
    return result;
}
```

図 3.1: 階乗の計算プログラム（繰り返しによる）

```
int fact(int n){
    if(n == 1) return 1;
    else return n*fact(n-1);
}
```

図 3.2: 階乗の計算プログラム（再帰による）

1 と書いてもいいでしょう。前者は添字を入力パラメータとしていますが、後者の方がパラメータが  
 2 明瞭ですから、このテキストではこれ以降、原則として後者の書き方を用いることとします。この  
 3 定義式は、実は図 3.2 のように、そのまま C 関数に書き下すことができます。このプログラムでは、  
 4 関数 `fact` を計算するプログラムの中で関数 `fact` を呼び出しています。このように、自分自身を  
 5 呼び出すことを再帰呼び出し、再帰的呼び出し（recursive call）と言います。また、このような関  
 6 数を「再帰的に定義されている（recursively defined）」関数、再帰的関数（recursive function）な  
 7 どと呼びます。そのプログラムのことを再帰プログラム、再帰的プログラム（recursive program）  
 8 と呼びます。

9 考察 図 3.2 の関数を 3.1.1 節の `main` 関数から呼び出したときの実行の様子を図や表を用いて表  
 10 示なさい。

11 上の考察の答えは図 3.3 のようになります。これはこの講義のための基礎知識として重要ですか  
 12 ら、以下に少し詳細に解説します。

13 1. まず、プログラムが起動すると、コンピュータのメモリ空間内に `main` 関数の作業領域が新  
 14 たに確保され、`main` の関数本体の実行が始まります。

15 2. `main` 関数内で関数呼び出し `fact(6)` が起きると、関数 `fact()` の作業領域が新たに確保さ  
 16 れます。このとき、作業領域内には局所変数 `n` を保持する変数領域が割り当てられます。そ  
 17 してそこには `main` 関数からの関数呼び出し `fact(6)` によって渡された整数値 6 がコピー  
 18 されます。引数値がコピーによって渡される場合を特に 値渡し（call by value）と呼んでい  
 19 ます。値渡しは、パラメータ受け渡しの最も基本的な仕組みです<sup>3</sup>。

<sup>3</sup>引数の渡し方には様々な方法が考案されており、他にはたとえば参照渡し（call by reference）がよく知られています。値渡し以外の引数の渡し方はプログラミング言語によって種々異なるため、言語毎に細心の注意を払う必要があります。

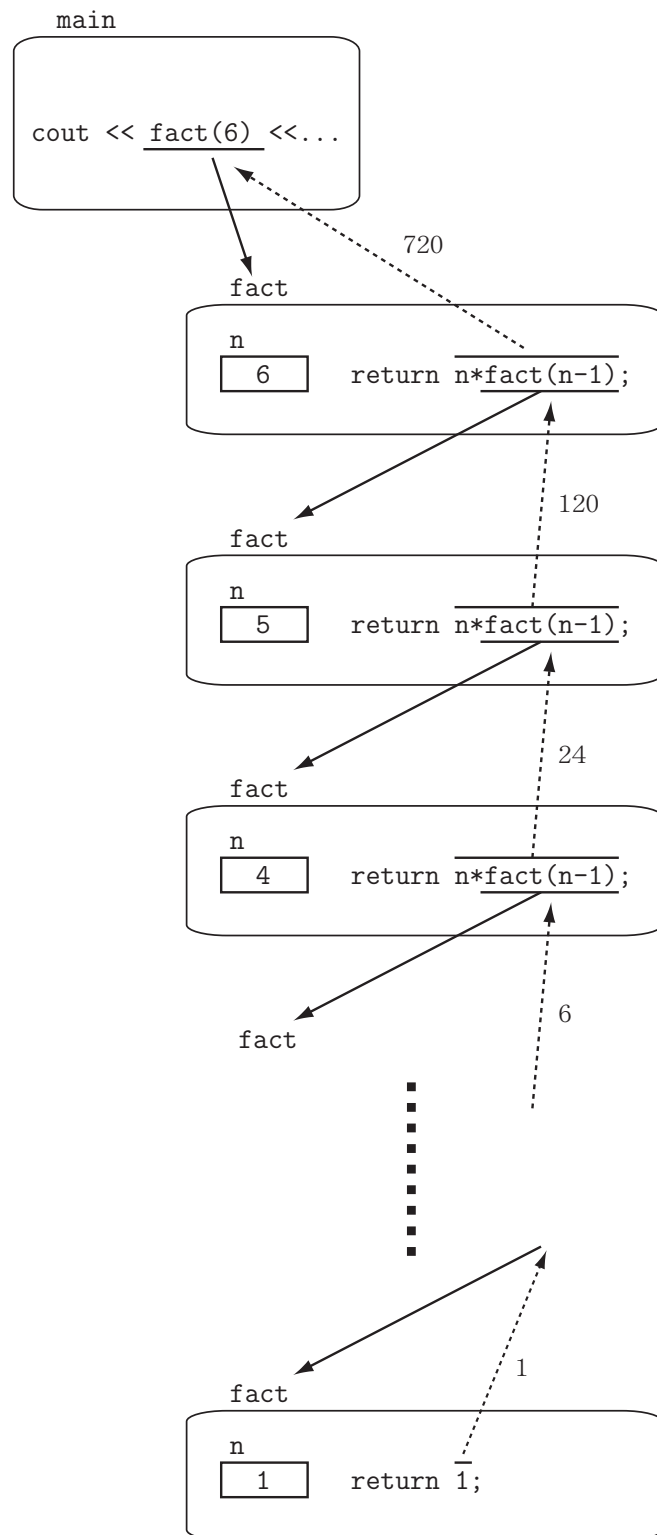


図 3.3: 再帰呼び出しの図式による理解

- 1     3. 関数 `fact()` の実行準備ができたところで関数本体の実行が始まります。この場合、`n` の値  
2       は 1 ではありませんから `if(n == 1)` は成り立たず、`else` 部の式 `return n*fact(n-1)` が  
3       評価されることになります。結果、その式の中の関数呼び出し `fact(n-1)` が次に起きます  
4       (その関数呼び出しを終えないことには式全体の値を評価できません)。このとき、`n` の値は  
5       6 ですから、`n-1` は 5 であり、実際には関数呼び出し `fact(5)` が起きます。
- 6     4. 再び関数 `fact()` の作業領域が新たに確保されます。ただし先ほど作った作業領域とは別の  
7       領域が確保されることに注意してください。その作業領域内には局所変数 `n` を保持する変数  
8       領域が割り当てられます。そしてそこには関数呼び出し `fact(n-1)` によって渡された 5 が  
9       コピーされます。この変数 `n` は先ほどの作業領域の `n` とは別ものであることに注意してくだ  
10      さい。名前は同じ `n` ですが、作業領域が異なればそれば別の変数であり、ひとつの作業領域  
11      から別の作業領域の変数は全く見えないことになっています。さて、現在の作業領域中の `n`  
12      の値は 1 ではありませんから `if(n == 1)` は成り立たず、再度、`return n*fact(n-1)` が実  
13      行されることとなり、関数呼び出し `fact(n-1)` が起きます。実際には `fact(4)` が起きます。
- 14    5. 同様のことを繰り返し、`fact()` の作業領域が次々と確保されていきます。そして、最終的  
15      には `n` の値が 1 であるような作業領域が作成され、その作業領域では値 1 が呼び出し側へ  
16      返されます (`return` されます)。なお、値を返した後の作業領域はもはや必要ないため、直  
17      後に消滅します (作業領域をシステムへ返還します)。
- 18    6. 呼び出し側へ返された値には `n` の値が乗じられ、`n*fact(n-1)` の値が計算され、それがさら  
19      に上位の呼び出し側関数へ返されます。最終的に `main` 関数へ `fact(6)` の値が返されます。

### 20 3.1.3 プログラムの特徴

21 図 3.1 と図 3.2 のプログラムは以下の点で対照的な特徴を持ちます。

22 **可読性** 図 3.1 では変数 `result` の値が `for` 文の実行と共にどのように変化していくかを理解せね  
23       ばなりません。図 3.1 はとても小さなプログラムなので、それを理解することに苦痛に感じ  
24       ることはありません。しかし一般には、時々刻々と変化していく事象を理解することは容易  
25       なことではありません。それに対して、図 3.2 は定義式をそのままプログラム化しているた  
26       め、プログラムの意味がストレートに読み取れます。プログラムの読み易さのことを可読性  
27       (readability) と言いますが、図 3.2 は図 3.1 よりも可読性に優れていると考えられます。し  
28       かし実際には図 3.1 の方が読みやすいと感じる人もいるでしょう。それは再帰を用いた記述  
29       法、思考法に慣れていないためです。

30 **実行速度** 図 3.1 は乗算と代入の繰り返しです。乗算と代入はコンピュータ (またはプロセッサ) の  
31       基本動作であるため、高速な実行が可能です。それに対して図 3.2 では関数呼び出しが頻繁

```
int fact(int n){
    return n*fact(n-1);
}
```

図 3.4: 誤った階乗の計算プログラム（無限再帰）

1 起こります。実は関数呼び出しは、多くのコンピュータにとって重たい処理（多くの実行  
2 時間を費やす処理）です。なぜならば、関数呼び出し前にはその関数内で使用する作業領域  
3 を確保し、呼び出し終了時に作業領域を解放するなどの本来の計算以外の処理が付随するか  
4 らです。そのため、呼び出し 1 回あたりの実行時間が長くなり、高速な実行ができません。  
5 結論を言えば、図 3.1 は図 3.2 よりも高速に実行可能です。

6 プログラムは可読性が高く、かつ高速であることが望まれます。しかし、可読性と高速性はしばし  
7 ば相反する性質です。相反する場合にはどちらかを優先されることになりますが、どのような状  
8 況でどちらを優先するか、そのトレードオフの見極めが重要です。一般論として述べれば、コン  
9 ピュータは年々高速になっていきます<sup>4</sup>が、プログラムを読み書きする人間の頭脳の進歩は微々た  
10 るものですから、数十年という長期に渡って使用されるプログラムでは可読性を重視する方が好ま  
11 しいでしょう。

#### 12 3.1.4 無限再帰

13 図 3.4 のプログラムは、図 3.2 と似ていますが、どのような挙動を示すでしょうか。

14 考察 このプログラムを 3.1.1 節の main 関数から呼び出したときの実行の様子を予想しなさい。  
15 あるいはコンピュータ上で実行してみなさい。

16 再帰関数の実行では再帰が止まらなくなることがあります。これを無限再帰 (infinite recursion)  
17 などと呼びますが、一般的には、これはプログラムのバグです。無限再帰を起こさない正しい再帰  
18 プログラムには

- 19 ● 必ず条件分岐が存在し、
- 20 ● 条件が成り立つならば再帰呼び出しを行わず、さもなければ再帰呼び出しを行う
- 21 という構造をしています。

<sup>4</sup>ムーアの経験則によると 1.5 年で 2 倍、10 年で 100 倍になります。

```

int fib(int n){
    if(n == 1) return 1;
    else if(n == 2) return 1;
    else return fib(n-1) + fib(n-2);
}

```

図 3.5: フィボナッチ数列の計算プログラム

## 1 3.2 複雑な再帰関数

2 再帰はきわめて強力な表現手段であって、様々な処理が再帰を用いて表現できます<sup>5</sup>。ここでは  
3 前節よりもさらに複雑な再帰プログラムを紹介します。

### 4 3.2.1 フィボナッチ数列

フィボナッチ数列  $fib(n)$  は以下のように定義される数列です。この数列は定義が簡単であるにも関わらず数学的に非常に奥が深く、様々な計算アルゴリズムが存在することで有名です。

$$\begin{aligned}
 fib(1) &= 1, \\
 fib(2) &= 1, \\
 fib(n) &= fib(n-1) + fib(n-2), \quad (n \geq 3)
 \end{aligned}$$

5 これをCの再帰関数でそのまま書き下すと図3.5の通りです。これを以下の main 関数から呼んで  
6 実行します。

```

7 int main(){
8     for(int i = 1; i <= 20; i++){
9         cout << "fib(" << i << ") = " << fib(i) << endl;
10    }
11 }

```

12 そうすると、次のような出力を得ます。数列の値が急速に増大していることに注意してください。

```

13 fib(1) = 1
14 fib(2) = 1
15 fib(3) = 2
16 fib(4) = 3
17 ...
18 fib(19) = 4181
19 fib(20) = 6765

```

フィボナッチ数列  $fib(n)$  は、 $n$  が十分大きいときに、

$$fib(n) = \alpha \times fib(n-1)$$

<sup>5</sup>理論上は、世の中の全てのプログラムはループを用いることなく、再帰だけで記述可能です。

を満たす等比数列になります。定数  $\alpha$  の値を求めるには、漸化式  $fib(n) = fib(n-1) + fib(n-2)$  に上の式を代入し、

$$\alpha^2 = \alpha + 1$$

1 を得ます。この二次方程式の解は  $\alpha = (1 \pm \sqrt{5})/2$  ですが、正数  $(1 + \sqrt{5})/2 = 1.6180339...$  が求め  
2 る  $\alpha$  です<sup>6</sup>。つまり、 $fib(n)$  の値は  $n$  の増加とともに約 1.618... のスピードで指数的に増大してい  
3 きます。

4 ところで、図 3.5 のプログラムの計算時間は、値がそうであるのと同様に、 $n$  の増加と共に指数  
5 的に増加していき、増加率はやはり  $\alpha = (1 + \sqrt{5})/2 = 1.6180339...$  です。つまり図 3.5 のプログ  
6 ラムは  $O(\alpha^n)$  の時間計算量を持っています。

7 考察 フィボナッチ数列を  $O(n)$  で計算する方法が知られています。興味のある人は調べてみな  
8 さい。

### 9 3.2.2 アッカーマン関数

数学者アッカーマン (Ackermann) が考えた以下の関数の計算は、階乗やフィボナッチ数列の計  
算よりも、本質的に複雑である<sup>7</sup>ことが知られています。

$$A(0, n) = n + 1,$$

$$A(m, 0) = A(m - 1, 1),$$

$$A(m, n) = A(m - 1, A(m, n - 1)), \quad (m \geq 1, n \geq 1).$$

というのも、階乗やフィボナッチ数列の関数では関数の引数 (入力パラメータ) が単純に減少し  
ながら再帰を繰り返していました。しかしアッカーマン関数では、3 行目の式の右辺はアッカーマ  
ン関数の引数としてアッカーマン関数が用いられており、引数の挙動が極めて複雑です。しかも、  
アッカーマン関数の関数値は、以下に示すように極めて大きな数です。

$$A(0, 0) = 1, \quad A(1, 0) = 2, \quad A(2, 0) = 3, \quad A(3, 0) = 5, \quad A(4, 0) = 13,$$

$$A(5, 0) = 65536 - 3, \quad A(6, 0) = 2^{65536} - 3, \quad A(7, 0) = 2^{2^{65536}} - 3, \dots$$

10 考察 この関数を C/C++ でプログラミングしなさい。

## 11 3.3 補足

12 この授業とは直接関係しませんが、関数呼び出しについて基本的な事項を確認しておきます。

<sup>6</sup>この数は黄金分割比 (golden ratio) と呼ばれ、辺の長さがこの比率を持つ長方形は最も美しい、飽きのこない長方形とされています。

<sup>7</sup>帰納関数論では、階乗やフィボナッチ数列の計算は原始帰納的関数 (primitive recursive function) というクラスに属することが知られています。それに対してアッカーマン関数は原始帰納的でなく、帰納関数というひとつ上のクラスに属することが知られています。

```

void f(int a){                                // 値を受け取る
    cout << "    &a = " << &a << endl; // a のアドレスの表示
    cout << "    a = " << a << endl; // 加算実行前の a の値の表示
    a += 3;                                    // 加算
    cout << "    a = " << a << endl; // 加算実行後の a の値の表示
}

int main(){
    int x = 2;
    cout << "&x = " << &x << endl; // x のアドレスの表示
    cout << " x = " << x << endl; // f(x) 実行前の x の値の表示
    f(x);                          // 関数呼び出し
    cout << " x = " << x << endl; // f(x) 実行後の x の値の表示
}

```

図 3.6: 値渡し の例

### 3.3.1 関数呼び出しにおいて引数を渡す仕組み

3.1.2 節で値渡しについて触れました。関数へ引数を渡す機構には他にも様々な方法があります。ここでは C/C++ で利用できるポインタ渡し (call by pointer)、参照渡し (call by reference) を値渡しと比較して解説します。

#### 値渡し

図 3.6 は、値渡しを説明するためのサンプルプログラムです。このプログラムは以下の手順が実行されます。

1. main 関数冒頭で変数 `x` を宣言し、2 で初期化します。
2. `x` のメモリアドレスを表示します。ここに '`&`' は変数のアドレスを求める演算子で、文字通り、アドレス演算子 (address operator, address-of operator) と呼ばれます。
3. `x` の値を出力します。
4. `x` を引数値として関数 `f()` を呼び出します。実際に関数に渡される値のことを実引数 (actual argument, actual parameter) と呼びます。この場合、`x` の保持する値 2 が実引数の値です。
5. 関数 `f()` の本体では以下の手順が実行されます。
  - (a) 関数 `f()` は局所変数 `a` の値として実引数の値を受け取ります。実引数を受け取る変数を特に仮引数 (formal argument, formal parameter) と呼びます。実引数、仮引数の違いをしっかりと理解しておきましょう。



```

void f(int *a){                                // アドレス値を受け取る
    cout << "    a = " << a << endl; // アドレス値の表示
    cout << "    *a = " << *a << endl; // アドレスの指すデータ実体の値の表示
    *a += 3;                                // アドレスの指すデータ実体への加算
    cout << "    *a = " << *a << endl; // アドレスの指すデータ実体の値の表示
}

int main(){
    int x = 2;
    cout << "&x = " << &x << endl;
    cout << " x = " << x << endl;
    f(&x);                                // アドレス値を渡す
    cout << " x = " << x << endl;
}

```

図 3.7: ポインタ渡し の例

1 (b) a のメモリアドレスを表示します。

2 (c) a の値を出力します。

3 (d) a に 3 を加算します。

4 (e) a の値を出力します。

5 6. 関数 f() の実行から戻った後、x の値を出力します。

6 実際にこのプログラムを講義担当者の PC で実行した場合の出力結果は以下の通りです。

```

7 &x = 0x7fff5c64381c
8 x = 2
9 &a = 0x7fff5c64379c
10 a = 2
11 a = 5
12 x = 2

```

13 x と a のメモリアドレスが異なることに注意してください。この二つの変数はメモリ内で別の実  
 14 体です。実体が異なるので、関数の内部で a の値を更新しても、関数呼び出しの前後で x の値は  
 15 不変です。

16 なお、メモリアドレス値はプログラムの実行環境毎に異なるので、その値自体に大きな意味はあ  
 17 りません。

## 18 ポインタ渡し

19 呼び出された関数（ここでは f()）の中で、呼び出した関数（ここでは main 関数）の変数の値  
 20 を変えたい場合、C/C++ではしばしばポインタ渡しが利用されます。そのサンプルプログラムが  
 21 図 3.7 です。このプログラムは図 3.6 のプログラムとは以下の点で異なります。

- 1 1. x のアドレス を実引数値として関数 `f()` を呼び出します。
- 2 2. 関数 `f()` の中では以下の手順が実行されます。
- 3 (a) 関数 `f()` では局所変数 `a` はポインタとして宣言されています。`a` には実引数として渡
- 4 された `x` のアドレスがコピーされます<sup>8</sup>。
- 5 (b) `a` の値を表示すると、それが `x` のアドレスに等しいことが確認できます。
- 6 (c) `*a` は、ポインタ `a` が指すメモリアドレスのデータ実体を表します。ここに `*` はポイン
- 7 タの指す先のデータや変数を取り出す演算子で、間接参照演算子 (dereference operator)
- 8 と呼ばれます。ポインタに `*` を付けることでデータ実体を求めることができますから、
- 9 `*` を付けることを参照外し (dereferencing) とも言います。さて、`*a` の値を出力する
- 10 ことは、`x` の値を出力することと同じです。
- 11 (d) `*a += 3;` によって、ポインタ `a` が指すメモリアドレスのデータ実体、すなわち `x` に
- 12 3 を加算します。
- 13 3. 関数 `f()` の実行から戻った後、`x` の値を出力すると、`x` の値が 3 増えていることに気づき
- 14 ます。

15 実際にこのプログラムを講義担当者の PC で実行した場合の出力結果は以下の通りです。

```

16 &x = 0x7fff55e2581c
17 x = 2
18 a = 0x7fff55e2581c
19 *a = 2
20 *a = 5
21 x = 5
```

## 22 参照渡し

23 図 3.7 のように、呼び出された関数の中で呼び出した関数の状態を更新できる機能はとても有用

24 です。しかし、ポインタを用いるため、プログラムの書き方が通常の変数を用いる場合とは少し異

25 なっており、可読性が落ちてしまいます。そこで、通常書き方でポインタ渡しと同等の機能を実

26 現する方法として、C++では参照渡しの機構が導入されています。

27 C++の参照渡しは、関数の仮引数のデータ型に `&` を付けるだけで実現でき、非常に簡単に値渡

28 しを参照渡しに変更できます。図 3.8 がそのプログラムです。このプログラムを講義担当者の PC

29 で実行した場合の出力結果は以下の通りです。

<sup>8</sup>アドレス値がコピーされる点に着目すれば、実はポインタ渡しは値渡しの特異形と見なすこともできます。しかしそれはあくまでも形式上の話であって、通常これを値渡しとは言いません。

```

void f(int & a){
    cout << "    &a = " << &a << endl;
    cout << "    a = " << a << endl;
    a += 3;
    cout << "    a = " << a << endl;
}

int main(){
    int x = 2;
    cout << "&x = " << &x << endl;
    cout << " x = " << x << endl;
    f(x);
    cout << " x = " << x << endl;
}

```

図 3.8: 参照渡し の例 (1 行目のみ図 3.6 と異なる)

```

1  &x = 0x7fff53b5981c
2  x = 2
3  &a = 0x7fff53b5981c
4  a = 2
5  a = 5
6  x = 5

```

7 図 3.8 のプログラムは 1 カ所を除いて図 3.6 と同じであることを確認してください。

8 なお、参照渡し の概念自体は C++ のオリジナルではありません。実は世界初の高水準プログラ  
9 ム言語 FORTRAN では引数は全て参照渡し でした。しかし、参照渡し を用いる図 3.8 のようなプ  
10 ログラムの場合、変数値の変更が安易に可能であるため、最近のプログラミング言語では値渡し が  
11 標準になっています。参照渡し は便利ですが、十分な注意が必要です。

## 1 第4章 Lispによるプログラミング

2 この章では、Lisp を用いた様々なプログラミングについて解説していきます。ただしこのテキ  
3 ストは言語マニュアルではありませんから、Lisp の使い方全てを解説する訳ではありません。関  
4 数型プログラミングのエッセンスを修得するために必要な事項のみを取り上げています<sup>1</sup>。

### 5 4.1 コンパイラとインタプリタ

6 コンパイラとインタプリタについては既に2章で説明しましたが、重要な事項ですから2章とは  
7 別の角度から再度概説します。

#### 8 4.1.1 コンパイラ

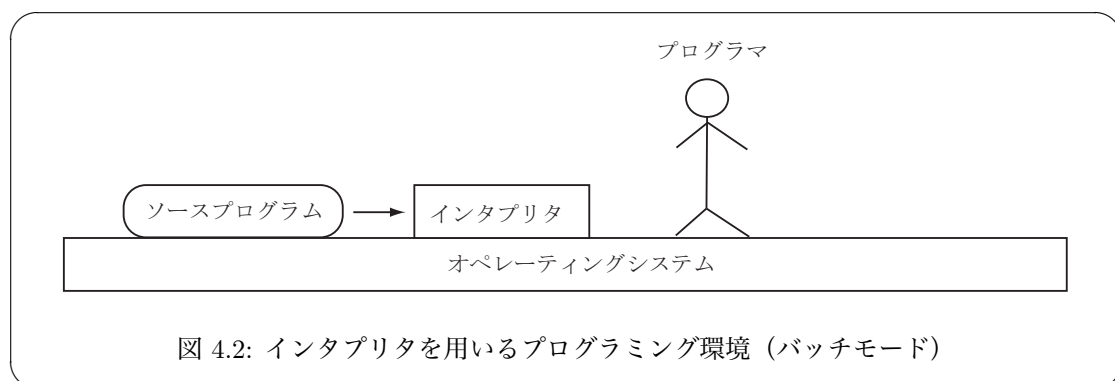
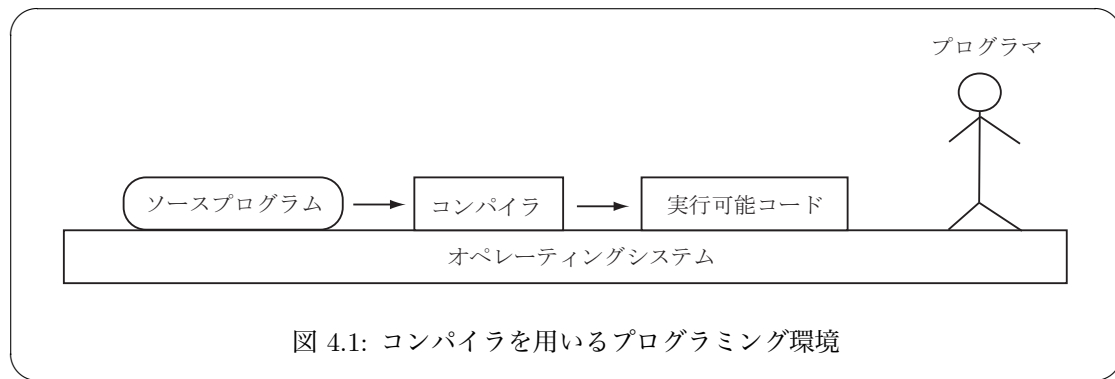
9 コンパイラはソースプログラムを実行可能コードへ変換するソフトウェアツールのことです。図  
10 4.1 にコンパイラを用いてプログラムの開発/実行を行う際の OS や関連ファイルを図式化しまし  
11 た。ここにソースプログラムは OS 管理下でエディタなどによって作成されるテキストファイルで  
12 す。コンパイラは OS の管理下で実行可能なアプリケーションプログラムの一種です。コンパイラ  
13 を OS やシステムプログラムの一部と見なす場合もあります。しかし、コンパイラは本来単なる翻  
14 訳ツールであって OS の特権モードを必要としないため、アプリケーションプログラムと見なす方  
15 が妥当です。さて、コンパイラはソースプログラムを入力としてそれを実行可能コードへ変換（翻  
16 訳）します。実行可能コードもアプリケーション・プログラムであり、OS 管理下で実行可能です。  
17 コンパイラを用いてプログラム開発/実行を行うプログラマの視点は OS のコンソール・モード  
18 の上（あるいは shell プログラムの上）にあります。コンパイラは、次に述べるインタプリタとは  
19 異なり、対話的なツールではありませんから、プログラマの位置がコンパイラの上や中に入ること  
20 はありません。

#### 21 4.1.2 インタプリタ

22 ここではインタプリタの動作を3種類に分けて解説します。

---

<sup>1</sup>この資料で取り上げない事柄はインターネットで「Lisp & ...」と検索すると種々ヒットします。自ら調べてください。

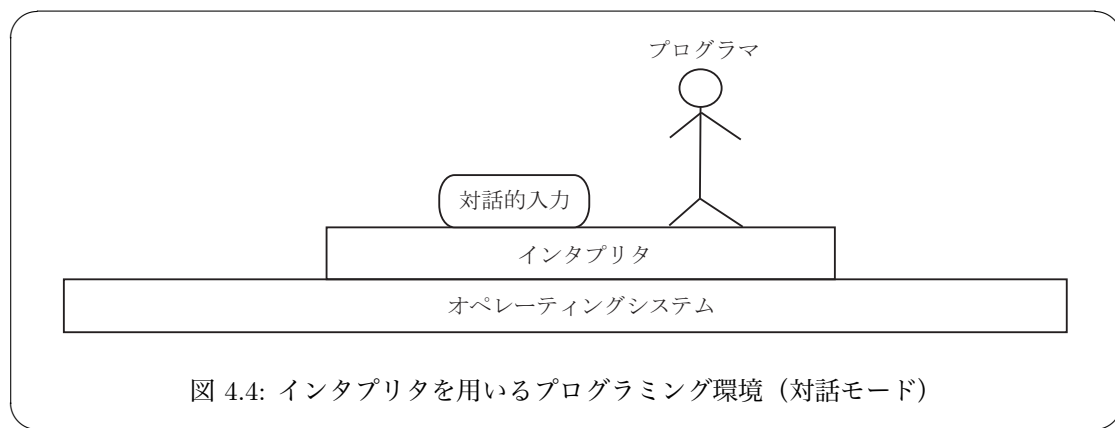
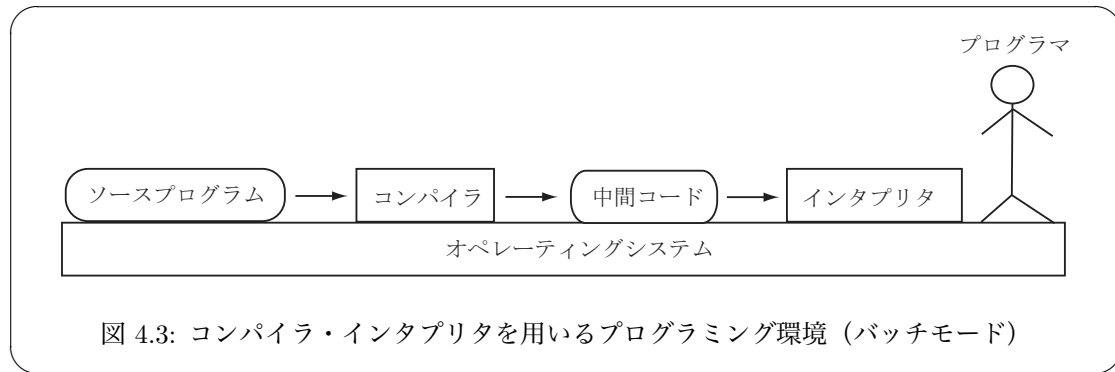


#### 1 バッチモード

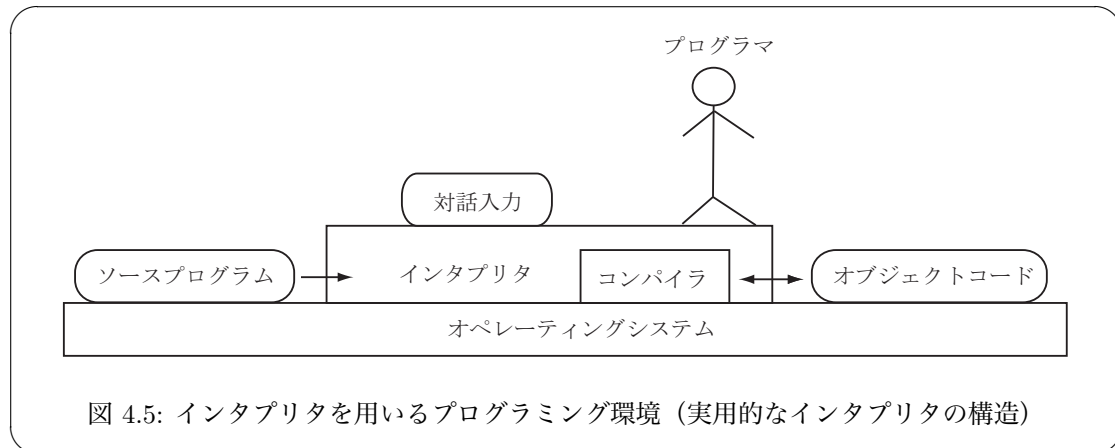
2 インタプリタはソースプログラムを解釈し、実行するソフトウェアツールです。ここで言う「解  
3 釈」とはコンパイラで言う「翻訳」とは異なり、プログラム中の文（やプログラム片）を実行直前  
4 に解析し、その意味を求めることを言います。たとえば、あるループについてループ本体の実行を  
5 1000 回繰り返すならば、解釈を 1000 回行うことになります。コンパイラの場合、そのループを 1  
6 回だけ翻訳しますが、インタプリタの場合、実行回数分だけ解釈を繰り返すため、インタプリタの  
7 プログラム全体の実行速度はコンパイラに比べて非常に遅くなります。

8 図 4.1 に、インタプリタを用いたプログラムの開発/実行の様子を図示しました。ソースプログ  
9 ラムは、コンパイラの場合と同様に、エディタ等を用いて別途作っておきます。そして、そのソー  
10 スプログラムをインタプリタに入力すれば、プログラムの実行が即座に開始されます。この実行の  
11 様子は、実行可能コードが作成されないこと、実行速度が遅いことを除けば、コンパイラの実行の  
12 様子と大差ありません。この実行方式を、次に述べる対話モードと対比させるため、特にバッチ  
13 モード (batch mode — 一括モードとも訳します) と呼びましょう。この場合のプログラマの視点  
14 は図 4.1 同様に OS のコンソール・モードの上にあります。

15 蛇足ですが、しばしばコンパイラとインタプリタを併用し、両者の長所を利用することが行われ  
16 ます (2 章参照)。Java の実行方法がそうです。これはソースプログラムを中間コードに翻訳し、



- 1 中間コードを解釈/実行する方法です (図 4.3 参照)。
- 2 対話モード
- 3 インタプリタは実行直前にプログラム片を解釈するため、コンソールからプログラム片を数行だ
- 4 け入力し、その入力が終了すると同時にそのプログラム片を実行するというプログラムの開発/実
- 5 行が可能です。これをインタプリタの対話モード (interactive mode) と呼びます。このモードで
- 6 は、プログラム入力直後にプログラムのバグが発見でき、かつ計算結果を得ることができるため、
- 7 小さなプログラムを効率的に開発することに向いています。
- 8 図 4.4 はその実行の様子を図示したものです。図 4.1 や図 4.2 との違いは、プログラム開発/実行
- 9 中のプログラマの視点がインタプリタの上にあることです。プログラマは OS のコンソールモード
- 10 で Shell プログラムと対話するのではなく、インタプリタと対話しながらプログラムを開発/実行
- 11 していきます。
- 12 この章では対話モードで Lisp インタプリタを用いていきます。



## 1 実用モード

2 対話モードは小さなプログラムを素早く作り、実行するには便利です。しかし数十行を超える  
 3 プログラムはバッチモードの場合のようにテキストファイル形式で編集/保存できると便利です。そ  
 4 のため、実用的なプログラム開発環境はバッチモードと対話モードを併用するような構成になっ  
 5 ています。またインタプリタの中にはコンパイラを内蔵するものもあります。これによって、デバッ  
 6 グの終わったプログラムをコンパイルし、オブジェクトコードに翻訳しておき、高速実行すること  
 7 も可能です。実際、次節以降に詳しく述べる Lisp インタプリタ gcl はそのような機構を持ってい  
 8 ます。

9 図 4.5 はそれを図式化したものです。プログラマはインタプリタの上からコンパイラを起動し、  
 10 ソースプログラムをコンパイルし、オブジェクトコードを作ることができます。作成されたオブ  
 11 ジェクトコードは必要に応じてインタプリタから呼び出され、高速に実行されます。

## 12 4.2 Lisp インタプリタのダウンロードと起動

13 この授業の演習/レポートで使用する処理系は Windows 用 gcl (Gnu Common Lisp) です。ホー  
 14 ムページは

15 <http://www.gnu.org/software/gcl/>

16 です。このページからも処理系をダウンロードできますが、gcl の処理系をダウンロードするには  
 17 以下のページが便利です。

18 <http://www.cs.utexas.edu/users/novak/gclwin.html>

19 このページの 2 段目 “For convenience, ... download [here](#) (3.4MB) .” の [here](#) をクリックすると、  
 20 gcl インタプリタ (gcl.exe) がダウンロードされます。この処理系はサイズがたった 3.4MB しか  
 21 ないため、ノートパソコンのハードディスクを圧迫することはないはずです。

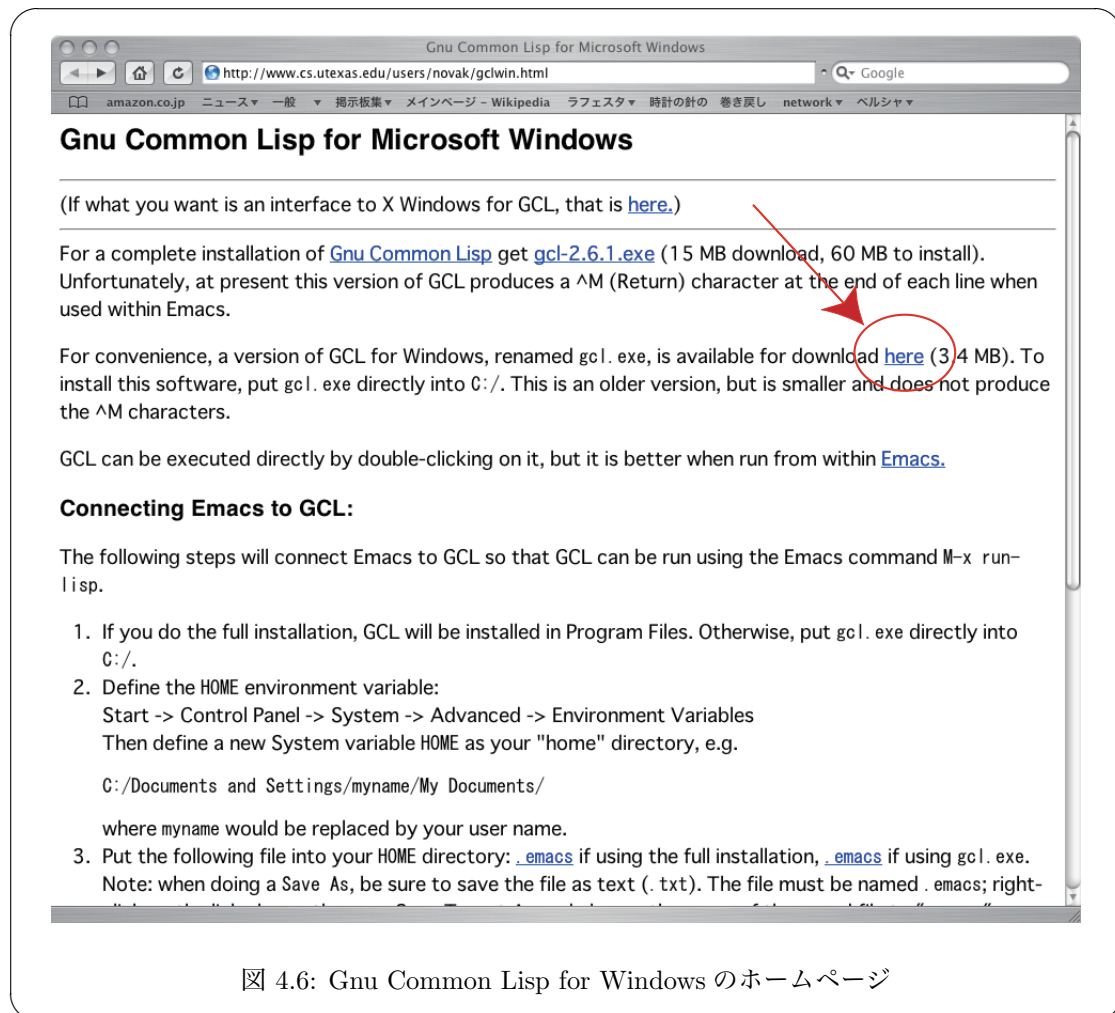


図 4.6: Gnu Common Lisp for Windows のホームページ

1     ダウンロードされた `gcl.exe` は、インストール・プログラムのようなものではなく、インタプリ  
2     タそのものです。そのため、これをダブルクリックすると、直ちに Lisp インタプリタのウイン  
3     ドウが開き、ウインドウに以下のように表示され、入力待ち状態になります。

