プログラミング言語の概念と構造 [新装版]

ラビ・セシィ 著
神林 靖 訳

For Dianne

who taught me yet another language

Programming Languages
Concepts and Constructs
Ravi Sethi

Contents

目次

意味が変わらないことを示している。2.6節では、同じ名前 x が異なる分脈で用いられる場合を調べる。

- 検査
  誤って2個の引数でなく1個の引数に関数 *pile* を適用したらどうなるであろうか。エラーが生じる。一般にプログラミング言語はいくつかの異なった型の対象を扱うので、関数が正しい数と型の引数に適用されているかどうかを検査する必要がある。2.7節ではデータ型を扱う。この章で考慮される以外の検査も言語は行っている。たとえばLittle Quiltは、*sew* (E_1, E_2) 中の式 E_1 と E_2 が同じ高さであるか検査しなければならない。

2.2　式の記法

二項（binary）演算子は、2つの演算数に適用される。中置記法（infix notation）では、二項演算子は $a + b$ のように演算数の間に書く。他の記法である前置記法（prefix notation）では、$+ab$ のように演算子は最初に書き、後置記法（postfix notation）では、$ab+$ のように演算子は最後に書く。

より詳細には、前置記法による式は次のように書かれる（後置記法の規則も同様）。

- 前置記法による定数や変数は、定数や変数そのものである。
- 部分式 E_1 と E_2 への演算子 **op** の適用は、前置記法では **op** E_1 E_2 と書く。

中置記法で式 E は、(E) のように括弧によって囲んでもその値に影響はない。前置記法では演算子の対する演算数は曖昧なく決定できるので、括弧は不要である。前置記法の式が + で始まるならば、+ の後の次の式は + の最初の演算数で、その次の式は + の2番目の演算数であるとこが分かっている[1]。

1 前置記法のこの解釈は、ある決まった数 $k > 0$ の演算数の場合にも拡張できる。演算子に対する演算数の数をアリティ（arity）と呼ぶ。アリティ $k \geq 0$ の演算子 $\mathbf{op}^k$ の $E_1, E_2, \ldots, E_k$ への適用は、前置記法で $\mathbf{op}^k\, E_1\, E_2 \ldots E_k$ と書ける。$1 \leq i \leq k$ の時左から右へ走査して $\mathbf{op}^k$ の右の i 番目の式は、$\mathbf{op}^k$ の i 番目の演算数である。Lispのように演算子のアリティが前もって分からないときは、括弧が必要となる。Lispの式については、7.1節で議論する。

⟨*expression*⟩ ::= *a* | *b*

定数 *a* =  と *b* =  .

⟨*expression*⟩ ::= *turn* (⟨*expression*⟩) | *sew* (⟨*expression*⟩ , ⟨*expression*⟩)

組み込み関数 turn と sew は図2.2に示されている。

⟨*expression*⟩ ::= **let** ⟨*declarations*⟩ **in** ⟨*expression*⟩ **end**
⟨*declarations*⟩ ::= ⟨*declaration*⟩ | ⟨*declaration*⟩ ⟨*declarations*⟩

let 式は <declarations> の効果を <expression> に局所的とする。つまり <declarations> 中で宣言された関数と変数は <expression> 中でのみ使用可能となる。
個々の宣言の後にオプションのセミコロンがあってもよい。

⟨*declaration*⟩ ::= **fun** ⟨*name*⟩ (⟨*formals*⟩) = ⟨*expression*⟩
⟨*formals*⟩ ::= ⟨*name*⟩ | ⟨*name*⟩ , ⟨*formals*⟩

この構造は関数宣言である。関数が適用されると（以下参照）引数で仮パラメータを置き換えた後に関数本体 <expression> が評価される。

⟨*expression*⟩ ::= ⟨*name*⟩ (⟨*actuals*⟩)
⟨*actuals*⟩ ::= ⟨*expression*⟩ | ⟨*expression*⟩ , ⟨*actuals*⟩

ここで <name> は、括弧内の実引数の式に適用される関数を表す。評価の順序は次のとおり。まず引数を評価する。その上で関数本体の対応する仮パラメータをその値で置換してから関数本体の値をリターンする。

⟨*declaration*⟩ ::= **val** ⟨*name*⟩ = ⟨*expression*⟩

この構造は値宣言である。これにより <expression> の値が <name> に関連付けられる。

⟨*expression*⟩ ::= ⟨*name*⟩

let 束縛中の値宣言により <name> の値は決定される。

図2.6 Little Quilt 言語のまとめ

□例題 2.1

x と y の和は、前置記法で $+xy$ と書かかれる。$+xy$ と z の積は、$*+xyz$ と書く。したがって + 20 30 は 50 に等しいので次のようになる。

$$* + 20\ 30\ 60 = * 50\ 60 = 3000$$

同様に次の式が成り立つ。

$$* 20 + 30\ 60 = *20\ 90 = 1800$$

□

結合性と優先順位

中置記法では、演算子は演算数の間に現れる。つまり和 $a+b$ で + は a と b の間にある。それでは式 $a+b*c$ はどのように解釈すればよいのであろうか。これは a と $b*c$ の和なのか、$a+b$ と c の積なのであろうか。

中置算術式でグループ化する伝統的習慣では、積算と除算は加算や減算に対して優先する、つまり積算と除算は、加算や減算の前に演算数を取る。これを積算と除算は、加算と減算よりも高い優先順位（higher precedence）を持つという。多くのプログラミング言語では演算子 * と / に等しい順位を与え、演算子 + と - にはともにそれよりも低い順位を与えている[2]。

括弧を使うことによって中置記法の構造が明らかとなる。$a * b + c$ は $(a * b) + c$ に等しく、$d + e * f$ は $d + (e * f)$ に等しい。同様に $b*b-4*a*c$ は次のものに等しい。

$$(b * b) - ((4 * a) * c)$$

演算子の間に相対的な順位を指定しないとすると、演算子がどの演算数を取るかを括弧を使ってすべて明示しなければならない。

等しい順位を持つ演算子間では、通常左から右にグループかが行われる。式 4–2–1 は (4–2)–1 とグループ化される。つまり 1 に等しい。次のとうり。

$$4 - 2 - 1 = (4 - 2) - 1 = 2 - 1 = 1$$

もし 4–2–1 を 4–(2–1) と評価してしまったならば、誤った結果 3 が得られてしまう。

演算子が複数現れるような部分式が左から右へグループ化される時、演算子は左結合

2 第 6 章で紹介する Smalltalk-80 プログラミング言語は、この規則の例外である。Smalltalk のすべての算術演算子は等しい順位を持ち、式は左から右へと読まれる。つまり $a+b*c$ は $(a+b)*c$ に等しく $a+b$ と c の積なのである。

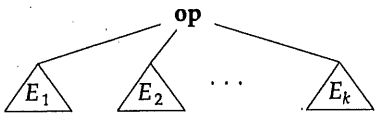
(left associative) であるといわれる。減算の演算子は左結合である。なぜなら 4–2–1 において左側の部分式が先に実行されるからである。+ , – , * , / といった算術演算子は、みな左結合である。

演算子が複数現れるような部分式が右から左へグループ化される時、演算子は右結合 (right associative) であるといわれる。たとえば指数は右結合である。次のとうり。

$$2^{3^4} = 2^{(3^4)} = 2^{81}$$

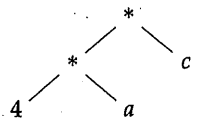
式の木表現

式の演算子/演算数の構造は、演算子を根として演算数が根の子となるような木によって表現することができる。定数や変数は木の葉として表す。つまり演算子 op を演算数 $E_1, E_2, \ldots, E_k$ ($k \geq 0$) に適用することによって式が作られるならば、その式の木は次のような形となる。



式 b*b–4*a*c に対応する木を図 2.7 に示す。この式は E_1–E_2 の形で、E_1 は b*b、E_2 は 4*a*c である。したがって演算子 – が図 2.7 の木の根となっている。根の子は、b*b と 4*a*c にそれぞれ対応する部分木である。

部分式 4*a*c はさらに E_1*E_2 の形をしていて、E_1 は 4*a、E_2 は c である。対応する木は次のようになり、根に * を持つ。



根の子は、4*a と c に対応する部分木である。

式の演算子/演算数の構造を示す木は、抽象構文木 (abstract syntax tree) と呼ばれることがある。それは式が、そもそも書かれた記法から独立した表現として構文構造を示しているからである。図 2.7 の木は、次の 3 種類の記法で書かれた同じ式の抽象構文木である。

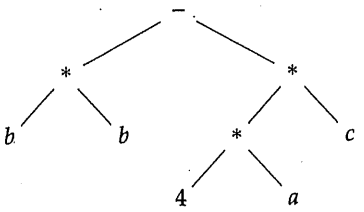


図2.7 $b*b–4*a*c$ の木表現

前置記法：　$–*bb**4ac$
中置記法：　$b*b–4*a*c$
後置記法：　$bb*4a*c*–$

抽象構文木の記法は、構造にふさわしい演算子からなる式のほかにも他の構造へと拡張できる。

2.3　式の評価

式 E_1 **op** E_2 は、次のように評価される。すなわち部分式 E_1 と E_2 をそれぞれの順位に従って評価し、E_1 と E_2 のそれぞれの結果の値に演算子 **op** を適用する。部分式 E_1 と E_2 は、どのような順序でも評価され得ることに注意されたい。

木の書き換えとしての式の評価

式の評価は、木の書き換えに対応する。7*7 が評価されて 49 を生じる時、7*7 に相当する部分木は 49 の部分木によって置き換えられる。図2.8に示す一連の木は、次

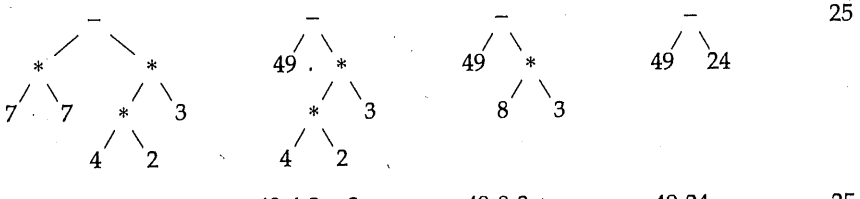


図2.8 木の書き換えの進行と対応する後置記法の式の評価の進行

の式の評価を表している。

$$
\begin{aligned}
&7 * 7 - 4 * 2 * 3 \\
&= 49 - 4 * 2 * 3 \\
&= 49 - 8 * 3 \\
&= 49 - 24 \\
&= 25
\end{aligned}
$$

図2.8の左から右への評価の進行は、後置記法ではもっと明かである。前述の式を次のように記述し直すと、最左の演算子をそれの演算数に適用することで次の行が得られることがよく分かるであろう。

$$
\begin{aligned}
&7\ 7 * 4\ 2 * 3 * - \\
&= 49\ 4\ 2 * 3 * - \\
&= 49\ 8\ 3 * - \\
&= 49\ 24 - \\
&= 25
\end{aligned}
$$

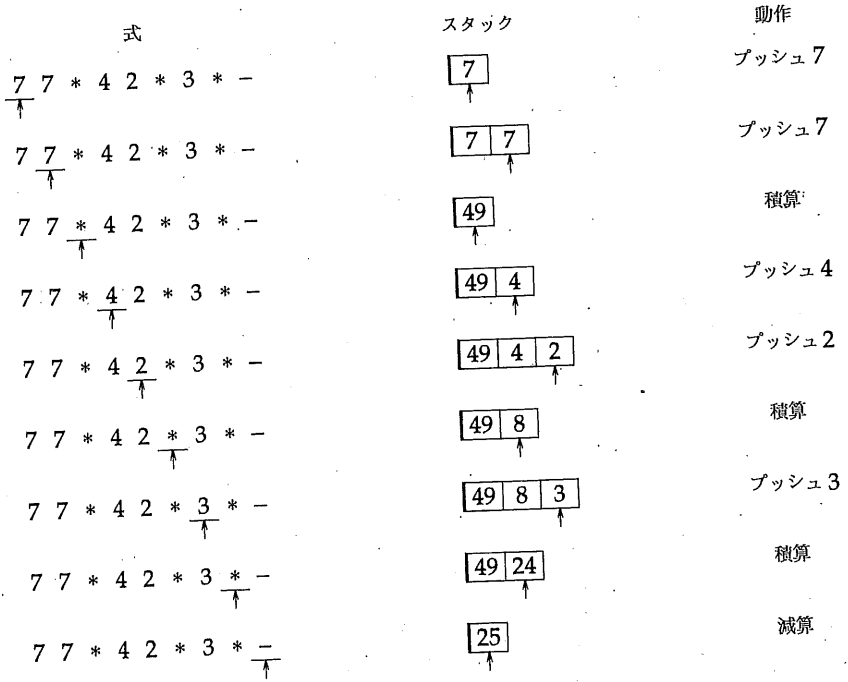


図2.9 スタックを用いた後置記法の式の評価

式の評価のスタックによる実現

木の書き換えは説明には便利であるが、式の評価を実際に木の書き換えで行うことはあまりない。次に示すスタックを補助に用いる評価のアルゴリズムは、現実の実現に近いものである。

1. 評価する式を後置記法に変換する。
2. 後置記法の式を左から右へ走査する（図2.9参照）。
   a. 定数があれば、スタックへプッシュする。
   b. 二項演算子があれば、スタックから値を2個ポップし演算子をそれらに適用する。結果はスタックへプッシュする。
3. 後置記法をすべて走査し終わると、式の値はスタック上にある。

図2.9の例題において、評価アルゴリズムの第2段階で走査される定数や演算子には下線を引いてある。スタックの上部は右にある。

2.4　関数の宣言と適用

関数は一旦宣言されると、式の中で演算子として使用できる。この節では、関数宣言と関数が適用された時に行われる計算について見ていくことにする。計算は必ず停止するとは限らず、停止しない計算の結果は定義されない。

写像としての関数

数学において全域関数（total function）fは、集合Aの各要素を集合Bの一つの要素に対応させる。Aはfの定義域（domain）であり、Bは値域（range）である。関数は、定義域の要素を値域の要素に写像（map）するという。

fの定義域Aと値域Bは、AからBへのすべての関数を表す記法$A \to B$中で示される。fがaをbに写像するのであれば、$f(a) = b$と書き、bをfのaにおける値（value）と呼ぶ。定義域A中の各aについて、B中のあるbに対して$f(a) = b$となる場合と、$b = f(a)$となるbが存在せず$f(a)$が定義されない場合がある時、そのようなfは部分関数（partial function）であるという。

関数を定義域から値域への写像としてとらえる見方では、aにおけるfの値がどのように計算されるかは指定しない。関数を定義する一つの方法は、次のように定義域の各要素についてその値をすべて列挙することである。

successor (0) = 1
successor (1) = 2
successor (2) = 3
successor (3) = 4
……

次の規則は、関数を指定するもう一つの方法を示している。

$g(x)$ は、$n^2 \leq x$ であるような最大の整数 $n \geq 0$ である。

この規則は、どのように x における g の値が計算されるか明示的に述べていない。

アルゴリズムとしての関数

プログラミング言語中の関数は、その定義域の各要素の関数値を計算するアルゴリズムとともに現れる。関数宣言は、次の3つの部分からなる。

1. 宣言された関数の名前
2. 関数のパラメータ
3. パラメータから結果を計算する規則

本章では、関数宣言の構文は次のものを使う。

fun <*name*> (<*formal-parameters*>) = <*body*> ;

たとえば次のとうり。

fun *successor* (*n*) = *n* + 1 ;

キーワード**fun** は関数宣言が始まることを示す。<*name*> は関数名で、<*formal-parameters*> は一つ以上のパラメータ名の並びである。<*body*> は評価される式である。

式の中で関数を使うことを関数の適用（application）と呼ぶ。宣言された関数の適用の規則には、次のように前置記法が用いられる。

<*name*> (<*actual-parameters*>)

ここでも <*name*> は関数名で、<*actual-parameters*> は関数の宣言内のパラメータの並びに対応する式の並びである。したがって次のものは、実パラメータ 2 + 3 に関数 *successor* を適用している。

successor (2 + 3)

仮パラメータと実パラメータは仮引数と実引数と呼ばれることもあり、インフォーマルには、仮パラメータに「パラメータ」という用語をそして実パラメータに「引数」という用語をあてる。

最内評価

最内評価（innermost-evaluation）規則のもとで次の関数適用

<*name*> (<*actual-parameters*>)

は、次のように計算される。

- <*actual-parameters*> 内の式を評価する。
- 関数本体内の仮引数を評価結果で置き換える。
- 関数本体を評価する。
- 評価した値を答としてリターンする[3]。

次の関数宣言

fun *successor* (*n*) = *n* + 1 ;

の後では、適用 *successor* (2 + 3) の最内評価は、次のように進められる。

- 引数 2+3 を評価する。
- 関数本体 *n* + 1 内の仮引数 *n* を値 5 で置き換える。
- その結果得られる式 5 + 1 を評価する。
- 答 6 をリターンする。

各関数本体の評価は関数の駆動（activation）と呼ばれる。*successor* (2 + 3) を *successor* (*puls* (2, 3)) と書き換えると、計算は次のように記述できる。

- *plus* (2, 3) を評価するために *plus* を駆動する。
- *plus* が結果 5 をリターンする。
- *successor* (5) を駆動する。
- 答 6 をリターンする。

関数本体の評価の前に引数を評価するアプローチは、値呼出し（call-by-value）と

3 この評価のステップは、言語の実現がどのような結果を計算しなければならないかを示すもので、どのように計算が行われなければならないかを示すものではない。引数の値は通常マシン・レジスタ中にあり、関数本体内のものと実際に置き換えられるわけではない。