```

4     GCL (GNU Common Lisp)  Version(2.5.0)  Thu Jan 14 14:23:46     EAST 2003
5     Licensed under GNU Public Library License
6     Contains Enhancements by W. Schelter
7
8     >

```

9     このプロンプト `>` に続いて、様々な Lisp の入力を行い、インタプリタと対話的に処理を進めてい  
10    きます。

11    インタプリタの実行を終了するときにはプロンプト `>` に続いて、

```

12    > (bye)

```

13    と入力します。 `bye` の代わりに、`by`、`quit` も使用できます。



$$\begin{aligned}
 (+ a_1 a_2 \dots a_n) &\equiv (\dots((a_1 + a_2) + a_3) \dots + a_n) \\
 (- a_1 a_2 \dots a_n) &\equiv (\dots((a_1 - a_2) - a_3) \dots - a_n) \\
 (* a_1 a_2 \dots a_n) &\equiv (\dots((a_1 \times a_2) \times a_3) \dots \times a_n) \\
 (/ a_1 a_2 \dots a_n) &\equiv (\dots((a_1 \div a_2) \div a_3) \dots \div a_n)
 \end{aligned}$$

図 4.7: 2 個以上の引数が与えられた場合の四則演算の評価

## 4.3 数式と変数

### 4.3.1 数式の評価

まず、Lisp インタプリタを電卓のように使ってみましょう。

以下は、 $1+2$  の足し算を入力した例です。プロンプトの後に、`"(+ 1 2)` と改行を入力します。

そうすると、実行結果が表示され、さらにプロンプトが表示され、次の入力待ち状態になります。

```
> (+ 1 2)
3
>
```

上は、 $+$  という加算を表す関数に引数  $1$  と  $2$  を与えた式をインタプリタが評価 (evaluate)<sup>2</sup> し、結果  $3$  を表示したものです。Lisp の式の表現方法では、 $a + b$  のような、四則演算子は二つの引数の間に演算記号を書く中値形式 (infix notation) は用いず、 $(+ a b)$  のような、引数の前に演算記号を書く前置形式 (prefix notation) を採用しています。これは単に構文の問題に過ぎませんが、他の多くのプログラミング言語とは異なる点です。このルールに則って、複数の演算を持つ式  $1+2*3$  は以下のように入力します。

```
> (+ 1 (* 2 3))
7
>
```

加減乗除記号  $+$ ,  $-$ ,  $*$ ,  $/$  全ての演算について、このような表記法なのです。Lisp では、この  $+$ ,  $-$ ,  $*$ ,  $/$  はさらに拡張されており、2 個以上の任意個数の引数を取ることができ、それぞれ図 4.7 のような意味になります。

ひとつの数値は、それ自身がひとつの式でもあります。よって、以下のように、数値を入力するとそれ自身が表示されます。

<sup>2</sup>関数型プログラミングでは、式の値を求めることを特に「評価」と呼びます。もとの英単語は動詞 evaluate ですので、「評価する」のことをしばしば「えばる」ということがあります。

```

1    > 777
2    777
3    >

```

4 もちろん負数や浮動小数点数も利用可能です。たとえば、

```

5    > -1e2
6    -99.999999999999986
7    >

```

8 となります。この場合、10 の二乗の負数は-100 ですが、まるめ誤差のため、-99.99... と評価されて  
 9 います（なお、システムのバージョンアップによって評価方法が変更されている場合があります）。

10 数値演算には、三角関数 `sin`, `cos`, `tan`、逆三角関数 `asin`, `acos`, `atan`、自然対数関数 `exp`、  
 11 べき乗 `expt`、対数 `log`、平方根 `sqrt`、絶対値 `abs` などが使用可能です。これらは、たとえば以下  
 12 のように用います。

```

13    > (sqrt 4.0)
14    2.0
15    >

```

16 その他のシステムに標準装備された関数については各自で調べてください。

### 17 4.3.2 変数と代入

18 変数 (variable) はデータを格納するためのメモリ領域を指定します。当面は C/C++ の変数と  
 19 同一視して構いません。変数名は、英文字、数字、記号から構成されます。ただし

- 20 • **Lisp** では、英大文字と英小文字を区別しない

21 点に注意が必要です。ちなみに、世の中の Lisp の解説書では、

- 22 • インタプリタへの入力には英小文字で行い、インタプリタからの出力は英大文字で得る

23 というスタイルのものが多いため、この資料でもそのスタイルで解説します。

24 代入 (assignment) は変数にデータを格納する処理です。C/C++ における代入は、

```
25     var = value ;
```

26 という形式で記述されますが、Lisp における代入には以下の形式を用います。

```
(setq var value)
```

27  
 28 Lisp の代入は演算同様に前置形式です。`setq` は代入を行う関数と考えてください（関連事項を後  
 29 に述べます）。ただし変数 `var` は全ての関数の外に宣言された大域変数 (global variable) と同じ  
 30 扱いになります。

31 たとえば以下は、`x` という名前の変数に整数値 123 を代入し、その `x` の値をそのまま評価/出力  
 32 した例です。

```

1    > (setq x 123)
2    123
3    > x
4    123
5    >

```

6 同じ変数への代入は繰り返して行うことができます。C/C++と同様に、それ以前に代入した値は  
7 上書きされます。よって、

```

8    > (setq x 123)
9    123
10   > (setq x 777)
11   777
12   > x
13   777
14   >

```

15 変数に代入された値は、数式中で参照可能です。よって

```

16   > (setq x 123)
17   123
18   > (+ x 100)
19   223
20   >

```

21 このように見てくると、形式は異なるものの、Lisp の代入が C/C++ のそれと同じように実行  
22 できることに気づくかもしれません。実際、C/C++ とほとんど同じ感覚で使用可能です。しか  
23 し、代入を繰り返し、変数値を更新しながら処理を進めていく手続き型のプログラミング・スタイ  
24 ルを Lisp に持ち込むことは（可能ですが）この講義では推奨しません。

## 25 型なし変数

26 上に「C/C++ とほとんど同じ感覚で...」と述べましたが、Lisp の変数と C/C++ の変数には大  
27 きな違いがあります。実は Lisp の変数は特定のデータ型の値を保持するのではなく、様々なデー  
28 タ型の値を自由に保持できます。このことを簡単に「型なし (typeless)」などと言いますが、実際  
29 にはデータ型がないのではなく、変数が保持できる値のデータ型に制限が無い (type-free) という方  
30 が正確です。

31 上の例では、変数 `x` を整数型であるかのように扱いましたが、それは `x` に代入したデータが整  
32 数型の値であったためです。`x` に浮動小数点型データを代入すれば、`x` は浮動小数点型データとし  
33 て参照されます。以下がその例です。数式  $(+ x 2)$  を 2 回評価していますが、それを評価すると  
34 きに `x` に代入されているデータが整数型ならば  $(+ x 2)$  の結果は整数値になります。`x` に代入さ  
35 れているデータが浮動小数点型ならば  $(+ x 2)$  の結果は浮動小数点数値になります（このとき、  
36 定数値 2 は言語処理系によって浮動小数点数値 2.0 に自動変換されて浮動小数点数型の加算が  
37 実行されます）。

```

1    > (setq x 1)
2    1
3    > (+ x 2)
4    3
5    > (setq x 1.0)
6    1.0
7    > (+ x 2)
8    3.0
9    >

```

10 ひとつの変数に様々なデータ型の値を代入できることはプログラミングの柔軟性に貢献します<sup>3</sup>  
 11 が、反面、データ型が厳密に定まっていないため、バグが入り込みやすいという欠点にもなりま  
 12 す<sup>4</sup>。

### 13 特殊形式

14 先ほど、「`setq` は代入を執り行う関数」と説明しました。しかし、より正確には、`setq` は Lisp  
 15 の特殊形式 (special form) と呼ばれており、関数とは異なるものです。その違いを簡単に説明し  
 16 ておきます。

17 `+` は Lisp の関数です。今、変数 `x` に浮動小数点数値 `1.0` が格納されているとしましょう。この  
 18 とき、インタプリタはたとえば以下のような評価を行います。

```

19    > (+ x (- 3 1))
20    3.0
21    >

```

22 関数 `+` の引数 `x` と `(- 3 1)` は、それが実行される前にそれぞれ浮動小数点数値 `1.0` と整数値 `2`  
 23 という値として求められ、その後に `(+ 1.0 2)` と同等の関数呼び出しが実行されます。関数呼び  
 24 出しの大原則は関数の全ての引数は、その関数の実行の前に評価されることです。これは Lisp に  
 25 限らず、C/C++においても成り立つ原則です。

26 特殊形式が関数と区別されるのは、その引数評価が関数と異なるためです。以下の式の実行を検  
 27 討しましょう。

```

28    > (setq x (- 3 1))
29    2
30    >

```

31 このとき、`setq` では、第二引数 `(- 3 1)` は事前に評価されますが、第一引数 `x` は評価されませ  
 32 ん。もしこれが評価されるならば、インタプリタは `(setq 1.0 2)` を実行することとなります。し  
 33 かし、これでは `x` という変数の情報は消えており、`x` への値の代入は不可能です。`setq` が代入操  
 34 作を行うためには、第一引数は評価せず、変数のまま保持する必要があります。

<sup>3</sup>簡単に言えば、プログラマはデータ型を気にすることなく、どんどんプログラミングできる訳です。

<sup>4</sup>バグを防ぐ意味で、変数にデータ型を定めることは重要です。C/C++などは厳格な型付き言語であり、最近の多くの言語は型について厳格な扱いをするものがほとんどです。

1    このように、関数とはやや異なる評価方法で動作するものを特殊形式と呼んでいます。後に 4.4  
2    節に登場する `if`、`cond` など特殊形式の例です。特殊形式の動作方法は個々の特殊形式ごとに異  
3    なりますが、それを憶えておく必要はありません。特殊形式の意味さえ正しく理解すれば、動作は  
4    自ずと明らかであり、個々の動作の違いに戸惑うことはほとんどありません。

### 5    4.3.3   その他のデータ型

6    `gcl` では整数型、浮動小数点型以外に、分数型、複素数型、文字型、文字列型、シンボル型、リ  
7    スト型など様々なデータ型が使用可能です。シンボル型、リスト型は Lisp の特徴的なデータ構造  
8    です。後で後に詳細に述べますが、分数型、複素数型、文字型、文字列型についてはこの授業では省  
9    略します。各自で調べてください。

10    以上、Lisp のデータ型と変数について基本的な事柄を知ることができました。次節からは、よ  
11    り複雑なプログラムを作るための仕組みについて学びます。

## 12   4.4   条件判定と分岐

13    条件判定を行い、その結果に応じて処理内容を選択することは複雑なプログラム実行に不可欠で  
14    す。考え方は C/C++ と大差ありませんが、記述法が異なります。

### 15   4.4.1   条件判定

#### 16   真偽値

17    Lisp では、`NIL` が偽値 (false value) を表す定数です。`NIL` 以外は全て真値 (true value) を表  
18    しますが、真値を代表する定数として `T`<sup>5</sup> がシステム定義されています。真値と偽値に関するこの  
19    辺の状況は、C 言語のそれ — C 言語では偽値は `0`、真値は `0` でない値 — とよく似ています。た  
20    だし、Lisp では `NIL` 以外は真値を表しますから、`0` は Lisp では真値を表しますから注意してくだ  
21    さい

#### 22   比較演算

23    二つの数値  $m$  と  $n$  の大小等号判定を行う Lisp の式は以下の通りです。

<sup>5</sup>4.3.2 節で述べたように、Lisp では英大文字と英小文字を区別しませんので、`t` も `T` を表します。

1	<code>(= m n)</code>	$m$ と $n$ は等しい。==ではないことに注意。
	<code>(/= m n)</code>	$m$ と $n$ は等しくない。
	<code>(&lt; m n)</code>	$m$ は $n$ よりも小さい。
	<code>(&lt;= m n)</code>	$m$ は $n$ よりも小さい、または $m$ と $n$ は等しい。
	<code>(&gt; m n)</code>	$m$ は $n$ よりも大きい。
	<code>(&gt;= m n)</code>	$m$ は $n$ よりも大きい、または $m$ と $n$ は等しい。

2 たとえば以下の通りです。

```
3 > (= 10 10)
4 T
5 > (< 5 2)
6 NIL
7 >
```

8 真偽値には以下の論理演算が使用可能です<sup>6</sup>。

9	<code>(not t)</code>	$t$ の否定を求める。
	<code>(and <math>t_1</math> <math>t_2</math> ... <math>t_n</math>)</code>	$n$ 個の引数 $t_1, \dots, t_n$ の論理積を求める。
	<code>(or <math>t_1</math> <math>t_2</math> ... <math>t_n</math>)</code>	$n$ 個の引数 $t_1, \dots, t_n$ の論理和を求める。

10 たとえば以下の通りです。

```
11 > (not 10)
12 NIL
13 > (not (not 10))
14 T
15 > (and t nil)
16 NIL
17 > (or t nil)
18 T
19 >
```

20 ここに、1行目の 10 は NIL ではないので真値と見なされ、その否定である NIL が結果となります。

## 21 4.4.2 条件分岐

22 以下の if 式は、条件判定した結果に応じて数式を選択する式です。

23 `(if t  $e_1$   $e_2$ )`

24 ここに、 $t$  が真値ならば  $e_1$  を評価し、偽値ならば  $e_2$  を評価し、式全体の値とします。たとえば

```
25 > (if t 1 2)
26 1
27 > (if nil 1 2)
28 2
29 > (if (= 1 1) (+ 1 2) 4)
30 3
31 >
```

<sup>6</sup>論理積、論理和の演算では C/C++ 同様にショートカットを行う場合があります。よって論理演算の引数の中で副作用を持つような演算を行うときには注意が必要です。詳細は各自で調べてください。

1 if 式は入れ子にできます。たとえば

```
2 > (setq x 2)
3 2
4 > (if (< x 5) (if (> x 1) x 0) 0)
5 2
6 > (if (> x 5) x (if (> x 1) (+ x 10) (- x 10)))
7 12
8 >
```

9 ところで上の例題の最後の入力式を C 言語風に書くと以下と同等です。

```
10 if(x > 5) return x;
11 else{
12     if(x > 1) return x + 10;
13     else return x - 10;
14 }
```

15 上のように else 部の中に入れ子の if 節がある場合、これを else if 節を用いて以下のように書  
16 き直した方が分かりやすくなります。

```
17 if(x > 5) return x;
18 else if(x > 1) return x + 10;
19 else return x - 10;
```

20 これと同様の書き方が Lisp に用意されており、一般に、C 言語のプログラム片:

```
21 if( $t_1$ ) return  $e_1$ ;
22 else if( $t_2$ ) return  $e_2$ ;
23 ...
24 else if( $t_n$ ) return  $e_n$ ;
```

25 に対応する Lisp のプログラム片は、cond 式を用いて以下のように書くことができます<sup>7</sup>。

```
(cond ( $t_1$   $e_1$ )
      ( $t_2$   $e_2$ )
      ...
      ( $t_n$   $e_n$ ))
```

27 ここに条件式  $t_1$  を評価し、それが真値ならば式  $e_1$  を評価し、この式全体の結果とします。もしそ  
28 れが偽値ならば、次に条件式  $t_2$  を評価し、それが真値ならば式  $e_2$  を評価し、この式全体の結果  
29 とします。もしそれが偽値ならば ... と評価を上から順に繰り返し、条件式  $t_i$  ( $1 \leq i \leq n$ ) が真  
30 値となったならば、式  $e_i$  を評価し、この式全体の結果とします。全ての条件式が成り立たなかつ  
31 たならば、NIL を全体の結果とします。

32 これを応用して、C 言語のプログラム片（最後の行が else if ではなく、else であることに  
33 注意）:

<sup>7</sup>もちろん、改行を行わず、(cond ( $t_1$   $e_1$ ) ( $t_2$   $e_2$ ) ... ( $t_n$   $e_n$ )) と書いても問題ありません。

```

1   if( $t_1$ ) return  $e_1$ ;
2   else if( $t_2$ ) return  $e_2$ ;
3   ...
4   else return  $e_n$ ;

```

5 に対応する Lisp のプログラム片は以下のように書くことができます。

```

(cond ( $t_1$   $e_1$ )
      ( $t_2$   $e_2$ )
      ...
      ( $t$   $e_n$ ))

```

7 つまり、最後の条件式を真値  $t$  にすればよいのです。

8 さて、cond 式を使えば、先ほどの例題は以下のように書くことができます。

```

9   (cond ((> x 5) x)
10         ((> x 1) (+ x 10))
11         (t      (- x 10)))

```

12 if 式を並べるよりも読み易いのではないでしょうか。

## 13 4.5 関数定義

14 Lisp は多くのシステム標準関数を持っており、それを用いて様々な処理が可能です。Lisp を関  
 15 数電卓の代わりに使うことは 4.3.1 節で述べた方法で可能です。しかし、より高度な処理を行うに  
 16 はプログラマ自らが関数を定義する必要があります。そもそも関数型プログラミングでは関数を定  
 17 義することがプログラミングであって、その意味では前節までは関数型プログラミングの核となる  
 18 内容についてはまだ全く述べていません。

19 プログラマが関数を定義するには以下の defun 式を用います。

```

(defun f ( $v_1$  ...  $v_n$ )  $e$ )

```

21 ここに  $f$  が関数名、 $v_1 \dots v_n$  が関数の引数、 $e$  が関数の本体です。 $n$  個の引数  $v_1 \dots v_n$  は、カンマ  
 22 ‘,’ではなく、空白‘ ’で区切ることにご注意してください。

23 簡単な例題を示します。以下は、引数の整数値に 1 を足す関数の定義とその使用例です。

```

24   > (defun succ (x) (+ x 1))
25   SUCC
26   > (succ 1)
27   2
28   > (succ)
29   Error: SUCC requires more than 0 arguments...
30   > (succ 1 2)
31   Error: SUCC requires less than 2 arguments...
32   >

```



1 まず、関数定義では、`defun`<sup>8</sup>の後に、関数名、仮引数リスト、関数本体を以下のように順に記述  
2 します。

```
3 > (defun succ (x) (+ x 1))
4     関数名 仮引数リスト 関数本体
```

5 関数呼び出しは、関数名とそれに続く実引数<sup>9</sup>リストを列挙することで起動します。

```
6 > (succ (+ 1 2))
7     関数名 実引数リスト
```

8 実引数は式でも構いません。その場合、全ての実引数値が評価された後に、この関数呼び出しの評  
9 価が行われます。実引数の数が仮引数の数と異なる場合にはエラーになります。

10 引数が2個以上の関数では、引数を空白で区切って仮引数リストに列挙します。

```
11 > (defun add (x y) (+ x y))
12 ADD
13 > (add 1 2)
14 3
15 >
```

16 引数のない関数を定義するときには仮引数リストを`()`と記述します。たとえば

```
17 > (defun zero () 0)
18 ZERO
19 > (zero)
20 0
21 >
```

22 引数が0であっても、関数呼び出し時には関数名を`()`で囲まねばなりません。言い方を換えれば、  
23 括弧`()`を用いることで関数呼び出しを表しています。上の場合、括弧を用いず、

```
24 > zero
```

25 と入力すれば変数 `zero` の評価を意味します。しかし

```
26 > (zero)
```

27 と入力すれば関数呼び出しを意味するのです。

28 関数呼び出しでは再帰定義が可能です。再帰の考え方は、C/C++であろうとも Lisp であろうと  
29 も変わるものではありません。階乗を計算するプログラムは C/C++ では以下の通りでした。

```
30 int fact(int n){
31     if(n == 1) return 1;
32     else return n*fact(n-1);
33 }
```

<sup>8</sup>`defun` は `define function` から作られた単語です。

<sup>9</sup>関数定義部に用いられる引数変数のことを仮引数 (formal argument) と呼びます。それに対して、関数呼び出し部に用いられる引数式のことを実引数 (actual argument) と呼びます。単に「引数」と呼ぶことが多いですが、両者は明確に区別すべきです。

1 これを Lisp の関数定義に変換すると以下の通りです。

```
2 > (defun fact (n) (if (= n 1) 1 (* n (fact (- n 1)))))
3 FACT
4 > (fact 1)
5 1
6 > (fact 4)
7 24
8 >
```

9 ところで、上の定義は `fact` の実引数の数値が1以上であることを暗に仮定しています。もし実引  
10 数に負数が与えられたならば、Lisp インタプリタはスタックオーバーフロー<sup>10</sup>を起こし、強制終  
11 了します（以下、参考）。

```
12 > (fact -1)
13 Error: Frame Stack Overflow
14 ...
15 >
```

16 上の関数定義では条件分岐に `if` を用いましたが、4.4.2 節に述べた `cond` を用いて以下のように  
17 定義することもできます。

```
18 > (defun fact (n) (cond ((= n 1) 1) (t (* n (fact (- n 1)))))
```

19 あるいは、読みやすさを考慮して

```
20 > (defun fact (n) (cond ((= n 1) 1)
21                        (t (* n (fact (- n 1)))))
```

22 または

```
23 > (defun fact (n)
24     (cond ((= n 1) 1)
25     (t (* n (fact (- n 1)))))
```

26 とインデントを使うとより読み易いでしょう。上の定義では、4.4.2 節の `(cond (t1 e1) (t2 e2))`  
27 というパターンを用い、特に判定式  $t_2$  には真値 `t` を指定しています。

28 `cond` の便利さを実感するには、条件式が2個以上並ぶ例題が適しています。以下は C/C++ で  
29 書かれたフィボナッチ数列の計算プログラムでした。

```
30 int fib(int n){
31     if(n == 1) return 1;
32     else if(n == 2) return 1;
33     else return fib(n-1)+fib(n-2);
34 }
```

<sup>10</sup>スタックは、関数呼び出し管理データや局所変数を格納するメモリ領域です。通常、処理系によって管理され、プログラマがそれを意識する必要はありませんが、プログラム実行をより深く理解の上では一応の知識を持つておくことが望まれます。

1 これに対応する Lisp の関数定義は以下の通りです。

```
2 > (defun fib (n)
3     (cond ((= n 1) 1)
4           ((= n 2) 1)
5           (t      (+ (fib (- n 1)) (fib (- n 2))))))
```

考察 以下のアッカーマン関数を Lisp を用いて定義しなさい。

$$A(0, n) = n + 1,$$

$$A(m, 0) = A(m - 1, 1),$$

$$A(m, n) = A(m - 1, A(m, n - 1)), \quad (m \geq 1, n \geq 1).$$

## 6 4.6 プログラムのファイル入力

7 関数定義式が数十行を超える場合、作業の度毎にそれをキーボード入力することはとても面倒で  
8 す。しかし、Lisp にはテキスト・ファイルからインタプリタへの入力を読み込む機能

```
(load "ファイル名")
```

9 があります。あらかじめテキスト・ファイルに Lisp のプログラムを作っておけばよいのです。

11 今、前節の関数 `succ` の定義と変数 `x` への代入式を、以下のようなテキスト・ファイル `"test.lsp"`  
12 に格納しておいたと仮定します。ただしファイルはインタプリタと同じ場所（ディレクトリ、フォ  
13 ルダ）に保管されているとします。

```
14 (defun succ (x) (+ x 1))
15 (setq x 777)
```

16 これを Lisp インタプリタ中に読み込むには、以下のように入力します。

```
17 > (load "test.lsp")
18 Loading test.lsp
19 Finished loading test.lsp
20 T
21 >
```

以上で、関数 `succ`、変数 `x` がインタプリタ上で使用可能になります。この操作は、図 4.5 に

ソース・プログラム → インタプリタ

22 と図示されている操作に相当します。

## 4.7 手続き的プログラミングの支援

Lisp は関数型プログラミング言語ですから、

プログラミング = 関数を定義すること

なのですが、しかし、手続き型言語のプログラミング・スタイルが全く使えないのは不便です。そのため、Lisp には「式を順番に記述しておく、その順番に式の評価が進む」ような手続き的プログラムを書く仕組みがいくつか用意されています。

以下の `progn` 式は、 $n$  個の式  $e_1$ 、...、 $e_n$  を順番に評価し、最後の  $e_n$  の値を式全体の値とします。

```
(progn  $e_1$  ...  $e_n$ )
```

たとえば以下のように用います。

```
> (progn (setq x 1) (setq y 1) (setq z (+ x y)) (+ x y z))
4
>
```

ここではまず  $x$  に 1 を代入し、次に  $y$  に 1 を代入し、 $z$  に  $(+ x y)$  の値、すなわち 2 を代入し、最後に  $(+ x y z)$  の値を求めると 4 になります。これはちょうど以下の C/C++ のプログラム片に似ています。

```
x = 1;
y = 1;
z = x + y;
return x + y + z;
```

実は 4.4.2 節で解説した `cond` にも同様の機能があります。4.4.2 節では `cond` の構文は

```
(cond ( $t_1$   $e_1$ )
      ( $t_2$   $e_2$ )
      ...
      ( $t_n$   $e_n$ ))
```

と紹介しましたが、正確には

```
(cond ( $t_1$   $e_1$   $e'_1$   $e''_1$  ...)
      ( $t_2$   $e_2$   $e'_2$   $e''_2$  ...)
      ...
      ( $t_n$   $e_n$   $e'_n$   $e''_n$  ...))
```

と、条件判定式の後に 0 個以上の式を並べることが可能です。

実は関数定義を行う `defun` にも同様の機能があります。4.5 節では `defun` の構文は

```
(defun  $f$  ( $v_1$  ...  $v_n$ )  $e$ )
```

と紹介しましたが、正確には

```
(defun f (v1 ... vn) e1 ... em)
```

と、仮引数リストの後に0個以上の式を並べることが可能です。関数値は、最後に評価した式  $e_m$  の値です。

以下の `loop` 式は、`progn` 式の拡張形ですが、式  $e_1$ 、...、 $e_n$  をこの順番に繰り返して評価し続けます。

```
(loop e1 ... en)
```

しかし、`loop` 式をそのまま実行しても無限ループになりますから、通常はループから抜け出すための `return` と組み合わせて使用します。たとえば、階乗を計算する関数を再帰を用いずに `loop` と `return` を用いて定義すると以下の通りです。

```
(defun fact (n)
  (setq i 1)
  (setq r 1)
  (loop (setq r (* r i))
        (if (= i n) (return))
        (setq i (+ i 1))
        r)
```

ここに `(return)` によって `loop` から抜け出します。この Lisp プログラムを C/C++ 風を書くならば以下の通りです。

```
int fact(int n){
  int i,r;
  i = 1;
  r = 1;
  while(1){
    r = r*i;
    if(i == n) break;
    i = i+1;
  }
  return r;
}
```

このように、Lisp を手続き型言語風に利用することも可能ですが、これについて詳しく述べてもこの授業の主旨から外れていくため、この辺りで終えることにします。

以上、4.3 節からこの節まで Lisp の基本を解説してきました。

ここからは、Lisp の名前の由来でもあるリスト処理 (List processing) について解説していきます。

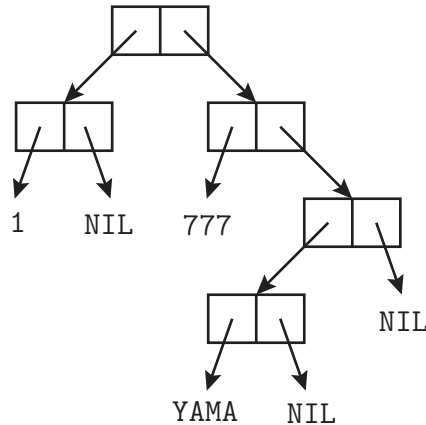


図 4.8: 2 進木の例

## 4.8 リスト処理

Lisp のリストとは、誤解を恐れず言えば、2 進木 (binary tree) の特殊な表現です。Lisp にはこの 2 進木を扱うための便利な機能が用意されています。

たとえば、図 4.8 は 2 進木 (= リスト) の例です。このようなデータ構造を Lisp の関数で操作することがこの節の目的です。

### 4.8.1 アトム

2 進木 (= リスト) は、葉節 (leaf node) と内部節 (inner node) で構成されます。その葉節になりうるデータのことをアトム (atom) と呼びます。たとえば図 4.8 の葉節は 1、YAMA、777、NIL であり、これらは全てアトムです。

アトムはそれ以上分解できない構成要素のことを意味しますが、Lisp におけるアトムとは次のようなデータです。

**数値:** 整数、浮動小数点数、分数、複素数などはいずれもアトムです。

**文字列:** ダブル・クォーテーションで囲まれた文字列はアトムです。

**NIL:** 4.4.1 節で紹介した NIL はアトムです。NIL は空リストを表す、特殊なリストでもあります。

**シンボル:** Lisp のプログラム空間で使用可能な 名前 のことをシンボル (symbol) と呼びます。図 4.8 の YAMA は、一見すると文字列に見えますが、文字列の場合には "YAMA" と表記しますが、YAMA は文字列ではありません。シンボルは変数名や関数名として使用可能ですが、必ずしも変数名、関数名である必要はありません。Lisp のプログラム中でシンボルを表記すると

1 きには、直前にシングルのクオーテーションを付けます。たとえば `'yama` です<sup>11</sup>。シンボル  
2 をインタプリタで評価すると以下の通りです。

```
3 > 'yama
4 YAMA
5 >
```

6 `'yama` はシンボルですが、シングルのクオーテーションを付けない `yama` は変数を意味しま  
7 す。よって

```
8 > (setq yama 10)
9 10
10 > yama
11 10
12 > 'yama
13 YAMA
14 > (+ yama 1)
15 11
16 > (+ 'yama 1)
17 Error: YAMA is not of type NUMBER
18 ...
19 >
```

20 シンボル名を変数名と解釈するために `symbol-value` という関数が用意されています。これ  
21 を用いるならば、シンボル名 `'yama` から変数 `yama` の値を取り出すことが可能です。よって

```
22 > (+ (symbol-value 'yama) 1)
23 11
24 >
```

## 25 4.8.2 ドット対

26 2進木 (= リスト) の内部節をあえて C/C++ の構造体で表すならば、以下のような定義<sup>12</sup> と類  
27 似しています。

```
28 struct Cons {
29     void *left;
30     void *right;
31 };
```

32 これを図示したものが図 4.9 です。この構造体のことを一般に **cons** セル (cons cell、コンス・セ  
33 ル)<sup>13</sup> あるいは簡単にセルと呼びます。

<sup>11</sup>実は `'yama` は `(quote yama)` の省略形です。quote はその引数を評価せず、その引数そのものを返す特殊形式です。

<sup>12</sup>構造体定義中の `void*` は、任意のデータを指すことのできるポインタを意味します。

<sup>13</sup>cons とは CONSTRUCTOR (構造体を作る演算子) に由来します。

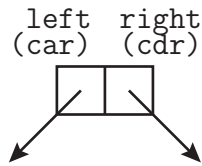


図 4.9: cons セル

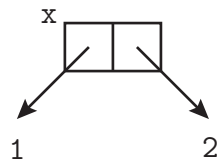


図 4.10: ドット対 (1 . 2)

セルを作るときには関数 `cons` を用います。この関数はセルをひとつ生成し、セルの二つのポインタが二つの引数のデータをそれぞれ指すように設定します。たとえば以下の入力では、まず変数 `x` に図 4.10 のデータを代入します<sup>14</sup>。

```
> (setq x (cons 1 2))
```

ここに (1 . 2) は、図 4.10 のデータのテキスト表示表現であって、このような表現を特にドット対 (dotted pair) と呼んでいます。さらに、以下の入力では変数 `y` に図 4.11 のデータを代入します。

```
> (setq y (cons x 3))
((1 . 2) . 3)
>
```

((1 . 2) . 3) もドット対です。もし以下のように入力したならば、図 4.12 のようなデータ構造となります。

```
> (setq y (cons 3 x))
(3 1 . 2)
>
```

なお、上の出力は、(3 . (1 . 2)) ではなく、(3 1 . 2) となっており、一見すると表示方法が一貫していない印象を受けるかもしれません。しかし、これは括弧 ( ) の数をできるだけ少なく表示するための `gcl` の仕様です。

<sup>14</sup>「変数 `x` に... データを代入します」という言い方では、`x` の中にそのデータがまるまる格納されているような印象を与えますが、むしろ「(ポインタとしての) `x` がヒープ領域内に生成されたデータを指している」という方が近いでしょう。



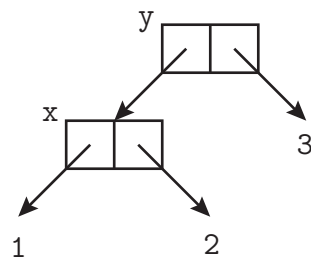


図 4.11: ドット対 ((1 . 2) . 3)

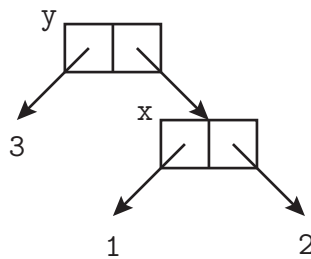


図 4.12: ドット対 (3 . (1 . 2)) (= (3 1 . 2))

### 4.8.3 リスト

2進木の中の最右の葉節のアトムがNILであるようなものを、特にリスト (list) と呼びます<sup>15</sup>。たとえば、図 4.8 の木はリストです。図 4.10 から図 4.12 の木はリストではありません。図 4.13 の木はいずれもリストです。これらのリストを作るには、cons を用いて以下のように入力します。ここでは (a)~(e) のリストを変数 a~e に代入します。