も呼ばれる。「最内評価」という用語は、*successor* (*puls* (2, 3)) のような式で入れ子構造の関数の駆動が内側から外に向かって行われることをいう。

選択的評価

式の一部を選択的に評価して他を無視する機能は、次の構造によって提供される。

if <*condition*> **then** <*expression*>$_1$ **else** <*expression*>$_2$

ここで <*condition*> は真 **true** か偽 **false** に評価される式で、そのような式はブール (boolean) 式と呼ばれる。<*condition*> が **true** と評価されれば <*expression*>$_1$ の値が構造全体の値となり、さもなければ <*expression*>$_2$ の値が構造全体の値となる。<*expression*>$_1$ と <*expression*>$_2$ の一方だけが評価されるのである。

数の絶対値を計算する関数は、次のように定義される。

fun *abs* (*n*) = **if** *n*≥0 **then** *n* **else** 0–*n*

もう一つの例として関数 *or* を示す。これはその引数の一方が **true** であれば **true** をリターンする。

```
fun or (x, y) =
    if x = true then true
    else if y = true then true
    else false
```

y の値が **true** か **false** に限られるならば、関数 *or* は次のようにより簡潔に宣言できる。

```
fun or (x, y) =
    if x = true then true else y
```

同様に引数が両方とも **true** の時 **true** をリターンし、それ以外は **false** をリターンする関数 *and* も定義できる。

最内評価のもとでは、*or* (*E*, *F*) 内の部分式 *E* と *F* は関数本体の中へ置き換えられる前に評価される。

演算子 **andalso** と **orelse** は、ブール式を短絡評価 (short-circuit evaluation) する。短絡評価とは、必要な時に限り右側の演算数を評価することである。式 *E* **andalso** *F* は *E* が偽であれば偽で、*E* と *F* の両方が真である時に真となる。*E* **andalso** *F* の評価は左から右へ行われるので、*E* が真である時に限り *F* が評価される。同様に *E* **orelse** *F* は *E* が真であれば真となり、*E* と *F* の両方が偽である時に偽となる。*E* が

偽である時に限り F が評価される。

演算子名 andalso と orelse は、この章でこれから使う式の構文とともにStandard ML プログラミング言語から借用したものである。

2.5 再帰的関数

関数 f がその関数本体の中に f の適用を含む時、関数 f を再帰的（recursive）であるという。一般的には他の関数を介してでも f が自分自身を駆動する時、f を再帰的であるという。再帰的関数の実用的な例は後の章で考えることにする。とりあえずこの節では、再帰を考えるために簡単な例題を用いる。

フィボナッチ数列 1, 1, 2, 3, 5, 8, ... の要素は、次の関数で計算できる。

fun *fib* (n) =
　if $n = 0$ **orelse** $n = 1$ **then** 1 **else** *fib* $(n-1)$ + *fib* $(n-2)$

この関数が再帰的であるというのは、関数本体

if $n = 0$ **orelse** $n = 1$ **then** 1 **else** *fib* $(n-1)$ + *fib* $(n-2)$

が2つの *fib* の適用 *fib* $(n-1)$ と *fib* $(n-2)$ を含むからである。関数 *fib* は次の等式を満たす。

fib $(0) = 1$, *fib* $(1) = 1$, *fib* $(2) = 2$, *fib* $(3) = 3$, *fib* $(4) = 5$, *fib* $(5) = 8$, ...

再帰的関数の適用を評価するのに何も新しい考え方は必要ない。今までどうり実パラメータが評価され関数本体の中へと置き換えられる。したがって *fib* (4) の適用は *fib* の本体の n を 4 で置き換えることによって評価され、次のようになる。

if $4 = 0$ **orelse** $4 = 1$ **then** 1 **else** *fib* $(4-1)$ + *fib* $(4-2)$

$4 = 0$ も $4 = 1$ もともに偽なので、この式は次のように単純化される。

fib $(4-1)$ + *fib* $(4-2)$

この2つの適用は、*fib* の本体中の n を 4–1=3 と 4–2=2 でそれぞれ置き換えることによって個別に評価される。結局 *fib* (3) は 3 をリターンし *fib* (2) は 2 をリターンするので、それらの和 5 が *fib* (4) の値となる。

ところでフィボナッチ数列の要素を計算するのに、関数 *fib* は効率のよくないアルゴリズムに基づいている。それというのは同じ要素を繰り返し再計算しているからである。ちょうど次の式

$$fib\ (4) = fib\ (3) + fib\ (2)$$

を見たが、同じく $fib\ (3) = fib\ (2) + fib\ (1)$ であるので、次のようになる。

$$fib\ (4) = fib\ (2) + fib\ (1) + fib\ (2)$$

この等式が暗示しているように、$fib\ (4)$ の評価は $fib\ (2)$ を再計算している。n が大きくなるにつれて、$fib\ (n)$ を評価する際に行われる再計算の量は急激に増加する。

n の値が負であれば、$fib\ (n)$ の評価は次のように停止しない。

$$fib\ (-1) = fib\ (-2) + fib\ (-3) = (fib\ (-3) + fib\ (-4)) + fib\ (-3) = \ldots$$

線形再帰

関数 f の駆動 $f(a)$ が最大1回新しい f を駆動する時、f の定義が線形再帰（linear recursive）であるという。

階乗を計算する次の関数 *factotrial* は線形再帰である。

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

引数 a で *factorial* を駆動すると、a が 0 であれば 1 がリターンされ、さもなければ引数 $a-1$ で *factorial* を新たに駆動する。したがって *factorial* の各駆動は最大1回の新しい *factorial* を駆動する。

線形再帰関数の評価は、次の2段階からなる。

1. 新しい駆動が行われる巻き取り段階（winding phase）。
2. 後入れ先出し法で駆動から制御が戻る、引き続く巻き戻し段階（unwinding phase）。

□例題 2.2

factorial を省略して f と書こう。つまり関数 f は次のように定義される。

fun $f(n)$ =
　if $n = 0$ **then** 1 **else** n * $f(n-1)$;

$n \geq 1$ の時この関数は階乗を計算するので、次のように期待できる。

$$f(n) = n * (n-1) * \ldots * 1$$

$f(3)$ の評価は次のように始まる。

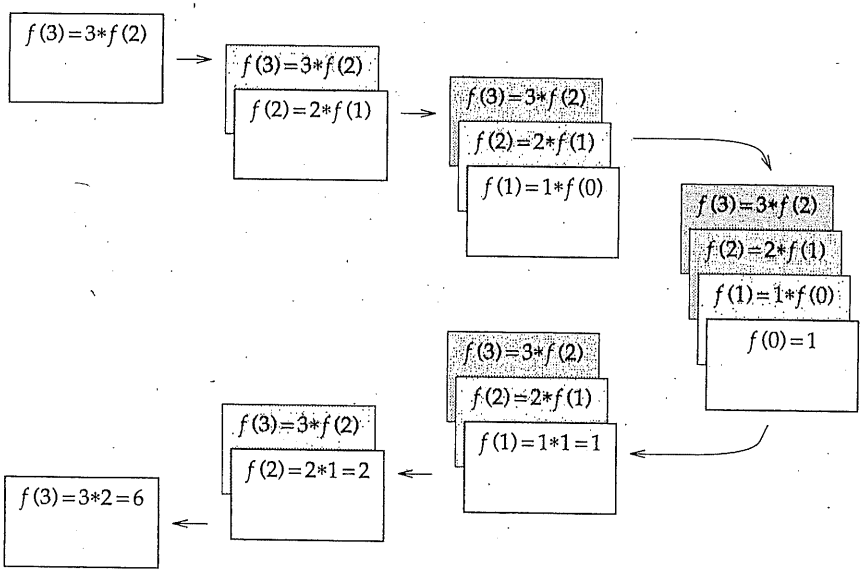


図2.10 階乗の関数の再帰的評価

if 3 = 0 **then** 1 **else** 3 * $f(3-1)$

この式は、f の本体中の仮引数 n を値 3 で置き換えることによって得られる。3 = 0 は偽なので、制御は式 3 * $f(3-1)$ へと達する。しかし次の積算

3 * $f(2)$

は、$f(2)$ の値が決まるまで実行できない。

$f(3)$ の評価に際して発生する駆動を図2.10に示す。次のように $f(3)$ を計算するには $f(2)$ を計算しなければならず、$f(2)$ を計算するためには $f(1)$ を計算しなければならない。

$$\begin{aligned} f(3) &= 3 * f(2) \\ f(2) &= 2 * f(1) \\ f(1) &= 1 * f(0) \end{aligned} \tag{2.1}$$

巻き取り段階は $f(0)$ で、それ以上新しい駆動がないので停止する。0 が仮引数 n を置き換えると次の式を得る。

if 0 = 0 **then** 1 **else** 0 * f(0 − 1)

関数本体中の判定 0 = 0 は真なので、f(0) の結果は 0 となる。

巻き戻し段階では、中断されていた駆動が後入れ先出し法により続行されて、(2.1) の右側の式がすべて評価される。f(3) の計算の要約は次のとうり。

$$\begin{aligned} f(3) &= 3 * f(2) \\ &= 3 * (2 * f(1)) \\ &= 3 * (2 * (1 * f(0))) \\ &= 3 * (2 * (1 * 1)) \\ &= 3 * (2 * 1) \\ &= 3 * 2 \\ &= 6 \end{aligned} \qquad (2.2)$$

□

末端再帰

再帰関数は、それが末端再帰（tail recursive）である時、効率的に実現できる。関数 f が再帰を必要とせずに値をリターンするか、あるいは単に再帰的駆動の結果をリターンする時、関数 f は末端再帰であるという。

次の関数 g は末端再帰である。なぜなら駆動 g (n, a) は、a をリターンするかあるいは再帰的駆動 g (n–1, n*a) の結果をリターンするからである。

fun g (n, a) =
　if n = 0 **then** a **else** g (n–1, n*a)

g が末端再帰である性質は、宣言を次のように書き換えるとより明白となろう。

$$\text{g}(n, a) = \begin{cases} a & \text{if } n = 0 \\ \text{g}(n-1, n*a) & \text{それ以外} \end{cases}$$

□例題 2.3

g (3, 1) の評価の際に発生する駆動を図 2.11 に示す。評価は次のように始まる。

if 3 = 0 **then** 1 **else** g (3–1, 3*1)

これはg (2, 3) と単純化できる。g (3, 1) の計算の要約は次のとうり。

g (3, 1) = g (2, 3) = g (1, 6) = g (0, 6) = 6

n のすべての値について式g (n, 1) は n! へと評価され、g (n, a) は n! * a へと評価される。n! は n の階乗を表す記法である。

□

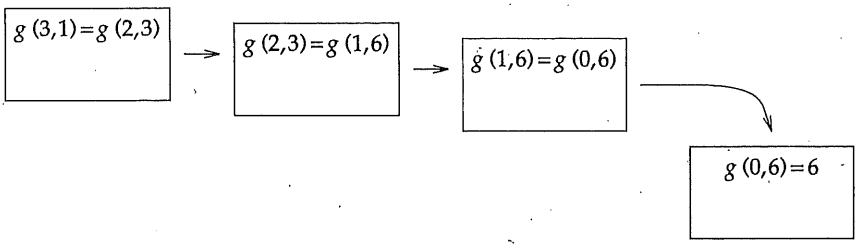


図2.11 階乗の関数の末端再帰的評価

線形末端再帰関数のすべての作業は、新しい駆動が発生する巻き取り段階で行われる。巻き戻し段階は簡単である。なぜなら最後の駆動で計算された値が評価全体の結果となるからである[4]。

対照的に、図2.10の関数 f は末端再帰ではない。なぜなら次の等式

$$f(3) = 3 * f(2)$$

で、駆動 $f(2)$ から制御が巻き戻った後で、積算が行われるからである。

2.6 字句有効範囲

一貫性を保つ限り変数名を変えても、式の値は影響を受けない。名前の変更は、局所変数または「束縛」変数の概念の導入によって精確となる。つまり束縛された変数の名前を変えてもプログラムの意味は変わらない。この名前変更の原則は、プログラム中の名前の意味を決定する「字句有効範囲」規則の基礎である。

後者関数の宣言中の仮パラメータを x から n に変えてみよう。

fun *successor* $(x) = x + 1$;
fun *successor* $(n) = n + 1$;

プログラムの意味はなんら影響を受けないはずである。式 *successor* (5) の値はどちら

---

4 第4章で、線形末端再帰関数がループへと変換できることを示す。$g(n, a)$ を計算するループは、n と a を代入可能な変数として扱う。

```
loop
    if n = 0 then return a ;
    else a := n * a ; n := n − 1 ;
    end;
end
```

の宣言を用いても 6 である。

微妙な問題は、関数宣言が非局所的な名前、つまり仮パラメータでない名前を参照している時に生じる。たとえば次の関数 $addy$ がリターンする結果は y の値によって左右される。

fun $addy(x) = x + y$;

y は非局所変数なので、分脈によってその値は決定される。問題は、どの分脈かということである。

字句有効範囲規則（lexical scope rule）は、非局所的な名前が評価される分脈を決定するのに関数宣言を囲んでいるプログラム・テキストを使用する。プログラム・テキストは実行時の状態と対照的に静的なものなので、そのような規則は静的有効範囲規則（static scope rule）とも呼ばれる。

この節では字句有効範囲規則を学ぶのに let 式を用いる。次のような一重の let 式を考えてみよう。

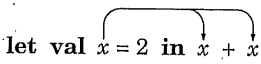
let val $x = 2$ **in** $x + x$ **end**

次のようにキーワード **val** の右にある x の出現は、x の束縛された出現（binding occurrence）と呼ばれる。

let val $x = E_1$ **in** E_2 **end**

E_2 中のすべての x はこの束縛の有効範囲（scope）にあると言われる。x の宣言自身もこの有効範囲に含まれる。この有効範囲内の x は束縛されている（bound）と言われる。名前が束縛されているとは、有効範囲内にあるその名前の出現が可視である（visible）ことを言う。

次の図の線は、x がどのように束縛されているかを示す。



「名前を変更できる変数」を精確に述べると次のようになる。「x の束縛の有効範囲中のすべての変数 x を別の変数で置き換えても、式中の値はなんら影響を受けない。」

したがって次の 2 つの式は同じ値を持つ。

let val $x = 2$ **in** $x + x$ **end**
let val $z = 2$ **in** $z + z$ **end**

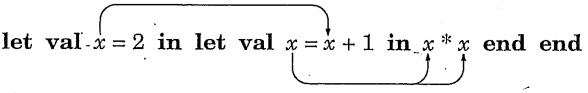
先の定義は、異なる変数を束縛するのにもそのまま使うことができる。

次の式 $x * x + y * y$ の x の有効範囲には x が2回出現する。

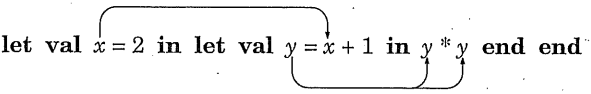
let val $x = 3$ **in let val** $y = 4$ **in** $x * x + y * y$ **end end**

y の有効範囲にも y は2回出現する。

次のように同じ変数に対して入れ子の束縛が行われた場合を考えてみよう。



最初に内側の束縛に対する名前の変更を適用する。内側の束縛内の x を y で置き換えると次の式を得る。



こうすることによって外側の x の有効範囲内の x は $x+1$ の x、内側の y の有効範囲内の y は $y * y$ の y となる。ここでも再び線は、有効範囲内で変数がどのように束縛されているかを示している。

2.7　型

式の型（type）とは、それがどのような値を取れるかと、それにどのような操作が適用できるかを示すものである。整数は加算できるが、ブール値 **true** と **false** はできない。したがって + は2つの整数型の式には適用できるが、2つのブール型の式には適用できない。

言語の設計で広く受け入れられている原理は、すべての式が唯一の型を持たなければならないというものである。この原理によって型は式を分類する機構となる。

マシン中の生データの唯一の構造は、メモリの物理的な配列だけである。同じビットの並びも、あるプログラムからは整数として、別のプログラムからは文字列の並び、そしてまた別のプログラムからはマシン命令として見られているかもしれない。たとえばマシンによっては、文字 @ に相当するビット・パターンと整数64のビット・パターンは同じである。このような柔軟性は汎用マシンの特徴である。プログラマにとって不幸なことは、マシンはあるデータがどのように使われているかチェックできないのでこの柔軟性がエラーを招くのである。

プログラミング言語における型は、3つのレベルから動機づけられている。

- マシン・レベル

  マシンが直接扱う値は、基本型（basic type）への分類できる。基本型とは、整数、文字、実数そしてブール値である。整数の加算命令と実数の加算命令とは通常マシン命令が異なるので、それらを含む式のマシン・コードを生成するのにコンパイラは型情報が必要である。

- 言語レベル

  基本型に加えて、言語は構造型（structured type）も利用できるようにしている。構造型とは、配列、レコード、リストといった単純な型から構築されるものである。構造型は、プログラムが扱うデータ構造を作成するのに使用される。型構成子（type constructor）は、構造型を構築する言語構造である。

- ユーザ・レベル

  ユーザ定義の型は、名前をつけたデータや関数をグループ化したものである。第5章および第6章で議論するクラスによって、ユーザは型を定義して問題に適するように言語を拡張することができる。

構造型

インフォーマルには、型は値の集合とその値への操作から構成される。型を定義したり使用する機構は言語によって大きく異なるので、ここでは値の集合を構築する方法を考えることで構造型を調べることにする。