```

6  > (setq a (cons 1 nil))
7  (1)
8  > (setq b (cons (cons 1 2) nil))
9  ((1 . 2))
10 > (setq c (cons 3 (cons 1 nil)))
11 (3 1)
12 > (setq d (cons (cons 1 nil) (cons 2 nil)))
13 ((1) 2)
14 > (setq e nil)
15 NIL
16 >

```

ここに nil (または NIL) は空リスト (シンボルを全く持たないリスト) を表します。括弧 ( ) も空リストを表します。

<sup>15</sup>純リスト (pure list) とも呼びます。

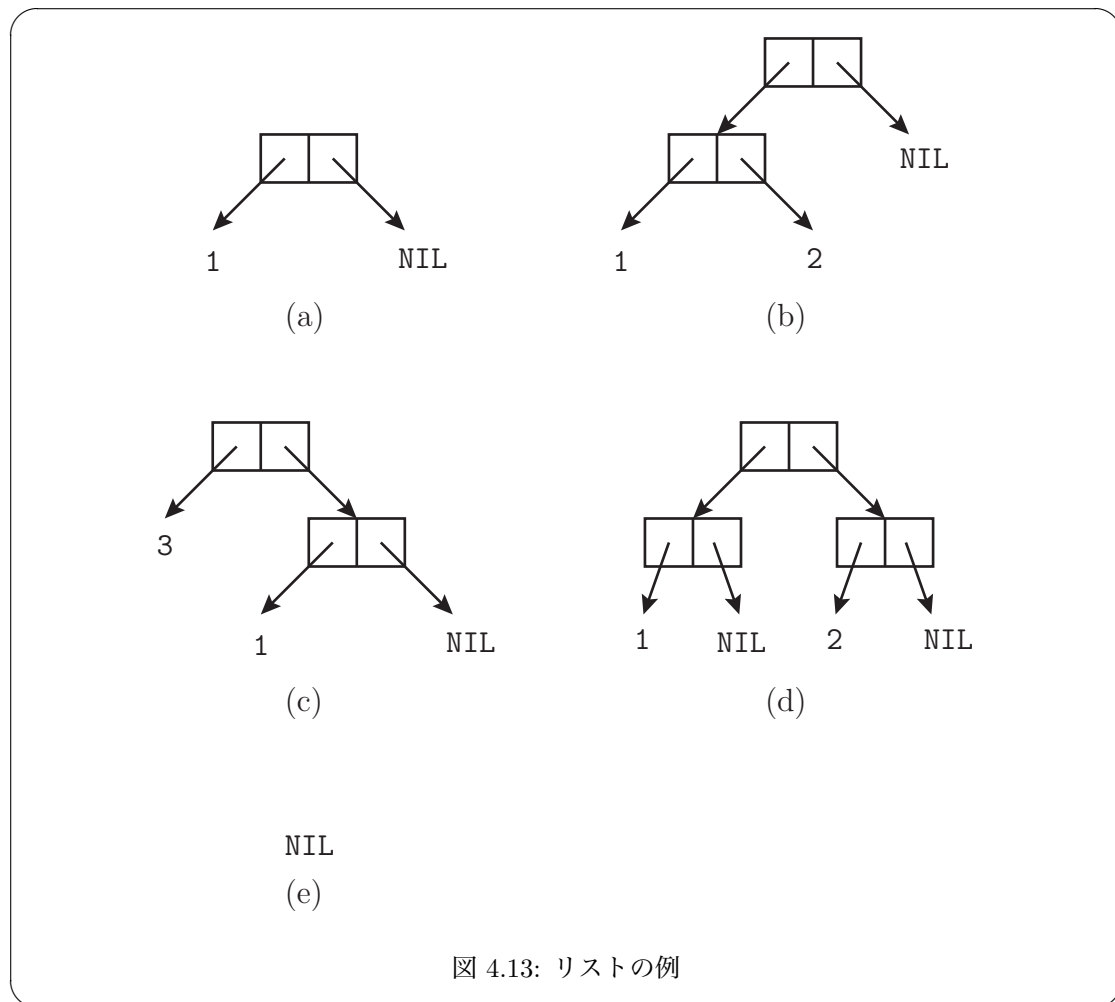


図 4.13: リストの例

一般にリストは図 4.14(a) のような形をしていると見なすことができます。ここに各  $\ell_i$  はアトムまたは 2 進木（一般にリストでなく、ドット対でも構いません）です。上の実行例に見られるように、このようなリストをインタプリタは

$(\ell_1 \ell_2 \dots \ell_n)$

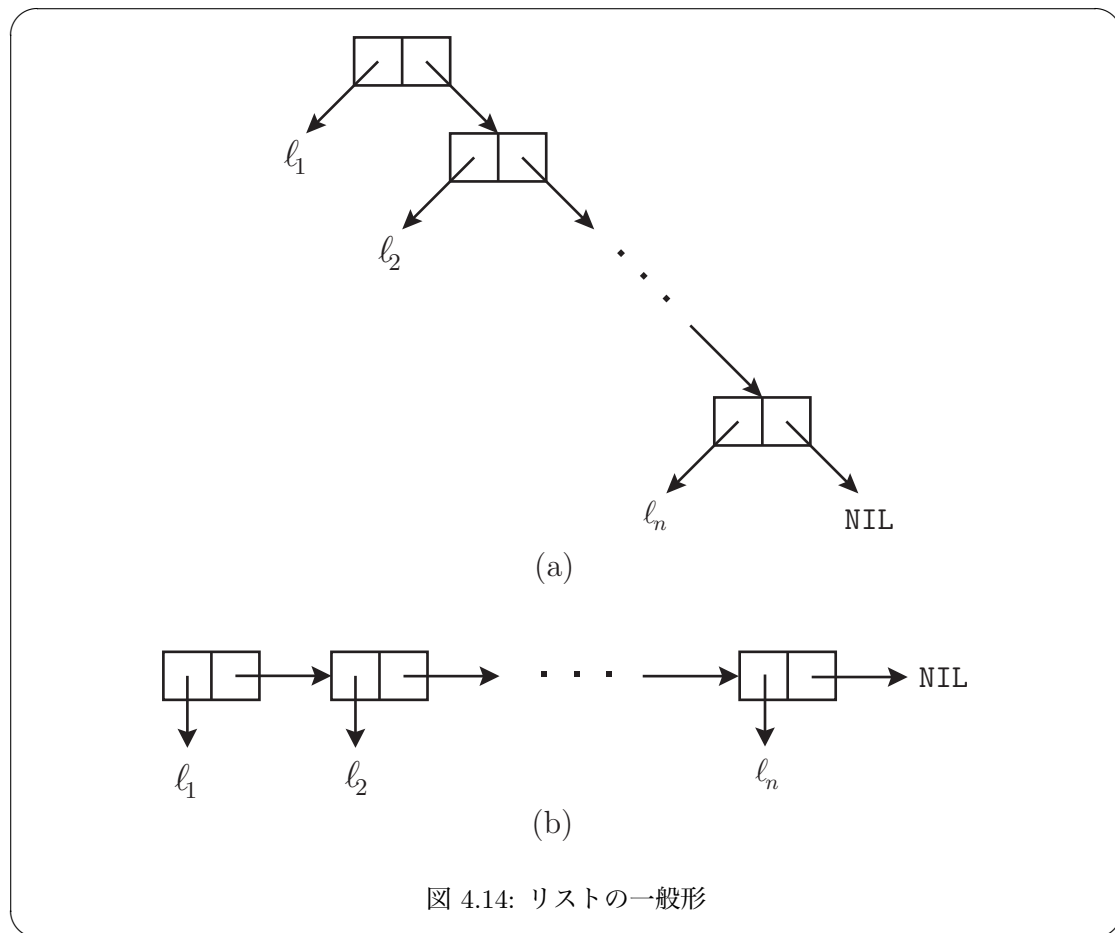
と出力します。

図 4.14(a) を図 4.14(b) のように描くこともあります。どちらの表現を用いても大差ありませんが、以下では主に (b) の表現を用いることとします。図から分かるように、Lisp のリストは、一般に線形リスト（linear list）、連結リスト（linked list）と呼ばれるデータ構造に相当します。

関数 `list` を用いると、リストの入力が軽減されます。図 4.14 のリストを作る場合、

`(list  $\ell_1 \ell_2 \dots \ell_n$ )`

と入力します。たとえば図 4.13 のリストを作るには以下の通りです。



```

1  > (setq a (list 1))
2  (1)
3  > (setq b (list (cons 1 2)))
4  ((1 . 2))
5  > (setq c (list 3 1))
6  (3 1)
7  > (setq d (list (list 1) 2))
8  ((1) 2)
9  > (setq e (list))
10 NIL
11 >

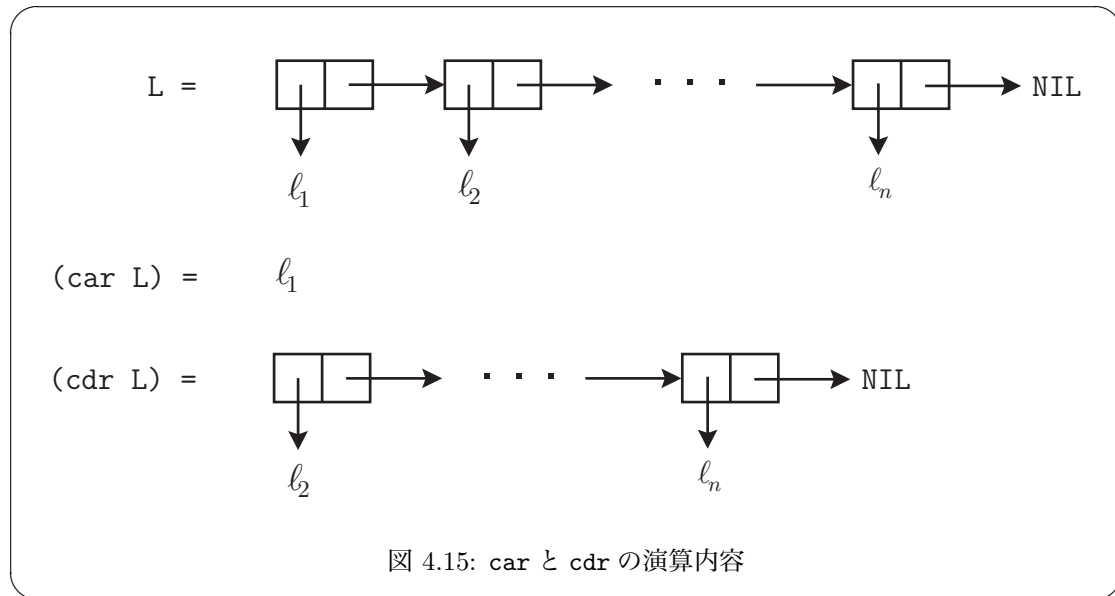
```

12 前節のドット対の場合と同様、シングル・クォーテーションを用いるとリストの入力を簡略化で  
 13 きます。以下は図 4.13 の (a)~(e) をシングル・クォーテーションを用いて入力した場合です。

```

14 > (setq a '(1))
15 (1)
16 > (setq b '((1 . 2))
17 ((1 . 2))
18 > (setq c '(3 1))
19 (3 1)
20 > (setq d '((1) 2))

```



```

1  ((1) 2)
2  > (setq e '())
3  NIL
4  >

```

5 なお、式が変数を含む場合にはシングルのクオーテーションを用いたのでは、その変数はシンボル  
6 と見なされ、変数の値を取り出すことができません。その場合には `cons` や `list` を用います。

#### 7 4.8.4 要素の取出し

8 与えられたリストからそのリストを構成する各部分を取り出すには二つの関数 `car` (「カー」と読  
9 みます)<sup>16</sup> と `cdr` (「クダー」と読みます)<sup>17</sup> を用います。

10 今、 $L$  がリスト  $(l_1 \ l_2 \ \dots \ l_n)$  であるとき、以下の式

```
(car L)
```

12 はリストの先頭の要素  $l_1$  を取出します (図 4.15 参照)。たとえば前節で定義した変数 `a`~`e` につ  
13 いて

```

14 > (car a)
15 1
16 > (car b)
17 (1 . 2)
18 > (car c)

```

<sup>16</sup>名称は、Lisp が初めて実装された IBM のコンピュータの命令：Contents of Address part of Register number に由来します。

<sup>17</sup>名称は、Lisp が初めて実装された IBM のコンピュータの命令：Contents of Decrement part of Register number に由来します。

```

1      3
2      > (car d)
3      (1)
4      > (car e)
5      NIL
6      >

```

7 空リストについては、`(car nil)=nil` であることに注意してください。

8  $L$  がリスト  $(l_1 \ l_2 \ \dots \ l_n)$  であるとき、以下の式

`(cdr L)`

9  
10 は先頭の要素を除いたリスト  $(l_2 \ \dots \ l_n)$  を取出します (図 4.15 参照)。たとえば前節で定義し  
11 た変数 `a~e` について

```

12      > (cdr a)
13      NIL
14      > (cdr b)
15      NIL
16      > (cdr c)
17      (1)
18      > (cdr d)
19      (2)
20      > (cdr e)
21      NIL
22      >

```

23 空リストについては、`(cdr nil)=nil` であることに注意してください。

24 `car`, `cdr` を連続して適用すれば、リストのより深い部分の要素を取出すことが可能です。たと  
25 えば

```

26      > (car (car b))
27      1
28      > (cdr (car b))
29      2
30      >

```

31 上の入力を簡単化するため、Lisp では `(car (car x))` を `(caar x)`、`(cdr (car x))` を `(cdar x)`  
32 と簡略化することが可能です。他に `cadr`, `caaar`, `cdddr`, `cdadr`, ... などが使用可能です。

33 なお、`car`, `cdr` をリスト以外に適用することができません。たとえば

```

34      > (car 1)
35      Error: 1 is not of type LIST.
36      ...
37      >

```

## 1 4.8.5 リストに関する条件判定

2 関数 `atom` は、引数のデータがアトムであるか否かを判定します。関数 `listp` は、引数のデータ  
3 がリスト（ドット対を含む）であるか否かを判定します。たとえば

```
4 > (atom 1)
5 T
6 > (atom 'yama)
7 T
8 >(atom '(1))
9 NIL
10 > (atom nil)
11 T
12 > (listp 1)
13 NIL
14 > (listp '(1 . 2))
15 T
16 > (listp nil)
17 T
18 >
```

19 ここでまぎらわしいのは `nil` です。`nil` は空リストを意味しますが、アトムでもあり、かつリス  
20 トでもあると定義されているため、上の二つの関数に対して `T` を返します。そこで、`nil` の判定に  
21 は関数 `null` を用います。この関数は引数が `nil` のときのみ真となります。たとえば

```
22 > (null nil)
23 T
24 > (null 'yama)
25 NIL
26 >(null '(1))
27 NIL
28 > (null 1)
29 NIL
30 >
```

31 二つのデータ（主にリストが対象）が等しいか否かの判定には関数 `eq` と `equal` を持ちます。`eq`  
32 は、二つの引数のデータを格納しているメモリ・アドレスが等しいか否かを判定します。それに対  
33 して `equal` は、二つの引数のデータの内容（リストならばリストの構造および葉節のアトム全て）  
34 が等しいか否かを判定します。たとえば

```
35 > (setq x '(1 2 3))
36 (1 2 3)
37 > (setq y '(1 2 3))
38 (1 2 3)
39 > (eq x y)
40 NIL
41 > (eq x x)
42 T
43 > (equal x y)
```

```
(defun member (x y)
  (cond ((null y)      nil)
        ((eq x (car y)) t)
        (t             (member x (cdr y)))))
```

図 4.16: 所属判定を調べる関数 member

```
1      T
2      > (equal x x)
3      T
4      >
```

5 アトムと同値性判定では eq も equal も正しい判定を行います。数値の比較では eq よりも "="  
6 (4.4.1 節参照) を用いるべきです。リストの同値性を eq で判定する場合には細心の注意が必要です。

7 以上で、Lisp を用いるための基本的な知識の説明を終えます。これ以降は、様々な問題を Lisp  
8 を用いてどのように表現し、解いていくのか、典型的な例題を挙げて説明していきます。

9 なお、先に、リストの一般形を

```
10      ( $l_1$   $l_2$  ...  $l_n$ )
```

11 と説明し、各  $l_i$  はアトムまたは2進木（リストでなくとも構いません）と定義しました。しかし、  
12 その定義をさらに制限し、 $l_i$  はアトムまたはリストであり、ドット対ではないと設定した方が議論  
13 が単純化します。しかも実用上もほとんど支障がありません。以下ではそのように制限したリスト  
14 について解説を進めていきます。もちろん、 $l_i$  がリストである場合にはその要素もアトムまたはリ  
15 ストであると再帰的に制限します。

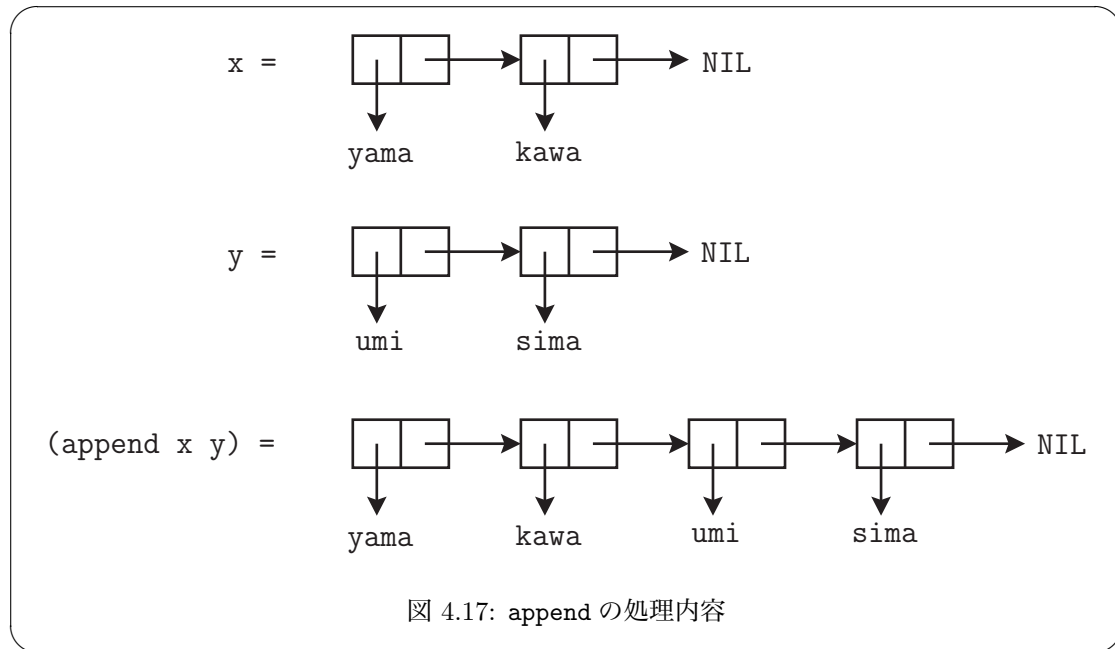
## 16 4.9 リストを用いた簡単な例題

17 この節ではリスト処理のよく知られた典型的な例題を紹介します。これによってリスト処理の勘  
18 所が分かってくるでしょう。

### 19 4.9.1 所属判定

20  $n$  個 ( $n \geq 0$ ) の要素を持つリスト ( $a_1 a_2 \dots a_n$ ) が与えられたとき、 $a_1$  から  $a_n$  の中に  $b$  と等し  
21 いデータが存在するか否かを判定し、もし存在するならば T、さもなければ NIL を返す関数 member  
22 を作りましょう。たとえば

```
23      > (setq x '(yama kawa umi))
24      (yama kawa umi)
25      > (member 'kawa x)
26      T
```



```
1  > (member 'sima x)
2  NIL
3  >
```

4 このような関数は図 4.16 のように定義できます。この定義の意味は以下の通りです。

- 5 1. もしリスト `y` が `nil` ならば、偽値 (`nil`) を返す。
- 6 2. もし `x` が `y` の先頭の要素 (`car y`) に等しい<sup>18</sup> ならば、真値 (`t`) を返す。
- 7 3. さもなくば、`x` が、`y` から先頭を除いた残りのリスト (`cdr y`) に含まれるかを調べる。

8 **考察** `member` は以下のように定義されている場合があります。図 4.16 と異なっても問題が生じ  
9 ない理由を述べなさい。

```
10 (defun member (x y)
11   (cond ((null y)      nil)
12         ((eq x (car y)) y)      //この行の最右の y は上の定義では t です。
13         (t              (member x (cdr y)))))
```

## 14 4.9.2 リストの接続

15 今、変数 `x`, `y` にそれぞれ以下のようなリストを代入したと仮定しましょう。

<sup>18</sup>関数 `eq` の代わりに `equal` を用いる定義もあります。



```
(defun append (x y)
  (cond ((null x) y)
        (t      (cons (car x) (append (cdr x) y)))))
```

図 4.18: 二つのリストを接続する関数 `append`

```
1  > (setq x '(yama kawa))
2  (yama kawa)
3
4  > (setq y '(umi sima))
5  (umi sima)
6  >
```

7 このとき、`x` のリストの後ろに `y` のリストを接続し、リスト:

```
8  (yama kawa umi sima)
```

9 を求めるような処理を一般に **append** と呼びます (図 4.17 参照)。append するリストはもっと複  
10 雑でも構いません。たとえば

```
11 > (setq x '((1 2) 3 4))
12 ((1 2) 3 4)
13
14 > (setq y '((5) (6 7)))
15 ((5) (6 7))
16 >
```

17 とするとき、この二つのリストを `append` した結果は以下の通りです。

```
18 ((1 2) 3 4 (5) (6 7))
```

19 `append` を実現する関数定義は図 4.18 の通りです。この定義の直感的な意味は以下の通りです。

- 20 1. `x` が空リスト (`nil`) ならば、append したリストは `y` そのものであるから、`y` を返す
- 21 2. さもなくば、まず `x` から先頭を除いたリスト (`cdr x`) に `y` を append したリストを求め、そ  
22 のリストの先頭に `x` の先頭の要素 (`car x`) を付けたリストを返す。

23 考察 上の例で、関数呼び出し (`append x y`) によって、どのように処理が進むか、図を用いて説  
24 明しなさい。

(append x y)

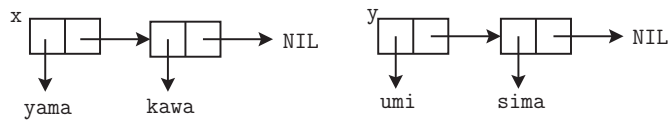


図 4.19: append の処理 (その 1)

1 補足説明：append 処理の詳細

2 append の実行の様子が分かりにくいという感想が毎年、多く寄せられます。そこで考察の解答  
3 を図を使って以下に詳細に解説します。

4 今、変数 x, y にそれぞれ以下のようなリストを代入したと仮定しましょう。

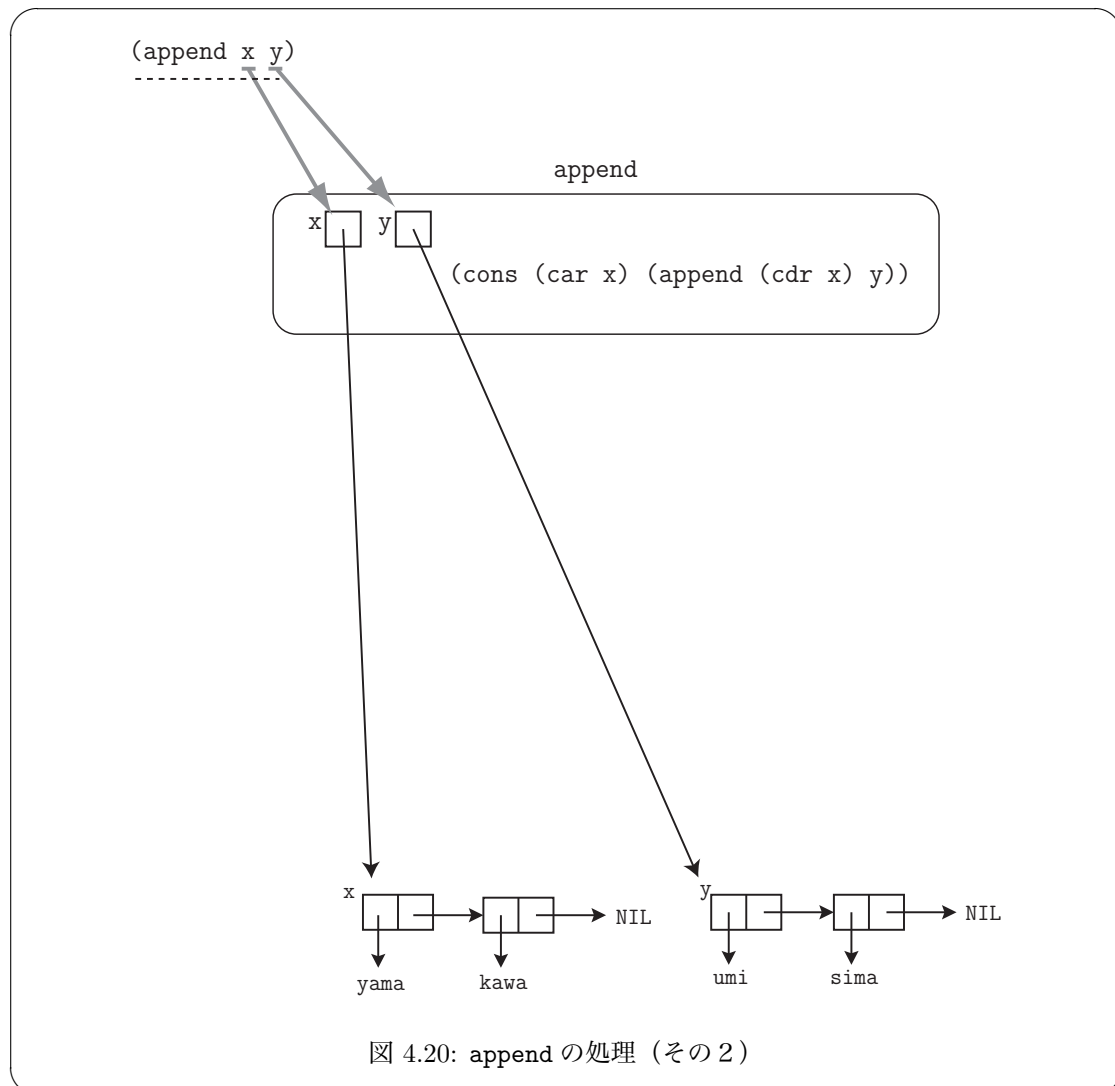
```

5 > (setq x '(yama kawa))
6 (yama kawa)
7 > (setq y '(umi sima))
8 (umi sima)
9 >
  
```

10 次に入力：

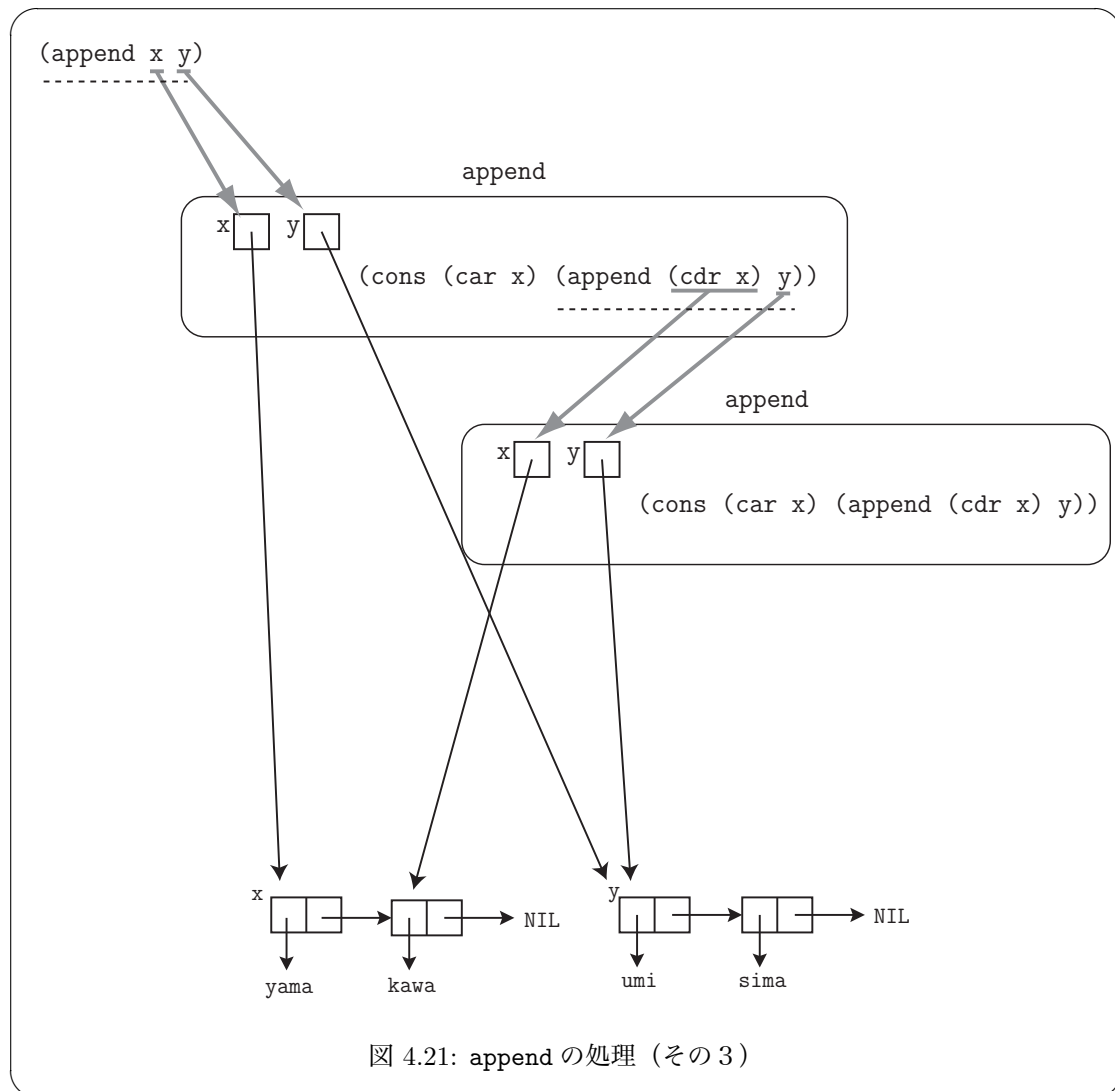
```