構造型を学ぶのに集合論が都合がよいというのはいくつかの理由がある。第1に集合を構成する方法は、プログラミング言語における型構成子とたいへん近い対応関係にある。第2に集合論は、集合の構造を記述するのに簡潔な表記法を提供する。最後に集合を構成するものに制限がないので議論を単純化できるという点がある。言語ではユーザの利便性とか変換の容易さそして実現効率等の問題が加わるが、それらについては別途に議論する。

次のような基本集合があるものとする。

bool　{ **true**, **false** }
color　{ *red*, *white*, *blue* }
int　整数
char　文字列の集合
real　実数の集合

次に3つの集合構成子を考える。プログラミング言語にもこれらに対応するものを見つけられる。すなわち直積、関数そして列構成子である。各構成子の記述は次の3つの部分からなる。

1. 構文
2. 構成される集合の要素
3. 構成された集合内の要素の構造を調べる操作

直積

2つの集合 A と B の直積（Cartesian product）A×B とは、(a, b) と表される順序対（orderd pair）から構成される。ここで a は集合 A の要素、b は集合 B の要素である。

集合 **bool** × **color** は6個の要素を持つ。

{ (**true**, *red*), (**true**, *white*), (**true**, *blue*),
(**false**, *red*), (**false**, *white*), (**false**, *blue*) }

集合 **int** × **int** は整数の対から構成される。

直積構成子×に関連づけられているのは、操作 **first** と **second** である。それぞれ対の第1要素と第2要素を抽出する。したがって対 (**true**, *blue*) に **first** を適用すると **true** が生じ、**second** の適用は *blue* を生じる。これらの操作は射影（projection）関数と呼ばれる。

n 個の集合の直積 $A_1 \times A_2 \times \ldots \times A_n$ は、$(a_1, a_2, \ldots a_n)$ と表される n 項組（n-tuple）から構成される。ここで a_i は、$1 \leq i \leq n$ であるような集合 A_i の要素である。関連づけられる操作は、n 項組から i 番目の要素を抽出する。

関数

集合 A から集合 B へのすべての関数（function）の集合を $A \rightarrow B$ と書く。集合 $A \rightarrow B$ に関連する唯一の操作は適用（application）である。適用は、$A \rightarrow B$ から関数 f を取り出し、A から要素 a を取り出して B の要素 b を生み出す。f を a に適用するのを通常の記法では $f(a)$ と書く。

集合 **color** → **bool** は、集合 **color** から集合 **bool** へのすべての関数から構成される。**color** には3個の要素が **bool** には2個の要素があるので、そのような関数の数は $2^3 = 8$ となる。そのような関数の一つ f は次の等式を満たす。

□例題2.4

この例題は算術式のための簡単な型システムを示すもので、最初のFortranマニュアルに基づいている。式は、変数、定数、あるいは2つの式に演算子 +, −, *, / を適用して作られたものである。式の型は **int** か **real** である。

式は、次の規則が式に当てはまる時に限り型を受け取る。

- 英文字 `I` から `N` で始まる変数名は **int** 型を持つ。それ以外のすべての名前は **real** 型を持つ。この暗示的規則では `COUNT` は **real** 型となる。
- 数字はそれが小数点を含んでいれば **real** 型である。そうでなければ **int** 型である。したがって `0.5` と `.5` は、`1.` や `1.0` 同様 **real** 型である。
- 変数や定数の **int** と **real** への分類は、そのまま式へ援用される。
  式 E と式 F が同じ型であれば、

  $E + F$

  $E - F$

  $E * F$

  E / F

  はそれと同じ型となる。

したがって `I+J` は **int** 型、`X+Y` は **real** 型となる。それでは `X+I` はどうであろうか。`X` と `I` が異なる型をもつので、この例題の型システムは `X+I` を却下する。`X+I` に当てはまる規則はないので、型を持つことができず却下されるのである。 □

例題2.4の型システムとModula-2やCのそれとの主な相違点は、変数に型を関連づける規則にある。ほとんどの言語では、変数に型を指定する明示的な宣言が必要である。

すべての型システムで基本となるのは、関数適用の次のような規則である。→ は関数構成子なので、$S \to T$ は型 S から型 T への関数の型である。つまり、

f が型 $S \to T$ の関数であり、かつ a の型が S であれば、
$f(a)$ の型は T である。(2.3)

この規則について以下に詳しく見ていこう。

算術演算子

各演算子 **op** には、式 E **op** F の型を E と F の型によって指定する規則が関連づけ

られている。たとえば次のようなものである。

> E と F が **int** 型であれば、
> E **mod** F もやはり **int** 型である。

中置記法を前置記法にして、式を **mod** を対 (E, F) に適用する **mod** (E, F) と書き換えると、この規則はより (2.3) に近くなる。

> **mod** が型 **int** × **int** → **int** の関数で、
> かつ対 (E, F) が型 **int** × **int** であれば、
> **mod** (E, F) は **int** 型である。

多重定義

+ や * といった身近な演算子記号は、多重定義 (overload) されている。多重定義とは、これらの記号が分脈によって異なる意味を持つことをいう。例題2.4では、+ が整数の加算にも実数の加算にも使われていた。つまり + は2個の可能な型を持つ。

> +: **int** × **int** → **int**
> +: **real** × **real** → **real**

したがって例題2.4における + の取り扱いは、次のような2対の規則によって言い替えることができる。

> E が **int** 型で、かつ F が **int** 型であれば、
> $E + F$ は **int** 型となる。

> E が **real** 型で、かつ F が **real** 型であれば、
> $E + F$ は **real** 型となる。

上記 **mod** 演算子のところでやったように前置記法にすると、これらの規則もより (2.3) に近くなる。

強制型変換

最初の Fortran 型システム（例題2.4参照）では、`X+I` や `2*3.142` のような式は却下されたが、この制限は後の版では取り外された。ほとんどのプログラミング言語では、`2*3.142` は `2.0*3.142` であるかのように扱われる。強制型変換とはある型を他の型に変換するもので、プログラミング言語によって自動的に挿入される。`2*3.142` では、積算が行われる前に整数2が実数へと強制型変換される。

$$f(red) = \textbf{false}$$
$$f(white) = \textbf{false}$$
$$f(blue) = \textbf{true}$$

記述の都合上、直積構成子×は関数構成子→より高い優先順位を持つこととする。したがって

$$\textbf{int} \times \textbf{int} \rightarrow \textbf{int}$$

は、整数の対から整数へのすべての関数の集合である。これらの関数には、整数を加算する + や整数を積算する * が含まれる。中置記法 2+3 は、2 と 3 への + の適用を表していることを思い出して頂きたい。したがって + を整数の対 (2, 3) から整数 5 へ写像する関数として取り扱うこともできる。同様に * も整数の対 (2, 3) を整数 6 に写像する関数である。

集合

$$\textbf{int} \times \textbf{int} \rightarrow \textbf{bool}$$

は、整数の対をブール値へ写像するすべての関数の集合である。これらの関数には、整数を比較する <、≤、=、≠、≥、> といったものがある。

列

集合 A のクリーネ閉包（Kleene closure）またはスター（star）とは A^* と書き、A の要素のすべての組から構成される。したがって **color*** は次の集合となる。

{ (), (*red*), (*white*), (*blue*), (*red*, *red*), (*red*, *white*), (*red*, *blue*), ... }

ここで () は要素のない空の組を表す。

クリーネ閉包は関数型言語のリスト構成子に関連するもので、要素の有限個のリストを構成する。リスト上での操作には、リストに要素がないことを判定する *null*、リストの第 1 要素を抽出する *head*、第 1 要素以外のすべての要素を抽出する *tail* 等がある。

型システム

言語の型システム（type system）とは、言語の式に型を関連づける規則の集合のことをいう。式に型が関連づけられない時、型システムはその式を却下（reject）する。

多相型

多相型（polymorphic）関数は、パラメータ化された型を取り総称（generic）型とも呼ばれる。多相型関数についての詳しい議論は第7章で行う。そのような関数はリスト操作で使用される。

型のエラー検査での使用

型システムの規則は、言語で各演算子が適切に使われるように指定する。型検査（type checking）は、プログラムで操作が正しく適用されることを確実にする。

型検査の目的は、エラーの防止である。たとえば整数が他の何かのように誤って扱われる等、実行時に操作が誤って適用されるとエラーが生じる。より精確には、型 S を持たない a に型 $S \to T$ である関数 f を適用すると、型エラー（type error）が生じる。型エラーなく実行されるプログラムは型安全（type safe）と言われる。

可能な限りプログラムは静的（static）に検査される。つまりソース・テキストを処理している時に検査される。例題2.4の規則でいうと、式`I+J`が評価される度に`+`が整数の対に適用されることを、Fortranコンパイラはソース・テキストから判断できる。

動的（dynamic）検査はプログラムの実行時に行われる。実際動的検査は、プログラムに余分なコードを挿入してさし迫ったエラーを発見するようにしている。動的検査のための余分なコードは時間と場所を取るので、静的検査に較べて実行時の効率が劣る。動的検査の重大な欠点は、実行時までエラーがプログラムの中に潜んでしまうことである。大規模なプログラムにはめったに実行されない部分があるものなので、動的検査が型エラーを発見するまでにプログラムが長く使われることもある。

静的検査は十分効果的であるし、動的検査は手間がかかり過ぎるので、言語の実現では、ソース・テキストから静的に検査できる属性だけ検査しているものが多い。ゼロによる除算や配列の添字が範囲内あるかどうかといった、実行時に計算される値によって変わる属性はほとんど検査されない。

強いとか弱いという用語は、型システムがエラーを予防する効果の度合いを述べている。型システムは、それが安全な式だけを受け入れる時強い（strong）といわれる。換言すると強い型システムによって受け入れられた式は、型エラーなく評価されることが保証されている。強くなければ、型システムは弱い（weak）といわれる。

強いとか弱いという言葉だけでは、ほとんど何の情報も与えない。図2.12にあるように強い型システムは安全なプログラムの部分集合を受け入れる。しかしながらこの部分集合の大きさについては何の情報もない。型システムが「転ばぬ先の杖」哲学をあま

第7章
関数型プログラミング

純関数型プログラミング（pure functional programming）は、次のようにインフォーマルな原則によって特徴づけられる。

> 式の値があれば、それはその部分式の値にだけ依存する。

a+bのような式の値は、単純にaとbの値の和である。この原則によって式の副作用が排除される。したがって純関数型プログラミングは、代入のないプログラミングと特徴づけることもできる。副作用がないので、式は評価される度にいつも同じ値を持つ。

ほとんどの関数型言語は代入操作があるので純粋ではない。それにもかかわらずそのプログラミング様式は、言語の純粋な部分によって支配されている。この章では、純関数型プログラミングを扱う。

関数型言語のもう一つの特徴として、ユーザがデータのメモリ領域に気を使わなくてもよいことがあげられる。次のように述べることができよう。

> 暗黙の領域管理
>
> データ領域に関する組み込み操作が、必要に応じてメモリ領域を確保する。アクセスできなくなったメモリ領域は、自動的に解放される。

領域解放のための明示的なコードがなくて済むため、プログラムは単純にそして短くなる。このアプローチを採った結果、言語処理系の実現は、アクセス不能となった領域を

再利用するための「ごみ集め」を実行しなければならなくなった。

最後に関数型プログラミングは、次のように関数を「一等市民」として扱う。

一等値としての関数

関数は、他のすべての値と同じ地位を持つ。関数は式の値となり得る。関数を引数として渡すこともできる。また関数をデータ構造の中に収めることもできる。

関数を一等値として扱うことによって、7.3節で見るようにデータの集合に対する強力な操作を生み出すことができるようになった。

本章では現実の言語として、Lispの方言であるSchemeと第2章で紹介したStandard MLを使用する。

7.1 Scheme、Lispの方言

この節では、Schemeの関数型の部分を紹介する。代入操作については触れない。リスト・データ構造について7.2節で調べ、関数については7.3節で詳しく考えることにする。

なぜLispなのか

Lispはどこでも利用できて、以下のような広範な分野で使用されているだけでなく、人工知能のようなアプリケーションで選択された言語である。

関数型言語一般とりわけLispは、言語定義でも独特の役割を果たしている。言語定義自身も、メタ言語（meta-language）または定義言語（defining language）と呼ばれるある種の表記法によって書かれるのだが、定義言語は関数型であることが多い。事実ほとんど偶然ではあるが、最初のLispの実現が行われた時、Lispがそれ自身を定義するのに使用されたのである。

関数型プログラミングの基本概念は、Lispとともに発生した。Lispは、1958年にMcCarthyによって設計されたFortranの次に最も古い言語である。Lispは、再帰、一等関数、ごみ集め、そして（Lisp自身による）形式的言語定義等を最初に実施した言語でもある。Lispの実現は、さらにエディタ、インタプリタ、デバッガを組み合わせた統合プログラミング環境をも導くことになった。

こんなにも多くの利点があるのなら、どうしてすべての人々がLispを使用しないのであろうか。まず第1に、Lispの構文はあまりに目新しかったので、「LispとはLots of

Silly Parentheses（多くのばかげた括弧）の意である。」というような中傷まで受けることになってしまった。しかしながらこの括弧こそが価値あるものなのである。Lispの構文の単一性こそが、プログラムをデータとして操作することを容易にしているのである。

第2に、Lispプログラムは型なしである。つまりLispでは型と式を対応づけていない。プログラミング上のエラーは型検査である程度発見できるので、型がないということは、実行時に問題が生じるまでエラーに気づかないということになってしまう。MLの経験は、プログラミングの柔軟性を犠牲にすることなく型検査の利点を享受できることを示している。

第3に、過去においてLispの実現が非効率であったことがあげられる。1960年代初期では、Lispは数値計算に関して「ものすごく遅かった」のである。今日では、性能のよい実現が利用できる。

なぜSchemeなのか

Scheme は比較的小規模な言語で、 Lisp の中核部分の言語構成を提供している。Schemeはこの章の性格にふさわしい2つの特徴を備えている。それは字句有効範囲の採用と真の一等関数の機能の提供である。初期のLispは一等関数を完全にサポートしていなかっただけでなく、プログラムの意味が関数内の局所名の選び方によって左右される動的有効範囲を採用していた。

Schemeインタプリタとの対話

Schemeを学ぶよい方法は、インタプリタと対話してその応答から学ぶことである。Schemeに対して、次のように2種類の対話を考える。

1. 評価すべき式を与える。
2. 名前を値で束縛する。

関数も値たり得ることに注意されたい。

この節では、インタプリタとの対話の記録を中心に話を進めていく。インタプリタからの応答は、字下げして斜体で示すことにする。次のようにタイプすると、

```
3.14159                    ; 評価されるべき数字
```

インタプリタは、次のように応答する。

```
3.14159
```

コメントは、セミコロンから行端までである。

名前**pi**は、次のようにして**3.14159**に束縛される。

```
(define pi 3.14159) ; 変数に値を束縛する
    pi
```

すると**pi**は、**3.14159**と評価される。

```
pi                          ; 変数の値を評価する
    3.14159
```

名前に関して、Schemeは大文字と小文字の区別をつけない。したがって**pi**、**Pi**、**pI**、**PI**は、次のようにみな同じ名前を表す。

```
pI                          ; piに束縛された値を応答する
    3.14159
```

Schemeの名前には、括弧以外の特殊文字があってもよい。次のものは正当な名前である。

```
long-name, research!emlin, back-at-5:00pm
```

一般に名前は文字で始まり、数字であってはいけない。

括弧で囲まれた前置記法の式

Schemeを含むLispの方言は、演算子とオペランドをともに囲む括弧の中に前置記法で式を記述する。算術式 5 * 7 は次のように記述される。

```
(* 5 7)
    35
```

Schemeの式の一般式は、次のようになる。

$(E_1\ E_2 \ldots\ E_k)$

ここで式E_1は、$E_2, \ldots, E_k$の値に適用される演算子を表現する。部分式$E_1, E_2, \ldots, E_k$を評価する順序は指定されていない。しかしながらE_1の値をそのオペランドに適用する前に、すべての部分式を評価しなければならない。換言すれば、Schemeは最内評価または値呼び出し評価を採用している。

複数の演算子を持つ算術式は、次のような部分式の構造によってSchemeへと変換で

きる。式4＋5＊7は、次のように4と5＊7の和である。

```
(+ 4 (* 5 7))
    39
```

括弧で囲まれた前置記法を統一的に使用することによって、括弧だらけになるという費用を支払いながらもSchemeを単純な構文としている。図7.1に本章で使用される構造をまとめてある。対応するMLの構造をコメントとして含める。

`(define pi 3.14159)`	; `3.14159` に名前 `pi` を与える
`(define (sq x) (* x x))`	; **fun** $sq(x) = x*x$
`(define sq (lambda (x) (* x x)))`	; **fun** $sq(x) = x*x$
`(lambda (x) (* x x))`	; 無名の関数値パラメータ x, 関数本体 $x*x$
`(* E1 E2)`	; E_1 * E_2
`(E1 E2 E3)`	; E_1 の値を関数として引数 E_2 と E_3 に適用
`(if P E1 E2)`	; **if** P **then** E_1 **else** E_2
`(cond (P1 E1) (P2 E2) (else E3))`	; **if** P_1 **then** E_1 ; **else if** P_2 **then** E_2 ; **else** E_3
`(let ((x1 E1) (x2 E2)) E3)`	; E_1 と E_2 を評価してからそれぞれの値に束縛 ; された x_1 と x_2 を使って E_3 を評価する
`(let* ((x1 E1) (x2 E2)) E3)`	; **let val** $x_1 = E_1$; **val** $x_2 = E_2$; **in** E_3 ; **end**
`(quote blue)`	; 記号 `blue`
`(quote (blue green red))`	; リスト`(blue green red)`
`(list E1 E2 E3)`	; E_1, E_2, E_3 の値のリスト

図7.1 Schemeの構造

引用

引用は、式をデータとして扱うために必須である。引用された項目は、それ自身として評価される。引用の構文は、記号を考えることによって明かとなろう。

記号（symbol）とは、綴りをもったオブジェクトである。引用は、ある綴りが記号として扱われるか変数名として扱われるかを選択するために使用される。

項目は、次の2つの方法によって引用され得る。

```
(quote <item>)
'<item>
```

引用されていないなまえは、次の**pi**が変数名であるようにある値に束縛されている。

```
pi
    3.14159
```

引用することによって、**pi**を綴りとして扱うことができる。

```
(quote pi)
    pi
'pi
    pi
```

引用されていない * は、次のように乗算関数を表現している。

```
(define f *)                ; f を関数として定義する
    f
(f 2 3)
    6
```

引用されると * は、次のように綴り * を表す記号となる。

```
(define f '*)               ; f を記号として定義する
    f
(f 2 3)
    ERROR: Bad procedure *
```

Schemeは関数を手続き（procedure）として参照する。

関数定義

次の構文は、関数を定義する。

```
(define (<function-name> <formal-parameters>) <expression>)
```

例を示す。

```
(define (square x) (* x x)) ; fun square (x) = x*x
    square
(square 5)                  ; 関数squareを5に適用する
    25
```

この定義は、関数値に名前**square**を関連づけている。関数は1個のパラメータを取り二乗する。

関数名の関数値への束縛は、次の構文によってより明かとなる。

```
(define <function-name> <function-value>)
```

この構文を使用するためには、関数値の表記法が必要となる。schemeには、無名の関数値に対して次のような表記法がある。

```
(lambda ( <formal-parameters> ) <expression> )
```

つまり仮パラメータ**x**と関数本体**(* x x)**の関数は次のように記述できる。

```
(lambda (x) (* x x))
```

関数を記述するのに**lambda**を用いる伝統は、チャーチのラムダ計算まで遡れる。ラムダ計算については、第12章で議論する。

lambdaを使って、**square**は次のように定義できる。

```
(define square (lambda (x) (* x x)))
    square
```

Schemeでは、次のように無名の関数が式中で演算子の位置を占めることもできる。

```
(square 5)
    25
((lambda (x) (* x x)) 5)    ; 無名の関数が、5に適用される
    25
```

7.3節で、さらに進んだ**lambda**の使用例について示す。

条件式

ブール値**true**と**false**は、それぞれ**#t**と**#f**と書かれる。条件式の中では**#f**が偽として扱われるだけで、他のすべての値は真として扱われる[1]。

1 Lispの伝統に従って、Schemeの実現の中には **()** を **#f** と等しく扱っているものもある。

述語（predicate）とは、真か偽に評価される式をいう。Schemeの述語名は、次のように?で終わるのが習慣である。

`number?`	引数が数であるかどうかテストする。
`symbol?`	引数が記号であるかどうかテストする。
`equal?`	引数同士が構造同値であるかどうかテストする。

条件式は、2種類の形式を取り得る。1つは次のとうり。

```
(if P E1 E2)                ; if P then E1 else E2
```

もう1つの形式は、次のように順序選択に対応する。

```
(cond (P1 E1)               ; if P1 then E1
      ...                   ; ...
      (Pk Ek)               ; else if Pk then Ek
      (else Ek+1)           ; else Ek+1
```

条件式は、次の階乗を計算する関数の定義のような再帰的関数には必須である。

```
(define (fact n)                    ; fun fact (n) =
   if (= n 0)                       ;    if n=0
      1                             ;    then 1
      (* n (fact (- n 1))) ))       ;    else n * fact (n-1)
```

let構造

`let`構造の構文は次のとうり。

```
(let ((x1 E1) (x2 E2) ... (xk Ek)) F)
```

`let`構造の値は次のようにして決定される。最初に$E_1, E_2, ..., E_k$がすべて評価される。その上でE_iの値を表すx_iを用いてFを評価する。結果はFの値となる。

`let`構造では、部分式が名前を持っていてもよい。次の式

```
(+ (square 3) (square 4))
   25
```

は、`(square 3)`に`three-sq`、`(square 4)`に`four-sq`という名前を付けて、次のように書き改めることができる。

```
(let ((three-sq (square 3))         ; let val three_sq = square (3)
      (four_sq (square 4)) )        ;     and four_sq = square (4)
  (+ three-sq four-sq) )            ; in three_sq + four_sq end
   25
```

let構造は、共通の部分式をくくり出すのにも使用することができる。次の式の中の共通の部分式(square 3)を考えてみよう。

```
(+ (square 3) (square 3))
   18
```

この部分式を2度計算することは、次のようにして回避できる。

```
(let ((three-sq (square 3)))
  (+ three-sq three-sq) )
   18
```

let構造に順序を導入したものは、キーワードlet*によって記述することができる。変数を束縛する前に式$E_1, E_2, ..., E_k$をすべて評価してしまうletと異なり、let*は、E_{i+1}を評価する前にx_iにE_iの値を束縛する。構文は次のとおり。

```
(let* ((x1 E1) (x2 E2) ... (xk Ek)) F)
```

letとlet*の違いは、次の応答によって明かであろう。

```
(define x 0)
   x
(let ((x 2) (y x)) y)      ; xを再定義する前にyを束縛する
   0
(let* ((x 2) (y x)) y)     ; xを再定義した後にyを束縛する
   2
```

7.2　リスト

Lisp系の言語においては、プログラムとデータは同じ形をとる。両者はともにリストとして表現される。

リストの要素

リスト（list）とは、ゼロ個以上の値の並びをいう。どのような値もリストの要素となり得る。Schemeでリストの要素となり得るのは、ブール値、数、記号、リスト、そして関数である（本章では、文字、文字列、ベクトルは無視することにする）。

リストは、その要素を括弧で囲むことによって記述される。空リスト（empty listまたはnull list）とは、ゼロ個の要素を持つリストで、()と書く[2]。次のリストは、it、

2 Lispによっては、nilを()の同義語としているものもあるが、Schemeでnilは、何物かに束縛され得る名前として扱われる。

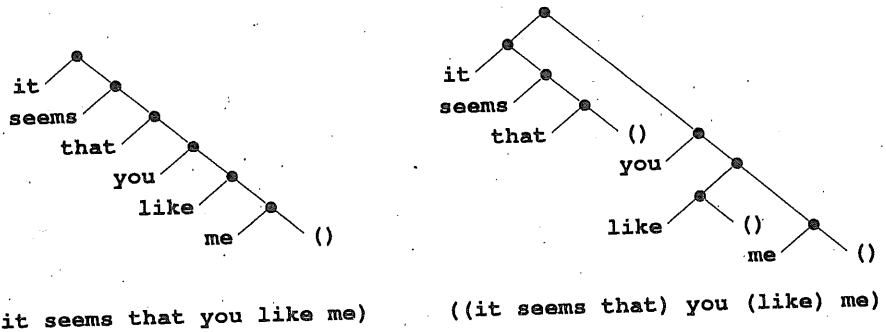


図7.2 リストの例

seems、thatの3つ記号を要素として持つ。

```
(it seems that)
```

次のリストは4つの要素を持ち、第1と第3要素はリストである。

```
((it seems that) you (like) me)
```

括弧は重要である。likeは記号であるが、(like)は要素1個のリストである。

リストの構造は、木表現で表すことによって理解しやすくなる。たとえば次の2つのリストは、木表現で図7.2のように表せる。

```
(it seems that you like me)
((it seems that) you (like) me)
```

リストと木表現の関係ついては、この章の後半で詳説する。

次のリストを考えてみよう。

```
(a ())
```

このリストは2つの要素を持ち、第1要素は記号aで、第2要素は空リスト()である。つまり要素を1個だけ持つリスト(a)は、(a ())とは異なるのである。

(+ 2 3)は、式であろうかリストであろうか。答はどちらも正しい。Schemeのインタプリタはこれを式として扱い、次のようにその値を応答する。

```
(+ 2 3)
5
```

Schemeのリストは括弧 (と) の間に書かれ、要素は空白で分かたれる。空リストは () で、リスト上の操作には次のようなものがある。

`(null? x)`	`x` が空リストであれば真、さもなければ偽
`(car x)`	非空リスト `x` の最初の要素
`(cdr x)`	リスト `x` から最初の要素を除いた残余
`(cons a x)`	`car` が `a` で、`cdr` が `x` である値、つまり次のとうり

```
(car (cons a x)) = a
(cdr (cons a x)) = x
```

図7.3 Schemeにおけるリスト

一方引用符が付いていれば、インタプリタは`(+ 2 3)`を次のようにリストとして扱う。

```
'(+ 2 3)
    (+ 2 3)
```

リストの前に1つの引用符 ' を置くことによって、それに続く構造全体がそれ自身を表すことが、次のように示される。

```
'(no quotes at (nested levels))
    (no quotes at (nested levels))
```

引用符直後の開括弧と照合する閉括弧によって、引用符の有効範囲は終了する。

リストの操作

どのような集合の要素に対しても、その構成要素を調べたり新しい集合を構成したりする操作は必要である。リストも例外ではない。リストに対する基本的操作を図7.3に示す。

述語`null?`は、次のように空リストに適用されると真をリターンし、さもなければ偽をリターンする。

```
(null? ())
    #t
```

空でないリストの構成要素を取り出す操作には、`car`と`cdr` (クダー "could-er"と発

式	省略形	値
x	x	*((it seems that) you (like) me)*
(car x)	(car x)	*(it seems that)*
(car (car x))	(caar x)	*it*
(cdr (car x))	(cdar x)	*(seems that)*
(cdr x)	(cdr x)	*(you (like) me)*
(car (cdr x))	(cadr x)	*you*
(cdr (cdr x))	(cddr x)	*((like) me)*

図7.4 carとcdrを使用してリストxの部分を取り出す

音する）がある。これらの名前はLispの伝統の一部である。carは、リストの頭部または第1要素を抽出する。cdrは、尾部または第1要素以外のすべてを取り出す。図7.4に、次のように定義されたリスト x に対してこれらの操作を適用した結果を示す。

```
(define x '((it seems that) you (like) me))
  x
```

図7.4の最初の行は、リスト x を示している。以降の行では、x から要素を抽出するのにcarとcdrを使用している。比較しやすいように、図中の同じ行に式が記述してある。

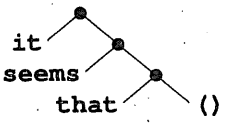
Lispは、carとcdrを連続して適用する次のような式に対して省略形を提供している。

```
(car (cdr x))
```

この式は、もとの演算子carとcdrを組み合わせた演算子cadrを使用して(cadr x)と書き改めることができる。

cons演算子はリストを構築する。(cons a x)は、頭部が a で尾部が x であるような値を生成する。ドット記法と呼ばれる表記法で(cons a x)を書くと、(a.x)となる。

リスト(it seems that)の次の木表現中のドット（点）は、cons操作に対応する。



これらの操作は、次のドット表現では明示的に示されている。

```
'(it . (seems . (that . ())))
   (it seems that)
```

厳密にいうと、**cons**操作はそのオペランドからドット対（dotted pair）と呼ばれる対（pair）を構築する。リスト（list）とは、空リストで終わるような対の連鎖であるといえる。つまり**x**に**cdr**を繰り返して適用することによって最終的に空リスト**()**が得られれば、**x**はリストなのである。

複数の要素を持つリストは、次のように要素に**list**演算子を適用することによっても構築することができる。

```
(list 'it 'seems 'that)
    (it seems that)
```

これは、次のものと等しい。

```
(cons 'it (cons 'seems (cons 'that '())))
    (it seems that)
```

7.3　有益な関数

この節では、リスト操作のための小規模な関数ライブラリを開発する。これらの関数は、共通の問題を解いたり、問題解決に適用できる便利なツールである。たとえばリストの要素を1つずつコピーする関数は、リストをコピーする過程で適用できる。

図7.5に示す等式は、ライブラリ内の関数を記述している（等式は、細い文字で印刷してある）。次の等式は、空リスト**nil**の長さが**0**で、リスト**(cons a y)**の長さはリスト**y**の長さよりも**1**だけ大きいことを述べている。

```
(length nil)        ≡ 0
(length (cons a y)) ≡ (+ 1 (length y))
```

この節では、図7.5の関数のSchemeによる実現を示す。**length**の等式は、次のように定義できる。

```
(define (length x)
  (cond ((null? x) 0)
        (else (+ 1 (length (cdr x))))))
```

ReverseとAppend

リストを要素の並びとして扱う関数は、線形再帰であることが多い。そのような関数は、リストの末端まで再帰を巻き取っていき、そしてはじめに向かってまた再帰を巻き

```
(length nil)                ≡ 0
(length (cons a y))         ≡ (+ 1 (length y))
```

```
(rev nil z)                 ≡ z
(rev (cons a y) z)          ≡ (rev y (cons a z))
```

```
(append nil z)              ≡ z
(append (cons a y) z)       ≡ (cons a (append y z))
```

```
(map f nil)                 ≡ nil
(map f (cons a y))          ≡ (cons (f a) (map f y))
```

```
(remove_if f nil)           ≡ nil
(remove_if f (cons a y))    ≡ (remove_if f y)          if (f a) is true
(remove_if f (cons a y))    ≡ (cons a (remove_if f y)) otherwise
```

```
(reduce f nil v)            ≡ v
(reduce f (cons a y) v)     ≡ (f a (reduce f y v))
```

図7.5 リスト操作の関数

戻してくる。2.5節でも述べたように、線形再帰関数の評価は次の2段階よりなる。

1. 新しい駆動が行われる巻き取り段階。
2. 後入れ先出し法で駆動から制御が戻る、引き続く巻き戻し段階。

ある関数がリストを構成する時、要素の並び順は、リストがどの段階で構成されるかによって左右される。これからリストを逆順に並べ換えるのに、なぜ図7.5中の関数**rev**が用いられて、**append**は用いられないかを考えることにする。インフォーマルにいうと、リストは巻き取り段階でコピーされると逆順になり、巻き戻し段階でコピーされると順序が変わらない。

関数**rev**は、Schemeの標準関数**reverse**と次のような関係にある。

```
(reverse x)     ≡  (rev x nil)
```

revと**append**の相違は、次の等式によって示される。

```
(rev (cons a y) z)      ≡  (rev y (cons a z))
(append (cons a y) z)   ≡  (cons a (append y z))
```

図7.6 リストを逆順にする

これらの等式は、次の関数**rev**と**append**の違いをよく捕らえている。

```
(define (rev x z)
   (cond ((null? x) z)
      (else (rev (cdr x) (cons (car x) z))) ))

(define (append x z)                          ; append版
   (cond ((null? x) z)
      (else (cons (car x) (append (cdr x) z))) ))
```

rev中の**cons**操作は、巻き取り段階で行われる。次の等式によって、引数**(b c d)**と**(a)**上での巻き取り段階が示される（図7.6参照）。

```
(rev '(b c d) '(a)) ≡ (rev '(c d) '(b a))
                    ≡ (rev '(d) '(c b a))
                    ≡ (rev nil '(d c b a))
                    ≡ '(d c b a)
```

結果のリストは、**rev**の引き続く呼び出しの第2引数として構築される。2.5節で触れたように、末端再帰は結果を戻すだけの単純な巻き戻し段階を持つ。**rev**のような末端再帰関数は、Schemeでは効率的に実現される。

同じ引数について**append**の巻き取り段階は、連続する呼び出しより構成される。

```
(append '(b c d) '(a)  は、(append '(c d) '(a))  を呼び出す。
(append '(c d) '(a)    は、(append '(d) '(a))    を呼び出す。
(append '(d) '(a)      は、(append nil '(a))     を呼び出す。
```

結果のリストは、次のように巻き戻し段階で構築される。

```
(append nil '(a))        ≡ '(a)
(append '(d) '(a))       ≡ '(d a)
(append '(c d) '(a))     ≡ '(c d a)
(append '(b c d) '(a))   ≡ '(b c d a)
```

リストの要素毎への関数の適用

フィルター（filter）とは、リストの各要素に必要な変更を施しつつリストをコピーする関数である。最も単純なフィルターは**copy**で、次のように何の変更も施さずにリストをコピーする。

```
(copy nil)          ≡ nil
(copy (cons a y))   ≡ (cons a (copy y))          (7.1)
```

フィルター copy-sq は、コピーに際してリストの要素を二乗する。

```
(copy-sq nil)       ≡ nil
(copy-sq (cons a y)) ≡ (cons (square a) (copy-sq y))   (7.2)
```

ここで、**square**はリストの要素がコピーされる際にそれらを変更するので、「変更関数」と考えることができる。

Schemeは、関数**f**をリスト**x**の要素それぞれに適用する関数**map**を提供している。つまり**map**は、**f**からフィルターを構築するツールである。たとえば次の関数を見てみよう。

```
(map square '(1 2 3 4 5))
    (1 4 9 16 25)
```

次の**map**のための等式と等式(7.1)や(7.2)との違いは、**map**が変更関数を引数として取っている点にある。

```
(map f nil)         ≡ nil
(map f (cons a y))  ≡ (cons (f a) (map f y))     (7.3)
```

関数**map**は、次のように定義できる。

```
(define (map f x)
  (cond ((null? x) nil)
        (else (cons (f (car x)) (map f (cdr x))))))
```

□例題 7.1

既存の関数を**map**とともに使えるようにするために、ラムダ記法は有効な方法である。**map**は変更関数が単項であることを、つまり1引数であることを前提としている。この例題では、ラムダ記法を用いて二項関数 * を採用できるようにする。