11 > (append x y)
  
```

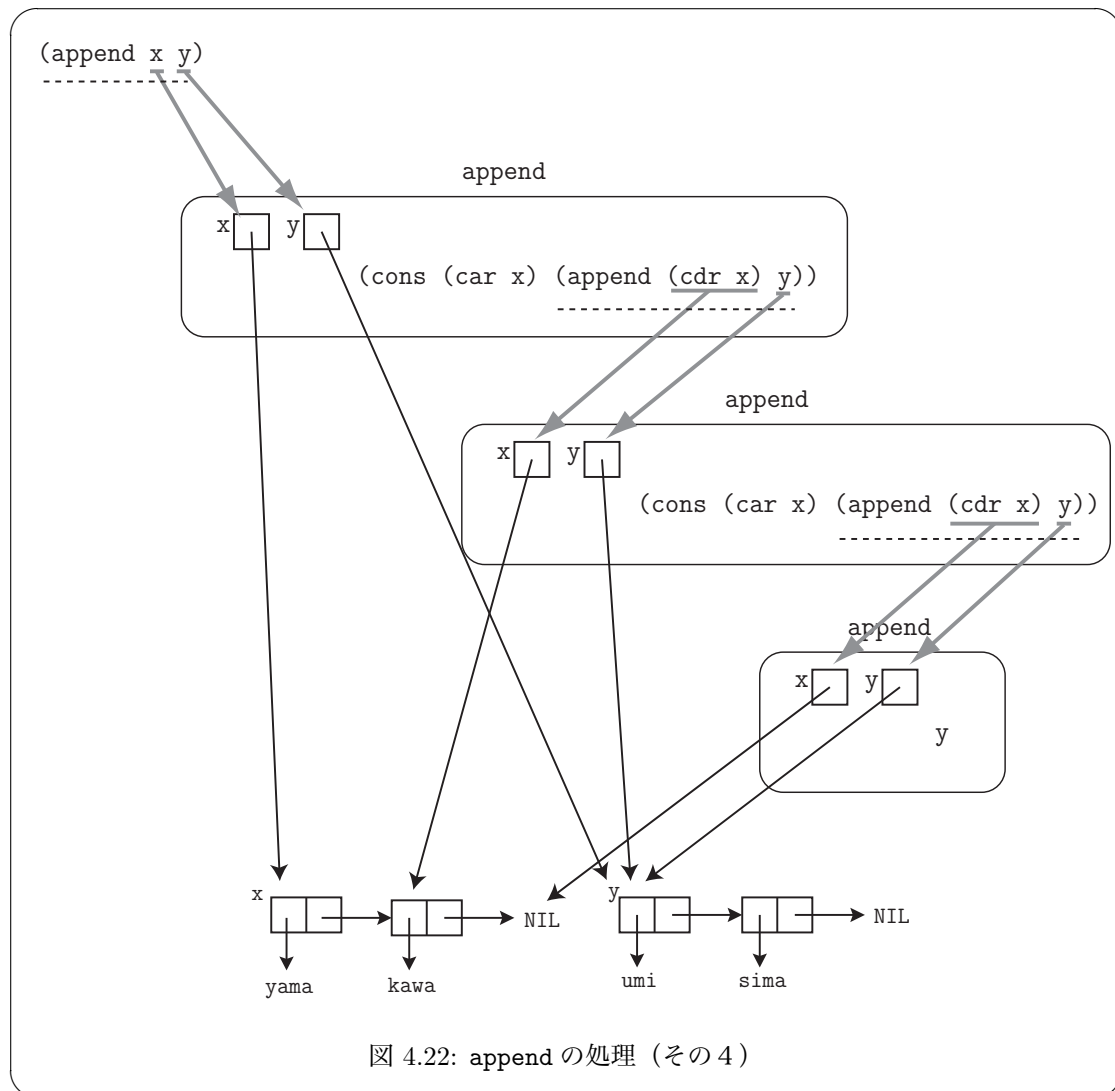


- 1 を実行したときのインタプリタの内部処理の様子を図式で補足説明します。
- 2 まず図 4.19 に (append x y) の実行が開始された直後のインタプリタの様子を図示します。メ
- 3 モリ内（正確には、メモリのヒープ領域内）には、あらかじめ作っておいた二つのリスト '(yama
- 4 kawa) と '(umi sima) が存在します。関数呼び出し (append x y) の実引数 x、y はそれぞれの
- 5 リストを意味します。
- 6 さて関数呼び出し (append x y) を実行するために、図 4.20 に示すように、インタプリタはプ
- 7 ログラムが実行される（抽象的な）空間内に append 関数の実行環境を構築します。その環境内に
- 8 は関数定義（上記 defun の式）の中の仮引数 x、y が存在します。そして、それらは関数呼び出し
- 9 (append x y) の実引数 x、y の値で初期設定されます。図 4.20 では、仮引数 x、y をポインタの
- 10 ように扱い、そのポインタがそれぞれのリストを指すように図示しました<sup>19</sup>。

<sup>19</sup>ポインタという言い方は C/C++ のプログラムの理解のための便宜的な表現です。C/C++ のポインタとは扱いが異なることに注意しましょう。

図 4.21: `append` の処理 (その 3)

- 1 この `append` 関数の実行環境内では実行は次のように進みます。まず、`x` が `NIL` であるか否かの
- 2 チェックを行います (`(null x)` の部分)。図からも明らかなように、`x` は `NIL` ではないため、結局、
- 3 この関数呼び出しの式全体の値は式 `(cons (car x) (append (cdr x) y))` の値となり、この式
- 4 の評価を行わねばなりません。そして、この式の評価のためには、関数呼び出し (`append (cdr`
- 5 `x) y)` を評価せねばなりません。
- 6 そこで関数呼び出し (`append (cdr x) y)` を実行するために、図 4.21 に示すように、インタプ
- 7 リタはプログラムが実行される (抽象的な) 空間内に `append` 関数の実行環境をさらに構築します。
- 8 その環境内の仮引数 `x`、`y` は、関数を呼び出した側の実引数 (`cdr x`)、`y` の値で初期設定されます。
- 9 この `append` 関数の実行環境内でも、`x` が `NIL` であるか否かのチェックを行います。図からも明
- 10 らかなように、`x` は `NIL` ではないため、結局、この関数呼び出しの式全体の値は式 `(cons (car x)`
- 11 `(append (cdr x) y))` の値となり、この式の評価を行わねばなりません。そして、この式の評価



- 1 のためには、再度、関数呼び出し (`append (cdr x) y`) を評価せねばなりません。
- 2 そこで関数呼び出し (`append (cdr x) y`) を実行するために、図 4.22 に示すように、インタプ
- 3 リタはプログラムが実行される (抽象的な) 空間内に `append` 関数の実行環境をさらに構築しま
- 4 す。その環境内の仮引数 `x`、`y` は、関数を呼び出した側の実引数 (`cdr x`)、`y` の値で初期設定され
- 5 ます。図 4.20、図 4.21 と異なるのは、仮引数 `x` の値が今回は `NIL` となる点です。
- 6 この `append` 関数の実行環境内でも、`x` が `NIL` であるか否かのチェックを行います。この場合、`x`
- 7 は `NIL` であるため、結局、この関数呼び出しの式全体の値は `y` です。

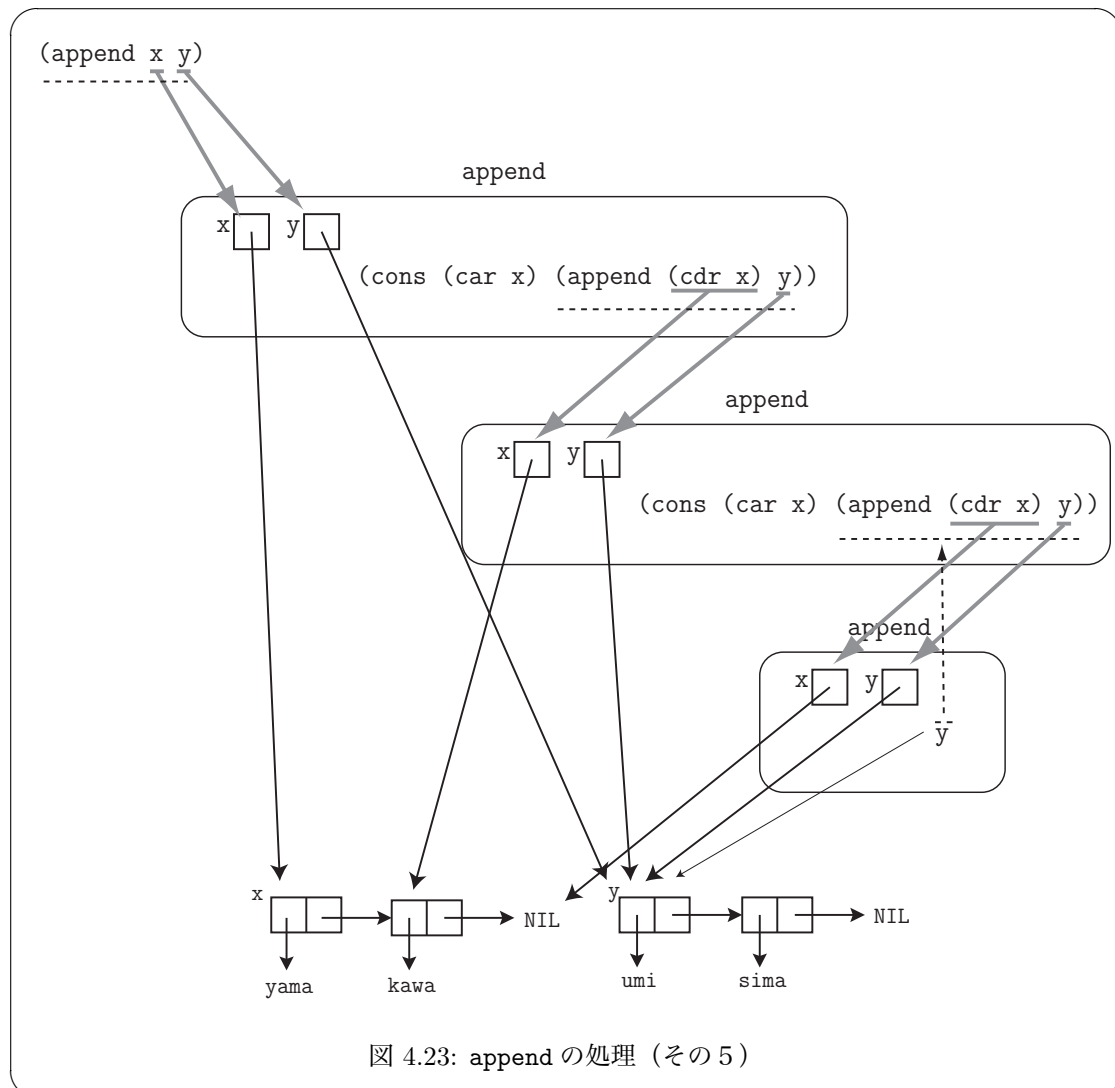


図 4.23: `append` の処理 (その 5)

- 1 求められた関数値は、その関数を呼び出した側へ戻されます (リターンされます)。つまり、図
- 2 4.23 の 3 層の `append` の実行環境の中で、最下層 (第 3 層) の実行環境中の `y` の値が第 2 層の実行
- 3 環境の関数呼び出し `(append (cdr x) y)` の値となります。
- 4 第 3 層の実行環境は関数値を第 2 層に引き渡した直後に消滅します。

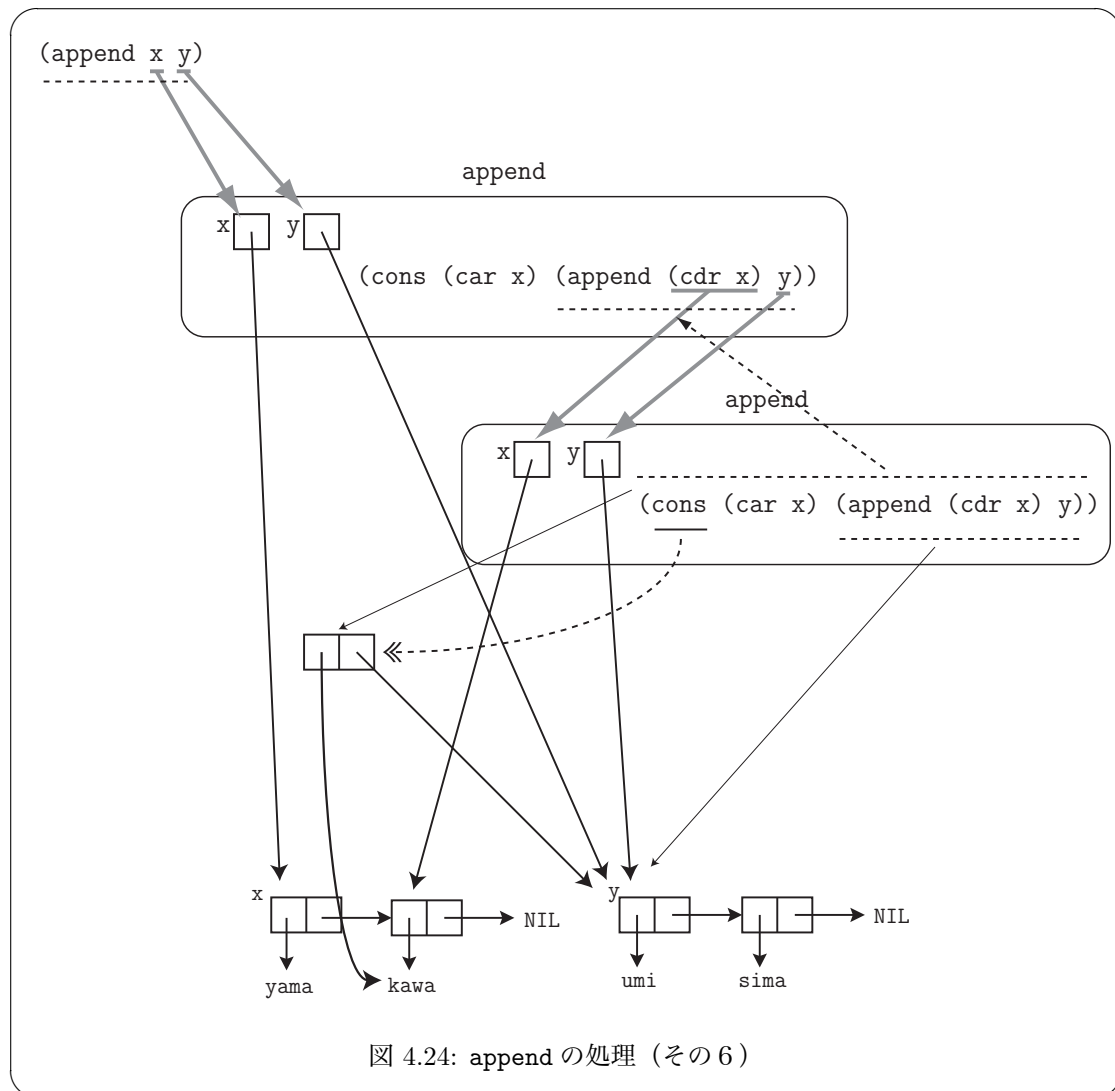
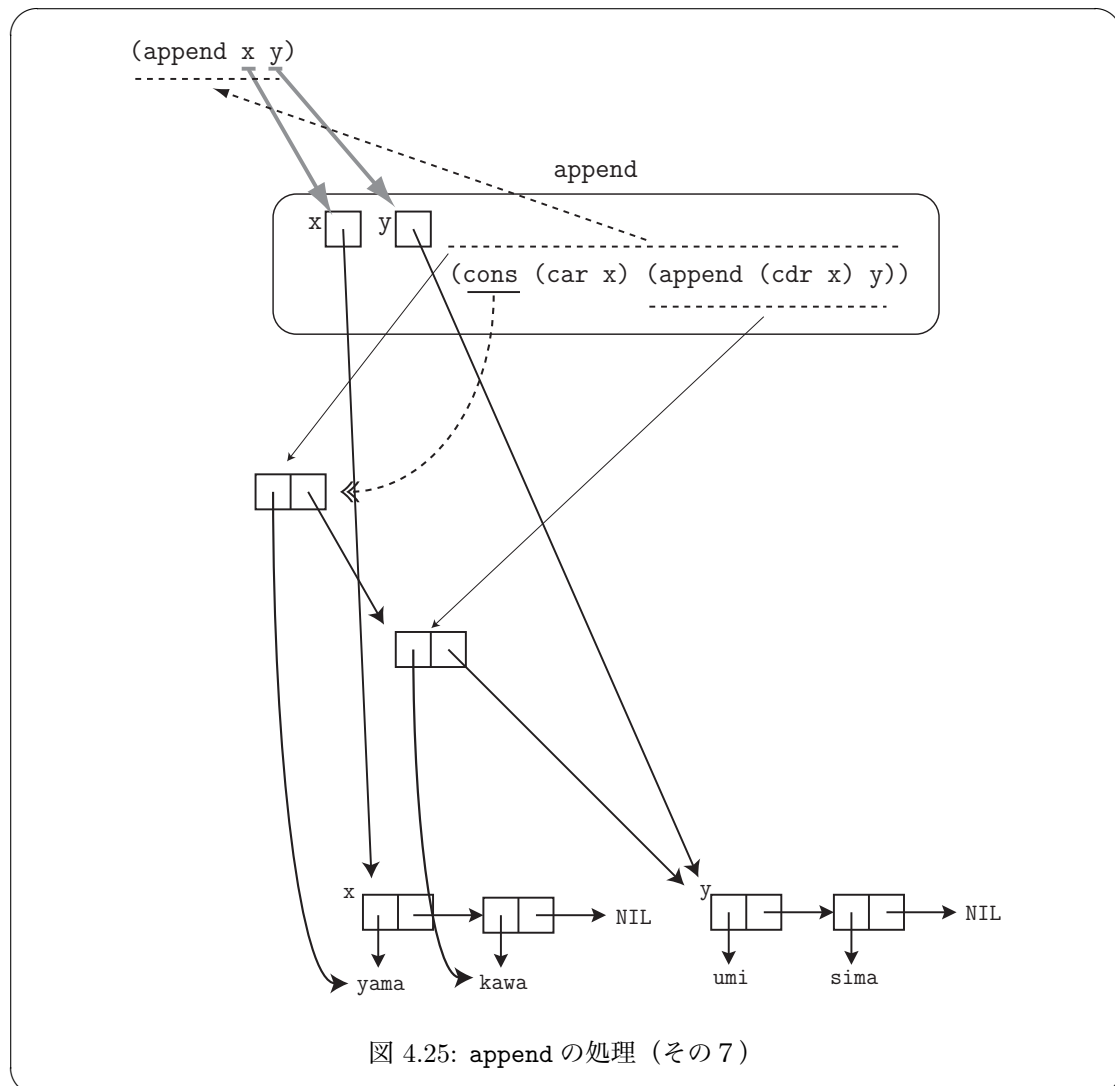


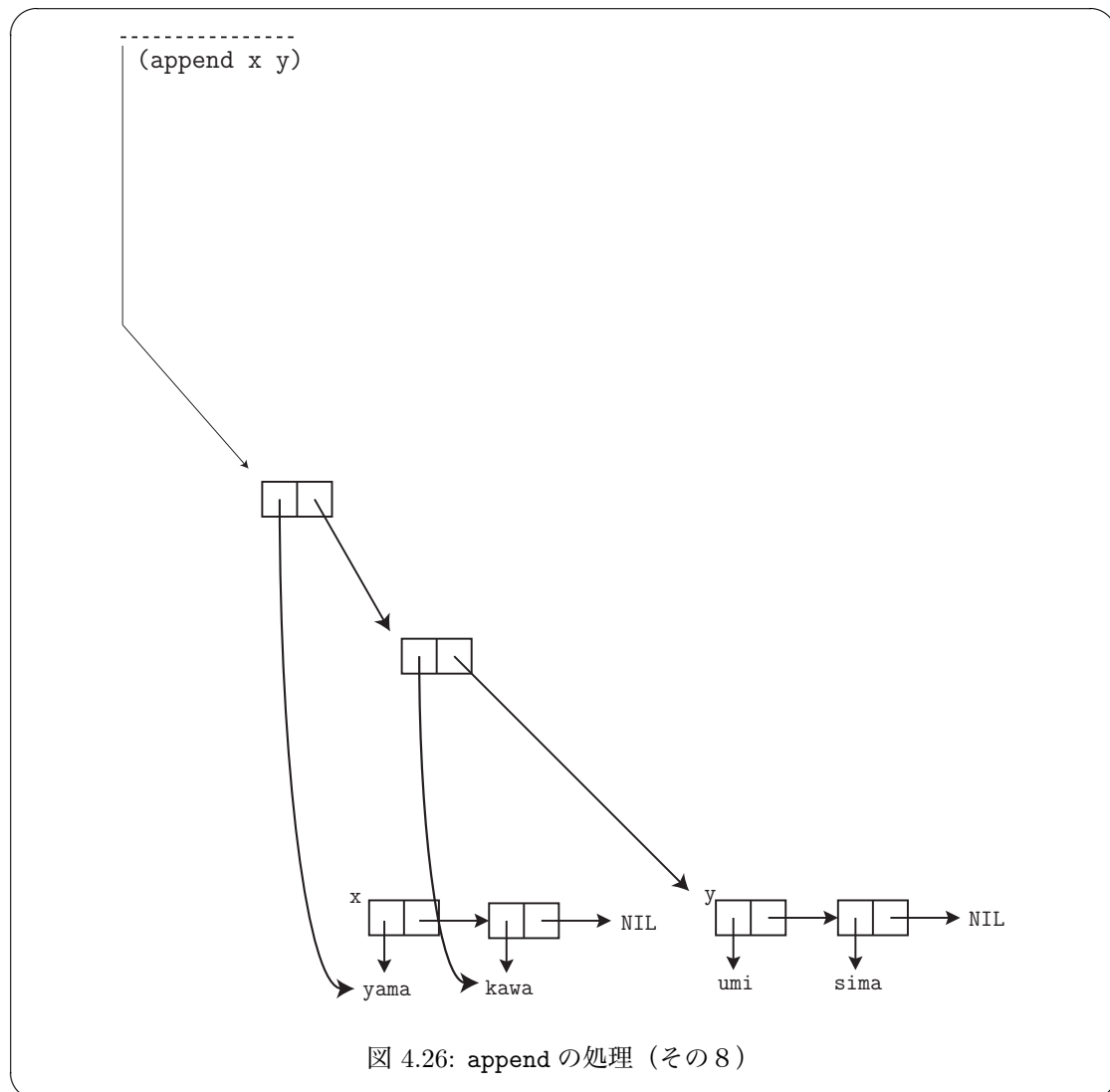
図 4.24: `append` の処理 (その 6)

- 1 次に、関数値を受け取った第2層では、最上層 (第1層) に渡す (リターンする) 関数値を計算
- 2 します。つまり、インタプリタは、`cons` 関数を用いて、`cons` セルをメモリ内にひとつ生成し、左
- 3 側の (`car` 側の) ポインタが (`car x`) の値 (=シンボル `kawa`) を指し、右側の (`cdr` 側の) ポイ
- 4 ンタは (`append (cdr x) y`) の値を指すようにデータを構築します。そして、それを第1層へ渡
- 5 します。
- 6 第2層の実行環境は関数値を第1層に引き渡した直後に消滅します。

図 4.25: `append` の処理 (その 7)

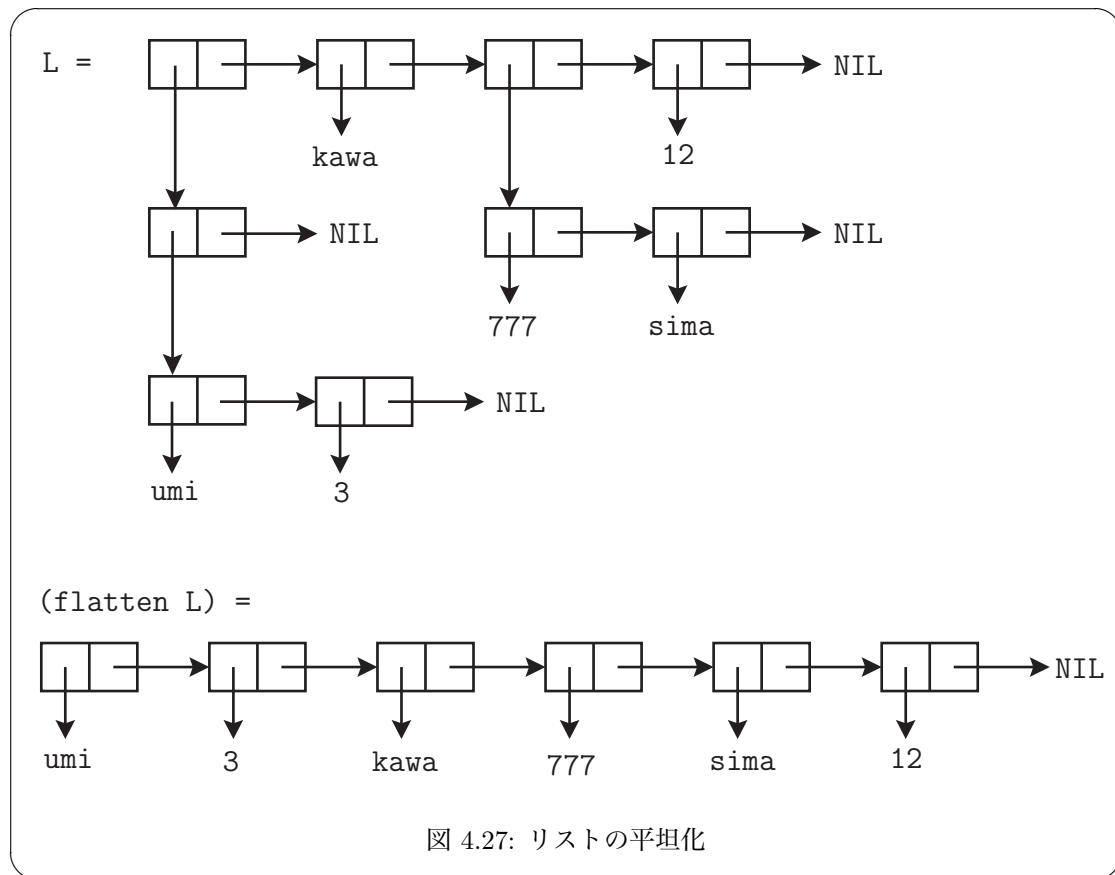
- 1 次に、関数値を受け取った第1層では、大元の関数呼び出しに渡す（リターンする）関数値を
- 2 計算します。この場合、その仕事は先ほどの第2層の実行環境とほとんど同じです。つまり、イン
- 3 タプリタは、`cons` 関数を用いて、cons セルをメモリ内にひとつ生成し、左側の（car 側の）ポイ
- 4 ンタが（car `x`）の値（=シンボル `yama`）を指し、右側の（cdr 側の）ポインタは（`append (cdr`
- 5 `x) y`）の値を指すようにデータを構築します。そして、それを上へ渡します。
- 6 そして、その直後に第1層の実行環境は消滅します。



図 4.26: `append` の処理 (その 8)

- 1 最終的に、関数呼び出し (`append x y`) の値として、目的とするリスト `'(yama kawa umi sima)`
- 2 が構築されました (図 4.26 で確認してください)。関数呼び出しの前と比較すると、新たに二つの
- 3 `cons` セルが作られています (図 4.19 と比較してください)。
- 4 また、もともと存在した二つのリスト `'(yama kawa)` と `'(umi sima)` には全く手が加えられて
- 5 いないことに注意してください。もし他の変数がこれらリストを参照している<sup>20</sup>としても、それ
- 6 らの値は全く不変であることが保証されます。この種の不変性はプログラムの動作内容を簡素化す
- 7 る点で非常に重要です。

<sup>20</sup> 正確に言えば、`x`、`y` 以外の他の変数が有するポインタがこれらリストまたはリストの一部を指していることを意味します。



```
(defun flatten (x)
  (cond ((null x) nil)
        ((atom x) (cons x nil))
        (t (append (flatten (car x)) (flatten (cdr x))))))
```

図 4.28: リストを平坦化する関数 `flatten`

### 4.9.3 リストの平坦化

リスト  $(a_1 a_2 \dots a_n)$  について、 $a_1, \dots, a_n$  が全てアトムであるとき、このリストを平坦 (flat) であると言います。ここでは平坦でないリストを平坦なリストへ変換する方法を検討します。たとえば図 4.27 の `L` は

```
'(((umi 3)) kawa (777 sima) 12)
```

という平坦ではないリストです。このリスト中の葉節のアトム (`NIL` を除く) を左から右へ順に列挙し、平坦なリストを作ると以下の通りです。

```
'(umi 3 kawa 777 sima 12)
```

平坦化する関数の名前を `flatten` とするとき、その定義は図 4.28 の通りです。そしてその直感

1 的な意味は以下の通りです。

2 1. `x` が空リスト (`nil`) ならば、それを平坦化したリストも空リストなので、`nil` を返す。

3 2. `x` がアトムならば、そのアトムだけを含むリストを作り、それを返す。

4 3. さもなくば、まず `x` の先頭 (`car x`) と、`x` から先頭を除いたリスト (`cdr x`) をそれぞれ平  
5 坦化し、その二つを `append` し、返す。

## 6 4.10 数をリストの長さで表して

7 リストを用いたプログラム例として、自然数を線形リストの長さで表し、その線形リストの上に  
8 加算、乗算等の演算を構築してみます。

### 9 4.10.1 自然数をリストで表す

10 まず、自然数  $n$  ( $= 0, 1, 2, 3, \dots$ ) を長さが  $n$  の Lisp のリスト `'(1 1 ... 1)` で表わすことを  
11 考えます。具体的には以下のような対応になります。

12  $0 = \text{NIL} = \text{長さ } 0 \text{ のリスト}$

13  $1 = '(1) = \text{長さ } 1 \text{ のリスト}$

14  $2 = '(1 1) = \text{長さ } 2 \text{ のリスト}$

15  $3 = '(1 1 1) = \text{長さ } 3 \text{ のリスト}$

16 以下同様

17 Lisp では整数型データを扱えますから、わざわざ上のようなことを考えるのは実用上は全く無駄  
18 です。しかし、ここでは実用を離れ、理論的な遊びをします。上のようなデータ構造の上に私た  
19 ちが日頃、慣れ親しんだ整数演算を再帰関数として定義し直し、それら演算の本質（演算の成り立  
20 ち）に触れることを目的とします。

### 21 4.10.2 加算

22 自然数を上のように表すとき、二つの数の加算を行う関数 `add` は、リストについて、以下のよう  
23 な等式を満たすべきです。

24  $(\text{add nil nil}) = \text{nil} \quad // \quad 0 + 0 = 0 \text{ の意味}$

25  $(\text{add nil '(1)}) = '(1) \quad // \quad 0 + 1 = 1 \text{ の意味}$

26  $(\text{add nil '(1 1)}) = '(1 1) \quad // \quad 0 + 2 = 2 \text{ の意味}$

27  $(\text{add '(1) nil}) = '(1) \quad // \quad 1 + 0 = 1 \text{ の意味}$

28  $(\text{add '(1 1) '(1 1 1)}) = '(1 1 1 1 1) \quad // \quad 2 + 3 = 5 \text{ の意味}$

```
(defun zero (n) (if (null n) t nil))

(defun succ (n) (cons 1 n))

(defun add (x y)
  (cond ((zero y) x)
        (t      (succ (add x (cdr y))))))
```

図 4.29: 加算を表す関数 add

ところで、以下は、加算  $+$  の再帰的な定義を与える式として有名です。

$$x + 0 = x$$

$$x + (y + 1) = (x + y) + 1$$

- 1 これら等式の中置記法  $x + y$  を前置記法  $\text{add}(x, y)$  に置き換え、 $x + 1$  を  $\text{succ}(x)$  に置き換える
- 2 と以下のように変形されます。

$$\text{add}(x, 0) = x$$

$$\text{add}(x, (y + 1)) = \text{succ}(\text{add}(x, y))$$

さらに左辺の  $y + 1$  を  $y$  に、右辺の  $y$  を  $y - 1$  に置き換えます。

$$\text{add}(x, 0) = x$$

$$\text{add}(x, y) = \text{succ}(\text{add}(x, y - 1)) \quad \text{ただし } y > 0$$

- 3 上の二つの等式は  $\text{add}(x, y)$  について「もし  $y$  が 0 ならば  $\text{add}(x, y)$  の値は  $x$  であり、さもな
- 4 く  $\text{add}(x, y)$  の値は  $\text{succ}(\text{add}(x, y - 1))$  であることを意味しています。
- 5 上の等式を満たす関数  $\text{add}$  を 補助的な関数  $\text{succ}$ 、 $\text{zero}$  と共に図 4.29 のように定義します。こ
- 6 こに図の最下行の  $(\text{cdr } y)$  が  $y - 1$  を表していることに注意してください。

- 7 考察 以下の関数呼び出しの実行の様子を図や表を用いて表しなさい。

8     `> (add '(1 1 1) '(1 1 1))`

### 9 4.10.3 乗算

- 10 二つの数の乗算を行う関数  $\text{mult}$  は、リストについて以下のような等式を満たすべきです。

11     `(mult nil nil) = nil` //  $0 \times 0 = 0$  の意味

```
(defun mult (x y)
  (cond ((zero y) nil)
        (t      (add (mult x (cdr y)) x))))
```

図 4.30: 乗算を表す関数 mult

```
(defun power (x y)
  (cond ((zero y) '(1))
        (t      (mult (power x (cdr y)) x))))
```

図 4.31: べき乗を表す関数 power

```
1      (mult nil '(1)) = nil // 0 × 1 = 0 の意味
2      (mult '(1) '(1 1)) = '(1 1) // 1 × 2 = 2 の意味
3      (mult '(1) nil) = nil // 1 × 0 = 0 の意味
4      (mult '(1 1) '(1 1 1)) = '(1 1 1 1 1 1) // 2 × 3 = 6 の意味
```

このような関係を満たす関数 mult を、add 同様に Lisp で再帰を用いて定義するには、mult に関する以下の再帰的を用います。

$$\begin{aligned} \text{mult}(x, 0) &= 0 \\ \text{mult}(x, y) &= \text{add}(\text{mult}(x, y-1), x) \quad \text{ただし } y > 0 \end{aligned}$$

- 5 これは Lisp では図 4.30 のようになります。
- 6 考察 以下の関数呼び出しの実行の様子を図や表を用いて表しなさい。
- ```
7      > (mult '(1 1 1) '(1 1 1))
```

#### 8 4.10.4 べき乗

- 9 乗算を用いてべき乗  $x^y$  を定義しましょう。
- べき乗を関数 power とするとき、power に関する再帰式は以下の通りでした。

$$\begin{aligned} \text{power}(x, 0) &= 1 \\ \text{power}(x, y) &= \text{mult}(\text{power}(x, y-1), x) \quad \text{ただし } y > 0 \end{aligned}$$

- 10 これを Lisp のプログラムに直せば図 4.31 の通りです。

```
(defun fact (x)
  (cond ((zero (cdr x)) '(1))
        (t (mult (fact (cdr x)) x))))
```

図 4.32: 階乗を表す関数 fact

```
(defun addmember (e s)
  (cond ((member e s) s)
        (t (cons e s))))
```

図 4.33: 集合への要素の追加を行う関数 addmember (注: 要素は重複しない)

### 1 4.10.5 階乗

階乗の再帰的定義は以下の通りです。

$$\begin{aligned} \text{fact}(1) &= 1 \\ \text{fact}(x) &= \text{mult}(\text{fact}(x-1), x) \quad \text{ただし } x > 1 \end{aligned}$$

2 これを Lisp のプログラムに直せば図 4.32 の通りです。

## 3 4.11 集合をリストで表す

4 この節の例題では集合をリストを用いて表します。たとえば集合  $\{\text{yama}, \text{kawa}, \text{umi}\}$  をリスト  
 5 '(yama kawa umi) で表すこととします。ただし、要素がリストに現れる順序は任意とし、たとえ  
 6 ばリスト '(yama kawa umi) とリスト '(umi yama kawa) は同じ集合を表すと考えます。ただし簡  
 7 単のため、集合の要素はアトムに限定し、リストなどの非アトムを要素とする集合は 4.11.5 節ま  
 8 で考えません。また、要素の重複はないとします。たとえば '(yama kawa yama) は集合を表すリ  
 9 ストとは見なしません。

10 このような集合について、要素の追加、包含関係、集合積、集合和等の演算を Lisp で定義しま  
 11 しょう。

### 12 4.11.1 要素の追加

13 まず集合 (つまりリスト)  $S$  に要素  $e$  を追加することを考えましょう。もし  $e$  が既に  $S$  に含ま  
 14 れているならば、 $e$  を  $S$  に追加する必要はありません。さもなくば、実際に  $e$  を  $S$  に追加します。  
 15 この関数を addmember とし、図 4.33 のように定義します。ここに member は 4.9.1 節に定義した  
 16 通りです。関数の評価例は以下の通りです。

```
(defun subset (s1 s2)
  (cond ((null s1) t)
        (t (and (member (car s1) s2) (subset (cdr s1) s2)))))
```

図 4.34: 部分集合か否かを判定する関数 subset (注: 等しい場合は真とする)

```
1 > (addmember 'kawa '(yama umi))
2 (kawa yama umi)
3 > (addmember 'kawa '(yama kawa))
4 (yama kawa)
5 >
```

#### 4.11.2 部分集合

集合  $S_1$ 、 $S_2$  について、 $S_1 \subseteq S_2$  が成り立つ否かの判定は次のように考えます。もし、 $S_1$  が空集合ならば、明らかに  $S_1 \subseteq S_2$  が成り立ちます。 $S_1$  が空集合でないならば、 $S_1$  の中の適当な要素  $e$  について  $e \in S_2$  が成り立ち、かつ  $S_1 - \{e\} \subseteq S_2$  が成り立つならば、 $S_1 \subseteq S_2$  が成り立ちます。

さて、関係  $S_1 \subseteq S_2$  を関数呼び出し (`subset  $S_1$   $S_2$` ) とするとき、関数 `subset` の定義は図 4.34 の通りです。関数の評価例は以下の通りです。

```
12 > (subset '(1 2 3) '(1 2 3 5))
13 T
14 > (subset '(1 kawa) '(1 kawa))
15 T
16 > (subset '(2 kawa) '(1 umi kawa))
17 NIL
18 >
```

#### 4.11.3 和集合

集合  $S_1$ 、 $S_2$  について、 $S_1 \cup S_2 = \{e \mid e \in S_1 \text{ または } e \in S_2\}$  を  $S_1$  と  $S_2$  の和集合と呼びます。これを Lisp で (`cup  $S_1$   $S_2$` ) と表すとき、以下のような関数評価が可能であるべきです。

```
22 > (cup '(1 2 3) '(1 3 4))
23 (1 2 3 4) (あるいは (2 1 3 4), (2 3 1 4), などなど、要素の出現順は不同でよい)
24
25 > (cup nil '(1 2 3))
26 (1 2 3) (あるいは (2 1 3), (3 2 1), などなど、要素の出現順は不同でよい)
27 T
28 > (cup '(kawa umi) '(sima yama))
29 (kawa umi sima yama) (あるいは... 以下同上)
30 >
```

```
(defun cup (s1 s2)
  (cond ((null s1) ?)
        (t ?)))
```

図 4.35: 和集合を求める関数 cup

```
(defun cap (s1 s2)
  (cond ((null s1) ?)
        ((member (car s1) s2) ?)
        (t ?)))
```

図 4.36: 積集合を求める関数 cap

- 1 考察 このような評価結果を与える関数 cup を図 4.35 のように定義した。図中の ? の部分に適切  
 2 な式を埋めなさい。なお、以下の関係を参考にしなさい。
- 3 1.  $S_1 \cup S_2$  は、もし  $S_1$  が空集合ならば、 $S_2$  に等しい。
- 4 2.  $S_1 \cup S_2$  は、もし要素  $e$  が  $S_1$  に含まれるならば、 $\{e\} \cup ((S_1 - \{e\}) \cup S_2)$  に等しい (これは  
 5  $(S_1 - \{e\}) \cup S_2$  に `addmember` を用いて  $e$  を追加したものである)。

#### 6 4.11.4 積集合

- 7 集合  $S_1$ 、 $S_2$  について、 $S_1 \cap S_2 = \{e \mid e \in S_1 \text{ かつ } e \in S_2\}$  を  $S_1$  と  $S_2$  の積集合と呼びます。
- 8 これを Lisp で `(cap S1 S2)` と表すとき、以下のような関数評価が可能であるべきです。
- 9 > (cap '(1 2 3) '(1 3 4))  
 10 (1 3) (あるいは (3 1))  
 11 > (cap '(kawa) '(umi))  
 12 NIL  
 13 > (cap '(1 3) nil)  
 14 NIL  
 15 > (cap '(1 kawa umi) '(kawa 1 umi))  
 16 (1 kawa umi)  
 17 (あるいは、(kawa 1 umi), (umi kawa 1) などなど、要素の出現順は不同でよ  
 18 い)  
 19 >

- 20 考察 このような評価結果を与える関数 cap を図 4.36 のように定義した。? の部分に適切な式を埋  
 21 めなさい。なお、以下の関係を参考にしなさい。

- 22 1.  $S_1 \cap S_2$  は、もし  $S_1$  が空集合ならば、空集合に等しい。



```

(defun f (e x)
  (cond ((null x) ?)
        (t  ?)))

(defun powerset (s)
  (cond ((null s) ?)
        (t  ?)))

```

図 4.37: ベキ集合を求める関数 powerset と補助関数 f

2.  $S_1 \cap S_2$  は、もし要素  $e$  が  $S_1$  に含まれ、かつ  $e$  が  $S_2$  にも含まれるならば、 $\{e\} \cup ((S_1 - \{e\}) \cap S_2)$  に等しい。もし  $S_1$  に要素  $e$  が含まれ、かつ  $e$  が  $S_2$  に含まれないならば、 $(S_1 - \{e\}) \cap S_2$  に等しい。

#### 4.11.5 ベキ集合

集合  $S$  について、 $2^S = \{S' \mid S' \subseteq S\}$  を  $S$  のベキ集合と呼びます。 $S$  のベキ集合は、 $S$  の部分集合の集合です。

これを Lisp で (powerset  $S$ ) と表すとき、以下のような関数評価が可能であるべきです。

```

> (powerset '(1 2 3))
(NIL (1) (2) (3) (1 2) (1 3) (2 3) (1 2 3))    (ただし順不同でよい)
> (powerset nil)
(NIL)
> (powerset '(kawa))
(NIL (KAWA))
>

```

**考察** このような評価結果を与える関数 powerset を図 4.37 のように定義した。? の部分に適切な式を埋めなさい。なお、上の関数 f は補助関数であり、アトム  $e$  を第 1 引数<sup>21</sup>とし、集合の集合 (実際にはリストのリスト)  $X$  を第 2 引数とする関数であり、 $e$  を  $X$  に含まれる各集合  $S$  に追加した集合の集合  $f(e, X) = \{\{e\} \cup S \mid S \in X\}$  を関数値とする。インタプリタ上では、たとえば、以下のように動作するものとする。

```

> (f 1 '((2) (3)))
((1 2) (1 3))
> (f 'yama '())
((yama))
> (f 'yama '())
NIL

```

このような  $f$  を用いて、ベキ集合  $2^S$  では次の関係が成り立つ。

<sup>21</sup>実際には  $e$  はアトムでなくても問題なく、リストでもよいのだが、話を簡単にするためにアトムと仮定する。

1.  $2^{\emptyset} = \{\emptyset\}$ 。
  2.  $2^S = 2^{S-\{e\}} \cup f(e, 2^{S-\{e\}})$ 。ただし、 $e$  は集合  $S$  の要素とする。
- 上の関係式 1. において、集合の集合  $\{\emptyset\}$  をリスト ' (nil) で置換し、関係式 2. において、 $e$ 、 $S - \{e\}$  をそれぞれ (car S)、(cdr S) で置換すればよい。

## 4.12 より進んだ内容

この節では、まだ紹介していない Lisp の機能の中で重要と思われる事項を解説します。最近のプログラミング言語にはこれらの事項を積極的に取り入れているものが多く見受けられます。これらが次世代のプログラミング言語で重要になっていく可能性があります。

### 4.12.1 無名関数

Lisp では、関数もデータの一種です。データとしての関数を作るには、以下のように、lambda 式を用います<sup>22</sup>。

```
(lambda (v1 ... vn) e)
```

これは、defun による関数定義の defun とそれに続く関数名を lambda に置き換えたものです。たとえば以下では、引数に 1 を加える関数をデータとして作り、変数 vinc に代入します。

```
> (setq vinc (lambda (x) (+ x 1)))
(LAMBDA-CLOSURE () () (X) (+ X 1))
>
```

データとして保持されているこの関数を実行するには、関数 funcall を以下のように実行します。

```
(funcall f e1 ... en)
```

ここに、 $f$  はデータとしての関数、 $e_1 \dots e_n$  は実引数の並びです。たとえば、上に定義した変数 vinc を用いて

```
> (funcall vinc 3)
4
>
```

上の例では、vinc という名前の関数を実行しているわけではありません。vinc はデータとしての関数を保持している変数であり、関数自体には特定の名前が付いてないことに注意してください。その意味で、このような関数を無名関数 (anonymous function, unnamed function) と呼びます。

<sup>22</sup>ラムダ入 という記号は、ラムダ算術 (lambda calculus) と呼ばれる関数型プログラミングの基礎理論から取られたものです。

逆に、名前の付いた関数から関数本体をデータとして取り出すことも可能です。これには Lisp の関数 `function` を用います。たとえば以下の通りです。

```

3    > (defun dec (x) (- x 1))
4    dec
5    > (funcall (function dec) 10)
6    9
7    > (setq vdec (function dec))
8    (LAMBDA-BLOCK (X) (- X 1))
9    > (funcall vdec 8)
10   7
11   >

```

#### 4.12.2 高階関数

Lisp で扱うデータや関数に以下のようにレベルを設定してみます。

レベル 0 ... 数値や文字列、リストなどの、通常、私たちがデータと認識している基本データ型や構造型データ

レベル 1 ... レベル 0 のデータを引数や戻り値とし、それらデータを加工する関数。通常の関数

レベル 2 ... レベル 1 以下のデータ（データ、関数）を引数や戻り値とし、それらデータを加工する関数。

レベル 3 ... レベル 2 以下の... 以下同様

一般にレベル 2 以上の関数を高階関数 (higher-order function) と呼びます。関数をデータとして扱うことのできる Lisp は、高階関数を取り扱うことが可能です。たとえば、任意に与えれた関数  $f(x)$  の三つの関数値  $f(0)$ 、 $f(1)$ 、 $f(2)$  の最大値を求める関数 `maxfval` は以下のように定義し、使用することができます。

```

24    > (defun maxfval (f) (max (funcall f 0) (funcall f 1) (funcall f 2)))
25    MAXFVAL
26    > (setq f1 (lambda (x) (* x x)))
27    (LAMBDA-CLOSURE () () () (X) (* X X))
28    > (maxfval f1)
29    4
30    > (defun f2 (x) (* x (- x 1)))
31    F2
32    > (maxfval (function f2))
33    2
34    >

```

ここに、`f1` は変数、`f2` は関数として定義しています。

関数を用いて関数を作ることも可能です。たとえば、数値  $n$  を引数として受け取り、関数  $x(x-n)$  を戻り値とする関数 `makefunction` は以下のように定義し、使用できます。