リスト要素を二乗する関数を再び見てみよう。

```
(map square '(1 2 3 4 5))
```

ここで**map**は、単項関数**square**をリストの各要素に適用している。リストの各要素を二乗する代わりに、各要素を2倍したいとする。引数に**2**を乗じる単項関数は、次のラムダ記法を用いて*****から生成できる。

```
(lambda (x) (* 2 x))
```

この無名の2倍する関数は、引数**x**を取り**2**と**x**に*****を適用する。この2倍する関数は、次の式中で**map**の第1引数となる。

```
(map (lambda (x) (* 2 x)) '(1 2 3 4 5))
    (2 4 6 8 10)
```

無名の2倍する関数は、次のように使用することもできる。

```
(define (double-all z)                ; 各要素を2倍する
   (let ((f (lambda (x) (* 2 x))))    ; 関数を生成し
      (map f z) ))                    ; リストの要素にmapする
```

ここでは**let**構造によって、**f**が無名の関数の略語となっている。 □

標準的なSchemeの拡張は、次のように$k \geq 1$個のリストの対応する要素に**map**が適用できるようにしている。

```
(map f x y z)
```

ここでは3つの引数を取る関数**f**が、リスト**x**、**y**、**z**の対応する要素に適用されている。

高階関数

関数の引数ないしはその結果が関数自身である場合、その関数は高階（higher order）であると呼ばれる。単純な関数から新しい関数を構築するツールは、高階関数である。

繰り返し子（iterator）とは、リストの要素に沿ってループまたは繰り返しを行って、要素それぞれに対して何ごとかを行う関数である。高階関数の1つである**map**についてはすでに述べた。もう1つの例は**remove-if**で、ある条件に合致する要素をリストから削除する関数である。厳密にいうと**remove-if**は、次のように**a**上で述語**f**が真とならない限り、リスト要素をコピーする。

```
(define (remove-if f x)
  (cond ((null? x) nil)
        ((f (car x)) (remove-if f (cdr x)))
        (else (cons (car x) (remove-if f (cdr x)))) ))
```

ここで、奇数を削除するのにSchemeの述語**odd?**を使用する。

```
(remove-if odd? '(0 1 2 3 4 5))
    (0 2 4)
```

Schemeの述語**even?**を用いて、偶数を削除することもできる。

```
(remove-if even? '(0 1 2 3 4 5))
    (1 3 5)
```

既存の述語を適用するのに、ラムダ記法を用いることもできる。次の無名の関数は、要素が**0**に等しいかどうかをテストするのに**equal?**を適用している。

```
(lambda (n) (equal? 0 n))
```

この関数は、次のように**0**を削除するのに使用される。

```
(remove-if (lambda (n) (equal? 0 n)) '(0 7 0 4 0))
    (7 4)
```

結果を蓄積する

関数**reduce**を導くような2個の特殊な場合を考えることから始めよう。この特殊な場合とは、整数のリストの和と積をそれぞれ計算することである。

空リスト**x**の和は**0**であり、非空リスト**x**の和は第1要素を残りの要素の和に加えたものである。次のようになる。

```
      (sum-all nil)      ≡ 0
(sum-all (cons a y))     ≡ (+ a (sum-all y))
```

同様に、空リストの積は**1**で、非空リスト**x**の積は第1要素を残りの要素の積に加えたものである。次のようになる。

```
      (product-all nil)      ≡ 1
(product-all (cons a y))     ≡ (* a (product-all y))
```

次に示す等式は、**sum-all**と**product-all**を一般化したものであるだけでなく、類似の関数のもとでもある。**reduce**の3つのパラメータは、二項演算子**f**、リストそし

て初期値 v である。もしリストが空であれば、値 v がリターンされる。さもなければ f が第１要素とリストの残りの要素から得られた結果に適用される。次のようになる。

```
(reduce f nil v)   ≡ v
(reduce f (cons a y) v)   ≡ (f a (reduce f y v))
```

reduceの実現は次のとうり。

```
(define (reduce f x v)
  (cond ((null? x) v)
        (else (f (car x) (reduce f (cdr x) v))) ))
```

reduceを + とリスト x そして 0 に適用することにより、関数 sum-allをシミュレートできる。

```
(reduce + '(1 2 3 4 5) 0)
    15
```

同様に、reduceを * とリスト x そして 1 に適用することにより、関数product-allをシミュレートできる。

```
(reduce * '(1 2 3 4 5) 0)
    120
```

□例題 7.2

関数 reduceと map の拡張を使用して、２つのリストを１つに混じり合わせることができる。次の２つのリストを用意する。

```
x
    (a b c)

y
    (1 2 3)
```

これらのリストから、リスト(a 1 b 2 c 3)を次のようにして構成できる。図7.7を参照されたい。

```
(reduce append (map list x y) nil)
    (a 1 b 2 c 3)
```

部分式(map list x y)は、xとyの対応する要素にlistを適用する。つまり関数 listは a と 1 に適用されてリスト(a 1)を作成し、b と 2 から(b 2)を、c と 3 から

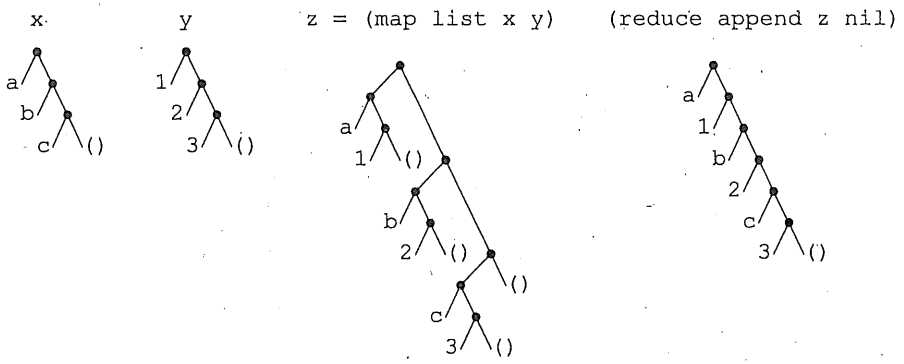


Fig. 7.7. Use of **map** and **reduce**.

図7.7 mapとreduceの使用

`(c 3)`をそれぞれ作成する。次のとうり。

```
(map list '(a b c) '(1 2 3))
    ((a 1) (b 2) (c 3))
```

その上で、reduceを使って部分リストを平坦にしている。 □

7.4 微分問題

Lispの、つまりSchemeの特徴のうちいくつかのものは、$x*(x+y+z)$のような式を微分する問題によって動機づけられている。この節の微分問題は、次の内容について説明する。

- 構文主導型変換
- データとしての式の表現
- 高階関数の使用

微分プログラムによって生成される式には、次の規則を適用して単純化する。

$$x+0=x$$
$$x*1=x \tag{7.4}$$

この節の締めくくりには、小規模な式を単純化するプログラムを示す。一般に式を単純化するプログラムは困難な問題である。

構文主導型微分プログラム

微分プログラムは、どのような形をとるであろうか。次の疑似コードは、式Eの変数xについての導関数を計算するのにEの構文を利用している。

fun $d\,(x, E) =$
　if E が定数 **then** ...
　else if E が変数 **then** ...
　else if E が加算式 $E_1 + E_2 + ... + E_k$ **then** ...
　else if E が乗算式 $E_1 * E_2 * ... * E_k$ **then** ...　(7.5)

この疑似コードは、次の関数へと展開することができる。

```
(define (d x E)
  (cond ((constant? E) (diff-constant x E))
        ((variable? E) (diff-variable x E))
        ((sum? E) (diff-sum x E))
        ((product? E) (diff-product x E))
        (else (error "d: cannot parse" E)) ))
```

述語 **constant?**、**variable?**、**sum?**、**product?**は、式 **E** が、定数、変数、加算式、乗算式であるかどうかを決定する。それぞれの場合について、実際の微分を計算する作業は、各場合に専念する問題へと委任される。

微分ルーチン **d** は、式 **E** を**sum?**と**diff-sum**のような関数の対を通してのみ扱うので、式 **E** がどのように表現されているかには左右されない。

定数

定数が数として表現されているとしよう。すると述語 **constant?**は、Schemeの述語 **number?**と同じになる。

```
(define constant? number?)
```

数の導関数はゼロである。**diff-constant**は式 **E** が定数の時だけ呼び出されて、常に**0**をリターンする。

```
(define (diff-constant x E) 0)
```

変数

変数は記号として表されているとしよう。すると述語 **varable?**は、Schemeの述語

symbol?と同じになる。

```
(define variable? symbol?)
```

変数**x**の**x**自身についての導関数は**1**である。したがって式**E**が変数**x**に等しければ、関数**diff-variable**は**1**をリターンする。さもなければ**E**は他の変数なので、**diff-variable**は**0**をリターンする。

```
(define (diff-variable x E)
  (if (equal? x E) 1 0) )
```

部分式のリスト

微分の例題へと進む前に、加算式と乗算式について復習しておこう。二項加算式と二項乗算式を表現するリストは、次のように演算子と2つのオペランドの3つの要素を持つ。

```
(+ 2 3)  ; 2 + 3 = 5
   5
(* 2 3)  ; 2 * 3 = 6
   6
```

Schemeも含まれるLisp一般は、**+**や*****は次のように任意の数の引数を取ることができる。

```
(+ 2 3 5)              ; 2 + 3 + 5 = 10
   10
(+ 2)                  ; 2 = 2
   2
(+)                    ; 何も加算しないと0となる。
   0
(* 2 3 5)              ; 2 * 3 * 5 = 30
   10
(* 2)                  ; 2 = 2
   2
(*)                    ; 何も乗算しないと1となる。
   1
```

McCarthyが1958年に開発した微分プログラムは、部分式のリストを含む加算式や乗算式も扱うものであった。そのようなリストを扱う必要性によって関数**map**が開発された。McCarthyは**map**について次のように述懐している。**map**は、「任意の数の部分項

の加算式を微分するのにどうしても必要であった。そして若干の変更を加えることにより、乗算式の微分にも適用できたのである。」

加算式と乗算式の微分規則

インフォーマルには、加算式E_1+E_2の微分は、部分式E_1とE_2の微分の和である。次の等式は、二項の加算式と乗算式に対する微分を表す。

$$d\,(x, E_1+E_2) = d\,(x, E_1) + d\,(x, E_2) \tag{7.6}$$

$$d\,(x, E_1*E_2) = d\,(x, E_1)*E_2 + E_1*d\,(x, E_2) \tag{7.7}$$

k個の部分式を持つ加算式に対する一般化は、次のとうり。

$$d\,(x, E_1+E_2+...+E_k) = \\ d\,(x, E_1) + d\,(x, E_2) + ... + d\,(x, E_k) \tag{7.8}$$

プログラミングを容易にするために、k個の部分式を持つ乗算式は、次のように二項式の場合を適用することにする。

$$d\,(x, E_1+E') = d\,(x, E_1)*E' + E_1*d\,(x, E') \\ \text{ただし}\quad E' = E_2 * ... * E_k \tag{7.9}$$

加算式の微分

加算式が、演算子 + と k≥0 個の部分式からなるリストとして表されているとしよう。Schemeには、オペランドが**cons**操作によって生成された対であるかどうかをテストする基本述語 **pair?**が備わっている。述語 **sum?**は、その引数 **E** が **car +** と対になっていれば真をリターンする。次のとうり。

```
(define (sum? E)
  (and (pair? E)
       (equal? '+ (car E)) ))
```

sum?の本体は、ブール式の左から右への短絡評価に依存している。つまり次の部分式は、**(pair? E)**が真の時に限り評価される。

```
(equal? '+ (car E))
```

関数**diff-sum**は、直接加算を扱うことはなく、補助的な関数**args**と**make-sum**を使用する。次の加算式

$(+\ E_1 E_2 \ldots E_k)$

が与えられたとすると、**args**は、次の部分式を抽出する。

$(E_1 E_2 \ldots E_k)$

そして**make-sum**は、その逆を行う。次のとうり。

```
(define (args E) (cdr E))
(define (make-sum x) (cons '+ x))
```

等式(7.8)より、E_1+E_2+...+E_kの導関数は、部分式の導関数の和である。関数**diff-sum**は、次のようにSchemeの基本関数**map**を用いて各部分式を微分する。

```
(define (diff-sum x E)
  (make-sum
   (map (lambda (expr) (d x expr))
        (args E) )))
```

ここで**d**を**map**とともに使用するためにラムダ記法を採用しているのは、298ページの例題7.1で2倍する関数を作るのに二項乗算関数 * を使用したのと類似である。次の無名の関数は、固定の **x** によって式を微分する単項関数である[3]。

```
(lambda (expr) (d x expr))
```

□例題 7.3

次のように**s**を**u**、**v**、**w**の和とする。

```
(define s (make-sum '(u v w)))
    s
```

関数 **d** を使った応答は次のようになる。

```
(d 'v s)
    (+ 0 1 0)
```

その理由を以下に説明する。**s** は加算式なので、微分関数 **d** は記号 **v** と式 **s** をともなって**diff-sum**を呼び出す。**diff-sum**の本体は、次のような無名の関数を生成する。

---

3 変数 **x** は、この無名の関数内で束縛されていないので自由変数である。Schemeの字句有効範囲規則のもとでは、この無名の関数がどこで適用されようとも **x** は**diff-sum**のパラメータ **x** を参照する。自由変数の扱いについては、第11章でSchemeのサブセットのためのインタプリタを開発するときに再び問題となる。

```
(lambda (expr) (d 'v expr))
```

その上でmapを使ってこの関数を(args s)の各要素に適用する。(args s)の各要素とはつまり(u v w)なので、結果はリスト(0 1 0)となる。最後にmake-sumがこのリストを和へと変換する。 □

乗算式の微分

述語product?は、sum?に類似である。

```
(define (product? E)
  (and (pair? E)
       (equal? '* (car E)) ))
```

関数diff-productは、部分式を持たない最も単純な式(*)に適用されると0をリターンし、1個の部分式を持つ(* E_1)に適用されるとE_1の導関数をリターンする。そしてそれ以外のすべての式に適用された時は、diff-product-argsを呼び出す。Eの部分式の数はargsを用いて、Eの部分式のリストを抽出した上でリストの長さを取ることで計算される。

```
(define (diff-product x E)
  (let* ((arg-list (args E))
         (nargs (length arg-list)) )
    (cond ((equal? 0 nargs) 0)
          ((equal? 1 nargs) (d x (car arg-list)))
          (else (diff-product-args x arg-list)) )))
```

nargsはarg-listを用いて定義されているので、逐次型のlet*構造を使用しなければならない。

ここで、等式(7.9)を繰り返しておくと便利であろう。

$$d(x, E_1+E') = d(x, E_1)*E' + E_1*d(x, E')$$
$$\text{ただし}\quad E' = E_2 * \ldots * E_k \tag{7.9}$$

関数diff-product-argsは、次のような名前を用いてこの等式を実現する。

E1	が、	E_1
EP	が、	E'
DE1	が、	$d(x, E_1)$
DEP	が、	$d(x, E')$
term1	が、	$d(x, E_1)*E'$
term2	が、	$E_1*d(x, E')$

diff-product-argsのプログラム・コードは次のとうり。

```
(define (diff-product-args x arg-list)
  (let* ((E1 (car arg-list))
         (EP (make-product (cdr arg-list)))
         (DE1 (d x E1))
         (DEP (d x EP))
         (term1 (make-product (list DE1 EP)))
         (term2 (make-product (list E1 DEP))) )
    (make-sum (list term1 term2)) ))
```

微分プログラムのまとめ

この節で示した微分プログラムは、関数dと、様々な式を扱う補助的なルーチンから構成されている。関数dは、式Eの構文を利用してEをその導関数へと変換する構文主導型変換を執り行う。

dを用いて変数vについての導関数を見つける例を次に示す。

```
(d 'v 'v)
    1
(d 'v 'w)
    0
(d 'v '(+ u v w))
    (+ 0 1 0)
(d 'v '(* v (+ u v w)))
    (+ (* 1 (* (+ u v w))) (* v (+ 0 1 0)))
```

式の単純化

微分プログラムの出力結果は、加算式から0を乗算式から1をそれぞれ削除して、かつ加算と乗算を「平坦化」することによって、より読み易くすることができる。たとえば次の加算式は、

```
(+ a (+ b c) d)
```

次のように書き直せる。

```
(+ a b c d)
```

これ以上の単純化は、式がリストとして表現されていることに由来する。たとえば(+)は0に、(*)は1へと単純化される。

この節の残りでは、次のような式

```
(+ (* 1 (* (+ u v w))) (* v (+ 0 1 0)))
```

を、次のように単純化する関数**simplify**を実現する。

```
(+ u v w v)
```

関数**simplify**は、次のように加算式と乗算式の単純化の作業をそれぞれ部分関数に託して、それ以外の式はそのままに残す。

```
(define (simplify E)
  (cond ((sum? E) (simplify-sum E))
        ((product? E) (simplify-product E))
        (else E) ))
```

加算式と乗算式の単純化の規則は対称的である。つまり加算式は0、乗算式は1を使用する。したがって**simplify-sum**と**simplify-product**は、次のように適当なパラメータを伴って共通ルーチン**simpl**を呼び出す。

```
(define (simplify-sum E)
  (simpl sum? make-sum 0 E))

(define (simplify-product E)
  (simpl product? make-product 1 E))
```

次の乗算式

```
(* 1 (* a (+ 0 b 0)))
```

に対して、関数**simpl**がどう振る舞うかを図7.8に示す。この乗算式を**E**と呼ぶことにする。つまりこの図は、次の式の評価を示している。

```
(simpl product? make-product 1 E)
```

評価は6つの段階を踏んで進む。

1. まず次のように**E**の部分式が抽出される。

   ```
   (args E)
   ```

2. 第1段階の結果をリストuとする。uの中の部分式それぞれが、次のようにして単純化される。

   ```
   (map simplify u)
   ```

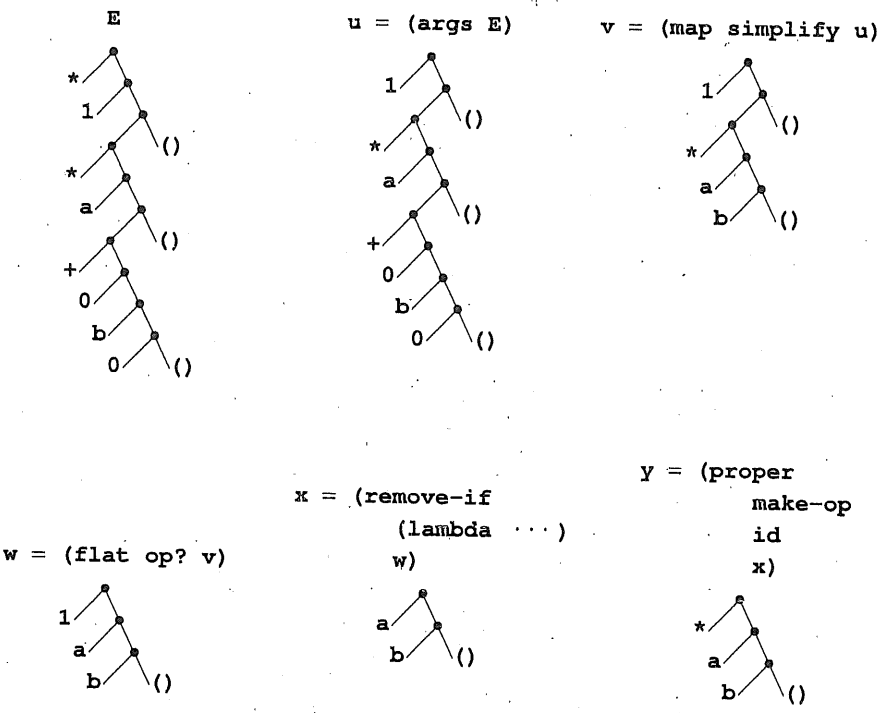


図7.8 (* 1 (* a (+ 0 b 0))) を (* a b) へと単純化する

図7.8では、1は自分自身に単純化され、`(* a (+ 0 b 0))`は`(* a b)`へと単純化される。

3. 第2段階での結果をリストvとする。補助関数flatが呼び出されて、次のように部分式のリストを平坦にする。

```
(flat op? v)
```

図7.8ではop?はproduct?であり、`(1 (* a b))`は`(1 a b)`へと平坦化される。

4. 第3段階での結果をリストwとする。wから次のようにしてすべてのidが取り除かれる。

```
(remove-if (lambda (z) (equal? id z)) w)
```

remove-ifのこのような使い方については、300ページを参照されたい。図7.8

第8章

論理プログラミング

論理プログラミングの概念は、Prologと呼ばれる言語に歴史的に結びついている。Prologは1972年に開発された言語であるが、いまだに一般に利用可能な唯一の論理プログラミング言語である。Prologは、当初自然言語処理へと応用された。爾来Prologは、アルゴリズムの記述、データベースの探索、コンパイラの記述、エキスパート・システムの構築、つまりLispのような言語が使用されるすべての応用分野で使用されるようになった。Prologは、パターン照合、後戻り探索、そして不完全な情報を扱うような応用にとりわけ向いている。

Prologは、実用的なツールである。これによって、論理プログラミングが実用となったのである。しかしながら実用性を追求するために、概念としての論理プログラミングと実用言語を区別する非純粋性を、Prologは導入せざるを得なかった。したがって論理プログラミングのインフォーマルな議論から、この章を始めることにする。

8.1 関係による計算

論理プログラミングでは、関数でなく関係を扱う。関係は引数と結果を同様に扱うので、関係によるプログラミングのほうが関数によるプログラミングよりも柔軟であるという前提に、論理プログラミングは基づいている。インフォーマルに言えば、関係には方向というものがなく、何が何によって計算されるということもない。

この節で使用するプログラム例は、リストの*append*関係である。Prolog自体は、8.2節で紹介するが、リストの記法はPrologのものを使用する。リストは、[と]の括弧の間に記述される。つまり[ ]は空リストであり、[b, c]は、b と c、2個の記号からなるリストである。Hを1個の記号、Tをリストとすると、[$H \mid T$]は、頭部をH尾部をTとするリストである。つまり、次のようになる。

$$[a, b, c] = [\, a \mid [b, c]\,]$$

関係

関係（relation）を具体的に表すと、$n \geq 0$ 列と無限も含む行の集合からなる表となろう。組$(a_1, a_2, ..., a_n)$は、a_iが、関係表のある行の$1 \leq i \leq n$ の i 列めに現れる関係にあるという。

関係 *append*は、(X, Y, Z)の形式の組の集合であり、ここでZは、Xの要素とそれに続くYの要素からなる。そのような組のいくつかを、次に示す。

append		
X	Y	Z
[ ]	[ ]	[ ]
[a]	[ ]	[a]
...	...	...
[a, b]	[c, d]	[a, b, c, d]
...	...	...

関係は、述語と呼ばれることもあるが、それは*rel*という名前の関係が、次の形式の判定式と考えることができるからである。

与えられた組が、*rel*の関係にあるか。

たとえば([a], [b], [a, b]) は*append*の関係にあるが、([a], [b], [ ])は*append*の関係にない。

この節の残りでは、関係*append*についてインフォーマルに語るために通常の書き言葉に近い表現を用いる。

規則と事実

関係は、規則（rule）によって指定できる。疑似コードによると、次のとうり。

P **if** Q_1 **and** Q_2 **and** ... **and** Q_k　ただし$k \geq 0$[1]

このような規則は、この分野の先駆的研究者Horn[1951]にちなんでホーン節（Horn clause）と呼ばれる。論理プログラミングの言語は、ホーン節に基づくことが多いが、それはホーン節が効率的な実現を導くからである。

事実（fact）は、k=0となる規則の特別な場合で、そこでPは無条件で成立する。つまり単に次のように記述される。

P.

関係 *append*は、2つの規則によって指定される。第1のものは、([ ], Y, Y)の形式の3つ組が*append*関係にあるという事実の記述である。疑似書き言葉によるこの事実の記述は、次のとうり。

[ ] と Y の *append*によって、Y を得る。

*append*の第2の規則は、完全性を示している。ここでは頭部 H と尾部 T からなるリストを現すのに [H | T] という記法を用いている。次のとうり。

[H | X_1] と Y の *append*によって[H | Z_1]を得る **if**
　X_1 と Y の *append*が Z_1である

この規則により、次のことがいえる。

[a, b] と [c, d] の *append*によって [a, b, c, d] を得る **if**
　[b] と [c, d] の *append*が [b, c, d] である

ここで、H=a、X_1=[b]、Y=[c, d]、Z_1=[b, c, d]である。[a | [b]] は [a, b] と同じリストであり、[a | [b, c, d]] は [a, b, c, d]と同じリストであることに注目されたい。

質問

論理プログラミングは、関係についての質問によって駆動される。最も単純な質問は、特定の組がある関係にあるかどうかを尋ねるものである。次の質問を見てみよう。

[a, b] と [c, d] の*append*は、[a, b, c, d]であるか。　(8.1)
　回答：yes

これは、組([a, b], [c, d], [a, b, c, d]) が*append*という関係にあるかどうかを尋ねている。

1 8.2節を先取りして簡単に説明しよう。P, Q_1, Q_2,..., Q_kは項である。項（term）は、定数であるか、変数であるか、n≥0の時 *rel* (T_1, T_2,..., T_n)の形式をとる。ここで *rel* は関係の名前、T_1, T_2,..., T_nは項である。慣例により、変数名は大文字で、定数と関係名は小文字で始めることになっている。

ホーン節は、否定情報を表現することができない。つまり、ある関係にないということを直接質問することはできない。したがってこの章での質問は、回答としてyes/noでなく、yes/失敗を持つ。「失敗」とは、yesという回答を導き出すのに失敗したという意味である。8.6節で、失敗による否定の限定された形式について考える。

次のような変数を含む質問は、より興味深い。

$[a, b]$ と $[c, d]$ を*append*して得られる Z が存在するか。　(8.2)
　回答：yes、$Z = [a, b, c, d]$

yes/失敗の質問に見えるものが、実は質問中の変数の値を求めるものであることがわかる。(8.2)の質問は、$([a, b], [c, d], Z)$ が*append*の関係になるような Zの値を求めるものなのである。

関係を用いる利点は、X と Y を*append*して Z が求まるならば、X、Y、Zのいずれをも他の2つから計算できることである。この性質から、関係による計算は誰が何から計算するかを指定しないので柔軟であると言われてきた。X は、次のようにして Y と Z から計算できる。

X と $[c, d]$ を*append*して$[a, b, c, d]$ が得られる。
　そのような X は存在するか。　(8.3)
　　回答：yes、$X = [a, b]$

同じように Y も X と Z から計算できる。

$[a, b]$ と Y を*append*して$[a, b, c, d]$が得られる。
　そのような Y は存在するか。　(8.4)
　　回答：yes、$Y = [c, d]$

(8.1)–(8.4)の質問は、同一の関係*append*のいくつかの異なった使い方を示している。既存の関係から新しい関係を定義することもできる。次の3つの規則 *prefix*、*suffix*、*sublist* において、変数 S、X、Y、Z はそれぞれリストの一部を示している（図8.1参照）。

X は Z の *prefix* である **if**
　ある Y があり、X と Y を *append* して Z が得られる時。
Y は Z の *suffix* である **if**
　ある X があり、X と Y を *append* して Z が得られる時。
S は Z の *sublist* である **if**
　ある X があり、X が Z の *prefix* である **and** S が X の *suffix* である時。

論理プログラミングとは何か

論理プログラミング（logic programming）という用語は、おおむね次のことを指している。

- 情報の表現のための事実と規則の使用
- 質問の応答への論理推論の使用

この2つの側面は、論理プログラミングにおけるプログラマと言語の分業を反映している。われわれが事実と規則を提供すると、言語が論理推論を用いて質問への回答を計算するわけである。Kowalski[1979b]は、この分業を次のようなインフォーマルな等式を用いて、巧みに表現している。

アルゴリズム = 論理 + 制御

ここで論理は事実とアルゴリズムを指定する規則を示し、制御は規則をどのような順序で適用するかによって、アルゴリズムをどのように実現するかを示している。われわれが論理部分を提供し、プログラミング言語が制御の部分を提供するわけである。

Prologにはいくつかの処理系があり、その中には制御に独自の記法を採用しているものもある。この章では、事実上の標準となっているEdinburgh Prologを使用する。処理系間における構文以外の相違は、次のような式によって表せるであろう。

$$\text{アルゴリズム}_D = \text{論理} + \text{制御}_D$$

ここでDは方言（dialect）を、そして制御$_D$はその方言の制御の記法を表す。

Edinburgh Prologの制御は、左から右へ進む。詳細は8.5節を参照してもらうとして、次の規則を見てみよう。

$$P \textbf{ if } Q_1 \textbf{ and } Q_2 \textbf{ and } \ldots Q_k. \qquad \text{ただし } k \geq 0$$

これは、次のように読むことができる。

algorithm = logic + control

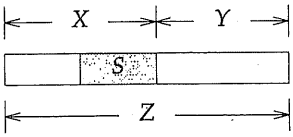


図8.1 リストの部分を参照する変数名

P を演繹するためには,
　Q_1を演繹する;
　Q_2を演繹する;
　...
　Q_kを演繹する;

この単純な方法は、驚くほど柔軟で用途が広い。もっとも不幸にして、時に無限ループに陥ることがあるし、否定に関連して誤りも導くこともある。

8.2 Prolog入門

この節では、アトミックな対象に関わる関係を考えることで、Prologを紹介しようと思う。データ構造は、次の節で考える。Prologのユニークな機能を利用したプログラムは、8.4節で扱う。

この節の例は、図8.2の矢印あるいはリンクに関わるものから採用する。

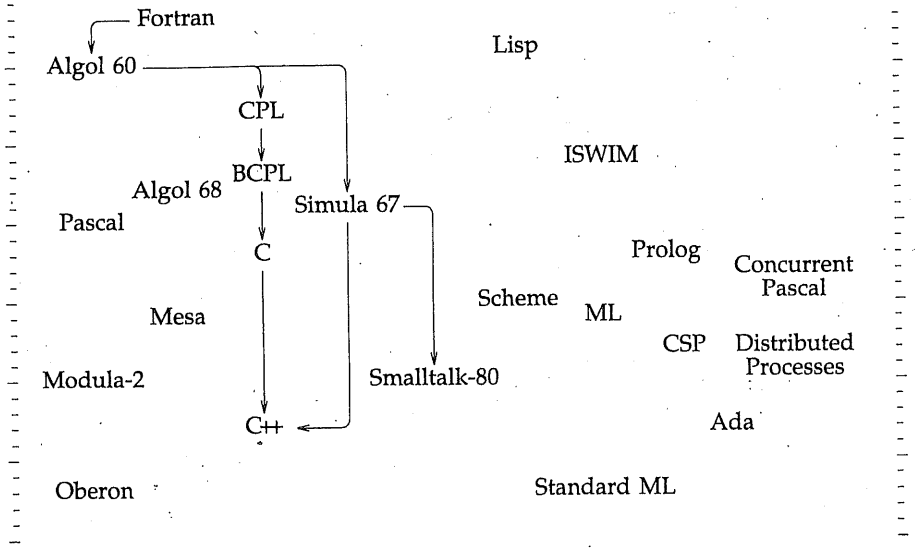


図8.2 プログラミング言語間のリンク

⟨*fact*⟩ ::= ⟨*term*⟩ .
⟨*rule*⟩ ::= ⟨*term*⟩ :- ⟨*terms*⟩ .
⟨*query*⟩ ::= ⟨*terms*⟩ .
⟨*term*⟩ ::= ⟨*number*⟩ | ⟨*atom*⟩ | ⟨*variable*⟩ | ⟨*atom*⟩ (⟨*terms*⟩)
⟨*terms*⟩ ::= ⟨*term*⟩ | ⟨*term*⟩ , ⟨*terms*⟩

図8.3 Edinburgh Prologの事実、規則、質問の基本構文

項

事実（fact）、規則（rule）、質問（query）は、項（term）を用いて指定される。Edinburgh Prologの基本構文を図8.3に示す。

単項（simple term）とは、数（number）、大文字で始まる変数（variable）、あるいはそれ自身を表すアトム（atom）である。単項の例は、次のとうり。

```
0   1972   X   Source   lisp   algol60
```

ここで`0`と`1972`は数、`X`と`Source`は変数、そして`lisp`と`algol60`はアトムである。

複合項（compound term）とは、アトムと括弧でくくられた部分項の並びからなる。そのようなアトムはファンクタ（functor）、そして部分項は引数（argument）と呼ばれる。次の複合項

```
link(bcpl, c)
```

において、ファンクタは`link`、引数は`bcpl`と`c`である。

複合項の構文は、後に必要に応じて拡張する。若干の演算子は、前置記法だけでなく、中置記法で書かれることもある。たとえば前置記法`=(X,Y)`は、`X=Y`と書き直すことができる。

特別な変数"_"は、無名の項のためのプレースホルダである。_のすべての出現は、互いに独立である。

Prologとの対話

この節の残りは、対話型セッションのスナップショットを利用する。第7章同様、システムの応答は斜体で表す。Prologを起動すると、システムは次のようなプロンプトを表示する。

```
?-
```

```
link(fortran,   algol60).
link(algol60,   cpl).
link(cpl,       bcpl).
link(bcpl,      c).
link(c,         cplusplus).
link(algol60,   simula67).
link(simula67,  cplusplus).
link(simula67,  smalltalk80).

path(L, L).
path(L, M) :- link(L, X), path(X, M).
```

図8.4 ファイルlinks中の事実と規則

これは、質問待ちの状態にあることを示している。

コンサルト（consult）構造は、事実と規則からなるファイルを読み込み、その内容を現在の規則データベースに追加する[2]。したがって

```
?- consult(links).
      links consulted ...
      yes
```

は、ファイル名**links**を読み込む。その内容は、図8.4のとうり。事実の並びは

```
link(fortran, algol60).
```

から始まりアトム間の**link**関係を指定している。この事実は**(fortran, algol60)**の対が**link**に属していることを示す。

存在の質問

質問

$\langle term \rangle_1, \langle term \rangle_2, \ldots, \langle term \rangle_k.$　　ただし $k \geq 1$

は、次の疑似コードに対応する。

$\langle term \rangle_1$ **and** $\langle term \rangle_2$ **and** ... **and** $\langle term \rangle_k$ であるか。

---

2 データベースの規則を上書きするためには、*reconsult*を用いる。システムによっては、特別なファイル名*user*を用いることによって、直接規則を入力できるようにしているものもある。

質問は、ゴール（goal）とも呼ばれる。質問文中の個々の項を「部分ゴール」呼ぶこともある。しかし厳密には、ゴールと部分ゴールの間に何の区別もないのは、項と部分項の間に何の区別もないのと同じである。

次の質問文

```
?- link(cpl,bcpl), link(bcpl,c).
     yes
```

には変数がないので、応答は単にyesである。

質問文中の変数は、適当な対象が存在するか否かを問うている。次の質問

```
?- link(algol60,L), link(L,M).
```

は、したがって次のように読むことができる。

link (*algol60*, *L*) **and** *link* (*L*, *M*)
　となるような L と M が存在するか。

質問への解（solution）は、質問文を真とするような変数の値への束縛である[3]。解を持つ質問文を充足可能（satisfiable）であるという。充足可能な質問に対してシステムは、次のように応答する。

```
?- link(algol60,L), link(L,M).
     L = cpl
     M = bcpl
```

ここで2つの選択岐がある。

- 改行文字をタイプする。Prologは*yes*と応え、別解が存在することを示す。その上で次の質問を受け付けるプロンプトを表示する。
- セミコロンを入力してから改行文字をタイプする。Prologは、別解を表示するか*no*と応答する。*no*を表示したときは、それ以上の解は見つけられなかったことを示す。

次の応答で、セミコロンは別解を次々と問い合わせている。

---

3 厳密には、質問文中のすべての変数は暗黙のうちに存在記号により限定されている。量化記号を明示するとこの質問は、次のようになる。

$\exists L, M.$ *link* (*algol60*, *L*) **and** *link* (*L*, *M*) ?

```
?- link(algol60,L), link(L,M).
     L = cpl
     M = bcpl ;

     L = simula67
     M = cplusplus ;

     L = simula67
     M = smalltalk80 ;

     no
```

変数は、質問文のどこに現れてもよい。次の質問

```
?- link(L, bcpl).
```

は、**bcpl**へリンクしている対象を、そして次の質問

```
?- link(bcpl, M).
```

は、**bcpl**がリンクしている対象を問い合わせている。

普遍的な事実と規則

規則

<term> **:-** <term>$_1$, <term>$_2$, ... , <term>$_k$.　　ただし$k \geq 1$

は、次の疑似コードに対応する。

<term> **if** <term>$_1$ **and** <term>$_2$ **and** ... **and** <term>$_k$

:- の左辺の項は頭部（head）と呼ばれ、**:-** の右辺の項は条件部（condition）と呼ばれる[訳注1]。

事実は、規則の特別な場合である。事実は頭部のみで条件部を持たない。

次の事実と規則は、関係**path**を指定している。

```
path(L, L).                                    (8.5)
path(L, M) :- link(L, X), path(X, M).          (8.6)
```

考え方は、経路（path）はゼロ個かそれ以上のリンクから構成されるというものである。**L**から**L**までのゼロ・リンクも経路として考える。**L**から**M**への経路は、**L**からある**X**へのリンクで始まり、その経路を辿ることによって**X**から**M**へ到着する。

訳注1：**:-**の右辺は、本体（body）と呼ぶのが一般的であるが、著者は条件部（condition）という用語を使っている。

規則の頭部の変数は、任意の対象で置き換えることができる。したがって事実(8.5)は、次のように読める。

すべての L について、
$path\ (L, L)$.