```

1  > (defun makefunction (n) (lambda (x) (* x (- x n))))
2  MAKEFUNCTION
3  > (setq f3 (makefunction 3))
4  (LAMBDA-CLOSURE ((N 3)) () (MAKFUNCTION BLOCK #<@10108e38>) (X) (* X (- X N)))
5  > (funcall f3 0)
6  0
7  > (funcall f3 1)
8  -2
9  > (maxfval f3)
10 0
11 >

```

12 このように、通常のデータと同様に取り扱うことのできる関数を第1級関数 (first-class function)  
 13 と呼んでおり、関数型言語には必須の機能と考えられています。また通常のデータと第1級関数を  
 14 含めて、広い意味で、第1級オブジェクト (first-class object) と呼んでいます。

15 Lispの組み込み関数 `mapcar` は第1級関数の例として著名です。これは、引数として関数とリ  
 16 ストを受け取り、その関数をリストの全ての要素に順番に適用し、結果をリストとして返す関数で  
 17 す。たとえば以下のように利用します。

```

18 > (setq succ (lambda (x) (+ x 1)))
19 (LAMBDA-CLOSURE () () (X) (+ X 1))
20 > (mapcar succ '(1 2 3))
21 (2 3 4)
22 > (defun succ2 (x) (+ x 2))
23 succ2
24 > (mapcar (function succ2) '(3 4 5))
25 (5 6 7)
26 >

```

### 27 4.12.3 リストの破壊的操作

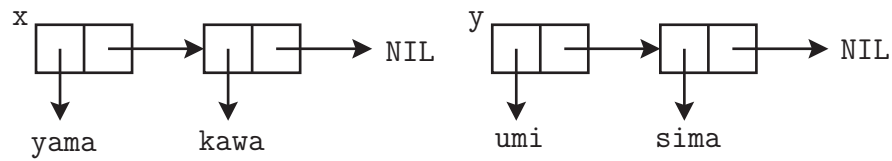
28 再度 `append` を検討します。以下の入力は、変数 `x`, `y` にリストを代入し、変数 `z` にそれを  
 29 `append` したリストを代入するものです。

```

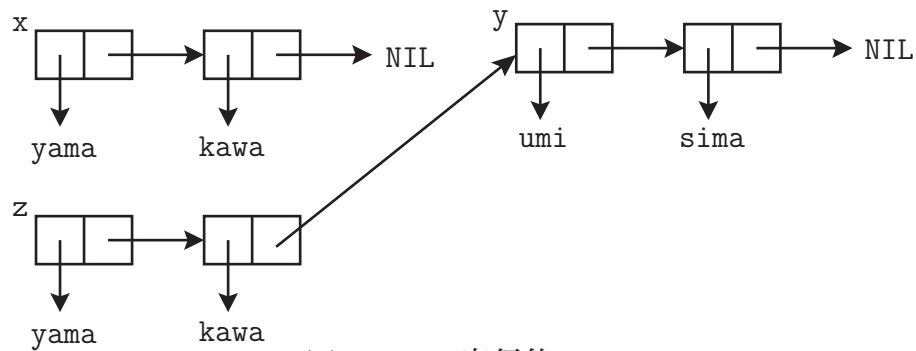
30 > (setq x '(1 2))
31 (1 2)
32 > (setq y '(3 4))
33 (3 4)
34 > (setq z (append x y))
35 (1 2 3 4)
36 > x
37 (1 2)
38 > y
39 (3 4)
40 >

```

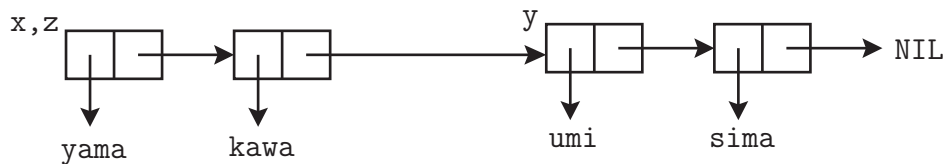
41 `append` 実行の前後でのリストの各セルの構造を図式化したものが図 4.38 (a)、(b) です。`append`  
 42 はリストを接続する関数ですが、しかし実際に `x` と `y` のリストを接続する訳ではありません。x



(a) append/nconc 実行前



(b) append 実行後



(c) nconc 実行後

図 4.38: append/nconc の実行とリストの構造変化

- 1 のリストのコピーを作り、そのコピーの末端が *y* のリストを指すように作るのです。これは一見  
 2 すると無駄なコピーをしているように思えますが、しかしコピーは *x* のリストを破壊しないとい  
 3 う利点があります。実際、`append` を実行した後の *x*、*y* の値は実行前と変わりません。  
 4 これに対して、`append` と類似した関数 `nconc` は *x* のリストを破壊します。以下は、上の入力  
 5 と同じことを `append` を `nconc` に変えて実行したものです。

```

6 > (setq x '(1 2))
7 (1 2)
8 > (setq y '(3 4))
9 (3 4)
10 > (setq z (nconc x y))
11 (1 2 3 4)
12 > x
13 (1 2 3 4)
```

```

1    > y
2    (3 4)
3    >

```

4 `nconc` 実行の後のリストの構造を図式化したものが図 4.38 (c) です。この場合、`x` のリストの末端  
 5 のポインタを `NIL` から `y` のリストの先頭へ付け替えています。`z` には接続したリストが代入され  
 6 ましたが、`x` のリストは破壊され、`nconc` 実行前とは変わってしまいました。もし `x` のリストを  
 7 別の用途にも利用したいと考えていたならば、この破壊は致命的です。しかし、不要なコピーがあ  
 8 りませんから、`nconc` は `append` に比べ高速に実行でき、またメモリの使用量も抑えられる利点  
 9 があります。

10 Lisp には他にも、`rplaca`、`rplacd` などの破壊的なリスト操作関数がいくつか定義されていま  
 11 す。これらはいずれもプログラムを高速化し、メモリ使用効率を高めるなどの利点を持ちますが、  
 12 関数型プログラミングのスタイルと相容れない側面がありますから、使用には十分注意してくだ  
 13 さい。

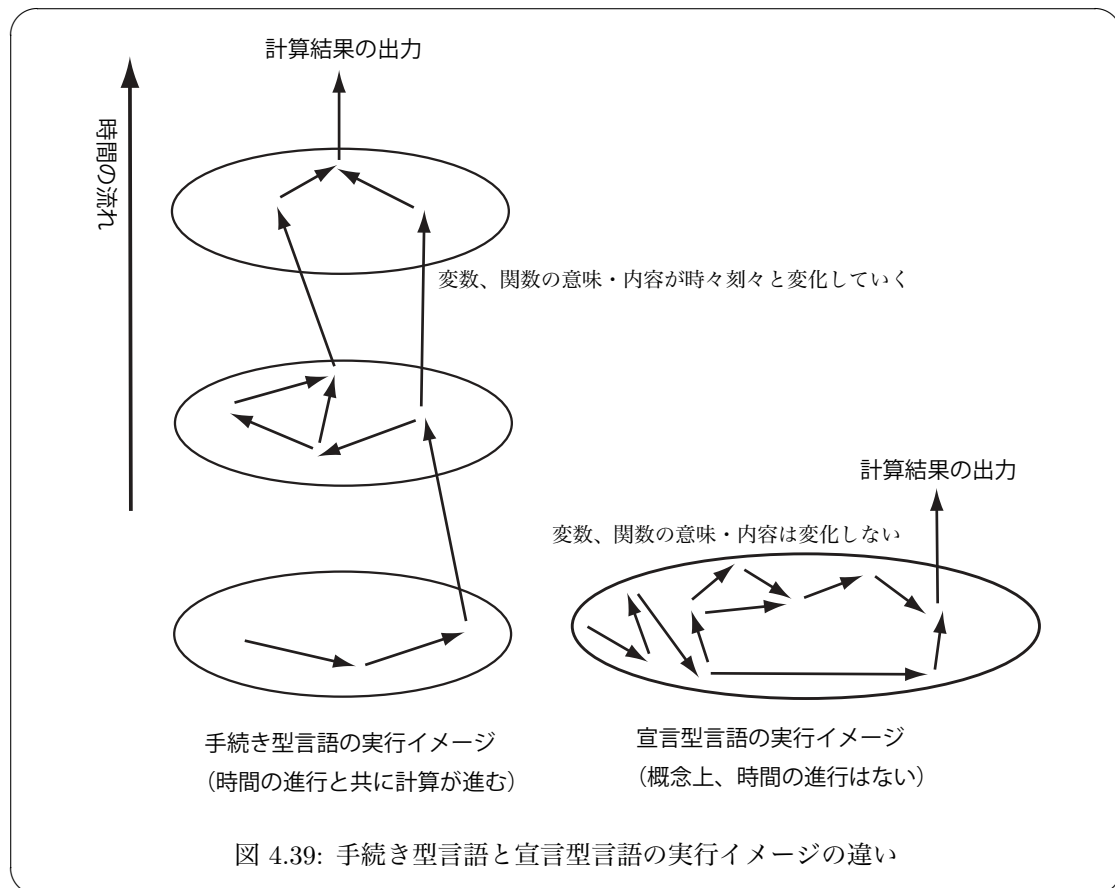
#### 14 4.12.4 参照透過性

15 上に示した `append` の破壊的操作で分かるように、データを更新していく操作（上の例ではポイ  
 16 ントを付け替える操作）はプログラマが意図しない状態変化を招く危険があり、好ましいものでは  
 17 ありません。

18 プログラムの実行において内部状態（変数の値など）が不可逆に更新されることを副作用（side  
 19 effect）と呼びます。C、C++などの手続き型言語での通常のプログラムの実行は、この副作用を  
 20 連続的に繰り返し、計算結果を求める方法です。よって、たとえばひとつの変数に着目するなら  
 21 ば、その変数の値は時間とともに変化していきますから、プログラマはその時間変化を正確に理解  
 22 せねばなりません。変数だけではなく、関数においても、もし関数が内部状態（たとえば C/C++  
 23 の `static` 変数など）を持つならば、ある時点で呼び出した関数の動作と別の時点で呼び出した関  
 24 数の動作は異なるかもしれず、時間変化、状況変化を理解する必要があります。この種の理解はプロ  
 25 グラムの可読性を落とすため、潜在的なバグにつながります。

26 これに対して、関数型言語では原則として副作用を用いずに計算を進めていきます。よって変数  
 27 の値は不変であり、関数の動作内容も時間に依存しません。図 4.39 にそのイメージの違いを示しま  
 28 す。このように変数、関数の内容が時間に依らず一定であることを参照透明（参照透過、Referential  
 29 transparent）であると言います。この性質が成り立つ場合、プログラムの可読性が高まるため、バ  
 30 グを防止しやすいと考えられています。

31 関数型言語ではプログラムを数学的に（宣言的に）記述するため、参照透明性が担保しやすい言  
 32 語になっています。対して手続き型言語では本質的に状態変化で計算を進めていくため、参照透明  
 33 性が得にくい性質があります。しかし 3 章で見たように、手続き型言語であっても再帰呼び出しを



- 1 意識したプログラミングを行うことで参照透明性を高めることが可能です。逆に関数型言語であつ
- 2 ても、4.7 節のような機能を用いれば、参照透明性の低いプログラミングに陥ってしまいます。

## 第5章 Prologによるプログラミング

### 5.1 Prologインタプリタのダウンロード

この授業の演習/レポートで想定する Prolog 処理系は Windows 用 SWI-Prolog です。ホームページは

<http://www.swi-prolog.org/>

です。このページは 2014 年 6 月 16 日現在、図 5.1 のようになっています。なお、レイアウトはしばしば変更されていますので、みなさんがアクセスしたときにはレイアウトが変わっているかもしれません<sup>1</sup>。このページの “Download” のメニューから “SWI-Prolog” を選ぶと、次のページ（図 5.2）へ移動します。このページの “Stable release” をクリックすると、OS を選択するページ（図 5.3）へ移動します。そのページで、たとえば “SWI-Prolog ... for Windows XP/Vista/7/8” などの自分の PC に適するものをクリックすると、インストールプログラムがダウンロードされます。

そのインストールプログラムを実行し、質問に全てデフォルトのまま答えれば、SWI-Prolog のインストールは数分で終了です。なお、インストールの途中で勝手に SWI-Prolog の処理系が起動されたりしますが、慌てず、そのまま数分間放置してください。この処理系のサイズは数十 MB ですからノート PC のハードディスクを圧迫することはないはずです。

処理系と関連ファイルはデフォルトでは ProgramFile フォルダの中に pl という名称のフォルダとして生成されます。そのフォルダの bin の中の plwin.exe が Prolog インタプリタです。

### 5.2 サザエさん一家

Prolog を紹介する (日本語の) 資料の多くは、サザエさん一家を例題に解説しています。実際、この例題は Prolog の初歩的な動作を学ぶことに適しています。この資料でもサザエさん一家を通して Prolog の導入を行います。

#### 5.2.1 親子関係の事実

サザエさん一家では以下の事実が成り立ちます。

---

<sup>1</sup>2016 年 6 月の時点で大きな変更は確認できませんでした。



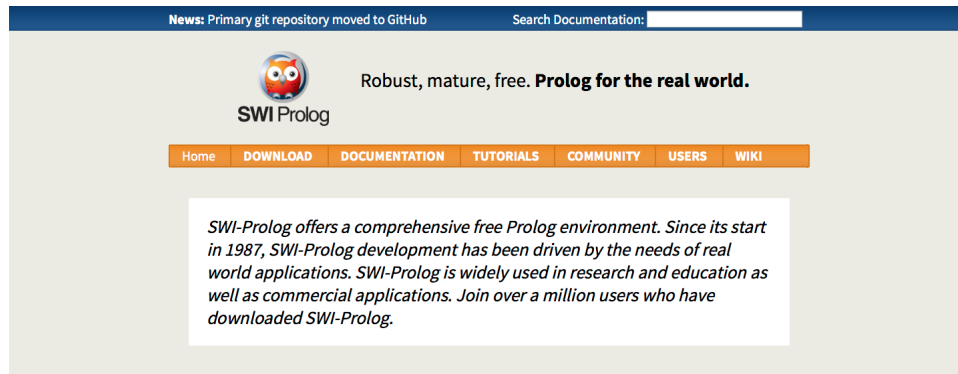


図 5.1: SWI-prolog のホームページ

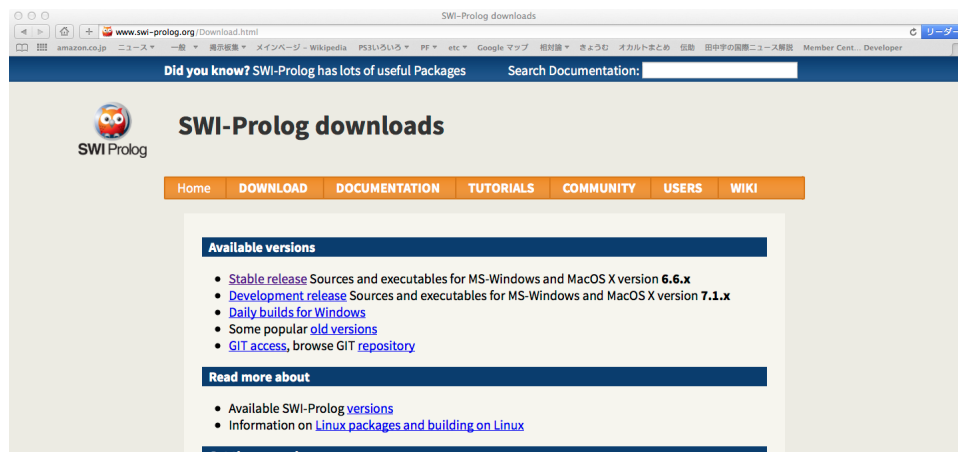


図 5.2: Download のページ

- 1 波平はサザエの親である。
- 2 波平はカツオの親である。
- 3 波平はワカメの親である。
- 4 フネはサザエの親である。
- 5 フネはカツオの親である。
- 6 サザエはタラの親である。
- 7 今、「X は Y の親である」という関係を述語式 `parent(X,Y)` と表すとき、上記の事実は以下の Prolog
- 8 プログラムとして表現できます。
- 9 `parent(namihei,sazae).`
- 10 `parent(namihei,katuo).`
- 11 `parent(namihei,wakame).`



図 5.3: Stable release のページ

```

1 parent(fune,sazae).
2 parent(fune,katuo).
3 parent(sazae,tarao).

```

4 ここに英小文字で始まる名前 `namihei`, `sazae`, `katuo`, `wakame`, `fune` はシンボル<sup>2</sup>を表します。

5 これに対して英大文字で始まる名前は変数 (variable) を表します。変数については後ほど述べま

6 す。一般に

7 述語名 (引数リスト)。

8 という形式の式を事実 (fact) または単位節 (unit clause) と呼びます。上の例題プログラムの場合、

9 サザエさん一家の親子関係について、まさに事実を表現している訳です。

## 10 5.2.2 プロンプトとプログラムのファイル入力

11 さて、上の事実群を Prolog インタプリタに入力してみましょう。

12 まず、SWI-Prolog インタプリタを起動した直後のインタプリタのウィンドウは以下の通りです。

<sup>2</sup>シンボルを文字定数と呼ぶことがありますが、C/C++の文字定数 (シングルクォーテーションで囲まれた一文字) と紛らわしいので、ここではシンボルと呼ぶことにします。Lisp のシンボル (4.8.1 節参照) とほぼ同じものと考えてください。

```

1 Welcome to SWI-Prolog
2 Copyright ...
3 SWI-Prolog comes with ...
4 ...
5 ?-

```

6 上の “?-” が Prolog インタプリタのプロンプトです。プロンプトの後に様々な入力を行いながら  
 7 プログラムを開発・実行していくことは Lisp と変わりません。なお、Prolog では、全ての入力は  
 8 論理式の形で行われ、入力の末尾はピリオド ‘.’ でなければなりません。慣れないうちは、よくピ  
 9 リオドを入力し忘れるので注意してください。インタプリタを終了させる<sup>3</sup>には、

```
10 ?- halt.
```

11 と入力します。“halt” は日本語で「止まる」の意味です。

12 これ以降、SWI-Prolog インタプリタの動作を順次説明していきます。講義テキストの説明はで  
 13 きるだけ最新のバージョンに沿うように更新しているつもりですが、みなさんがダウンロードした  
 14 インタプリタの動作がこの講義テキストの説明とやや異なる場合があるかもしれませんので、了解  
 15 してください。

16 プログラムをインタプリタに入力する方法は三つあります。

17 ひとつは、あらかじめテキストファイルに作っておいたプログラムをインタプリタに読み込ませ  
 18 る方法です。サフィックスが .pl になっているファイルは OS によって SWI-Prolog のファイルと  
 19 認識されます（そのようにインストールされています）。そこで、そのファイルをダブル・クリッ  
 20 クして開くとインタプリタが自動的に起動し、かつファイルの内容を読み込みます。これが最も簡  
 21 単なインタプリタへの入力方法です。

22 もう一つは述語 `consult` を用いてインタプリタに読み込む方法です。これは、Lisp の関数 `load`  
 23 を用いる方法に相当します。たとえば以下は Prolog のソースファイル ‘`sazae.pl`’ をインタプリ  
 24 タに読み込みます。

```

25 ?- consult('sazae.pl').
26 % sazae.pl ...
27 true.
28 ?-

```

29 インタプリタに読み込まれたプログラムを表示するには、以下のように述語 `listing` を用いま  
 30 す。引数には表示させたい述語名を指定します<sup>4</sup>。

```

31 ?- listing(parent).
32 parent(namihei,sazae).

```

<sup>3</sup>もうひとつの方法は、インタプリタのウインドウを閉じるという操作です。この方法でも特に重大な問題は生じないようですが。

<sup>4</sup>引数なしで `?-listing.` と指定すると、読み込まれた全ての述語の定義が表示されます。現時点での仕様では、システムで定義された述語の情報も表示されるため、膨大な表示となり、使い勝手が良くありません。

```

1 parent(namihei,katuo).
2 parent(namihei,wakame).
3 parent(fune,sazae).
4 parent(fune,katuo).
5 parent(sazae,tarao).
6 true.
7 ?-

```

8 これによって、その時点でインタプリタが読み込んでいるプログラムの内容を確認できます。

9 述語 `edit` は、Prolog インタプリタ内からエディタを起動します。たとえば、以下のように入力  
10 します<sup>5</sup>。

```
11 ?- edit.
```

12 プログラムを編集した後にエディタの実行を終了すると、そのプログラムがインタプリタ内に自動  
13 的に読み込まれます。

14 プログラムを入力する三番目の方法は、組み込み述語 `assert` を使って1行ずつキーボードから  
15 入力する方法です。たとえば以下の入力によって単位節 `parent(namihei,sazae).` がインタプリ  
16 タ中に読み込まれます。

```

17 ?- assert(parent(namihei,sazae)).
18 true.
19 ?-

```

20 この方法はプログラムを動的に入力していく場合に便利ですが、一般には `consult`、`make`、`edit`  
21 を用いればよいでしょう。

22 SWI-Prolog には以上述べた他にも様々な方法が用意されています。各自で調べてください。

### 23 5.2.3 問い合わせ

24 さて、上に述べたサザエさん一家の親子関係をインタプリタ内に読み込んだとしましょう。これ  
25 以降、Prolog インタプリタに対してサザエさん一家の親子関係について様々な問い合わせを行う  
26 ことができます。

27 たとえば以下は、述語式 `parent(namihei,sazae)`、`parent(fune,wakame)` の真偽の問い合  
28 せです。

```

29 ?- parent(namihei,sazae).
30 true.
31 ?- parent(fune,wakame).
32 false.
33 ?-

```

---

<sup>5</sup>または `?- edit('test.pl').` と、ファイル名を指定し、そのファイルを編集することも可能です。

1 まず、`parent(namihei,sazae)` は事実としてプログラム中に存在しますから、インタプリタは真  
2 (すなわち `true`) と返答します。

3 これに対して、`parent(fune,wakame)` はプログラム中に存在しません。本来、存在しない (知  
4 らない) 事実の真偽は不明ですが、論理プログラミングでは、「知らないことは偽」という立場を  
5 取ります<sup>6</sup> から、上のようにインタプリタは偽 (すなわち `false`) と返答します。実際のサザエさ  
6 ん一家では「フネはワカメの親である」という命題は真ですが、プログラム中に記述されてい  
7 ないため、インタプリタは偽と答えるのです。私たちの常識とは少し食い違いますが注意してく  
8 ださい。

9 次に、以下は `parent(sazae,X)` が真であるような `X` の値を求める問い合わせです。`X` は 英大文字  
10 で始まる名前であり、変数です。論理式中に変数を用いることで、その論理式を真とするような変  
11 数の値を求めることができます。

```
12 ?- parent(sazae, X).
13 X = tarao.
14 ?-
```

15 インタプリタは "`X = tarao`" ならば、`parent(sazae, X)` が真であることを回答しています。

16 この実行はある種の計算であることに注意してください。`parent(sazae, X)` は方程式に相当  
17 し、"`X = tarao`" はその解に他なりません。Prolog インタプリタは、ユーザからの「方程式を解  
18 け」という指示に対して、何らかの方法で解を求めたことになるのです。「何らかの方法」とは論  
19 理学でいう一階述語論理における線形導出法 (linear resolution method) ですが、その詳細につ  
20 いては後に述べます。

21 同様に、`namihei` についても実行してみましょう。

```
22 ?- parent(namihei,X).
23 X = sazae
```

24 上の実行では、インタプリタが "`X = sazae`" を出力したところで一時停止し、ユーザの指示を待  
25 ちます。このとき、ユーザがキーボードから改行キー (CR) を打鍵すると、インタプリタはこの  
26 問い合わせ処理を終了します。すなわち、"`X = sazae`" の後ろにピリオドを出力し、プロンプト  
27 が返ります。

```
28 ?- parent(namihei,X).
29 X = sazae . // 改行キーを打鍵した。
30 ?-
```

31 これに対して、改行キーの代わりにセミコロン・キー ';' を打鍵すると、インタプリタは "`X =`  
32 `sazae`" 以外で `parent(namihei,X)` を真にする別の `X` の探索を再開します。そして、上の例では、

<sup>6</sup> これを閉世界仮説 (closed world assumption) と呼びます。

```

1  ?- parent(namihei,X).
2  X = sazae ; // セミコロン・キーを打鍵した。
3  X = katuo

```

4 と実行が進み、インタプリタは再び一時停止し、ユーザの指示を待ちます。もしセミコロン・キー  
5 を続けて打鍵するならば、以下のように、全ての解を求めることができます。

```

6  ?- parent(namihei,X).
7  X = sazae ; // セミコロン・キーを打鍵した。
8  X = katuo ; // セミコロン・キーを打鍵した。
9  X = wakame. // ここは改行キーを打鍵していないが、上記以外の別解がないことを
10              // インタプリタが判断し、自動的に実行を終了した。
11  ?-

```

12 結局、X = sazae, katuo, wakame の三つの解を順に求めることができました。ところで、この  
13 解の順番が katuo, sazae, wakame になることはありません。解の順番はプログラムの書き方や  
14 インタプリタの実行方式に依存します。詳細は後に述べます。

15 これ以降も様々なプログラムの実行例を紹介、解説していきますが、インタプリタが一時停止す  
16 る場面では 常にセミコロン・キーを打鍵する と約束しておきます。

17 上の例では、parent の第2引数を変数にしましたが、第1引数を変数にすることも可能です。  
18 たとえば、

```

19  ?- parent(X, sazae).
20  X = namihei ;
21  X = fune.
22  ?-

```

23 全ての引数を変数にすることも可能です。その場合、その論理式を満たす全ての事実を求めるこ  
24 とができます。なお、変数名には (C/C++ の変数名のように) 英大文字で始まる英数字の列を使  
25 用できます。以下の例では、英大文字で始まる Parent と Child を変数名に採用しました。

```

26  ?- parent(Parent, Child).
27  Parent = namihei,
28  Child = sazae ;
29  Parent = namihei,
30  Child = katuo ;
31  ... 中略...
32  Parent = sazae,
33  Child = tarao.
34  ?-

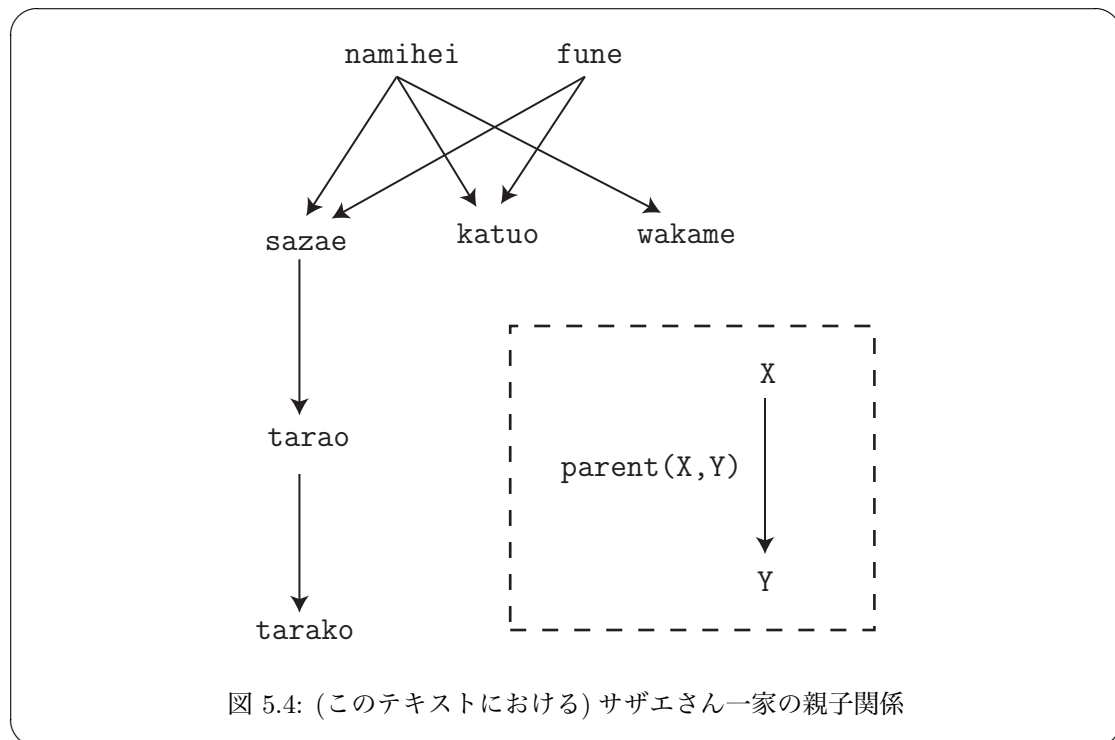
```

35 以下の問い合わせは上の例とは異なります。

```

36  ?- parent(Person, Person).
37  false.
38  ?-

```



1 この問い合わせでは第1引数と第2引数が同じであるような解を問い合わせています。しかし、そ  
 2 のような解は存在しないため、インタプリタは直ちに `false` と返答しています。この例のように、  
 3 同じ変数が複数箇所に現れる論理式を用いることである種の制約条件を表わすことができます。こ  
 4 れは Prolog における重要なプログラミングテクニックです。

5 たとえば、以下のような問い合わせも可能です。この場合、波平とフネの共通の子供を問い合わ  
 6 せています。

```

7 ?- parent(namihei, Child), parent(fune, Child).
8 Child = sazae ;
9 Child = katuo ;
10 false.
11 ?-

```

12 なお、ワカメがフネの子供である事実がインタプリタには知らされていないので、インタプリタは  
 13 `Child = wakame` を答えません。

14 扱う関係式がだんだん複雑になってきました。そこで確認しやすいように、このテキストで扱  
 15 う<sup>7</sup> サザエさん一家の親子関係を図 5.4 にまとめておきます。なお、実際のサザエさん一家には  
 16 `tarako` はいませんが、ここでは `tarao` の子供として新たに想定しておきます。後に登場します。

<sup>7</sup> このテキストではサザエさん一家の親子、親戚関係の一部のみを扱うため、マスオさんらは登場させませんでした。

#### 5.2.4 祖父母・孫関係の規則

さて、親子関係を二代に渡って調べれば、祖父母・孫関係が分かります。

いま、「 $X$  は  $Y$  の祖父母である」という関係を述語式 `grandparent(X,Y)` と表すとして、このとき、以下の論理的な関係<sup>8</sup> が成り立ちます。

• `parent(X,Y)` かつ `parent(Y,Z)` ならば、`grandparent(X,Z)` である。

これは特定の定数 (`namihei`, `sazae`, ...) に関する事実ではなく、それらの上に成り立つ規則です。この「`○○○`ならば`△△△`」という形式の規則を Prolog では `△△△ :- ○○○.` と表現します。ここに “:-” (コロン ‘:’ と等号記号 ‘=’ をつないだ文字列) は含意 (implication) を表わす Prolog の論理記号です。具体的には、上記の祖父母の関係は以下のようなプログラムとして表現します。

```
grandparent(X,Z) :- parent(X,Y),parent(Y,Z).
```

このような式を確定節 (definite clause) と呼びます。確定節の右边が条件式であり、左边がその条件が成り立つときの帰結に相当します。含意記号 `:-` の左边を確定節の頭部 (head)、右边を体部 (body) と呼びます。上の例題の確定節の左边には変数  $X$ ,  $Z$  が現れており、右边には変数  $X$ ,  $Y$ ,  $Z$  が現れています。変数  $X$ ,  $Z$  が両辺に現れていることに注意してください。既に述べたように、複数箇所に現れる変数はある種の制約条件を表わします。

上の確定節を Prolog のプログラムに加えることで、サザエさん一家の祖父母・孫関係の問い合わせについて問い合わせることができるようになります。たとえば、以下の通りです。

```
?- grandparent(namihei, X).
X = tarao ;
false.
?- grandparent(X, Y).
X = namihei,
Y = tarao ;
X = fune,
Y = tarao ;
false.
?-
```

このような解を具体的に求めるには、インタプリタはかなり複雑な処理をせねばなりませんが、その方法 (Prolog インタプリタの実行方式) の詳細は後に述べます。

#### 5.2.5 先祖・子孫関係

親子関係、祖父母・孫関係を延長すれば、先祖・子孫関係が得られます。いま、「 $X$  は  $Y$  の先祖である」という関係を述語式 `ancestor(X,Y)` と表すとき、以下が成り立ちます。

<sup>8</sup>より正確には「任意の変数  $X$ ,  $Z$  について、ある変数  $Y$  が存在し、`parent(X,Y)` かつ `parent(Y,Z)` ならば、`grandparent(X,Z)` である」と言い表すことができます。



1     • `parent(X,Y)` ならば、`ancestor(X,Y)` である。

2     • `parent(X,Y)` かつ `ancestor(Y,Z)` ならば、`ancestor(X,Z)` である。

3     これらの規則を確定節で表すと以下の通りです。

```
4     ancestor(X,Y) :- parent(X,Y).
5     ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

6     再帰定義になっていること（`ancestor` を定義するために `ancestor` を用いていること）に注意し  
7     てください。問い合わせ例は以下の通りです。

```
8     ?- ancestor(X, tarao).
9     X = sazae ;
10    X = namihei ;
11    X = fune ;
12    false.
13    ?-
```

14    もし以下の事実：

```
15    parent(tarao, tarako).
```

16    が加わったならば、以下のような問い合わせ例も可能です。

```
17    ?- ancestor(X, tarako), ancestor(fune, X).
18    X = tarao ;
19    X = sazae ;
20    false.
21    ?-
```

22    これらの解が論理的に正しいことを確認してください。

## 23    5.3    より複雑なデータ型

24    前節では、シンボル `namihei`、`sazae`、... 間の論理的な関係を種々取り扱いましたが、実用  
25    的なプログラムを書くにはシンボル以外のデータ型も必要です。この節ではそれを簡単に紹介しま  
26    す。前章の Lisp で学んだリストは、Prolog でも重要なデータ型です。リストについても触れます。

27    まだ実行方式を厳密に定義していないため、この節のプログラムの意味を十分には納得できない  
28    かもしれません。しかし、この講義テキストでは、当面、厳密な理解よりも直感的な理解を優先  
29    し、実行方式の定義を後回しにし、典型的な Prolog プログラムを紹介します。

### 5.3.1 数値

Prolog は数値計算のための言語ではありませんが、数値を全く扱えないのも不便です。そのため、ごく簡単な数値計算の仕組みが用意されています。

単純に「1 たす 2 を計算する」、「1 たす 2 が 3 であることを確認する」には、インタプリタに以下のように入力します。

```
?- X is 1+2.
X = 3.
?- 3 is 1+2.
true.
?- 1+2 is 1+2.
false.
?-
```

ここに、`is` は Prolog の組み込み述語であり、`is` の右辺の数式を評価し、もし左辺が変数ならばその変数に右辺の式の値を代入し、もし左辺が数値ならば両辺の値が等しいか否かを判定します。ただし、`1+2 is 1+2` のように、左辺が変数や数値定数以外（この例では数式）の場合には実行結果は偽 (`false`) となります。

右辺の数式に変数が現れた場合などの右辺の数式の値が計算できない場合にはエラーと見なされます。たとえば

```
?- 3 is X+2.
ERROR: is/2: Arguments are not sufficiently instantiated
?-
```

これらの性質から、`is` は代入演算子に近いと考えてよいでしょう。

`is` に類似した述語として等号記号 `=` が使用可能です。等号記号は、数値に限らず、任意のデータ間で等号記号の両辺が等しいか否かを判定する、重要な組み込み述語です。しかし等号記号は両辺を 単純に 比較するため、数式の評価は行いません。たとえば、以下の通りです。

```
?- X = 1+2.
X = 1+2.
?- 1+2 = X.
X = 1+2.
?-
```

この場合、`X = 1+2` の両辺の式の単純なパターンマッチングが行います。よって `X = 1+2` の解は `1+2` という数式そのものであり、`3` という演算結果ではありません。また、左辺と右辺の役割に違いがないので、`1+2 = X` でも同等の効果を持ちます。以下の場合には実行結果は偽 (`false`) です。

```
?- 3 = 1+2.
false.
?- 2+1 = 1+2.
false.
?-
```

```

distance(usizu,   saga,    9.2).    // (1)
distance(saga,    kanzaki, 9.3).    // (2)
distance(kanzaki, tosu,    15.7).   // (3)
distance(tosu,    hakata,  28.6).   // (4)

sum(X, X, 0).                                // (5)
sum(X, Y, D) :- distance(X, Z, D1), sum(Z, Y, D2), D is D1+D2. // (6)

```

図 5.5: 長崎本線の駅間の距離

1 何故ならば、等式  $3 = 1+2$  の左辺は数値 3 ですが、右辺は数式であり、その数式の値は計算され  
 2 ないため、両辺は等しくありません。よって `false` と評価されます。また、等式  $2+1 = 1+2$  の両  
 3 辺はともに数式ですが、形が異なるため、やはり両辺は等しくありません。

4 数値計算を含む簡単な例題プログラムを示しましょう。図 5.5 は、長崎本線の牛津駅、佐賀駅、  
 5 神崎駅、鳥栖駅、博多駅間の各距離 (Km を表す浮動小数点数値) を述語 `distance` で事実として  
 6 記述し、離れた駅間の距離を述語 `sum` で求めるプログラムです。