規則(8.6)では、変数は条件部に現れるだけで、頭部には現れない。そのような変数は、条件を充足する何がしかの対象を示している[4]。したがって規則(8.6)は、次のように読める。

すべての L と M について、
$path\ (L, M)$ **if**
次のような X が存在する
$link\ (L, X)$ **and** $path\ (X, M)$

失敗による否定

Prologは、質問文を充足するのに失敗すると、質問に*no*と応える。失敗による否定(negation as failure)の仮定とは、次のように言っているのと同じである。

「証明することができなければ、偽に違いない」

図8.4の事実は、**lisp**から**scheme**へのリンクについて何も述べていない。したがって次の質問には、*no*と応答が返る。

```
?- link(lisp,scheme).
   no
```

同様に**not**演算子も真の論理否定でなく、失敗による否定を表す。質問 **not**(P) は、システムがPを演繹できなければ真として扱われる。失敗による否定は単純な場合には機能する。複雑な場合についての問題は8.6節で扱う。次の例題は**not**の適用を含む。

□例題 8.1

図8.2中の2つの言語 L と M で、同一の言語 N にリンクしているものが存在するか。この質問を素朴に書くと、次のようになる。

---

4 厳密にいえば、事実と規則中のすべての変数は暗黙のうちに全称記号で限定されている。量化記号を明示すると、この規則は次のようになる。

$\forall L, M, X.\ path\ (L, M)$ **if** $(link\ (L, X)$ **and** $path\ (X, M))$

X は頭部に現れないので、この規則は、論理的に次の式に等しい。

$\forall L, M.\ path\ (L, M)$ **if** $(\exists X.\ link\ (L, X)$ **and** $path\ (X, M))$

```
?- link(L,N), link(M,N).
     L = fortran
     N = algol60
     M = fortran
```

次に、変数Lと Mは異なった値でなければならないという条件を追加する。

```
?- link(L,N), link(M,N), not(L=M).
     L = c
     N = cplusplus
     M = simula67 ;

     L = simula67
     N = cplusplus
     M = c ;

     no
```

この2つの解は明らかに異なっている。なぜなら個々の変数がそれぞれ異なった値を持っているからである。

経験的に、notは既知の値かあるいは、あるいはnotが適用される前に値が決まる場合の判定に使用できることがわかっている。質問文の項の順序を、次のように変えてみよう。

```
?- not(L=M), link(L,N), link(M,N).
     no
```

これは失敗する。なぜなら質問が始まった時点で変数Lと Mの値は不明だからである。不明な値は等しくなり得るので、not(L=M)は失敗する。 □

単一化

Prologは、次の式をどのように解くであろうか。

```
?- f(X,b) = f(a,Y).
     X = a
     Y = b
```

項Tの具体値（instance）は、Tの中の変数を部分項によって置換して求められる。ただしある変数のすべての出現を、同一の部分項によって置換しなければならない。

したがってf(a,b)は、f(X,b)の具体値である。なぜならこれはf(X,b)中の変数Xを部分項aによって置き換えることによって得られるからである。同様にf(a,b)は、

f(a,Y)の具体値でもある。なぜならこれは、f(a,Y)中の変数Yを部分項bによって置き換えることによって得られるからである。

別の一例として、g(a,a)とg(h(b),h(b))は、ともにg(X,X)の具体値である。しかしながらg(a,b)は、g(X,X)の具体値ではない。なぜなら同一の変数Xのひとつの出現をaで置き換え、もうひとつの出現をbで置き換えることはできないからである。

Prologにおける演繹は、単一化の概念に基づいている。単一化とは、2つの項T_1とT_2が共通の具体値Uを持つ時に単一（unify）とされるというものである。T_1とT_2の両方に変数が現れる時には、T_1とT_2中の変数のすべての出現が同じ部分項によって置換されなければならない。

項f(X,b)とf(a,Y)は、共通の具体値f(a,b)を持つので単一化される。

規則が適用される時、単一化は暗黙のうちに行われている。関係identityが、次の事実によって定義されるとする。

```
identity(Z,Z).
```

次の質問の応答を計算するために、単一化が使用される。

```
?- identity( f(X,b), f(a,Y) ).
     X = a
     Y = b
```

応答は、identity(Z,Z)を次の文と単一化することによって計算される。

```
identity( f(X,b), f(a,Y) )
```

これによって、Zがf(X,b)とf(a,Y)に単一化される。つまりf(X,b)とf(a,Y)が、単一化されるのである。

算術演算

演算子 = は、Prologでは単一化を表す。つまり

```
?- X = 2+3.
     X = 2+3
```

では、単に変数Xが項2+3に束縛されるだけである。

中置記号is演算子は、次のように式を評価する。

```
?- X is 2+3.
     X = 5
```

is 演算子によって X は 5 へと束縛されるので、次の質問

```
?- X is 2+3, X = 5.
     X = 5
```

は充足される。しかしながら

```
?- X is 2+3, X = 2+3.
     no
```

は失敗する。なぜなら2+3を5とは単一化できないからである。項2+3は、演算子 + の引数2と3への適用であるのに対し、5は単に整数の5である。したがって2+3と5を単一化するわけにはいかない。

8.3 Prologのデータ構造

Lispで使われるようなリストを表現するのに、Prologはいくつかの記法を用意している。記法を調べた後では、それらが単に項に対する構文上の飾りにすぎないことがわかる。つまりこれらの構文は、何か特別な機能をつけ加えているわけではないのである。

項をデータとして見るにつれて、リストよりも他のデータ構造、とりわけ木が都合のよいことがわかる。

Prologのリスト

リストを記述する最も単純な方法は、要素を並べることである。3つのアトムa、b、cからなるリストは、次のように記述できる。

```
[a, b, c]
```

空リストは[]と書く。

リストの始めの方と後に続くものを | で分けて指定することもできる。つまりリスト[a,b,c]は、次のように記述することもできる。

```
[a, b, c | []]
[a, b | [c]]
[a | [b, c]]
```

この記法の特別な場合が頭部 H、尾部 T のリストであり、[H|T]と書く。頭部とはリストの最初の要素であり、尾部とは頭部を除いた残りの要素からなるリストである。

単一化はリストから要素を取り出すのに利用できるので、頭部と尾部を取り出すため

の明示的な演算子は必要ない。次の質問

```
?- [H|T] = [a,b,c].
     H = a
     T = [b,c]
```

の解は、変数 H をリスト [a,b,c] の頭部に、変数 T を尾部にそれぞれ束縛している。次の質問

```
?- [a|T] = [H,b,c].
     T = [b,c]
     H = a
```

は、部分的にしか指定されていない項を取り扱うPrologの能力を示している。項 [a|T] は、頭部が a で変数 T で示される不明の尾部からなるリストの部分仕様である。同様に [H,b,c] は、不明の頭部 H と尾部 [b,c] からなる部分仕様である。これら2つの仕様を単一化するためには、H が a を、T が [b,c] をそれぞれ表示するようにしなければならない。

□例題 8.2

リストのappend関係は、次の規則で定義される。

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X, Y, Z).
```

この規則は、8.1節の疑似コードのProlog版である。どういうことかというと、空リスト [] とリスト Y の連結は Y である。また X と Y の連結の結果が Z であれば、[H|X] と Y の連結は [H|Z] となる。

次の質問の応答は、X、Y、Zのいずれもが、appendの規則を用いて他の2つから計算できることを示している。

```
?- append([a,b], [c,d], Z).
     Z = [a,b,c,d]
?-- append([a,b], Y, [a,b,c,d]).
     Y = [c,d]
?- append(X, [c,d], [a,b,c,d]).
     X = [a,b]
```

次の質問は、引数の不整合が却下されることを示している。

```
?- append(X, [d,c], [a,b,c,d]).
     no
```

8.4節で議論するリストへの別のアプローチである差分リストによって、appendはより効率的に実現することができる。 □

データとしての項

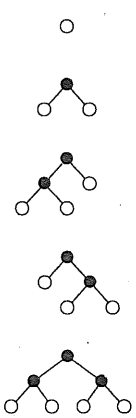
リストと項の関係は、次のとうり。`[H|T]`は、項 `.(H,T)`と構文が異なるにすぎない。つまり、

```
?- .(H, T) = [a,b,c].
     H = a
     T = [b,c]
```

したがってドット演算子あるいはファンクタ"."は、Lispのconsに相当し、リストは項であるといえる。リスト`[a,b,c]`の項は、次のとうり。

```
.(a, .(b, .(c, [])))
```

木と項の間には、一対一の対応がある。つまりいかなる木も項として書け、かついかなる項も木として描ける。しがたって木を使ってシミュレートできるデータ構造なら何でも項によってシミュレートできる。2.6節の例題を用いて、二分木を項で書いてみよう。図8.5に示すように葉にはアトム`leaf`を、2個の引数を持つ分岐節点にはファンクタ`nonleaf`を対応させる。



```
leaf
nonleaf(leaf, leaf)
nonleaf(nonleaf(leaf, leaf), leaf)
nonleaf(leaf, nonleaf(leaf, leaf))
nonleaf(nonleaf(leaf, leaf), nonleaf(leaf, leaf))
```

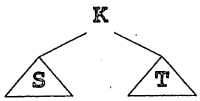
図8.5 二分木を表現する項

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

図8.6 二分探索木上の関係member

□例題 8.3

61ページの例題2.5の二分探索木は、直接対応する Prolog の表現を持つ。アトム**empty**が空な二分探索木を表現し、項**node(K,S,T)**が次の木を表現するものとしよう。



この木は、整数値 K を根に持ち、そして左部分木 S と右部分木 T を持つ。

図8.6の規則は、ある整数が木のいずれかの節点に現れるか否かを判定する関係**member**を定義している。**member**の2つの引数は、整数と木である。次の事実

```
member(K, node(K,_,_)).
```

は、K が根にあれば、木の中に現れると解釈できる。特別な変数 _ のそれぞれの出現は、別々の無名項のためのプレースホルダである。つまり**node(K,_,_)**は、根に K を持ち、無名の左部分木と右部分木を持つ二分探索木を表現している。次のように書き改めることにより、**member**が整数 K と木 U からなる対の関係であることを強調できる。

```
member(K, U) :- U = node(N,S,T), K = N.
```

規則

```
member(K, node(N,S,_)) :- K < N, member(K, S).
```

は、次のように解釈できる。

> K は、木 *node* $(N, S\_)$ 中にある **if**
> $k < N$ で、K が左部分木 S 中にある。

次の関係**insert**を定義することは、読者に委ねる。

> K を S に挿入して T を得る。

任意の整数を木から削除する関係 **delete**の定義も、同様に読者に委ねよう。　□

例題8.3のアトム**empty**とファンクタ**node**は、例題2.5から採った次のデータ型宣言中の値構成子*empty*と*node*をシミュレートしている。

datatype *searchtree* = *empty* | *node* **of int** * *searchtree* * *searchtree* ;

MLの他のデータ型も、同じようにシミュレートすることができる。

木よりも一般的なデータ構造も可能である。Prologの変数を活用することで、項は共有部分を持つデータ構造を実現できる。項**node(K,S,S)**は、次のグラフを表している。

この章では取り上げないが、項によって閉路を持つグラフも表現できる。

8.4　プログラミング技法

この節で調べるプログラミング技法は、Prologの長所すなわちバックトラッキングと単一化である。バックトラッキングによって、もし解が存在すればそれを発見できる。単一化によって、値が後で決まるデータのプレースホルダとして変数を使用できる。

この節の技法を注意深く使用することによって、効率のよいプログラムを作成できる。この節のプログラムは、Prologの部分ゴールを左から右へ評価する規則によっている。したがってプログラムの形を変えると、正しく動作しなくなることがある。その理由は、8.5節でPrologにおける制御を議論する際に明らかにする。

推測と確認

推測と確認の質問は、次の形式を取る。

guess (*S*)　**and**　*verify* (*S*)
　　となる *S* が存在するか。

ここで*guess* (*S*)と*verify* (*S*)は、部分ゴールである。このような質問に対してPrologは、*verify* (*S*)を充足するものが見つかるまで*guess* (*S*)で解を生成することによって応答する。このような質問は、生成と判定（generate-and-test）問い合わせとも呼ばれる。

同様に推測と確認（*guess-and-verify*）規則は、次の形式を採る。

conclusion (...) **if** *guess* (...,*S*,...) **and** *verify* (...,*S*,...)

□例題 8.4

この例題での推測と確認規則は、次のとうり。

```
overlap(X, Y) :- member(M, X), member(M, Y).
```

どういうことかというと、X と Y の両者のメンバーであるような M が存在すれば、リスト X と Y は交わるという。最初のゴール**member(M, X)**はリスト X 中のある M を推測し、2番目のゴール**member(M, Y)**が、リスト Y 中に M があるかどうかを確認している。

memberの規則は、次のとうり。

```
member(M, [M|_]).
member(M, [_|T] :- member(M, T).
```

最初の規則により、リストの頭部が M であれば、M はそのリストのメンバーである。2番目の規則により、M がリストの尾部のメンバーであれば、M はそのリスト全体のメンバーである。

次の質問

```
?- overlap([a,b,c,d], [1,2,c,d]).
    yes
```

が、どうして*yes*と応答するかは、次の質問を見ると解る。

```
?- member(M, [a,b,c,d]), member(M, [1,2,c,d]).
```

質問の最初のゴールが解を生成し、第2のゴールがその解が受け入れられるものかどうかを判定している。最初のゴールにより生成される解は次のとうり。

```
?- member(M, [a,b,c,d]).
    M = a ;
    M = b ;
    M = c ;
    M = d ;
    no
```

初めの2個は受け入れられないが、3番目の解は、次のように第2のゴールによって受け入れられる。

```
?- member(a, [1,2,c,d]).
     no
?- member(b, [1,2,c,d]).
     no
?- member(c, [1,2,c,d]).
     yes
```

□

ヒント

Prologの計算は左から右へ実行されるので、推測と確認質問の部分ゴールの順序は効率に影響する。解の数が少ない部分ゴールを推測ゴールに選ぶべきである。

ゴールの順序が効率に大きく影響する極端な例を見てみよう。次の2つの質問を考えて頂きたい。

```
?- X = [1,2,3], member(a,X).
     no
?- member(a,X), X = [1,2,3].
     [無限の計算へ陥る]
```

関係**member**は、例題8.4と同じである。質問

```
?- X = [1,2,3], member(a,X).
     no
```

の推測ゴール**X=[1,2,3]**は解を1つしか持たず、その解は**member(a,X)**を充足しない。なぜなら**a**は、リスト**[1,2,3]**中にないからである。他方、次の質問