7 図 5.5 の (1) 行目の単位節は、「牛津駅 (`usizu`) と佐賀駅 (`saga`) の距離 (`distance`) は 9.2  
 8 Km である」ことを表わします。(2)~(4) 行目も同様です。(5) 行目の確定節は、「同じ駅 (変数  
 9 `X`) 間の距離の合計 (`sum`) は 0 Km であることを表わします。(6) 行目は、「`X` 駅と `Z` 駅の距離を  
 10 `D1` とし、`Z` 駅から `Y` 駅までの距離の合計を `D2` とし、`D` を `D1+D2` とするならば、`X` 駅から `Y` 駅ま  
 11 での距離の合計は `D` である」ことを表わします。これらの意味が、正しい距離の計算になってい  
 12 ることを確かめてください。また、(6) 行目の確定節が、`sum()` に関する再帰定義になっているこ  
 13 とに注意してください。

14 この定義の下、たとえば、佐賀駅と鳥栖駅間の距離を求めるには、以下のように問い合わせ  
 15 ます。

```

16 ?- sum(saga, tosu, D).
17 D = 25.0 .
18 ?-

```

19 なお、この問い合わせの解は唯一なので、セミコロン・キーは打鍵していません。仮に打鍵して  
 20 も、別解は見つかりません。

### 21 5.3.2 リスト

22 Lisp と同様に Prolog でもリストを扱うことができます。リストは Lisp の中核をなすデータ型  
 23 ですが、Prolog においても非常に重要なデータ型です。ただし、Prolog は関数型言語では

表 5.1: Lisp と Prolog のリストの表記法

|        | Lisp の表記法                        | Prolog の表記法                         |
|--------|----------------------------------|-------------------------------------|
| 空リスト   | nil または ()                       | []                                  |
| ドット対   | (cons X Y)                       | [X Y]                               |
| car 演算 | (car L)                          | [X Y] となる X                         |
| cdr 演算 | (cdr L)                          | [X Y] となる Y                         |
| リスト    | ( $\ell_1 \ell_2 \dots \ell_n$ ) | [ $\ell_1, \ell_2, \dots, \ell_n$ ] |
| 例 1    | (cons 'a 'b)                     | [a b]                               |
| 例 2    | (cons 'a (cons 'b ())) = '(a b)  | [a [b []]] = [a, b]                 |
| 例 3    | '(a (b c) d)                     | [a, [b, c], d]                      |
| 例 4    | (car '(a b c))                   | [a, b, c] = [X Y] となる X             |

- 1 ありませんから、cons、car、cdr を使ったリスト処理はそのままでは実現できません。代わりに
- 2 Prolog ではパターンマッチングを巧妙に用いるプログラミングを行います。

### 3 Lisp と Prolog のリストの表記法

- 4 Lisp と prolog のリストは概念的には全く同じものののですが、表記法が異なります。ここでは、
- 5 Lisp と prolog のリストを対比しながら解説を行います。

- 6 表 5.1 が両者を対比した表です。

- 7 まず、空リストは、Lisp では nil または () で表わしました。これに対して Prolog では [] と
- 8 表わします。Lisp のリストの丸括弧 () は Prolog ではカギ括弧 [] に対応します。

- 9 ドット対は、Lisp では関数 cons を用いて構築しましたが、Prolog ではパターン [ | ] (カギ
- 10 括弧と縦棒) を用いて構築します。

- 11 リスト  $\ell$  が与えられたと仮定しましょう。 $\ell$  の先頭要素を得るには、Lisp では関数呼び出し
- 12 (car  $\ell$ ) を実行しました。 $\ell$  から先頭要素を取り除いたリストを得るには、関数呼び出し (cdr  $\ell$ )
- 13 を実行しました。しかし、Prolog では関数を用いることができません。その代わりに、リスト  $\ell$
- 14 と変数を含むドット対 [X|Y] とのパターンマッチングを行い、それによって求められた X の部分
- 15 がリストの car の部分に対応し、Y の部分が cdr の部分に対応します。

- 16 Lisp での一般的なリストの表記法は ( $\ell_1 \ell_2 \dots \ell_n$ ) でした (4.8.3 節参照)。ここに要素間は
- 17 空白で区切ります。これに対応して Prolog では丸括弧を角括弧に置き換え、区切り記号としてカ
- 18 ンマ ‘,’ を使い、[ $\ell_1, \ell_2, \dots, \ell_n$ ] と表わします。

- 19 以上が Lisp と Prolog におけるリスト表記の違いです。このような説明だけでは要領を得ない
- 20 かもしれませんが、後の例題プログラムを見れば明らかになるはずです。また、表 5.1 には例 1～
- 21 例 4 に具体的な対比例を示していますので、参考にしてください。

```

member(X, [Y|Z]) :- X=Y.           // (1)
member(X, [Y|Z]) :- member(X, Z). // (2)
または
member(X, [X|Z]).                  // (3)
member(X, [Y|Z]) :- member(X, Z). // (4)
または
member(X, [X|_]).                  // (5)
member(X, [_|Z]) :- member(X, Z). // (6)

```

図 5.6: 所属判定

## 1 所属判定

2 リストを用いる Prolog プログラムの最初の例題として、4.9.1 節で取り上げた所属判定問題を  
3 Prolog でも検討しましょう。

4 述語式 `member(b, ℓ)` は、リスト  $\ell = [a_1, a_2, \dots, a_n]$  が与えられたときに、もし  $\ell$  の要素  $a_1$  か  
5 ら  $a_n$  の中に  $b$  が存在するならば真、さもなければ偽となるものとします。図 5.6 はその述語 `member`  
6 を定義する Prolog のプログラム（確定節の並び）です。

7 図 5.6 には 3 種類のプログラムを載せました。まず、確定節 (1)、(2) が最も単純な定義ですが、  
8 その直感的（論理的）な意味は以下の通りです。

9 (1) 述語式 `member( $a_1, a_2$ )` について、第 1 引数  $a_1$  が  $X$  であり、第 2 引数  $a_2$  がリスト `[Y|Z]` の  
10 形をしている（つまり、 $a_2$  が `[Y|Z]` とパターン・マッチできる）とき、もし  $X$  が  $Y$  に等し  
11 いならば、`member( $a_1, a_2$ )` は真である。しかも、このとき、 $Z$  は何であってよい。

12 (2) 述語式 `member( $a_1, a_2$ )` について、第 1 引数  $a_1$  が  $X$  であり、第 2 引数  $a_2$  がリスト `[Y|Z]` の  
13 形をしているとき、もし `member( $X, Z$ )` が真であるならば、`member( $a_1, a_2$ )` は真である。し  
14 かも、このとき、 $Y$  は何であってよい。

15 たとえば、以下のような実行が可能です。

```

16 ?- member(b, [a,b,c]).
17 true .
18 ?- member(b, [a,X,c]).
19 X = b ;
20 false.
21 ?-

```

22 最初の問い合わせ例とインタプリタの答えは明らかでしょう。2 番目の問い合わせについては、  
23 リスト `[a,X,c]` の不定箇所（2 番目の要素の位置の変数  $X$ ）がシンボル `b` であるならば述語式  
24 `member(b, [a,X,c])` が真になることをインタプリタが発見しています。この種の穴埋め問題を解  
25 くことは通常の手続き型言語や関数型言語では難しい処理です。

```

append([], X, X).                                // (1)
append([A|X], Y, [A|Z]) :- append(X, Y, Z).      // (2)

```

図 5.7: リストの接続

ところで、図 5.6 の確定節 (1) の右辺  $X=Y$  は、 $X$  と  $Y$  が同じデータであることを示しているに過ぎません。よって、確定節左辺  $\text{member}(X, [Y|Z])$  に現れる  $Y$  を  $X$  に置き換えて、左辺を  $\text{member}(X, [X|Z])$  と書き直し、右辺を削除することができます<sup>9</sup>。この改造したプログラムが図 5.6 の確定節 (3)、(4) です。この (3)、(4) の意味は (1)、(2) と等価です。

次に、図 5.6 の確定節 (3) においては、変数  $Y$  はただ一箇所にしか現れていません。変数  $Y$  の位置には任意のデータが現れてもよく、それがどのようなデータであろうとも確定節中の他の箇所に何の影響も与えません。確定節 (4) における変数  $Y$  も同様です。確定節中にただ一箇所にしか現れておらず、その  $Y$  の位置にどのようなデータが現れようとも他の箇所に何の影響も与えません。Prolog ではそのような変数を下線文字（アンダーバー、アンダースコア）‘ $\_$ ’だけで表し、「その位置の変数がそれほど重要ではない」ことを積極的に示すことができます。その流儀に従ったプログラムが、図 5.6 の確定節 (5)、(6) です。実はこれが Prolog の教科書等に広く知られている  $\text{member}$  の定義です。なお、ひとつの確定節に複数の下線文字‘ $\_$ ’が現れてもそれらは個別の独立した変数を意味すると約束されています。

さて、4.9.1 節において紹介した Lisp の関数  $\text{member}$  の定義は以下のようなものでした。

```

(defun member (x y)
  (cond ((null y)      nil)                // (L1)
        ((eq x (car y)) t)                // (L2)
        (t             (member x (cdr y))))) // (L3)

```

この関数定義では三つの場合 (L1)、(L2)、(L3) に分けて所属判定を行っています。(L2) の条件は、第 1 引数  $x$  が第 2 引数  $y$  の  $\text{car}$  部に等しいという条件であり、これは図 5.6 の確定節 (5)（あるいは (1)、(3)）の確定節に相当します。(L3) は、第 2 引数の  $\text{cdr}$  部について再帰呼び出しを行っており、これは図 5.6 の確定節 (6)（あるいは (2)、(4)）の確定節に相当します。実は Prolog のプログラムには (L1) に相当する確定節は含まれていません。というのは、(L1) は第 2 引数が空リストの場合に関数値が偽値 ( $=\text{nil}$ ) であることを示すものです。Prolog の場合、第 2 引数が空リストの場合には図 5.6 のどの確定節の左辺ともマッチしないため、(L1) のような内容をわざわざ記述する必要はないのです。

## 27 リストの接続

次に、4.9.2 節において紹介した Lisp の関数  $\text{append}$  を Prolog で定義しましょう。

<sup>9</sup>あるいは、 $X$  を  $Y$  に置き換えて、左辺を  $\text{member}(Y, [Y|Z])$  と記述することもできます。

1 Lisp の `append` は以下のような動作を行う関数です。

```
2 > (append '(yama kawa) '(umi sima))
3 (yama kawa umi sima)
4 >
```

5 関数の定義は以下の通りでした。

```
6 (defun append (x y)
7   (cond ((null x) y)
8         (t      (cons (car x) (append (cdr x) y)))))
```

9 この関数に対応する Prolog の `append` は 3 引数の以下のような動作を行う述語です。

```
10 ?- append([yama, kawa], [umi, sima], X).
11 X = [yama, kawa, umi, sima].
12 ?-
```

13 この述語の第 1 引数、第 2 引数が関数 `append` の第 1、第 2 引数に対応し、述語の第 3 引数が関数  
14 値に対応します。述語の値は、関数値とは異なり、真偽値以外の値を持ちませんから、述語には関  
15 数値に相当する引数を加える必要があります。もし関数  $f$  が  $n$  個の引数を持つならば、それに対  
16 応する述語  $p$  は一般には  $n + 1$  個の引数を持たねばなりません。上に紹介した `member` は関数値  
17 が真偽値であるため、例外的に述語の引数の数と関数の引数の数が同じになります。

18 この述語の定義は図 5.7 の通りです。これは Lisp の `append` の定義を単純に論理プログラムに  
19 読み替えたものになっています。すなわち、

20 (1) 述語式 `append( $a_1, a_2, a_3$ )` について、第 1 引数  $a_1$  が空リスト `[]` であり、第 2 引数と第 3 引  
21 数が共に  $X$  である（つまり、第 2 引数と第 3 引数が等しい）とき、`append( $a_1, a_2, a_3$ )` は真  
22 である。

23 (2) 述語式 `append( $a_1, a_2, a_3$ )` について、第 1 引数  $a_1$  がリスト `[A|X]` の形をしており、第 2 引  
24 数  $a_2$  が  $Y$  であり、第 3 引数  $a_3$  がリスト `[A|Z]` の形をしているとき、`append( $X, Y, Z$ )` が  
25 真である<sup>10</sup> ならば、`append( $a_1, a_2, a_3$ )` は真である。

26 Prolog の場合には以下のような逆方向の計算も可能です。これは Lisp にはマネのできない実行  
27 です。

```
28 ?- append([yama, kawa], X, [yama, kawa, umi, sima]).
29 X = [umi, sima].
30 ?-
```

31 このような実行が可能な理由は次節において解説します。

<sup>10</sup>すなわち、 $X$  と  $Y$  を接続したものが  $Z$  である。

4.9 節では、Lisp によるリスト処理の典型的な例題として、`member`、`append`、`flatten` の三つを取り上げました。ここでは Prolog による同等の処理として、`member`、`append` を紹介しました。Lisp と同様の処理がほぼ同様の再帰の手順で Prolog でも可能であることがわかったのではないのでしょうか。残る `flatten` についてはここでは紹介しません。`flatten` には Prolog 特有の難しさがあり、紹介には馴染まないためです。また、`member`、`append` についても実行の様子を詳細に解説した訳ではありません。そのために、次節においてインタプリタの動作を厳密に定義します。

## 5.4 Prolog インタプリタの動作方法

この節では Prolog インタプリタの動作原理を詳細に述べ、ここまで紹介してきた Prolog プログラムがインタプリタによってどのように処理されていったのかを解説します。

### 5.4.1 Prolog プログラムの構文

論理型プログラミング言語では、ホーン節 (Horn clause) と呼ばれる特殊な形式の論理式を用います。

そのホーン節で利用できる定数は、数値定数、シンボル、リストなどです。数値定数には整数と浮動小数点数があり、既に、5.3.1 節で取り扱った通りです。シンボルは、英小文字で始まる名前<sup>11</sup>であり、サザエさん一家の例で用いました。リストは数値やシンボルとは異なり、構造を持つデータですが、変数を含まないリストは定数リストであり、5.3.2 節で取り扱った通りです。

変数 (variable) は、英大文字で始まる名前<sup>12</sup>です。定数、変数、そして変数を含むデータ (たとえば、`[X|Y]` のような変数を含むリスト) をまとめて項 (term) と呼びます。

述語記号 (predicate symbol、以下単に述語とも呼びます) は、英小文字で始まる名前 (たとえば、既に出てきた `parent`、`ancestor`、`member` など) です。述語の後に、0 個以上の項からなる引数リストを付いたものを原子論理式 (atomic formula) と呼びます (たとえば `parent(X,sazae)` など)。引数の個数は述語毎に固定されていると考えるのが自然ですが、Prolog では必ずしもそれに限定しておらず、たとえば述語 `p` について `p(1)` と `p(1,2)` という異なる個数の引数を持つ原子論理式を混用することも可能です。

$A, B_1, B_2, \dots, B_n$  ( $n \geq 0$ ) を原子論理式とするとき、以下の形式：

$$A :- B_1, B_2, \dots, B_n.$$

を確定節と呼びます (たとえば、既に出てきた

`ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).`

<sup>11</sup> プログラム中に使用する名前のことを総じて識別子 (identifier) と呼ぶことがあります。

<sup>12</sup> 下線 ‘\_’ で始まる名前も変数と見なされます。たとえば `_1`、`_ABC` などです。ただし、5.3.2 節で述べたように、下線 ‘\_’ のみの変数は特殊な役割を持つので注意が必要です。



```

parent(namihei,sazae).    // (1)
parent(namihei,katuo).    // (2)
parent(namihei,wakame).   // (3)
parent(fune,sazae).       // (4)
parent(fune,katuo).       // (5)
parent(sazae,tarao).      // (6)
parent(tarao,tarako).     // (7)

grandparent(X,Z) :- parent(X,Y),parent(Y,Z). // (8)

ancestor(X,Y) :- parent(X,Y).                // (9)
ancestor(X,Z) :- parent(X,Y),ancestor(Y,Z).   // (10)

```

図 5.8: サザエさん一家の親子関係とその拡張

1 などです)。この確定節は、論理的には

2  $B_1 \wedge B_2 \wedge \dots \wedge B_n \rightarrow A$

3 と等価であり、「 $B_1$  かつ  $B_2$  かつ  $\dots$   $B_n$  が成り立つならば  $A$  が成り立つ」ことを意味します。特  
 4 に  $n = 0$  の、

5  $A :- .$

6 のような場合、これを単位節、事実と呼び、「無条件に  $A$  が成り立つ」ことを意味します。単位節  
 7 では  $:-$  を省略し、簡単に

8  $A .$

9 と略記可能です（たとえば、既に出てきたように

10 `parent(namihei,sazae).`

11 など)。Prolog のプログラムは確定節の集合です。サザエさん一家に関する確定節を全て集めたプ  
 12 ログラム例を図 5.8 に再掲します。左辺が同じ述語記号であるような確定節全体で、その述語記号  
 13 を定義していると思えます。

14 次に、 $B_1, B_2, \dots, B_n$  ( $n \geq 0$ ) を原子論理式とするとき、以下の形式：

15  $?- B_1, B_2, \dots, B_n .$

16 を 問い合わせ、質問、ゴール節 (goal clause) と呼びます（たとえば

17 `?- ancestor(X, tarako), ancestor(fune, X).`

など)。そして、Prolog における実行とは、簡単に言えば、与えられた確定節の集合の下で「問い合わせに答えること」、言い換えれば「ゴール節の論理式  $B_1$  かつ  $B_2$  かつ ...  $B_n$  が真になる証明を見つけること」です。より正確に、論理的な立場から言えば、ゴール節：

$$?- B_1, B_2, \dots, B_n.$$

は以下の論理式：

$$B_1 \wedge B_2 \wedge \dots \wedge B_n \rightarrow$$

と等価であり、さらにこれは以下の論理式と等価です。

$$\neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n$$

これは「 $B_1$  が成り立たない、または  $B_2$  が成り立たない、... または  $B_n$  が成り立たない」ことを意味しており、Prolog の実行とは、確定節の集合とゴール節が論理的に矛盾することを反駁過程 (refutation process) として証明することです。

確定節とゴール節をまとめてホーン節と呼びます。論理プログラミングは、ホーン節上に構築されたプログラミング・パラダイムであり、Prolog はその手法を言語処理系として実現したものです。

## 5.4.2 Prolog プログラムの実行

以下、Prolog インタプリタの実行について詳細に述べますが、その準備として、代入と単一化の概念を導入します。

代入

$X_1, \dots, X_k$  を変数、 $t_1, \dots, t_k$  を項とすると、以下の形式：

$$[X_1 := t_1, X_2 := t_2, \dots, X_k := t_k]$$

を、 $X_1, \dots, X_k$  をそれぞれ  $t_1, \dots, t_k$  に置き換える代入演算 (substitution operation) と呼びます。単に代入、あるいは置換演算 (replacement operation)、置換と呼ぶ場合もあります。上記の代入には  $k$  種類の代入 ( $X_1$  に  $t_1$  を代入、...、 $X_k$  に  $t_k$  を代入) が含まれていますが、代入操作は同時に一回だけ行くと約束します。よって、項  $t_i$  の中に変数  $X_j$  ( $i \neq j$ ) が含まれるときに、 $t_i$  の中の  $X_j$  をさらに  $t_j$  で置き換えるというような連鎖的な代入は行いません。

以下では、代入演算をしばしばギリシャ文字  $\theta$  を用いて表わします。すなわち以下の通りです。

$$\theta = [X_1 := t_1, X_2 := t_2, \dots, X_k := t_k]$$

- 1 この代入  $\theta$  を項  $t$  に適用するときには、 $\theta$  を  $t$  の後ろについて付けて  $t\theta$  と表すと約束します<sup>13</sup>。  
もし変数  $X_i$  に上の代入  $\theta$  を適用するならば

$$X_i\theta = X_i[X_1 := t_1, X_2 := t_2, \dots, X_k := t_k] = t_i$$

が成り立ち、 $X_i$  が  $t_i$  に置き換えられます。どの  $X_i$  とも異なる変数  $Y$  にこの代入を適用するならば、 $Y$  は置換されず、

$$Y\theta = Y[X_1 := t_1, X_2 := t_2, \dots, X_k := t_k] = Y$$

となります。任意の定数（数値やシンボル） $c$  についても

$$c\theta = c[X_1 := t_1, X_2 := t_2, \dots, X_k := t_k] = c$$

が成り立ち、 $c$  は置換されません。変数  $X_j$  を含むリスト  $\ell(X_j)$  に  $\theta$  を適用するならば、以下の  
ように、リストの中の  $X_j$  が  $t_j$  に置換されます。

$$\ell(X_j)\theta = \ell(X_j)[X_1 := t_1, X_2 := t_2, \dots, X_k := t_k] = \ell(t_j)$$

具体的な代入の例は以下の通りです。

$$\begin{aligned} \text{Person}[\text{Person} := \text{sazae}] &= \text{sazae}, \\ \text{Person}[\text{Person} := \text{Who}] &= \text{Who}, \\ \text{sazae}[\text{Person} := \text{Who}] &= \text{sazae}, \\ \text{Child}[\text{Person} := \text{Who}] &= \text{Child}, \\ 1.23[\text{Person} := \text{Who}] &= 1.23, \\ [\text{Car}|\text{Cdr}][\text{Car} := \text{yama}, \text{Cdr} := []] &= [\text{yama}|[]] \\ &= [\text{yama}], \\ [\text{Car}|\text{What}][\text{Car} := \text{yama}, \text{Cdr} := []] &= [\text{yama}|\text{What}], \\ [\text{kawa}, 1, \text{Who}, 3|\text{What}][\text{Who} := \text{yama}, \text{What} := [4, 5]] &= [\text{kawa}, 1, \text{yama}, 3|[4, 5]] \\ &= [\text{kawa}, 1, \text{yama}, 3, 4, 5] \end{aligned}$$

複数の代入  $\theta_1, \theta_2$  を項  $t$  に連続して適用することを  $t\theta_1\theta_2$  と表します。これは  $(t\theta_1)\theta_2$  の意味です。たとえば

$$\begin{aligned} [\text{Car}|\text{What}][\text{Car} := \text{yama}, \text{Cdr} := []][\text{What} := \text{kawa}] &= [\text{yama}|\text{What}][\text{What} := \text{kawa}] \\ &= [\text{yama}|\text{kawa}] \end{aligned}$$

<sup>13</sup>関数  $f$  を値  $x$  に適用するときには  $f(x)$  と記述するのが一般的です。これを前置形式 (prefix notation) と呼びます。それに対して  $(x)f$  や  $xf$  という記述法を後置形式 (postfix notation) と呼びます。関数呼び出しを記述する場合、本来はどちらの形式を選んでも大差なく、代入  $\theta$  について後置形式を用いるのはこの分野の単なる慣用にすぎません。

表 5.2: 項の単一化の例

| # | 項 $s$    | 項 $t$     | 可否 | 単一化子 $\theta$                                                                                                      |
|---|----------|-----------|----|--------------------------------------------------------------------------------------------------------------------|
| 1 | sazae    | sazae     | ○  | 任意                                                                                                                 |
| 2 | namihei  | sazae     | ×  | —                                                                                                                  |
| 3 | Person   | Who       | ○  | [Person := sazae, Who := sazae],<br>[Person := Who] *                                                              |
| 4 | Person   | sazae     | ○  | [Person := sazae] *,<br>[Person := sazae, Who := sazae] <sup>14</sup>                                              |
| 5 | 1        | 2         | ×  | —                                                                                                                  |
| 6 | [X yama] | [kawa Y]  | ○  | [X := kawa, Y := yama] * <sup>15</sup>                                                                             |
| 7 | [yama X] | [kawa Y]  | ×  | — <sup>16</sup>                                                                                                    |
| 8 | [X, a]   | [b Y]     | ○  | [X := b, Y := [a]] * <sup>17</sup>                                                                                 |
| 9 | [a X]    | [a, b, Y] | ○  | [X := [b, Z], Y := Z] * <sup>18</sup> ,<br>[X := [b, c], Y := c],<br>[X := [b, [d]], Y := [d]],<br>[X := [b, Y]] * |

\* は最汎単一化子を表す。

## 1 項の単一化

二つの項  $s$  と  $t$  について、ある代入  $\theta$  が存在し、

$$s\theta = t\theta$$

2 が成り立つならば、 $s$  と  $t$  は  $\theta$  で単一化 (unify) できる、単一化に成功すると言います。また、 $\theta$   
3 を  $s$  と  $t$  の単一化子 (unifier) と呼びます。

4 単一化とは「同一のものに変形できる」という意味です。「パターンマッチ」がこれに近い言葉  
5 であり、実はテキストのこれ以降、単一化の代わりにパターンマッチを使っても大きな誤解は生じ  
6 ません。具体的な単一化の例は表 5.2 の通りです。これらの例から単一化に関するいくつかの事実  
7 が分かります。

8 まず、同じ定数項は代入に依らず、必ず単一化できます (1 番目の例)。逆に、異なる定数項は  
9 どのような代入を用いようとも決して単一化できません (2 番目、7 番目の例)。定数項は代入を適  
10 用しても不変なので、もともと異なる項どうしは代入によって同じ項になることはありません。

<sup>14</sup>Who := sazae は不要ですが、不要なものが付加されていても単一化に問題ありません。

<sup>15</sup>何故ならば、 $s\theta = [X|yama][X := kawa, Y := yama] = [kawa|yama]$ 、 $t\theta = [kawa|Y][X := kawa, Y := yama] = [kawa|yama]$

<sup>16</sup>リストの先頭要素が異なるシンボル yama と kawa ですから、変数 X、Y をどのように置き換えても二つのリストは等しくなりません。

<sup>17</sup>何故ならば、 $s\theta = [X, a][X := b, Y := [a]] = [b, a]$ 、 $t\theta = [b|Y][X := b, Y := [a]] = [b|[a]] = [b, a]$

<sup>18</sup>何故ならば、 $s\theta = [a|X][X := [b, Z], Y := Z] = [a|[b, Z]] = [a, b, Z]$ 、 $t\theta = [a, b, Y][X := [b, Z], Y := Z] = [a, b, Z]$

二つの項が単一化できるとき、適用できる代入は唯一とは限りません。1 番目の場合、任意の代入が単一化子と見なせます。3 番目の場合、二つの項、Person と Who は共に変数なので、Person と Who に同じ項を代入すれば必ず単一化できます。一方の変数 Person を他方の変数 Who に置換することでも単一化できています。わざわざ特定の定数 sazae に置換しなくとも単一化できるのです。4 番目の場合、Person を定数 sazae に置換すれば単一化できます。しかし代入  $[Person := sazae, Who := sazae]$  には単一化に全く寄与しない代入  $[Who := sazae]$  が含まれており、その意味でこの単一化子は不必要に複雑です。9 番目の例でも様々な単一化子が表示されています。その中で、最も制限されていない（変数を不必要に定数化しない）単一化子は  $[X := [b, Z], Y := Z]$  と  $[X := [b, Y]]$  です。

#### 10 最汎単一化子

項  $s$  と  $t$  について、上に解説した意味での最も一般性の高い（必要最小限の代入しか行わない、最も制限の少ない）単一化子  $\theta$  を  $s$  と  $t$  の最汎単一化子（most general unifier）と呼びます。数学的に正確に言えば、 $s$  と  $t$  の任意の単一化子を  $\theta'$  とするとき、任意のリスト  $u$  について、

$$u\theta' = (u\theta)\sigma$$

が成り立つような代入  $\sigma$  が必ず存在するならば、上の式の  $\theta$  を  $s$  と  $t$  の最汎単一化子と呼びます。つまり、単一化子  $\theta$  と適当な代入  $\sigma$  の合成  $\theta\sigma$  によって任意の単一化子  $\theta'$  が表現できるならば、 $\theta$  を全ての単一化子の中で最も一般性の高い単一化子と見なすのです。実際、リストどうしの任意の単一化では、最汎単一化子が必ず存在することが知られています。

ただし、最汎単一化子は唯一ではありません。たとえば、表 5.2 の 9 番目の例の場合、 $\theta = [X := [b, Z], Y := Z]$  と  $\theta' = [X := [b, Y]]$  は共に最汎単一化子です。ここに、

$$\theta[Z := Y] = \theta', \quad \theta'[Y := Z] = \theta$$

が成り立ちます。つまり、最汎単一化子では変数名の付け替えの任意性は残るのです。その任意性を無視し、これらを同じ単一化子と見なせば、二つの項の間の最汎単一化子は唯一となります。

与えられた二つの項  $s$  と  $t$  について最汎単一化子を求める操作を最汎単一化（most general unification）と呼びます。

#### 19 Prolog における最汎単一化

Prolog の等号記号  $=$  は、式の両辺を最汎単一化する組み込み述語です。これを明示的に用いることで、Prolog の単一化の様子を知ることができます。たとえば以下のような実行になります。

```

1  ?- [X|a] = [b|Y].
2  X = b,
3  Y = a.
4  ?- [a|X] = [b|Y].
5  false.
6  ?- [X, a] = [b|Y].
7  X = b,
8  Y = [a].
9  ?- [a|X] = [a, b, Y].
10 X = [b, Y].
11 ?-

```

12 以下のような場合は単一化は失敗します。その理由は各自で考えてください。

```

13 ?- [X, a, Y] = [a, b, c].
14 false.
15 ?- [X|a] = [a].
16 false.
17 ?- [X] = [a,b].
18 false.
19 ?-

```

20 以下は、下線文字 ‘\_’ だけの変数を含む場合の単一化です。‘\_’ は、そこに何らかのデータが存在す  
 21 ることのみを示し、それと単一化されたデータ自体は無視されます。よって、以下のような実行と  
 22 なります。

```

23 ?- _=1.
24 true.
25 ?- [X|_]=[a,b,c].
26 X = a.
27 ?-

```

## 28 原子論理式の単一化

ここまでは二つの項の単一化を考えましたが、次にそれを二つの原子論理式の単一化へ自然に拡張します。すなわち、二つの原子論理式  $p(s_1, s_2, \dots, s_m)$  と  $q(t_1, t_2, \dots, t_n)$  が与えられたとき、ある代入  $\theta$  について、

$$p(s_1, s_2, \dots, s_m)\theta = q(t_1, t_2, \dots, t_n)\theta$$

が成り立つならば、 $p(s_1, s_2, \dots, s_m)$  と  $q(t_1, t_2, \dots, t_n)$  は  $\theta$  で単一化できると言います。原子論理式が単一化できるのは、それらに含まれる述語記号および引数が単一化できるときに限られます。よって、以下が成り立たねばなりません。

$$p = q, \quad m = n, \quad s_i\theta = t_i\theta \quad (1 \leq i \leq m)$$

29 最汎単一化についても同様の議論になります。

表 5.3: 原始論理式の最汎単一化の例

| # | 原始論理式 $s$                            | 原始論理式 $t$                            | 可否 | 最汎単一化子 $\theta$                              |
|---|--------------------------------------|--------------------------------------|----|----------------------------------------------|
| 1 | <code>parent(X,sazae)</code>         | <code>parent(namihei,Y)</code>       | ○  | $[X := namihei, Y := sazae]$                 |
| 2 | <code>parent(X,X)</code>             | <code>parent(namihei,Y)</code>       | ○  | $[X := namihei, Y := namihei]$               |
| 3 | <code>ancestor(X,sazae)</code>       | <code>parent(namihei,Y)</code>       | ×  | —                                            |
| 4 | <code>parent(fune,X)</code>          | <code>parent(namihei,Y)</code>       | ×  | —                                            |
| 5 | <code>member(X, [X Y])</code>        | <code>member(b, [b,c])</code>        | ○  | $[X := b, Y := [c]]$                         |
| 6 | <code>member(X, [X _])</code>        | <code>member(b, [b,c])</code>        | ○  | $[X := b]$ <sup>19</sup>                     |
| 7 | <code>append([A X], Y, [A Z])</code> | <code>append([a,b], [c,d], W)</code> | ○  | $[A := a, X := [b], Y := [c,d], W := [a Z]]$ |

1 原子論理式どうしの最汎単一化の例は表 5.3 の通りです。

## 2 インタプリタの実行手順

3 Prolog の実行の理論的拠り所は、一階述語論理の証明手法のひとつである線形導出法による反  
4 駁証明法です。その方法の非決定的な動作を深さ優先探索でアルゴリズム化したのが Prolog イン  
5 タプリタの実行方式です。

6 Prolog インタプリタの実行はゴール節による問い合わせからスタートします。

7 ゴール節  $?- G_1, G_2, \dots, G_k.$  が与えられたときの Prolog インタプリタの 1 回の実行ステップ  
8 は以下の通りです。

9 1. もし  $k \geq 1$  ならば（すなわち、ゴール節に少なくとも 1 個の原子論理式が含まれるならば）

10 (a) プログラム中の確定節  $A :- B_1, \dots, B_n.$  について、ゴール節最左の原子論理式  $G_1$  と  
11 確定節頭部  $A$  が単一化できる確定節を見つけます。そのような確定節が複数個ある場  
12 合には、プログラム中の最も上の行に書かれている確定節を選びます。もし全ての確定  
13 節の頭部が  $G_1$  と単一化できないならば、バックトラック <sup>20</sup> (backtracking) を行いま  
14 す（バックトラックの詳細は後述します）。

(b) 次に、上の  $G_1$  と  $A$  の最汎単一化子を  $\theta$  とします。このとき、ゴール節：

$$?- G_1, G_2, \dots, G_k.$$

を以下のゴール節：

$$?- B_1\theta, \dots, B_n\theta, G_2\theta, \dots, G_k\theta.$$

<sup>19</sup> この最汎単一化では変数  $_$  への代入も考慮せねばなりませんし、実際、代入  $[_ := [c]]$  が適用されますが、Prolog インタプリタは変数  $_$  への代入をプログラマに見せることはしません。

<sup>20</sup> 「後戻り」とも言います。

に置換します。そして置換されたゴール節について、1.に戻り、実行を継続します。なお、もし  $n = 0$  ならば（すなわち、選択された確定節が単位節であるならば）、確定節は右辺を持たないので、置換されるゴール節は以下のようになります。

$$?- G_2\theta, \dots, G_k\theta.$$

1       その結果、ゴール節中の原子論理式の数ひとつ減ることとなり、実行過程は成功に一  
2       歩近づきます（以下の2.を参照）。

3       2. もし  $k = 0$  ならば（すなわち、ゴール節中の原子論理式が消滅したならば）、この実行過程  
4       は成功しました。このとき

5       (a) 初期のゴール節に変数が含まれているならば、

6           i. その変数が実行途中の最汎単一化子によって置換された値を出力し、キーボードか  
7           らの入力待ち状態に入ります。なお、置換された値を得るために、各実行ステップ  
8           では過去に適用したを最汎単一化子別途、記録しておかねばなりません。

9           ii. キーボードから改行 (CR) が打鍵されたならば、実行を終了します。セミコロン・  
10          キー ; が打鍵されたならば、強制的にバックトラックを行います。

11       (b) 初期のゴール節に変数が含まれていないならば、ディスプレイ上に単に `true.` と出力  
12       し、実行を終了します。

13       なお、バックトラックでは、インタプリタの状態を上の実行過程のひとつ前のステップに戻しま  
14       す。そして上記 1. (a) では、まだ適用していない確定節の中で、プログラム中の最も上の行に書  
15       かれている確定節  $A:- B_1, \dots, B_n.$  の次の候補を新たに見つけ、上記 1. (b) の処理を行います。  
16       もしそのような確定節が無いならば、さらにバックトラックを行います。バックトラックによって  
17       初期状態まで戻り、初期状態のゴール節の最左の原子論理式と頭部が単一化できる確定節が見つ  
18       からないならば、ディスプレイに `false.` と出力し、プログラムの実行を終了します。

19       前節サザエさん一家の問い合わせ例のいくつかを図5.8のプログラムについて実際に実行してみ  
20       ましょう。実行状況を把握するために、例題の各行には

- 21       ● 実行ステップの深さ（0以上の整数値）
- 22       ● 現在のゴール節,
- 23       ● [単一化子によって具体化された変数の値のリスト]
- 24       ● (実行に関する注釈)



1 の順に表示します。実行ステップの深さは、実行がひとつ進む度に1増え、バックトラックの度に  
2 1減ります。これを用いてバックトラックの様子を知ることができます。

3 最初の例として以下のゴール節を検討しましょう。

0 ?- grandparent(X,Y)., []

4 まず、原子論理式 grandparent(X,Y) と頭部が単一化できる確定節は図 5.8 (8) のみですから、そ  
5 れを適用します。

(8) を適用

1 ?- parent(X,Y2),parent(Y2,Y)., []

6 ここに、上の「(8) を適用」とは「直前のゴール節の最左の原子論理式に図 5.8 の (8) の確定節  
7 を適用する」ことを縮めて述べたものです。変数 Y2 は新たに導入した変数です。(8) の右辺に現  
8 れる変数名 Y は既にゴール節中に使用されています。そこで、変数名の混乱が生じないように、(8)  
9 の右辺に現れる変数名を全て Y2 に名前替えしました。この名前替えは、実行の様子を簡単に説明  
10 するための便宜上の措置であり、実行の方式とは関係しません。実際の名前替えは Prolog インタ  
11 プリタがシステム内部で自動的に行うため、プログラマが意識する必要はありません。

12 実行はさらに以下のように続きます。

(1) を適用

2 ?- parent(sazae,Y)., [X = namihei]

(6) を適用

3 ?- ., [X = namihei, Y=tarao]

13 ここでゴール節が空になりましたから、実行は成功しました。よって、インタプリタは、変数値  
14 X = namihei、Y = tarao をディスプレイに出力します。

15 次に、もしキーボードからセミコロンのキーが打鍵されたならば、バックトラックが起きます。

バックトラック

2 ?- parent(sazae,Y)., [X = namihei] ((6) は適用不可)

16 上に記された「バックトラック」とは「ゴール節の状態を1ステップだけ直前の状態へ戻すこと」  
17 の意味です。また注釈の「((6) は適用不可)」とは、既に直前の第2ステップで(6)は適用済みで  
18 あり、この状況で(6)を再度適用することはできない<sup>21</sup>ことを注記したものです。

19 さて、上の原子論理式 parent(sazae,Y) に適用可能な確定節は(6)以外にはありませんから、  
20 上記の状況を先へ進めることはできません。そこでインタプリタはさらにバックトラックし、別の  
21 確定節の適用を試みます。

バックトラック

1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1) は適用不可)

(2) を適用

2 ?- parent(katuo,Y)., [X = namihei]

バックトラック

1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1), (2) は適用不可)

<sup>21</sup>再度の(6)の適用は同じことの繰り返しとなり、全く無意味です。

```

(3) を適用
バックトラック      2 ?- parent(wakame,Y)., [X = namihei]
(4) を適用          1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1),(2),(3) は適用不可)
(6) を適用          2 ?- parent(sazae,Y)., [X = fune]
                    3 ?- ., [X = fune, Y = tarao]

```

- 1 ここでゴール節が空になり、実行は成功しました。そこで変数値  $X = \text{fune}$ 、 $Y = \text{tarao}$  を出力し  
 2 ます。もしここでキーボードからセミコロン・キーを打鍵したならば、またバックトラックが起き  
 3 ます。

```

バックトラック      2 ?- parent(sazae,Y)., [X = fune] ((6) は適用不可)
バックトラック      1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1)~(4) は適用不可)
(5) を適用          2 ?- parent(katuo,Y)., [X = fune]
バックトラック      1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1)~(5) は適用不可)
(6) を適用          2 ?- parent(tarao,Y)., [X = sazae]
(7) を適用          3 ?- ., [X = sazae, Y = tarako]

```

- 4 ここでゴール節が空になりましたから、実行は成功です。変数値  $X = \text{sazae}$ 、 $Y = \text{tarako}$  を出力  
 5 します<sup>22</sup>。もしキーボードからセミコロン・キーを打鍵したならば、さらにバックトラックが起き  
 6 ます。

```

バックトラック      2 ?- parent(tarao,Y)., [X = sazae] ((7) は適用不可)
バックトラック      1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1)~(6) は適用不可)
(7) を適用          2 ?- parent(tarako,Y)., [X = tarao]
バックトラック      1 ?- parent(X,Y2),parent(Y2,Y)., [] ((1)~(7) は適用不可)
バックトラック      0 ?- grandparent(X,Y)., [] ((8) は適用不可)

```

バックトラック (失敗)

- 7 ここに「(失敗)」とは、初期ゴールに適用可能な規則が無い場合、実行を進めることができず、プ  
 8 ログラム実行が終了することを意味します。インタプリタは `false` と出力し、停止します。

- 9 もうひとつの例題として、5.2.5 節の二つ目の問い合わせの実行を解説しましょう。この実行は  
 10 以下のように進みます。

```

(9) を適用          0 ?- ancestor(X,tarako),ancestor(fune,X)., []
(7) を適用          1 ?- parent(X,tarako),ancestor(fune,X)., []
(9) を適用          2 ?- ancestor(fune,tarao)., [X = tarao]
バックトラック      3 ?- parent(fune,tarao)., [X = tarao]
(10) を適用         2 ?- ancestor(fune,tarao)., [X = tarao] ((9) は適用不可)
(4) を適用          3 ?- parent(fune,Y),ancestor(Y,tarao)., [X = tarao]
(9) を適用          4 ?- ancestor(sazae,tarao)., [X = tarao]
                    5 ?- parent(sazae,tarao)., [X = tarao]

```

<sup>22</sup>なお、5.2.4 節では図 5.8 (7) の規則を想定していなかったため、この出力は起きませんでした。

(6) を適用

6 ?- ., [X = tarao]

この時点でインタプリタは「X = tarao」を出力。セミコロン・キーによってバックトラック

バックトラック

5 ?- parent(sazae,tarao)., [X = tarao] ((6) は適用不可)

(10) を適用

4 ?- ancestor(sazae,tarao)., [X = tarao] ((9) は適用不可)

(6) を適用

5 ?- parent(sazae,Y),ancestor(Y,tarao)., [X = tarao]

(9) を適用

6 ?- ancestor(tarao,tarao)., [X = tarao]

バックトラック

7 ?- parent(tarao,tarao)., [X = tarao]

(10) を適用

6 ?- ancestor(tarao,tarao)., [X = tarao] ((9) は適用不可)

7 ?- parent(tarao,tarao),ancestor(tarao,tarao).,

[X = tarao]

バックトラック

6 ?- ancestor(tarao,tarao)., [X = tarao]

((9), (10) は適用不可)

バックトラック

5 ?- parent(sazae,Y),ancestor(Y,tarao)., [X = tarao]

((6) は適用不可)

バックトラック

4 ?- ancestor(sazae,tarao)., [X = tarao] ((9), (10) は適用不可)

バックトラック

3 ?- parent(fune,Y),ancestor(Y,tarao)., [X = tarao]

((4) は適用不可)

(5) を適用

4 ?- ancestor(katuo,tarao)., [X = tarao]

(9) を適用

5 ?- parent(katuo,tarao)., [X = tarao]

バックトラック

4 ?- ancestor(katuo,tarao)., [X = tarao] ((9) は適用不可)

(10) を適用

5 ?- parent(katuo,Y),ancestor(Y,tarao)., [X = tarao]

バックトラック

4 ?- ancestor(katuo,tarao)., [X = tarao] ((9), (10) は適用不可)

バックトラック

3 ?- parent(fune,Y),ancestor(Y,tarao)., [X = tarao]

((4), (5) は適用不可)

バックトラック

2 ?- ancestor(fune,tarao)., [X = tarao] ((9), (10) は適用不可)

バックトラック

1 ?- parent(X,tarako),ancestor(fune,X)., [] ((7) は適用不可)

バックトラック

0 ?- ancestor(X,tarako),ancestor(fune,X)., [] ((9) は適用不可)

(10) を適用

1 ?- parent(X,Y),ancestor(Y,tarako),ancestor(fune,X)., []

...

- 1 以下同様に進み、さらに相当数のステップを経た後に 2 番目の解 X = sazae を得ることができ
- 2 ます。

### 3 Prolog プログラミングの勘所

- 4 上のような実行過程を 1 ステップずつ丁寧に追いかけることは非常に骨の折れる作業です。Prolog
- 5 プログラムを作るときに、上のような実行過程をいちいち追いながらプログラムを作ること
- 6 ません。詳細は述べませんが、Prolog の実行では論理的に正しくない答えは導出されないことが
- 7 証明されています。そこで、まず

- 8 ● 確定節を論理式として読み、プログラムの表明する内容に論理的に間違いがないかチェック
- 9 する。

1 ことが重要です。しかし、論理的に正しい答えが必ず全て導出できるとは限らないことも知られて  
 2 います。これは、Prolog の実行が深さ優先探索であり、しかも確定節の適用される順序が確定節  
 3 がプログラムに記述されている順序に固定されているためです。答えを確実に導出するにはプログ  
 4 ラマが Prolog の実行方式を意識したプログラミングを行う必要があります。

5 確定節の記載順序が実行に影響する様子を知るために、たとえば、`ancestor` を定義するふたつ  
 6 の確定節の順序を以下：

```
7 ancestor(X,Z) :- parent(X,Y),ancestor(Y,Z). // (11)
8 ancestor(X,Y) :- parent(X,Y). // (12)
```

9 のように入れ替えて、図 5.8 の (9) と (10) の代わりに上の (11) と (12) を用いるとします。そう  
 10 すると、入れ替えによって論理的な意味は変わっていないにも関わらず、上の例題の実行過程は以  
 11 下のように変化します。

|                 |                                                           |
|-----------------|-----------------------------------------------------------|
|                 | 0 ?- ancestor(X,tarako),ancestor(fune,X)., []             |
| <u>(11) を適用</u> |                                                           |
| <u>(1) を適用</u>  | 1 ?- parent(X,Y),ancestor(Y,tarako),ancestor(fune,X)., [] |
|                 | 2 ?- ancestor(sazae,tarako),ancestor(fune,namihei).,      |
|                 | [X = namihei]                                             |
| <u>(11) を適用</u> | 3 ?- parent(sazae,Y),ancestor(Y,tarako),                  |
|                 | ancestor(fune,namihei)., [X = namihei]                    |
| <u>(6) を適用</u>  | 4 ?- ancestor(tarao,tarako),ancestor(fune,namihei).,      |
|                 | [X = namihei]                                             |
| <u>(11) を適用</u> | 5 ?- parent(tarao,Y),ancestor(Y,tarako),                  |
|                 | ancestor(fune,namihei)., [X = namihei]                    |
| <u>(7) を適用</u>  | 6 ?- ancestor(tarako,tarako),ancestor(fune,namihei).,     |
|                 | [X = namihei]                                             |
| <u>(11) を適用</u> | 7 ?- parent(tarako,Y),ancestor(Y,tarako),                 |
|                 | ancestor(fune,namihei)., [X = namihei]                    |
| <u>バックトラック</u>  | 6 ?- ancestor(tarako,tarako),ancestor(fune,namihei).,     |
|                 | [X = namihei] ((11) は適用不可)                                |
| <u>(12) を適用</u> | 7 ?- parent(tarako,tarako),ancestor(fune,namihei).,       |
|                 | [X = namihei]                                             |
| <u>バックトラック</u>  | 6 ?- ancestor(tarao,tarako),ancestor(fune,namihei).,      |
|                 | [X = namihei] ((11),(12) は適用不可)                           |
| <u>バックトラック</u>  | 5 ?- parent(tarao,Y),ancestor(Y,tarako),                  |
|                 | ancestor(fune,namihei).,[X = namihei]                     |
|                 | ((7) は適用不可)                                               |
| <u>バックトラック</u>  | 4 ?- ancestor(tarao,tarako),ancestor(fune,namihei).,      |
|                 | [X = namihei] ((11) は適用不可)                                |
| <u>(12) を適用</u> | 5 ?- parent(tarao,tarako),ancestor(fune,namihei).,        |
|                 | [X = namihei]                                             |
| <u>(7) を適用</u>  | 6 ?- ancestor(fune,namihei)., [X = namihei]               |
|                 | ...                                                       |

1 既に見たように、もとの実行過程では初期ゴールから8ステップ目で最初の解  $X = \text{tarao}$  に到達  
 2 していますが、上の実行過程では14ステップを経てもなお解に到達せず、さらに実行を続けて行  
 3 くと、約110ステップ後ようやく  $X = \text{tarao}$  に到達します。この例で実行ステップ数が増大し  
 4 た原因は、適切でない確定節を先に適用し、結果としてバックトラックが増えたためです。この例  
 5 は確定節を記述する順番が重要であることを示唆しています。一般には

- 6 ● 単純な確定節を上の方に記述し、再帰的に定義された確定節を下の方に記述する。
- 7 ことによって、万が一、誤った確定節を先に適用しても早い段階でバックトラックが起き、結果と  
 8 して高速に解に到達できることが知られています。

9 確定節の右辺に複数の原子論理式が含まれるときには、右辺の中での順番にも注意しなければな  
 10 りません。たとえば、(11)の確定節の右辺の  $\text{parent}(X,Y)$  と  $\text{ancestor}(Y,Z)$  を以下のように入  
 11 れ替えてみます。

```
12 ancestor(X,Z) :- ancestor(Y,Z),parent(X,Y). // (13)
13 ancestor(X,Y) :- parent(X,Y). // (14)
```

14 このときの実行過程は以下の通りです。

```

(13) を適用      0 ?- ancestor(X,tarako),ancestor(fune,X)., []
(13) を適用      1 ?- ancestor(X,Y),parent(Y,tarako),ancestor(fune,X)., []
                  2 ?- ancestor(X,Y2),parent(Y2,Y),parent(Y,tarako),
(13) を適用                  ancestor(fune,X)., []
                  3 ?- ancestor(X,Y3),parent(Y3,Y2),parent(Y2,Y),
(13) を適用                  parent(Y,tarako),ancestor(fune,X)., []
                  4 ?- ancestor(X,Y4),parent(Y4,Y3),parent(Y3,Y2),
                  parent(Y2,Y),parent(Y,tarako),ancestor(fune,X)., []
                  ...
```

15 ゴール節の原子論理式が増え続けていることに気づくでしょう。実は、この実行過程では決して解  
 16 に到達することはなく、コンピュータのメモリを使い切る<sup>23</sup>まで実行し続けます。一般には

- 17 ● 確定節の左辺の述語記号と同じ述語記号を持つ原子論理式を右辺の最左に置くと、無限再帰  
 18 に陥って実行が止まらなくなる場合がある。

19 ことが知られています。

20 上の二つの●に述べた性質はあくまでも一般論であり、常に成り立つ訳ではありません。しか  
 21 し、Prolog インタプリタが融通の効かない杓子（しゃくし）定規な方法でプログラムを実行する  
 22 ことは上に述べた通りです。プログラマは、確定節の記述順序や確定節右辺の原子論理式の記述順  
 23 序に十分に注意を払う必要があります。

<sup>23</sup>Prolog の実行スタックを使い切ったときには、**ERROR: Out of local stack** が出力され、実行が強制的に中断され  
 ます。

### 5.4.3 トレース・モード

Prolog インタプリタには、プログラムの実行の様子を逐次表示しながら実行するトレース・モードが用意されており、プログラムのデバッグに便利です。その表示方法は、上に示した実行ステップの表示とは異なりますが、実行の様子が容易に理解できるように工夫されています。たとえば以下は、トレース・モードの実行例です。トレース・モードに入るには組み込み述語 `trace` を用います。

```
?- trace.      //左記の入力でトレースモードに入る。
true.

[trace] ?- ancestor(X,tarako),ancestor(fune,X).
Call: (8) ancestor(_G585, tarako) ? creep    //ここで改行 (CR) を打鍵、以下同様
Call: (9) parent(_G585, tarako) ? creep
Exit: (9) parent(tarao, tarako) ? creep
Exit: (8) ancestor(tarao, tarako) ? creep
Call: (8) ancestor(fune, tarao) ? creep
Call: (9) parent(fune, tarao) ? creep
Fail: (9) parent(fune, tarao) ? creep
Redo: (8) ancestor(fune, tarao) ? creep
Call: (9) parent(fune, _L218) ? creep
Exit: (9) parent(fune, sazae) ? creep
Call: (9) ancestor(sazae, tarao) ? creep
Call: (10) parent(sazae, tarao) ? creep
Exit: (10) parent(sazae, tarao) ? creep
Exit: (9) ancestor(sazae, tarao) ? creep
Exit: (8) ancestor(fune, tarao) ? creep

X = tarao ;
Redo: (9) ancestor(sazae, tarao) ? creep
Call: (10) parent(sazae, _L246) ? creep
... (以下、同様に続く)
```

なお、上に現れる `_G585`、`_L218` などはインタプリタが内部で使用する変数名で、必要に応じてインタプリタが自動的に生成します。通常の実行では、ユーザがこのような内部の変数を見ることはありません。

## 5.5 より複雑なデータ型 (再考)

ここでは、5.3 節で紹介した例題プログラムの実行を詳細に検討します。5.3 節の説明に疑問を持った人はよく確認してください。

### 5.5.1 数値

図 5.9 に 5.3.1 節の図 5.5 のプログラムを再掲します。このプログラムについて、ゴール節 `?- sum(saga,tosu,D).` の実行過程を以下に載せます。

```

distance(usizu,  saga,  9.2).    // (1)
distance(saga,   kanzaki, 9.3).  // (2)
distance(kanzaki, tosu,  15.7).  // (3)
distance(tosu,   hakata, 28.6).  // (4)

sum(X, X, 0).                      // (5)
sum(X, Y, D) :- distance(X, Z, D1), sum(Z, Y, D2), D is D1+D2. // (6)

```

図 5.9: 長崎本線の駅間の距離 (再掲)

```

(6) を適用      0 ?- sum(saga,tosu,D)., []
(2) を適用      1 ?- distance(saga,Z,D1),sum(Z,tosu,D2),D is D1+D2., []
(6) を適用      2 ?- sum(kanzaki,tosu,D2),D is 9.3+D2., []
                 3 ?- distance(kanzaki,Z,D12),sum(Z,tosu,D22),D2 is D12+D22,
                   D is 9.3+D2., []
(3) を適用      4 ?- sum(tosu,tosu,D22),D2 is 15.7+D22,D is 9.3+D2., []
(5) を適用      5 ?- D2 is 15.7+0,D is 9.3+D2., []
組み込み述語 is を評価      6 ?- D is 9.3+15.7., []
組み込み述語 is を評価      7 ?- ., [D = 25.0]

```

D = 25.0 を出力。

- 1 ここに D12, D22 は、既に存在する D2 と名前が衝突しないように導入した新しい変数名です。上
- 2 の実行において、組み込み述語 is を評価する直前には、is の右辺が定数式になっていることに
- 3 注意してください。
- 4 このプログラムの場合、距離から駅を見つける逆計算も可能です。

```

(6) を適用      0 ?- sum(X, Y, 25.0)., []
(1) を適用      1 ?- distance(X,Z,D1),sum(Z,Y,D2),25.0 is D1+D2., []
(5) を適用      2 ?- sum(saga,Y,D2),25.0 is 9.2+D2., [X = usizu]
                 3 ?- 25.0 is 9.2+0., [X = usizu, Y = saga]
組み込み述語 is を評価 (失敗)
バックトラック
... 中略...
バックトラック 1 ?- distance(X,Z,D1),sum(Z,Y,D2),25.0 is D1+D2., [] ((1) は適用不可)
(2) を適用      2 ?- sum(kanzaki,Y,D2),25.0 is 9.3+D2., [X = saga]
(5) を適用      3 ?- 25.0 is 9.3+0., [X = saga, Y = kanzaki]
組み込み述語 is を評価 (失敗)
バックトラック
(6) を適用      2 ?- sum(kanzaki,Y,D2),25.0 is 9.3+D2., [X = saga] ((5) は適用不可)
                 3 ?- distance(kanzaki,Z,D1),sum(Z,Y,D22),D2 is D1+D22,
                   25.0 is 9.3+D2., [X = saga]

```

```
member(X,[X|_]). // (5)
member(X,[_|Z]):-member(X,Z). // (6)
```

図 5.10: 所属判定 (再掲)

(3) を適用

```
4 ?- sum(tosu,Y,D22),D2 is 15.7+D22,25.0 is 9.3+D2.,
[X = saga]
```

(5) を適用

組み込み述語 is を評価

```
5 ?- D2 is 15.7+0,25.0 is 9.3+D2., [X = saga, Y = tosu]
```

組み込み述語 is を評価

```
6 ?- 25.0 is 9.3+15.7., [X = saga, Y = tosu]
```

```
7 ?- ., [X = saga, Y = tosu]
```

X = saga, Y = tosu を出力

## 1 5.5.2 リスト

## 2 5.5.3 所属判定

3 5.3.2 節で紹介した述語 `member(b,L)` について詳細な実行の様子を見てみましょう。

4 たとえば、ゴール節：

5 `?- member(b,[a,b,c]).`

6 を図 5.10 のプログラムを用いて実行してみます。

```
(6) を適用      0 ?- member(b,[a,b,c])., []
(5) を適用      1 ?- member(b,[b,c])., []
                  2 ?- ., []
```

true. を表示、実行終了。

7 逆計算では以下のように実行が進みます。

```
(6) を適用      0 ?- member(b,[a,X,c])., []
(5) を適用      1 ?- member(b,[X,c])., []
                  2 ?- ., [X = b]
X = b を出力。セミコロン・キーを打鍵したならばバックトラック
(6) を適用      1 ?- member(b,[X,c])., [] ((5) は適用不可)
(6) を適用      2 ?- member(b,[c])., []
                  3 ?- member(b,[],)., []
バックトラック  2 ?- member(b,[c])., [] ((6) は適用不可)
バックトラック  1 ?- member(b,[X,c])., [] ((5),(6) は適用不可)
バックトラック  0 ?- member(b,[a,X,c])., [] ((6) は適用不可)
```

バックトラック (失敗)



```

append([],X,X).                                // (1)
append([A|X],Y,[A|Z]):-append(X,Y,Z).         // (2)

```

図 5.11: リストの接続 (再掲)

```

flatten([],[]).                                // (1)
flatten(X,[X]):-atomic(X).                     // (2)
flatten([X|Y],Z):-flatten(X,X2),flatten(Y,Y2),append(X2,Y2,Z). // (3)

```

図 5.12: リストの平坦化 (その 1)

#### 5.5.4 リストの接続

- 次に、5.3.2 節で紹介した述語  $\text{append}(\ell_1, \ell_2, \ell_3)$  の実行の様子を図 5.11 のプログラムについて見てみましょう。

```

(2) を適用      0 ?- append([yama, kawa], [umi, sima], X)., []
(2) を適用      1 ?- append([kawa], [umi, sima], X2)., [X = [yama|X2]]
(1) を適用      2 ?- append([], [umi, sima], X3)., [X = [yama, kawa|X3]]
                3 ?- ., [X = [yama, kawa, umi, sima]]
X = [yama, kawa, umi, sima] を出力。セミコロン・キーを打鍵したならばバックトラック
                2 ?- append([], [umi, sima], X3)., [X = [yama, kawa|X3]]
                ((1) は適用不可)
バックトラック 1 ?- append([kawa], [umi, sima], X2)., [X = [yama|X2]]
                ((2) は適用不可)
バックトラック 0 ?- append([yama, kawa], [umi, sima], X)., [] ((2) は適用不可)
バックトラック (失敗)

```

- 上記実行では最初の解に到達するまで全くバックトラックが発生しておらず、効率的な計算ができています。なお、この実行過程を理解する上で、 $[yama, kawa|X3] = [yama|[kawa|X3]]$  であることに注意してください。また単一化子  $\theta = [X3 := [umi, sima]]$  について、 $[yama, kawa|X3]\theta = [yama, kawa, umi, sima]$  であることにも注意してください。
- 逆方向の計算についても見てみましょう。

```

(2) を適用      0 ?- append([yama, kawa], X, [yama, kawa, umi, sima])., []
(2) を適用      1 ?- append([kawa], X, [kawa, umi, sima])., []
(1) を適用      2 ?- append([], X, [umi, sima])., []
                3 ?- ., [X = [umi, sima]]
X = [umi, sima] を出力

```

### 1 5.5.5 リストの平坦化

2 `member()`、`append()` と同様に、リストを平坦化する Prolog プログラムも 4.9.3 節の Lisp プロ  
3 グラムを翻訳するだけで作ることができます。

4 以下が Lisp の関数定義でした。

```
5 (defun flatten (x)
6   (cond ((null x) nil)
7         ((atom x) (cons x nil))
8         (t (append (flatten (car x)) (flatten (cdr x))))))
```

9 これを翻訳したものが図 5.12 です。ここに確定節 (2) の `atomic(X)` は `X` がアトム（シンボル、数  
10 値、空リスト）であるときに真になる組み込み述語です。

11 実行例は以下の通りです。別解が出力されることに注意してください。

```
12 ?- flatten([[[umi, 3]], kawa, [777, sima], 12], X).
```

```
13
14 X = [umi, 3, kawa, 777, sima, 12] ;
```

```
15
16 X = [umi, 3, kawa, 777, sima, 12, []] ;
```

```
17
18 X = [umi, 3, kawa, 777, sima, [], 12] ;
```

```
19
20 X = [umi, 3, kawa, 777, sima, [], 12, []] ;
```

```
21
22 (中略)
```

```
23
24 X = [umi, 3, [], [], kawa, 777, sima, [], 12, []] ;
```

```
25
26 false.
```

```
27 ?-
```

28 最初の解 `X = [umi, 3, kawa, 777, sima, 12]` は予想された解です。しかし空リストを途中に  
29 含むような別解が存在することが分かります。何故、このような結果になったのでしょうか。理由  
30 は後節で述べます。

## 31 5.6 数をリストに置き換えて

32 4.10 節においても考察したように、自然数を線形リストで表し、その線形リストの上に加算、乗  
33 算等の演算を Prolog の述語として構築しましょう。

### 34 5.6.1 自然数をリストで表す

35 まず、自然数  $n$  ( $= 0, 1, 2, 3, \dots$ ) を長さが  $n$  の Prolog のリスト `[1, 1, ..., 1]` で表現します。

36 具体的には以下のような対応になります。

```

1      0 = [] = 長さ 0 のリスト
2      1 = [1] = 長さ 1 のリスト
3      2 = [1, 1] = 長さ 2 のリスト
4      3 = [1, 1, 1] = 長さ 3 のリスト
5      以下同様

```

### 6 5.6.2 加算の再帰的定義

7 自然数を上のように表すとき、二つの数の加算を行う述語 `add` は、以下のようなゴール節の実  
8 行に成功すべきです。

```

9  ?- add([], [], []).      // 0 + 0 = 0 の意味
10 true.
11 ?- add([], [1, 1], X).    // 0 + 2 = ? の意味
12 X = [1, 1] ;
13 false.
14 ?- add([1], [], X).       // 1 + 0 = ? の意味
15 X = [1] ;
16 false.
17 ?- add([1, 1], [1, 1, 1], X). // 2 + 3 = ? の意味
18 X = [1, 1, 1, 1, 1] ;
19 false.
20 ?-

```

21 また Prolog では逆計算も可能ですから、以下のような実行も可能であるべきです。

```

22 ?- add(X, [], []).        // X + 0 = 0 を満たす X は？
23 X = [] ;
24 false.
25 ?- add(X, [1, 1], [1, 1, 1, 1]). // X + 2 = 4 を満たす X は？
26 X = [1, 1] ;
27 false.
28 ?- add([1], X, [1, 1]).    // 1 + X = 2 を満たす X は？
29 X = [1] ;
30 false.
31 ?-

```

32 考察 上のような性質を満たす述語 `add` を確定節で定義しなさい。なお、以下は、逆計算はできな  
33 いが、同等の計算が可能な Lisp のプログラムである。これを参考にしなさい。

```

34 (defun zero (n) (if (null n) T nil))
35 (defun succ (n) (cons 1 n))
36 (defun add (x y)
37   (cond ((zero y) x)
38         (t      (succ (add x (cdr y))))))

```

### 5.6.3 乗算の再帰的定義

二つの数の乗算を行う述語 `mult` は、以下のようなゴール節の実行ができるべきです。

```

?- mult([],[],X).      // 0 x 0 = ? の意味
X = [] ;
false.
?- mult([], [1],X).    // 0 x 1 = ? の意味
X = [] ;
false.
?- mult([1],[1,1],[1,1]). // 1 x 2 = 2 の意味
true.
?- mult([1,1],[1],X).  // 2 x 0 = ? の意味
X = [] ;
false.
?- mult([1,1],[1,1,1],X). // 2 x 3 = ? の意味
X = [1,1,1,1,1,1] ;
false.
?-

```

また以下のような逆計算もできるべきです。

```

?- mult([1,1],X,[1,1,1,1]). // 2 x X = 4 を満たす X は?
X = [1, 1] ;
false.
?-

```

**考察** 上のような性質を満たす述語 `mult` を確定節で定義しなさい。

べき乗や階乗の計算も同様に考えることができます。余裕のある人は実際にプログラムを作成し、その動作を検討してください。

## 5.7 集合をリストで表す

4.11 節と同様に、この節では集合をリストを用いて表します。集合  $\{\text{yama}, \text{kawa}, \text{umi}\}$  を Prolog のリスト `[yama, kawa, umi]` で表すこととします。4.11 節と同様に、要素がリストに現れる順序は任意とし、リスト `[yama, kawa, umi]` とリスト `[umi, yama, kawa]` は同じ集合を表すと考えます。また、要素の重複はないとします。

このような集合について、要素の追加、包含関係、集合積、集合和等の演算を Prolog で定義しましょう。

### 5.7.1 要素の追加

原子論理式 `addmember(e, S1, S2)` は、集合 (リスト)  $S_1$  に  $e$  を加えた集合が  $S_2$  であるような関係を表すとしてします。そのような述語 `addmember` の定義は以下の通りです。

```

1  addmember(X,Y,Y):-member(X,Y),!.
2  addmember(X,Y,[X|Y]).

```

3 このプログラムの直感的な意味は以下の通りです。

- 4 1. もし  $X$  が  $Y$  に含まれる ( $\text{member}(X,Y)$  が成り立つ) ならば、 $Y$  に  $X$  を加えた集合は  $Y$  である。
- 5 2. さもなくば、 $Y$  に  $X$  を加えた集合は  $[X|Y]$  である。

6 プログラム 1 行目の "!" はカット演算子と呼ばれ、インタプリタの実行を制御します。詳細は 5.8  
7 節で述べることにし、ここでは「さもなくば」を明示的に表す演算子である位に考えておきましょ  
8 う。上の Prolog プログラムは、4.11.1 節の Lisp プログラム：

```

9  (defun addmember (e s)
10    (cond ((member e s) s)
11          (t      (cons e s))))

```

12 をそのまま Prolog に変換したものになっています。実行例は以下の通りです。

```

13 ?- addmember(yama,[kawa],X).           //問い合わせ 1
14 X = [yama,kawa] ;
15 false.
16 ?- addmember(yama,[kawa,yama,umi],X).   //問い合わせ 2
17 X = [kawa,yama,umi] ;
18 false.
19 ?- addmember(yama,[kawa,yama],[kawa,yama]). //問い合わせ 3
20 true.
21 ?- addmember(yama,[kawa,yama],[yama,kawa]). //問い合わせ 4
22 false.
23 ?- addmember(yama,X,[yama,kawa]).        //問い合わせ 5
24 X = [yama,kawa] ;
25 false.
26 ?- addmember(X,[yama],[kawa,yama]).      //問い合わせ 6
27 X = kawa ;
28 false.
29 ?- addmember(X,[yama],[yama,kawa]).      //問い合わせ 7
30 false.
31 ?-

```

32 上の実行例を見ると、問い合わせ 1, 2, 3, 6 については正しい解を得ています。問い合わせ 4 の答  
33 えは本来ならば true. と返るべきです。しかしそうならない理由は、リストを集合と見なす立場  
34 からは  $[kawa,yama]$  と  $[yama,kawa]$  は同じ集合を表すのですが、インタプリタは異なるリストと  
35 判断するためです。この問い合わせに true. と解答させるには、異なるリストを同じ集合と見な  
36 すような、Prolog プログラムを追加せねばなりません。問い合わせ 5, 6, 7 は、いわゆる逆計算で  
37 す。問い合わせ 6 では正しい解答を得ていますが、これは様々な幸運が重なった偶然であって、こ  
38 こで定義した addmember のプログラムは逆計算には適しないと考えた方がよいでしょう。

- 1 上の実行例から判断すると、原子論理式 `addmember(e, S1, S2)` は  $e$  と  $S_1$  が入力引数であり<sup>24</sup>、  
 2  $S_2$  が出力引数 (つまり  $S_2$  は変数) であるときに正しく動作すると考えます<sup>25</sup>。

### 3 5.7.2 部分集合

- 4 原子論理式 `subset(S1, S2)` は、関係  $S_1 \subseteq S_2$  を表すとします。そのための述語定義は以下の通  
 5 りです。

```
6 subset([], _).
7 subset([X|Y], Z):-member(X,Z),subset(Y,Z).
```

- 8 これは以下の Lisp プログラム (4.11.2 節参照) を変換したものです。

```
9 (defun subset (s1 s2)
10   (cond ((null s1) t)
11         (t (and (member (car s1) s2) (subset (cdr s1) s2)))))
```

- 12 関数の評価例は以下の通りです。

```
13 ?- subset([1,2,3],[1,2,3,5]).
14 true.
15 ?- subset([1,kawa],[1,kawa]).
16 true.
17 ?- subset([2,kawa],[1,umi,kawa]).
18 false.
19 ?- subset(X,[yama,kawa]).
20 X = [] ;
21 X = [yama] ;
22 X = [yama,yama] ;
23 ...
```

- 24 4 番目の問い合わせでは `[yama,kawa]` の部分集合を全て列挙することを試みましたが、要素が重  
 25 複する解が出てしまいました。この述語も二つの引数が共に入力引数であることに使用するのが無  
 26 難でしょう。

### 27 5.7.3 和集合

- 28 原子論理式 `cup(S1, S2, S3)` は、 $S_1 \cup S_2 = S_3$  であるような関係を表すとします。この述語 `cup`  
 29 は、たとえば以下のような関数評価を行うべきです。

```
30 ?- cup([1,2,3],[1,3,4],X).
31 X = [1,2,3,4] ; (あるいは [2,1,3,4], [2,3,1,4] などなど、要素の出現順は不同でよい)
32 false.
33 ?- cup([], [1,2,3], X).
```

<sup>24</sup>正確に言えば、第一引数と第二引数が定数に具体化 (インスタンス化) されていることです。

<sup>25</sup>もともと Lisp のプログラムから作ったプログラムですから、Lisp と同様の実行でのみ正しく動いたとしても不思議ではありません。

```

1 X = [1,2,3] ;    (あるいは [2,1,3], [2,3,1] などなど、要素の出現順は不同でよい)
2 false.
3 ?- cup([kawa,umi],[sima,yama],X).
4 X = [kawa, umi, sima, yama] ; (あるいは... 以下同上)
5 false.
6 ?-

```

7 考察 上のような評価結果を与える述語 `cup` を定義しなさい。ただし逆計算 (たとえば

8 `?-cup([1,2,3],X,[1,2,3,4])` . などの問い合わせへの解答) が正しく実行できることは要求し

9 ない。

#### 10 5.7.4 積集合

11 原子論理式  $\text{cap}(S_1, S_2, S_3)$  は、 $S_1 \cap S_2 = S_3$  であるような関係を表すとしてします。この述語 `cap`

12 は、たとえば以下のような関数評価を行うべきです。

```

13 ?- cap([1,2,3],[1,3,4],X).
14 X = [1,3] ;    (あるいは (3 1))
15 false.
16 ?- cap([yama],[umi],X).
17 X = [] ;
18 false.
19 ?- cap([1,3],[],X).
20 X = [] ;
21 false.
22 ?- cap([1,kawa,umi],[kawa,1,umi],X).
23 X = [1,kawa,umi] ;
24 (あるいは、[kawa,1,umi], [umi,kawa,1] などなど、要素の出現順は不同でよい)
25 false.
26 ?-

```

27 考察 上のような評価結果を与える述語 `cap` を定義しなさい。ただし逆計算 (たとえば

28 `?-cap([1,2,3],X,[1])` . などの問い合わせへの解答) が正しく実行できることは要求しない。ま

29 た必要ならば組み込み述語 `not` を用いなさい。`not` は原子論理式を引数とし、その原子論理式を評

30 価した結果が真であれば偽と評価され、偽であれば真と評価される述語です (以下、参照)。

```

31 ?- not(member(a,[a,b,c])).
32 false.
33 ?- not(member(d,[a,b,c])).
34 true..
35 ?-

```

#### 36 5.7.5 べき集合

37 原子論理式  $\text{powerset}(S_1, S_2)$  は、 $2^{S_1} = S_2$  であるような関係を表すとしてします。この述語 `powerset`

38 は、たとえば以下のような関数評価を行うべきです。

```

1  ?- powerset([1,2,3],X).
2  X = [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]] ; (ただし順不同でよい)
3  false.
4  ?- powerset([],X).
5  X = [[]] ;
6  false.
7  ?- powerset([kawa],X).
8  X = [[], [kawa]] ;
9  false.
10 ?-

```

11 考察 上のような評価結果を与える述語 `powerset` を定義しなさい。ただし逆計算 (たとえば  
12 `?-powerset(X, [[], [1]])` などの問い合わせへの解答) が正しく実行できることは要求しない。

## 13 5.8 カット演算子

14 Prolog のバックトラックは複数解の探索には必須の機能です。仮に解がひとつだけであったと  
15 しても、その解への到達経路が単純でなく、事前に予測できない場合にはバックトラックは効果的  
16 な実行機構です。しかし、複数解を必要としない場合にはバックトラックは必要ありません。また  
17 バックトラックを意図的に抑制したい場合もあります。そのような要求のために Prolog にはカッ  
18 ト演算子 `!` が組み込まれています。この「カット」は探索木を刈り込む (cut) ことを意味してい  
19 ます。

### 20 5.8.1 カット演算子の使用法

一般に、以下の確定節：

$$A : -B_1, \dots, B_i, !, B_{i+1}, \dots, B_n.$$

21 のように右辺にカット演算子がある場合、プログラム実行中にこの確定節が選択適用され、かつ  
22  $B_1$  から  $B_i$  までの実行に成功した<sup>26</sup> ならば、以後

- 23 1. この確定節を選択適用した分岐点 (ゴール最左の原子論理式とこの確定節の頭部が単一化でき  
24 て、この確定節が選択適用された時点) では、他の確定節によるバックトラックを行わない。
- 25 2.  $B_1, \dots, B_i$  に関係する部分探索木の分岐点でもバックトラックを行わない。

26 と約束されています。

27 しかし上に説明だけでは実際のカット演算子の利便性を知ることは困難です。この演算子の最  
28 も一般的な使用法として、手続き型プログラミング言語の if-then-else に例えると分かりやすいで  
29 しょう。

<sup>26</sup> 正確に言えば、それ以降の実行によって、 $B_1$  から  $B_i$  に関連する原子論理式が実行途中のゴール節から消去された。



たとえば以下は、引数の数値の3種類の排他的な範囲（引数が0より小さい、0以上1より小さい、または1以上）によって戻り値（-1/0/1）を選択するCの関数です。

```

3 int f(int x){
4     if(x < 0) return -1;
5     if((0 <= x)&&(x < 1)) return 0;
6     if(1 <= x) return 1;
7 }

```

しかし上のプログラムは冗長な書き方であって、多くのプログラマは以下のように書くはずです。

```

9 int f(int x){
10     if(x < 0) return -1;
11     else if(x < 1) return 0;
12     else return 1;
13 }

```

このelseの機能をPrologで実現するのがカット演算子です。

前者のC関数に対応するPrologプログラムは以下の通りです。

```

16 f(X,-1):-X < 0.
17 f(X, 0):-0 <= X, X < 1.
18 f(X, 1):-1 <= X.

```

しかし、カットを用いるならば、これを以下のように書くことができます。

```

20 f(X,-1):-X < 0,! .
21 f(X, 0):-X < 1,! .
22 f(X,1).

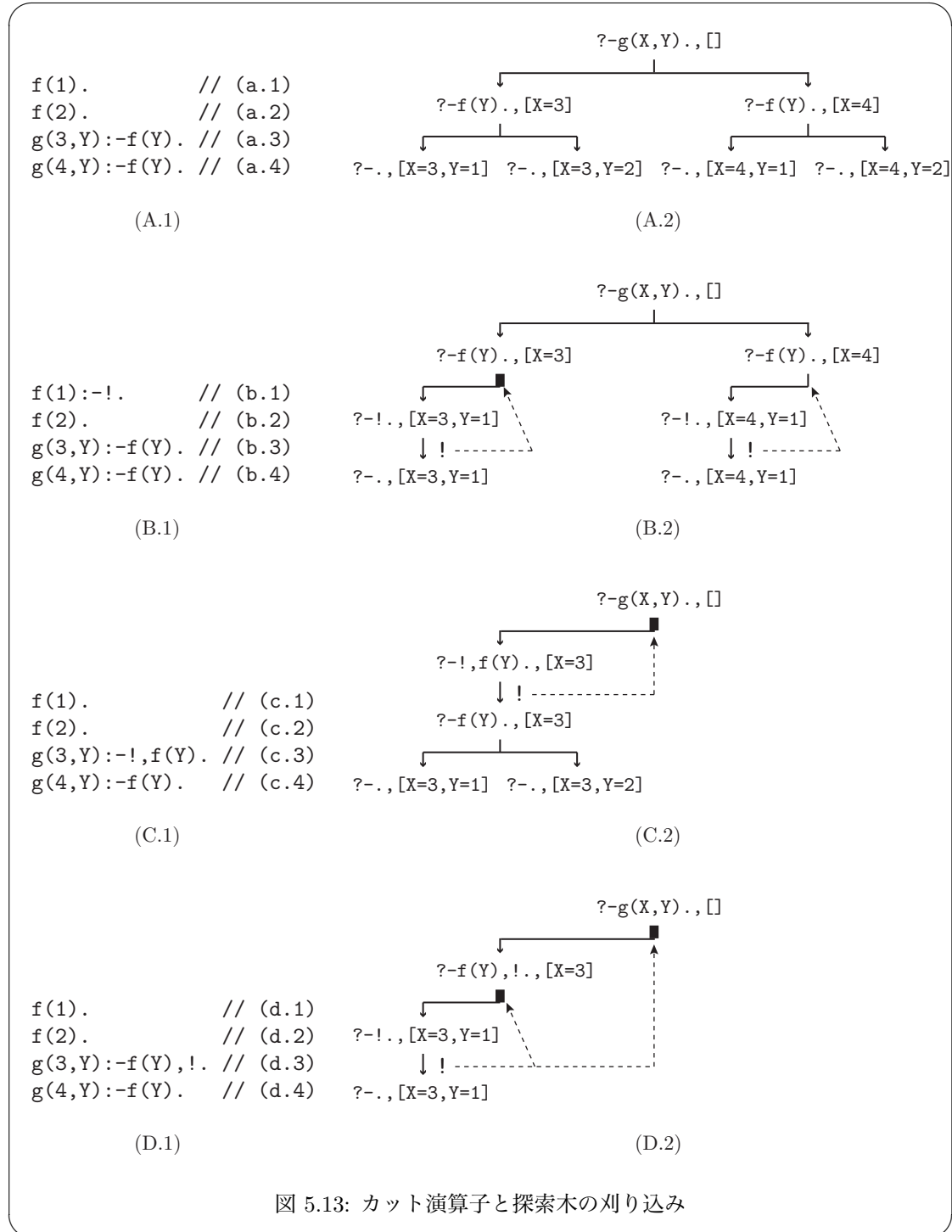
```

つまり、1行目の確定節右辺において、 $X < 0$  が成り立ったならば、プログラムの実行はカット演算子を通して、その右側へ移動しますから、バックトラックが起きても2行目、3行目は適用しません。しかし $X < 0$  が成り立たなかったならば、プログラムの実行はカット演算子を超えませんから、そこからバックトラックし、2行目の確定節の適用を試みます。2行目の右辺においても同様です。

いかなる数値も三つの排他的な範囲（0より小さい、0以上1より小さい、または1以上）のいずれかひとつの範囲にしか属しません。そのような場合にカットを用いることで、不必要な探索を減らし、プログラムの実行速度を上げることができます。

### 5.8.2 カット演算子の詳細な効果

次に、カット演算子のより詳細な説明のために図5.13を用います。図の左上、(A.1)のプログラムが基点となります。このプログラムでは以下のような4種類の解が出力されます。



```

1  ?- g(X,Y).
2  X = 3
3  Y = 1 ;
4  X = 3
5  Y = 2 ;
6  X = 4
7  Y = 1 ;
8  X = 4
9  Y = 2 ;
10 false.
11 ?-

```

その探索木は図 5.13 (A.2) の通りです。この探索木の頂点は初期ゴール節  $?- g(X,Y).$  です。この原子論理式  $g(X,Y)$  に図 5.13 (A.1) の (a.3) の確定節を適用し、さらに (a.1) を適用すると、解  $X = 3, Y = 1$  が得られます。そこからバックトラックし、(a.1) の代わりに (a.2) を適用すると、解  $X = 3, Y = 2$  が得られます。さらにバックトラックし、(a.3)、(a.2) の代わりに (a.4)、(a.1) を順に適用すると解  $X = 4, Y = 1$  が得られます。さらにバックトラックし、(a.1) の代わりに (a.2) を適用すると、解  $X = 4, Y = 2$  が得られます。これはカット演算子を用いない場合の動作であって、前節までに述べてきた通りの動作です。

次に、図 5.13 (A.1) の確定節 (a.1)  $f(1).$  を図 5.13 (B.1) の (b.1)  $f(1):-!$  に置き換えたプログラムを検討しましょう。この確定節のカット演算子は直感的には以下のような意味を持ちます。

(b.1) プログラム実行途中にゴール節中の最左の原子論理式  $f(t)$  (ここに  $t$  は任意の項) に (b.1) が適用された<sup>27</sup> ならば、その後にバックトラックしても原子論理式  $f(t)$  には如何なる確定節を適用しない。

さて初期ゴール節  $?- g(X,Y).$  に関するプログラム (B.1) の探索木は図 5.13 (B.2) の通りです。(b.2) が適用されないため、解は  $X = 3, Y = 1$  と  $X = 4, Y = 1$  の2種類に限定されます。カット演算子  $!$  は論理的には真値と見なしますから、インタプリタは状態：

```

27  ?-!, [X = 3, Y = 1]

```

から状態：

```

29  ?-., [X = 3, Y = 1]

```

へ直ちに遷移します。カット演算子  $!$  は、それがゴール節中に最初に出現する状態の直前の分岐を枝刈りするため、確定節 (b.2) は適用されないのです。なお、図 5.13 (B.2) の点線矢印はカット演算子が作用を及ぼす探索分岐点 (太線) を示しています。

次に図 5.13 (A.1) の (a.3)  $g(3,Y):-f(Y).$  を図 5.13 (C.1) の (c.3)  $g(3,Y):-!,f(Y).$  に置き換えたプログラムを検討しましょう。このカット演算子  $!$  の直感的な意味は以下の通りです。

<sup>27</sup> 正確に言えば、原子論理式  $f(t)$  と (b.1) の左辺  $f(1)$  が単一化可能であり、その結果 (b.1) が適用されて、 $f(t)$  を (b.1) の右辺  $!$  に置き換えた。

```

flatten([], []).                                     // (1)
flatten(X, [X]):-atomic(X), X \== [].               // (2)
flatten([X|Y], Z):-flatten(X, X2), flatten(Y, Y2), append(X2, Y2, Z). // (3)
または
flatten([], []):-!.                                 // (4)
flatten(X, [X]):-atomic(X).                         // (5)
flatten([X|Y], Z):-flatten(X, X2), flatten(Y, Y2), append(X2, Y2, Z). // (6)

```

図 5.14: リストの平坦化 (その 2)

1 (c.3) プログラム実行途中にゴール節中の最左の原子論理式  $g(t_1, t_2)$  (ここに  $t_1, t_2$  は任意の項) に  
 2 (c.3) が適用されたならば、その後にバックトラックしても原子論理式  $g(t_1, t_2)$  には如何なる  
 3 確定節を適用しない。

4 探索木は図 5.13 (C.2) の通りです。点線矢印の始点から終点の範囲のバックトラックが抑制され  
 5 ます。

6 3 番目の例として、(a.3)  $g(3, Y):-f(Y).$  を図 5.13 (D.1) の (d.3)  $g(3, Y):-f(Y), !.$  に置き換  
 7 えたプログラムを検討しましょう。このカット演算子  $!$  の直感的な意味は、以下の通りです。

8 (c.3) プログラム実行途中にゴール節中の最左の原子論理式  $g(t_1, t_2)$  (ここに  $t_1, t_2$  は任意の項) に  
 9 最汎単一化子  $\theta$  を使って (d.3) が適用されたと仮定する。そしてその後の実行ステップにお  
 10 いてゴール節最左に現れる原子論理式 <sup>28</sup>  $f(Y)\theta$  の実行に成功したならば、その後にバック  
 11 トラックしても、 $f(Y)\theta$  には別の如何なる確定節も適用せず、かつ  $g(t_1, t_2)$  にも如何なる  
 12 確定節も適用しない。

13 よって、図 5.13 (D.1) のプログラムの探索木は図 5.13 (D.2) のように枝刈りされています。点  
 14 線矢印が 2 カ所の分岐点でバックトラックを抑制していることに注意してください。

### 15 5.8.3 flatten の改良

16 さて、5.5.5 節の例題で予想しない解が出てしまったことを思い出しましょう。そうなった理由  
 17 は、図 5.12 の確定節 (2) の右辺で使用されている組み込み述語 `atomic(X)` が  $X$  が空リストのとき  
 18 にも真になるためです。そのため、`flatten([], X)` のような、第 1 引数が空リストであるような  
 19 原子論理式には図 5.12 の (1)、(2) が共に適用可能となってしまう、それが予想外の結果を招いた  
 20 のです。この不都合を解決するには、ひとつは図 5.14 (2) のように  $X \backslash== []$  と条件を追加する方

<sup>28</sup>(d.3) の右辺の原子論理式に最汎単一化子  $\theta$  を適用した式

法が考えられます。ここに `\==` は項間の不等号を調べる演算子です。もうひとつは、図 5.14 (4) のようにカット演算子を用い、一度 (4) の適用に成功したならばバックトラックして (5)、(6) を適用しないように制御する方法です。

また、5.7.1 節のプログラム：

```
5      addmember(X,Y,Y):-member(X,Y),!.      // (1)
6      addmember(X,Y,[X|Y]).                  // (2)
```

では十分な説明をしないままにカット演算子を用いました。ここで使用されるカット演算子は、(1) の確定節を適用し、`member(X,Y)` の実行に成功したならば、バックトラックしても (2) を適用しないことを意味します。逆に言えば、`member(X,Y)` が成り立たないときにのみ (2) を適用するのです。

本来、論理プログラミング言語には `else` に相当する機能が用意されていません。原理的には `else` を用いなくとも（上の例で見たように）等価なプログラムを作成することは可能ですから、それは致命的な問題ではありません。しかしプログラムの実行速度を重視する場合や後に述べるような技巧的なプログラム作成にはカットはとても便利です。

## 5.9 組み込み述語

Prolog インタプリタにあらかじめ定義されている特殊な述語として、これまでも `assert`、`trace`、`is`、`=`、`atomic` などを紹介しました。ここではその他の代表的な組み込み述語をいくつか紹介します。

### 5.9.1 出力用述語

述語 `write` は変数値や定数値を標準出力へ出力する述語です。たとえば以下のように実行されます。

```
22  ?- write(1).      // 定数 1 を出力
23  1
24  true.
25  ?- write(abc).    // シンボル abc を出力
26  abc
27  true.
28  ?- write([a,b,c]). // リスト [a,b,c] を出力
29  [a,b,c]
30  true.
31  ?- write('X Y Z'). // 文字列 'X Y Z' を出力
32  X Y Z              // 文字列の囲みにはシングル・クォーテーションを用いる
33  true.
34  ?- write(X).      // 変数 X の値を出力。
35  _G242             // 定数値に単一化されていないので、内部変数名を出力する
```

```

1 true.
2 ?- X=1, write(X). // 変数 X の値を出力
3 1                // X は 1 に単一化されているため、1 を出力
4 X = 1 ;
5 false.
6 ?- write(X), X=1. // 変数 X の値を出力
7 _G301           // write(X) が評価される時には X は 1 に単一化されていない
8                // ので、内部変数名を出力
9 X = 1 ;         // X=1 によって単一化したため、解 X=1 を出力
10 false.
11 ?-

```

12 また、nl は標準出力の改行を行う述語です。たとえば

```

13 ?- write(1),nl,write(2). // 定数 1,2 を改行をはさんで出力
14 1
15 2
16 true.
17 ?-

```

18 次に紹介する例題は、「ハノイの塔」という名称で有名です<sup>29</sup>。ハノイの塔は、3本の棒(以下、  
19 それぞれを左、中央、右の棒と呼びます)と  $n$  枚の大きさが異なる円盤からなります。初期状態では、  
20 全ての円盤は半径が大きい円盤の順に左の棒に刺さって重なっています(図 5.15(a) 参照)。こ  
21 の円盤をひとつひとつ、ひとつの棒から他の棒へ移動させながら、最終的に中央の棒に半径が大き  
22 い円盤の順に重ねる(図 5.15(b) 参照)ことが目的です。ただしこのとき、

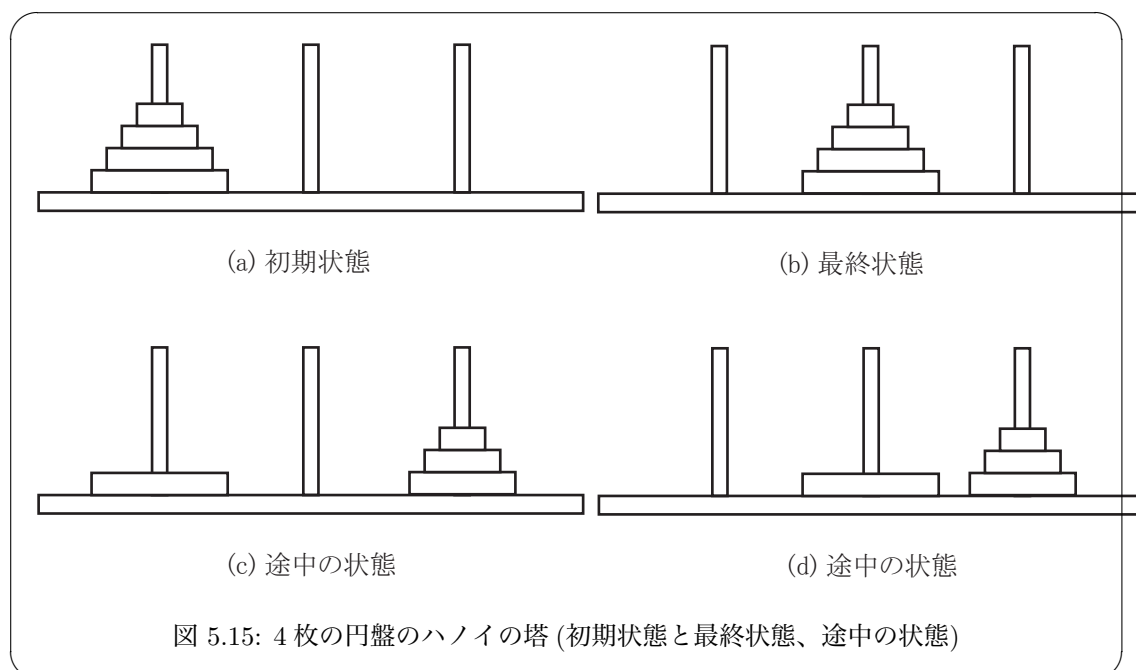
- 23 1. 一回に一枚の円盤しか動かすことができません。
- 24 2. 半径の小さな円盤の上に大きな円盤を置くことはできない。
- 25 3. 3本の棒は自由に使用して構いません。

26 ハノイの塔は再帰を使えば簡単に解ける問題です。

27 今、 $n$  内の円盤について考えます。これを  $n$  枚のハノイの塔と呼びましょう。さて、(1) 左の棒  
28 の上から  $n-1$  枚の円盤を右の棒へ移動させることを考えます(図 5.15(c) 参照)。それに成功した  
29 ならば、(2) 次に左の棒の最も大きな円盤を中央の棒へ移し(図 5.15(d) 参照)、(3) さらに右の棒  
30 の  $n-1$  枚を中央へ移動すれば完成です。この三つの手順の中の(1)と(3)は  $n-1$  枚のハノイの  
31 塔の変形問題です。と言うのも、(1)では左の棒から右の棒へ、(3)では右の棒から中央の棒へ移動  
32 させる点がオリジナルの問題の左の棒から中央の棒へ移動させることと異なるだけです。

33 もし  $n$  が 1 ならば、すなわち 1 枚のハノイの塔を考えるならば、その移動手順は自明です。単  
34 にその 1 枚を左の棒から中央の棒へ移動させるだけです。

<sup>29</sup> この例題の解説は、[http://www.geocities.jp/m\\_hiroi/prolog/index.html](http://www.geocities.jp/m_hiroi/prolog/index.html) を参考にしました。



```

hanoi(1,From,To,_):-
    write('move '),write(From),write(' to '),write(To),nl.
hanoi(N,From,To,Via):-
    N1 is N - 1,
    hanoi(N1,From,Via,To),
    write('move '),write(From),write(' to '),write(To),nl,
    hanoi(N1, Via, To, From).

```

図 5.16: ハノイの塔

- 1 この手順を Prolog 化したものが図 5.16 です。述語 `hanoi(N,From,To,Via)` は、 $N$  枚のハノイの  
 2 塔について、`From` の棒から `To` の棒へ移動させる問題解くものです。ただし、`Via` の棒を作業用に  
 3 用いることができます。このプログラムは以下のように動きます。

```

4 ?- hanoi(1,left,center,right).
5 move left to center
6 true.
7 ?- hanoi(2,left,center,right).
8 move left to right
9 move left to center
10 move right to center
11 true.
12 ?- hanoi(3,left,center,right).
13 move left to center
14 move left to right
15 move center to right

```

```

1 move left to, center
2 move right to left
3 move right to center
4 move left to center
5 true.
6 ?-

```

## 5.9.2 入力用述語

述語 `read` は、出力用述語 `write` とは逆に標準入力から数値、シンボル、リストを読むために使われます。

この述語が呼び出されるとインタプリタの画面上に"`|:`"という形のプロンプトが表示されます。そこで、キーボードからデータを入力し、さらにピリオド"."を打鍵し、改行すると入力が完了し、入力された値が `read` の実引数と単一化されます。たとえば以下のような実行になります。

```

13 ?- read(X).
14 |: 1.                // 数値の入力
15 X = 1 ;
16 false.
17 ?- read(1).
18 |: 1.                // 数値の入力
19 true.                // 実引数 1 と入力値 1 の単一化に成功
20 ?- read(1).
21 |: 2.                // 数値の入力
22 false.               // 実引数 1 と入力値 1 の単一化に失敗
23 ?- read(X).
24 |: Y.                // 変数の入力
25 X = _G234 ;          // 実引数 X は内部変数と単一化
26 false.
27 ?- read([1|X]).
28 |: [1,2].            // リストの入力
29 X = [2] ;            // 実引数 [1|X] と入力リスト [1,2] を単一化した結果 X =[2] となった
30 false.
31 ?-

```

## 5.10 より進んだ話題：失敗と繰り返し

Prolog の最後の話題として、ここまでの Prolog の話からは少し逸脱してみます。

Prolog は本来「論理式をそのままプログラムと見なす」立場のプログラミング言語ですが、実際には

- 確定節の右辺の論理式は、左から順に評価する。
- 確定節は上の行に書かれたものから順に適用する。



```
x(1).
x(2).
x(3).
getallx:-x(X),write(X),nl,fail.
```

図 5.17:  $x(X)$  の全解探索

1 という二種類の実行順序が加わっています。そこで、この順序制御を積極的に利用することで「論  
2 理式=プログラム」の図式を超えたプログラム表現が可能になります。

述語 `fail` は失敗を明示的に表す組み込み述語です。この述語が評価されると、実行はその時点で失敗し、バックトラックが起きます。このバックトラックが起動することを利用して全解探索プログラムを書くことが可能です。図 5.17 はそのような典型的なプログラム例です。確定節

```
6      getallx:-x(X),write(X),nl,fail.
```

7   では、まず右辺の最左の `x(X)` によって、変数 `X` の値をひとつ求めます。求めた値は `write(X),nl`  
8   によって標準出力へ出力されます。そして `fail` によって実行はここで失敗し、バックトラックが  
9   起きます。バックトラックすると、`x(X)` を満たす次の値をひとつ求めます。そして、求めた値は  
10   `write(X),nl` によって出力され...、再度 `fail` によってバックトラックが起きます。このようにし  
11   て、`x(X)` を満たす全ての `X` を出力できるのです。以下は実行結果です。

```
12  ?- getallx.  
13  1  
14  2  
15  3  
16  false.  
17  ?-
```

上の全探索プログラムを改造すると、Prolog に「繰り返し」機構を導入することが可能です。

以下は、数字の 1 を無限に出力するゴールの実行です。

[illegible]

23 ここに `repeat` は何回でも成功する組み込み述語であって、以下の定義と等価です。

```

24     repeat.
25     repeat:-repeat.

```

1 行目は常に成功することを表し、2 行目は単純な無限再帰になっています。よって `repeat` は何回評価されても、必ず成功します。

28 実用的なプログラムでは無限の繰り返しはありません。以下は、条件(入力値が0である)が  
29 成り立つまで出力を繰り返すゴールの実行です。

```

1  ?- repeat,read(X),write(X),nl,X = 0,!,fail.
2  |: 1.
3  1
4  |: abc.
5  abc
6  |: 0.
7  0
8  false.
9  ?-

```

一般に、以下のゴールによって条件が成り立つまでの `do until` 型のループが実現できます。

`repeat, 実行文, 条件,!, fail`

10 しかしながら同様のことは以下のように定義した述語 `do` でも可能です。

```

11      do:-read(X),write(X),nl,check(X).
12      check(0):-!,fail.
13      check(_):-do.

```

14 ですから、`repeat` が他に比類ない便利な機能を提供しているという訳ではありません。

15 同様の発想から、Prolog に `for` ループに類似した表現を実現することも可能です。以下は 1 か  
16 ら 5 までの数の出力を行うゴールの実行です。

```

17 ?- between(1,5,X),write(X),nl,fail.
18 1
19 2
20 3
21 4
22 5
23 false.
24 ?-

```

25 述語 `between(L,H,X)` は、`L` から `H` までの整数を順に変数 `X` に割り付ける組み込み述語です。こ  
26 れは以下のように定義されています。

```

27      between( L, H, L ) :- L <= H.
28      between( L, H, X ) :- L < H,L1 is L+1,between( L1, H, X )

```

29 以上、この節ではバックトラックを積極的に用いた全探索の方法について簡単に述べました。

## 1 第6章 関数型プログラムと論理型プログラムの関係

2 この章はこの講義テキストの最後の話題として、前2章を互いに関連付けるものです。

3 前章の Prolog プログラムの説明では Lisp プログラムとの類似性をいくつも指摘しました。こ  
4 の章ではその類似性を整理し、Lisp と Prolog のプログラムの相互変換する方法について概説しま  
5 す。これによって構文構造が全く異なるプログラミング言語の数理的共通性を理解することができ  
6 ます。

### 7 6.1 Lisp から Prolog への変換

8 ここでは二つの変換例を紹介します。

#### 9 6.1.1 階乗計算

3章で述べたように、階乗を計算する関数は再帰を用いて以下のように表すことができます。

$$f(1) = 1 \quad (6.1)$$

$$f(n) = n \times f(n-1) \quad (n \geq 2) \quad (6.2)$$

10 これを Lisp に書けば

```
11 (defun f (n)
12   (cond ((= n 1) 1)
13         (t (* n (f (- n 1))))))
```

14 となります。これに対応する Prolog のプログラムは以下のように導出することができます。この  
15 方法はひとつの事例に過ぎませんが、変換の考え方を教えてくれます。

まず、等式  $f(n) = m$  が成り立つときに限り真になる述語  $p$  を新たに考えます。すなわち以下が成り立つとします。

$$f(n) = m \Leftrightarrow p(n, m) \text{ は真}$$

このことは踏まえ、式 (6.1)、(6.2) を以下のように書き換えます。

$$p(1, 1). \quad (6.3)$$

$$p(n, n \times f(n-1)). \quad (n \geq 2) \quad (6.4)$$

- 1 しかし、式 (6.4) の第2引数  $n \times f(n-1)$  には三つの関数  $\times$ 、 $f$ 、 $-$  が使用されており、関数と論
- 2 理が混在した表現です。これを Prolog のプログラムと見なすことはできません。

そこで関数  $\times$  については等式  $z = x \times y$  に相当する原子論理式  $z \text{ is } x \times y$  を導入し、式 (6.4) を以下のように置き換えてみます。is は 5.3.1 節で紹介した Prolog の代入演算子です。

$$p(n, m) \leftarrow m \text{ is } n \times f(n-1). \quad (n \geq 2)$$

ここに  $\leftarrow$  は含意記号です。上の式は、直感的には「 $m \text{ is } n \times f(n-1)$  が真ならば、 $p(n, m)$  は真である」と読むことができます。次に  $m \text{ is } n \times f(n-1)$  の中の関数呼び出し  $f(n-1)$  を変数  $x$  に置き換え、かつ、 $f(n-1) = x$  を論理式中に追加します。

$$p(n, m) \leftarrow f(n-1) = x \wedge m \text{ is } n \times x. \quad (n \geq 2) \quad (6.5)$$

- 3 ここに  $\wedge$  は論理積記号です。

さて、式 (6.5) の等式  $f(n-1) = x$  は  $p(n-1, x)$  と等価です。よって、以下のように置換します。

$$p(n, m) \leftarrow p(n-1, x) \wedge m \text{ is } n \times x. \quad (n \geq 2)$$

上の式の右辺にはまだ関数  $-$  が使用されています。そこで等式  $z = x - y$  に相当する原子論理式  $z \text{ is } x - y$  を導入し、上の式を以下のように置き換えます。

$$p(n, m) \leftarrow y \text{ is } n-1 \wedge p(y, x) \wedge m \text{ is } n \times x. \quad (n \geq 2)$$

結果、以下の二つの確定節が  $p$  を定義する論理式として求められます。

$$p(1, 1). \quad (6.6)$$

$$p(n, m) \leftarrow y \text{ is } n-1 \wedge p(y, x) \wedge m \text{ is } n \times x. \quad (n \geq 2) \quad (6.7)$$

- 4 上の確定節は、次の点に注意しつつ Prolog プログラムに変換できます。
- 5 1. 含意記号  $\leftarrow$  を  $:-$  に置き換えます。
- 6 2. 変数は全て大文字に直します。
- 7 3. 式 (6.7) の確定節は、左辺  $p$  の第1引数  $n$  が1のときには適用できないため、式 (6.6) の右
- 8 辺にカット演算子を加えて排他制御を行います。

1 変換後の Prolog プログラムは以下の通りです。

```
2           p(1,1):-!.
3           p(N,M):-Y is N-1, p(Y,X), M is N*X.
```

4 2 番目の確定節の右辺の原子論理式の並びを変えた確定節：

```
5           p(N,M):-M is N*X, p(Y,X), Y is N-1.
```

6 などは Prolog インタプリタでは正常に動作しないことに注意してください。Prolog プログラムの  
7 右辺の原子論理式は左から順に処理されていくため、その順に変数に値が代入されていく並びに調  
8 整しておく必要があるからです。上の変換過程ではそれに留意し、あらかじめ適切な原子論理式の  
9 配置順を決めました。

## 10 6.1.2 append

11 2 番目の例として、以下の append を考えます。

```
12           (defun append (x y)
13             (cond ((null x) y)
14                   (t      (cons (car x) (append (cdr x) y))))))
```

まずこれを以下のような等式風の書き方に直します。

$$\text{append}(x, y) = y \quad \text{if } x = \text{nil} \quad (6.8)$$

$$\text{append}(x, y) = \text{cons}(\text{car}(x), \text{append}(\text{cdr}(x), y)) \quad \text{if } x \neq \text{nil} \quad (6.9)$$

次に、関数 append に対応する述語  $\text{append}_p$  を導入し、

$$\text{append}(x, y) = z \Leftrightarrow \text{append}_p(x, y, z) \text{ は真}$$

が成り立つと仮定します。そうすると、上の二つの等式は以下の確定節に置き換えることができます。

$$\text{append}_p(x, y, y) \leftarrow x = \text{nil}. \quad (6.10)$$

$$\text{append}_p(x, y, \text{cons}(\text{car}(x), \text{append}(\text{cdr}(x), y))) \leftarrow x \neq \text{nil}. \quad (6.11)$$

さて、式 (6.10) の右辺の等式  $x = \text{nil}$  は  $x$  が  $\text{nil}$  であることを要請していますから、 $x$  に  $\text{nil}$  を代入すれば  $x = \text{nil}$  を削除できます。よって式 (6.10) は以下のように等価変換されます。

$$\text{append}_p(\text{nil}, y, y). \quad (6.12)$$

次に、式 (6.11) の関数 `cons` を Prolog のドット対構築子 `[ | ]` に置換します。

$$\text{append}_p(x, y, [\text{car}(x)|\text{append}(\text{cdr}(x), y)]) \leftarrow x \neq \text{nil}.$$

ところで、上の論理式では関数呼び出し `car(x)`、`cdr(x)` を用いていますから、 $x$  は空リスト `nil` ではありません、必ず  $x = [a|b]$  という形式です。そこで、 $x$  を  $[a|b]$  に置き換え、`car(x)`、`cdr(x)` をそれぞれ  $a$ 、 $b$  に置き換えます。

$$\text{append}_p([a|b], y, [a|\text{append}(b, y)]) \leftarrow [a|b] \neq \text{nil}.$$

上の式の右辺の不等式  $[a|b] \neq \text{nil}$  は恒真です。よってこれを削除しても構いません。結果、

$$\text{append}_p([a|b], y, [a|\text{append}(b, y)]).$$

次に、変数  $z$  を導入し、関数呼び出し `append(b, y)` を左辺の原理論理式の引数部分から右辺に取り出します。

$$\text{append}_p([a|b], y, [a|z]) \leftarrow \text{append}(b, y) = z.$$

論理式 `append(b, y) = z` は `append_p(b, y, z)` に置換できます。よって

$$\text{append}_p([a|b], y, [a|z]) \leftarrow \text{append}_p(b, y, z). \quad (6.13)$$

- 1 以上で主要な変換は終了です。最後に式 (6.12)、式 (6.13) の述語名 `append_p` を改めて `append`  
 2 に置き換え、論理式を Prolog 風書き直すと、以下を得ます。

```
3      append([], Y, Y).
4      append([A|B], Y, [A|Z]) :- append(B, Y, Z).
```

- 5 考察 以下の Lisp の関数 `member` を上と同様にして Prolog のプログラムへ変換しなさい。その結  
 6 果を 5.3.2 節のプログラムと比較しなさい。

```
7      (defun member (x y)
8        (cond ((null y)      nil)
9              ((eq x (car y)) t)
10             (t              (member x (cdr y))))))
```

## 11 6.2 述語から関数へ

- 12 ここでは前節とは逆に Prolog のプログラムを Lisp のプログラムへ変換する方法を考察しましょう。

### 1 6.2.1 flatten

2 以下は、5.8.3 節の図 5.14 の Prolog プログラムです。

3 `flatten([],[]):-!. // (4)`

4 `flatten(X,[X]):-atomic(X). // (5)`

5 `flatten([X|Y],Z):-flatten(X,X2),flatten(Y,Y2),append(X2,Y2,Z). // (6)`

Lisp への変換のために述語 `flatten` に対応する関数 `flattenf` を導入し、

$$\text{fatten}_f(x) = y \Leftrightarrow \text{flatten}(x, y) \text{ は真}$$

が成り立つと仮定します。そうすると、上の三つ確定節は以下の等式に置き換えることができます。

$$\text{flatten}_f(x) = [] \quad \text{if } x = []$$

$$\text{flatten}_f(x) = [x] \quad \text{if } \text{atomic}(x)$$

$$\text{flatten}_f(w) = z \quad \text{if } w = [x|y] \wedge \text{flatten}(x, x') \wedge \text{flatten}(y, y') \wedge \text{append}(x', y', z)$$

3 番目の式に現れる `flatten(x, x')`、`flatten(y, y')` はそれぞれ `fattenf(x) = x'`、`fattenf(y) = y'` と等価ですから、その置換を行うと

$$\text{flatten}_f(w) = z \quad \text{if } w = [x|y] \wedge \text{flatten}_f(x) = x' \wedge \text{flatten}_f(y) = y' \wedge \text{append}(x', y', z)$$

次に、`flattenf(x) = x'` を削除し、式中の `x'` を `flattenf(x)` に置き換えます。また `flattenf(y) = y'` を削除し、式中の `y'` を `flattenf(y)` に置き換えます。よって

$$\text{flatten}_f(w) = z \quad \text{if } w = [x|y] \wedge \text{append}(\text{flatten}_f(x), \text{flatten}_f(y), z)$$

述語 `append` は関数 `appendf` との間に以下が成り立ちます。

$$\text{append}_f(x, y) = z \Leftrightarrow \text{append}(x, y, z) \text{ は真}$$

そこで上の式を以下のように置き換えます。

$$\text{flatten}_f(w) = z \quad \text{if } w = [x|y] \wedge \text{append}_f(\text{flatten}_f(x), \text{flatten}_f(y)) = z$$

等式 `appendf(flattenf(x), flattenf(y)) = z` を上の式から消去し、変数 `z` を `appendf(flattenf(x), flattenf(y))` に置換すると

$$\text{flatten}_f(w) = \text{append}_f(\text{flatten}_f(x), \text{flatten}_f(y)) \quad \text{if } w = [x|y]$$

次に、式 `w = [x|y]` から `x = car(w)`、`y = cdr(w)` が成り立ちます。そこで `w = [x|y]` を消去し、`x`、`y` を置換すると

$$\text{flatten}_f(w) = \text{append}_f(\text{flatten}_f(\text{car}(w)), \text{flatten}_f(\text{cdr}(w)))$$

さらに上の式中の変数  $w$  の名称を  $x$  に改名すると、

$$\text{flatten}_f(x) = \text{append}_f(\text{flatten}_f(\text{car}(x)), \text{flatten}_f(\text{cdr}(x)))$$

結局、以下の三つの等式を得ることができます。

$$\text{flatten}_f(x) = [] \quad \text{if } x = []$$

$$\text{flatten}_f(x) = [x] \quad \text{if } \text{atomic}(x)$$

$$\text{flatten}_f(x) = \text{append}_f(\text{flatten}_f(\text{car}(x)), \text{flatten}_f(\text{cdr}(x)))$$

1 これを Lisp のプログラムへ読み替えます。このとき、以下に留意します。

2 1.  $[]$  は Lisp の `nil` に置換します。

3 2.  $x = []$  は Lisp の `(null x)` に置換します。

4 3.  $[x]$  は Lisp の `(cons x nil)` に置換します。

5 4. `atomic(x)` は Lisp の `(atom x)` に置換します。

6 5. if 以下の条件式を持つ 1 行目と 2 行目の等式では、それぞれの条件が成り立つときにのみそ  
7 の値を関数値とするようにプログラムを作ります。

8 6.  $f(x)$ 、 $g(x, y)$  という形式の関数呼び出しを Lisp の  $(f\ x)$ 、 $(g\ x\ y)$  に形式変換します。

9 よって

```
10 (defun flatten (x)
11   (cond ((null x) nil)
12         ((atom x) (cons x nil))
13         (t (append (flatten (car x)) (flatten (cdr x))))))
```

14 考察 以下の Prolog プログラムを Lisp プログラムへ変換しなさい。ただし、`member` の引数の数  
15 は、Lisp、Prolog 共に 2 であることを注意しなさい（通常、Prolog の述語の引数の数は対応する  
16 Lisp の関数の引数の数よりも 1 だけ多い）。

```
17 member(X, [X|_]).
18 member(X, [_|Z]) :- member(X, Z).
```



## 1 6.2.2 まとめ

2 上に示した三つの変換の手法がそのまま全ての Lisp プログラム、Prolog プログラムに適用可能  
3 な訳ではありません。

4 しかし、関数型プログラムと論理型プログラムが

5 
$$\text{関数定義} \Leftrightarrow \text{等式} \Leftrightarrow \text{論理式} \Leftrightarrow \text{確定節}$$

6 という図式でつながっていることを強く示唆してします。Lisp のような関数型プログラムを Prolog  
7 のような論理型プログラムへ変換するには、関数定義を等式へ置き換え、その等式を論理式へ変形  
8 することで可能です。逆に論理型プログラムを関数型へ変換するには、論理式を等式へ置き換え、  
9 それを関数定義へ変形することで可能です。つまり、関数型、論理型は上の構図の共通基盤の上に  
10 成り立っている訳です。

11 ただし、両者にはひとつだけ大きな違いがあります。Lisp に限らず関数型プログラミングでは  
12 一般に高階関数を利用可能であって、高階関数は関数型プログラミングの必須機能と見なされてい  
13 ますが、論理型プログラムでは高階述語はほとんど利用されていません