```
?- member(a,X), X = [1,2,3].
     [無限の計算へ陥る]
```

の推測ゴール**member(a,X)**は、次のように無限個の解を持つ。

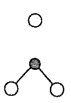
```
X = [a|_] ;          a が X の最初の要素の場合
X = [_,a|_] ;        a は X の2番目の要素の場合
X = [_,_,a|_] ;      a は X の3番目の要素の場合
```

これらのいずれも**X**をリスト**[1,2,3]**に束縛することはない。それゆえ**X=[1,2,3]**を充足するものを求めてPrologがいつまでも解を試すので、無限の計算に陥ってしまうのである。

プレースホルダとしての変数

これまで変数は規則と質問の中で使われてきたが、対象を表す項の中では用いられていなかった。変数を含む項は、変更可能なデータ構造をシミュレートするのに使用できる。変数は、後で記入される部分項のためのプレースホルダの役割を果たす。そのような項は、例題8.5で効率の良い待ち行列を実現するのに用いられる。

8.3節の項による、次のような二分木の表現を思い出して頂きたい。



leaf

nonleaf(leaf, leaf)

項**leaf**と**node(leaf,leaf)**は、完全に指定されている。それと対照的に変数**X**を含むリスト**[a,b|X]**は、部分的に指定されているという。なぜなら**X**が何を表すか不明だからである。このリスト**[a,b|X]**では、**a**が第1要素、**b**が第2要素で、変数**X**がリストの残りを表している。もし**X**が**[]**と単一化されるならば、**[a,b|X]**は**[a,b]**を表すことになり、もし**X**が**[c]**と単一化されるならば、**[a,b|X]**は**[a,b,c]**を表すことになる等、**X**はいろいろな可能性を持つ。

開リスト（open list）とは、変数で終わるリストをいう。そこで変数は、リストのエンドマーカ変数（endmarker variable）と呼ばれる。空な開リストは、エンドマーカ変数のみからなる。リストが開リストでなければ、閉じている（closed）といわれる。

内部的にPrologは、アンダースコア_が先行する整数で示されるマシン生成の変数を用いる。次の対話で、マシン生成の変数_1は、エンドマーカ**X**に対応している。

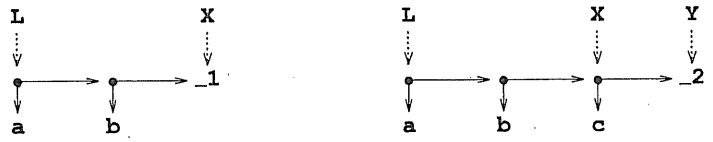
```
?- L = [a,b|X].
     L = [a,b|_1]
     X = _1
```

Prologは、質問に答えたり規則を適用する度に新しい変数を生成する。

エンドマーカを単一化することで、開リストは変更される。次の質問は、**L**をエンドマーカ**Y**を持つ新しい開リストへと拡張している（図8.7参照）。

```
?- L = [a,b|X], X = [c,Y].
     L = [a,b,c|_2]
     X = [c|_2]
     Y = _2
```

エンドマーカ変数の単一化は、変数への代入に似ている。図8.7でリストLは、**_1**を



(a) X の束縛前　　(b) X=[c|Y]. の実行後

図8.7 エンドマーカを単一化することによる開リストの拡張

```
setup(q(X,X)).
enter(A, q(X,Y), q(X,Z)) :- Y = [A|Z].
leave(A, q(X,Z), q(Y,Z)) :- X = [A|Y].
wrapup(q([],[])).
```

図8.8 待ち行列操作の規則

[c|_2]と単一化することで、[a,b|_1]から[a,b,c|_2]へと変化している。

開リストを用いる利点は、リストの終わりへのアクセスが速い、つまりエンドマーカを通じて一定時間でできるところにある。次の例題では、待ち行列を実現するのに開リストを用いている。

□例題8.5

この例題では、図8.8の待ち行列を操作する規則について議論する。

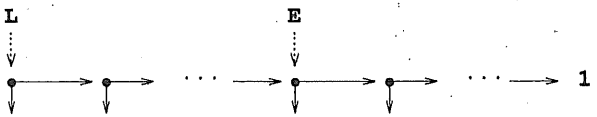
関係enter(a,Q,R)は、インフォーマルに次のように記述できる。

待ち行列Qに要素aが入ると、待ち行列Rとなる。

同様にleave(a,Q,R)は、次のとうり。

待ち行列Qから要素aを取り除くと、待ち行列Rとなる。

待ち行列は生成されると、q(L,E)の形式の項として表される。ここでLは、エンドマーカEを持つ開リストである。引き続く操作により、一般にリストLは、次の図のように展開される。



したがってq(L,E)中のLは開リストを、EはLの接尾辞ということになり、待ち行列q(L,E)の内容は、EにないLの要素ということになる。

待ち行列の実現を図8.9に示す。この図は、次の質問の処理を示している。

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
     leave(X,S,T), leave(Y,T,U), wrapup(U).
```

最初のゴールsetup(Q)は、次のように空な待ち行列を生成する。

```
?- setup(Q).
     Q = q(_1,_1)
```

図8.9中で待ち行列q(L,E)は、LからEへの破線で示される。したがってQからの矢印は、エンドマーカ_1を持つ空の開リスト_1へと向かっている。

それでは第2のゴールenter(a,Q,R)を考えることにしよう。規則

```
enter(A, q(X,Y), q(X,Z)) :- Y = [A|Z].
```

は、次のように読める。

> 待ち行列 $q(X,Y)$ に A を挿入するためには、
> Y を新しいエンドマーカ Z を持つリスト $[A|Z]$ へと束縛し、
> その結果の待ち行列 $q(X,Z)$ をリターンする。

setup(Q)がQをq(_1,_1)へと初期化した後に、第2の規則は次の質問

```
?- setup(Q), enter(a,Q,R).
     Q = ...
     R = q([a|_2],_2)
```

で、_1と[a|_2]を単一化している。ここで_2は新しいエンドマーカである。

第3のゴールenter(b,R,S)は、_2と新しい変数_3を持つ[b|_3]を単一化することでbをq([a|_2,_2)へ挿入している。

要素が取り除かれると、その結果の待ち行列q(L,E)は、Lの代わりにLの尾部を持つ。leave(X,S,T)の右側の図で、待ち行列Tの開リストがSの開リストの尾部であることに注目されたい。同様にleave(Y,T,U)の右側の図で、Uの開リストはTの開リストの尾部である。

最後のゴールwrapup(U)は、挿入削除の操作によってUが初期状態q(L,E)になっていることを検査する。ここでLはエンドマーカEを持つ空な開リストである。さもなければq(L,E)は、次の事実中でq([],[])と単一化されない。

```
wrapup(q([],[])).
```

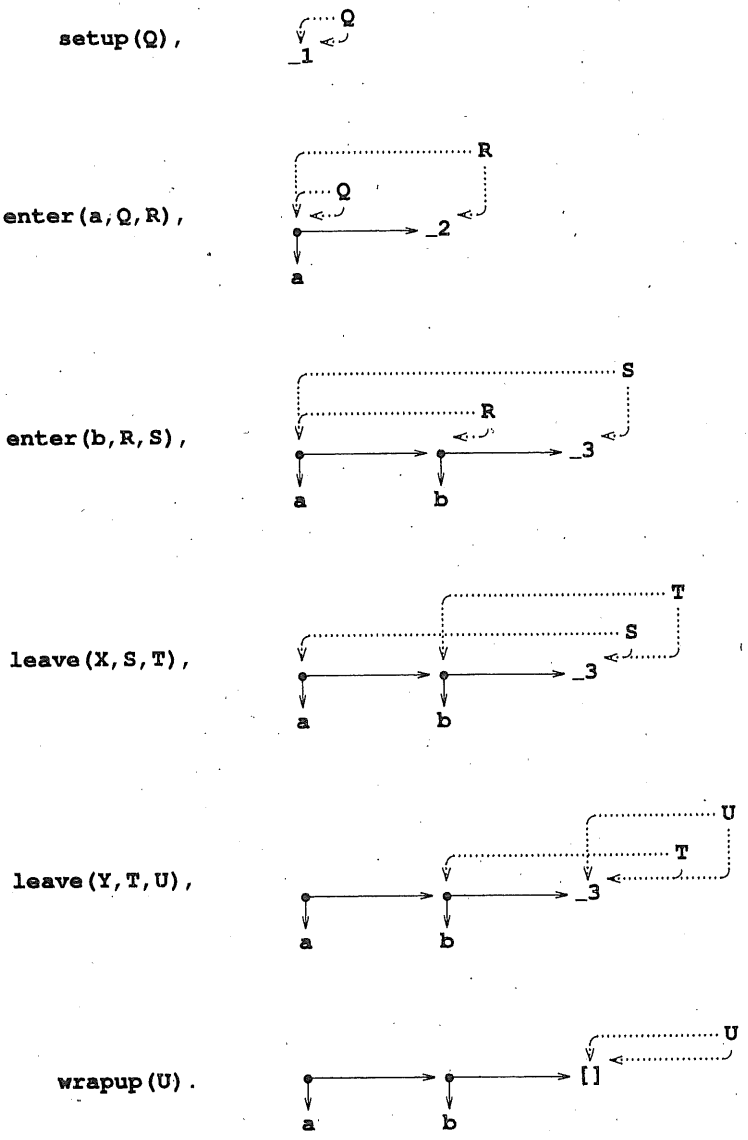


図8.9 待ち行列上の演算

図8.9中でUはq(_3,_3)であり、wrapup(q(_3,_3))は_3を[]と単一化している。□

驚くべきことに図8.8の待ち行列の規則は、「赤字」の待ち行列を許している。つまり要素を挿入する前に削除できる。厳密にいえば、変数で表される不明の要素を削除しておいて、後に挿入操作で埋め合わせることができるのである。次の質問では、後にゴールenter(a,R,S)によってXがaを表すとわかる前に、要素Xが初期待ち行列から削除されている。

```
?- setup(Q), leave(X,Q,R), enter(a,R,S), wrapup(S).
     Q = q([a],[a])
     X = a
     R = q([],[a])
     S = q([],[])
```

差分リスト

リストを用いるアプリケーションは、開リストを使うように変更できる。開リストを用いる場合には少し注意を要するが、効率は向上する。注意しなければならないとは、エンドマーカが単一化される際に開リストが変化するからである（図8.7参照）。差分リストとは、そのような変化を利用する技法である。

差分リスト（defference list）は、2つのリストLとEからなる。ここでEは、Lの尾部と単一化される。差分リストの内容は、Lにあって Eにない要素からなる。この差分リストをdl(L,E)と書くことにしよう。リストLとEは、どちらも開リストでもよいし閉じていてもよい。通常は開リストを用いる。

内容が[a,b]の差分リストの例は、次のとうり。

```
dl([a,b], []).
dl([a,b,c], [c]).
dl([a,b|E], E).
dl([a,b,c|F], [c|F]).
```

dl(L,E)中の変数によって、内容の終わりが直接参照できる。差分リスト上の連結操作は、非再帰的規則を用いることによって一定時間で実現できる。次の規則をインフォーマルにいうならば、XによってLがMへと展開され、YによってMがNへと展開されるならば、XとYを連結した結果のZによってLはNへと展開される。

```
append_dl(X, Y, Z) :-
        X = dl(L,M), Y = dl(M,N), Z = dl(L,N).
```

この節を終わるにあたって、8.5節で議論するPrologの制御に関連する例を示しておきたい。`dl(L,E)`の内容が`L`にあって`E`にない要素であることを、形式的に示す規則を考えよう。次のように定義したくなるであろう。

```
contents(X, dl(L,E)) :- append(X, E, L).
```

次の質問は、それぞれの差分リストが`[a,b]`であることを確かめている。

```
?- contents([a,b], dl([a,b,c], [c])).
     yes
?- contents([a,b], dl([a,b,c|F], [c|F])).
     F = _1
     yes
```

`dl([a,b|E],E)`の内容を直接尋ねようとすると、次のような応答を得る。

```
?- contents(X, dl([a,b|E], E)).
     X = []
     E = [a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b, ...
```

こうなる理由は、8.5節で説明する。

8.5 Prologにおける制御

再び、例のインフォーマルな式を見てみよう。

アルゴリズム = 論理 + 制御

ここで「論理」は論理プログラムの規則と質問を、そして「制御」は質問に答えるためにどのように言語が計算するかを指している。図8.10の疑似コードは、Prologの制御の概略を示している。

Prologの制御は、図8.10の次のような2個の決定によって特徴づけられる。

1. ゴールの順序　　最左ゴールを選択
2. 規則の順序　　最初の適用可能な規則を選択

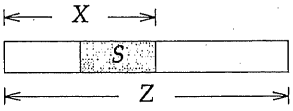
質問への応答は、質問中でのゴールの順序と事実の規則のデータベース中での規則の順序の両者によって影響される。

```
現在のゴールから質問を開始する;
while 現在ゴールが空でない do
    最左部分ゴールを選ぶ;
    if 部分ゴールに規則が適用可能 then
        最初の適用可能な規則を選ぶ;
        新しい現在ゴールを形成する
    else
        backtrack
    end if
end while;
succeed
```

図8.10 Prologの制御

□例題 8.6

この節の例題は、図8.11の規則を用いる。Zの部分リストS



は、次の一見等しい定義によって指定できる。

X は Z の接頭辞（prefix） **and** S は X の接尾辞（suffix）。
S は X の接尾辞（suffix） **and** X は Z の接頭辞（prefix）。

それぞれに対応するPrologの質問は、通常同じ応答を得る。しかしながら S が Z の部分リストでない時は、次のように異なる応答となる。

```
?- prefix(X,[a,b,c]), suffix([e],X).
     no

?- suffix([e],X), prefix(X,[a,b,c]).
     [無限の計算へ陥る]
```

この節では、接尾辞—接頭辞ゴールの順序について詳しく見ることにする。

規則を適用する順序によっても結果は変わる。次のように、要求に応じて新しい解が生成される。

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).

prefix(X,Z) :- append(X,Y,Z).

suffix(Y,Z) :- append(X,Y,Z).

appen2([H|X], Y, [H|Z]) :- appen2(X,Y,Z).
appen2([], Y, Y).
```

図8.11 8.5節の例題用の規則のデータベース

```
?- append(X, [c], Z).
     X = []
     Z = [c] ;

     X = [_1]
     Z = [_1,c] ;

     X = [_1,_2]
     Z = [_1,_2,c]

     yes
```

（yesは、まだ別解が存在することを示す。）

図8.11の関係**append2**は、**append**と同じ規則であるが逆順となっている。その応答は次のようになるが、その理由もこの節で説明する。

```
?- append2(X, [c], Z).
     [無限の計算へ陥る]
```

単一化と置換

Prologの制御において単一化は最も重要なことなので、342ページよりも詳しく厳密に定義することにしよう。

変数から項への関数を置換（substitution）という。置換を$X \rightarrow T$の形式の要素の集合として記述することにする。ここで変数Xは項Tへと写像される。特別にことわらない限り、ある置換がXをTに写像する時、項Tの中に変数Xは現れないものとする。$\{V \rightarrow [b, c], Y \rightarrow [a, b, c]\}$は、置換の1例である。

置換σを項Tに適用した結果を$T\sigma$と記述する。項に置換を適用した結果は、次の規則で得られる。

$$
\begin{array}{ll}
X\sigma = U & \sigma \text{中に} X \rightarrow U \text{がある時} \\
X\sigma = X & \sigma \text{中に} X \rightarrow U \text{がない時} \\
(f(T_2, T_2))\sigma = f(U_1, U_2) & T_1\sigma = U_1, T_2\sigma = U_2 \text{の時}
\end{array}
$$

この定義は、ファンクタ f が $k \geq 0$ 個の引数を持つ場合に一般化できる。どういうことかというと、σ が $X \rightarrow U$ を要素に持てば変数 X に σ を適用した結果は U であり、さもなければ $X\sigma$ はそのまま X となる。項 $f(T_1, ..., T_k)$ $(k \geq 0)$ に σ を適用した結果は、σ を個々の部分項に適用することによって得られる。たとえば次のとうり。

$$Y\{V \rightarrow [b,c],\ Y \rightarrow [a,b,c]\} = [a,b,c]$$
$$Z\{V \rightarrow [b,c],\ Y \rightarrow [a,b,c]\} = Z$$
$$(append([\,],Y,Y))\{V \rightarrow [b,c],\ Y \rightarrow [a,b,c]\} = append([\,], [a,b,c], [a,b,c])$$

ある置換 σ に対して $U = T\sigma$ であれば、項 U は T の具体値（instance）であるという。ある置換 σ に対して $T_1\sigma$ と T_2 が同一となれば、項 T_1 と T_2 は単一化されるという。この時 σ を T_1 と T_2 の単一子（unifier）と呼ぶ。他のすべての単一子 σ' に対して $T_1\sigma$ が $T_1\sigma'$ の具体値となる時、その置換 σ を T_1 と T_2 の最汎単一子(most general unifier)という[5]。

項 $append([\,], Y, Y)$ と $append([\,], [a \mid V], [a, b, c])$ は、単一化される。なぜならこれらの項は、共通の具体値 $append([\,], [a, b, c], [a, b, c])$ を持つからである。この場合の最汎単一子は、置換 $\{V \rightarrow [b, c]$、$Y \rightarrow [a, b, c]\}$ である。

ゴールへの規則の適用

図8.12の疑似コードは、Prologの制御を置換と単一化によって書き改めたものである。まずバックトラッキングなしで成功する例を考えることで調べていこう。バックトラッキングについては、後に計算を木で表すことによって述べる。

図8.12において規則 A :- $B_1, ..., B_n$ は、その頭部 A が部分ゴール G と単一化される時に G に適用（apply）される。規則中の変数は、部分ゴール中の変数と混同しないように、単一化の前に名前を書き換えられる。

□例題 8.7

次の質問と応答

```
?- suffix([a],L), prefix(L,[a,b,c]).
     L = [a]
```

は、図8.13に示すようにバックトラッキングなしで計算される。初期状態での現在ゴー

5 最汎単一子の定義において、$T_1\sigma$ が $T_1\sigma'$ の具体値であるというだけで十分である。σ と σ' はともに単一子なので、$T_2\sigma$ は自動的に $T_2\sigma'$ の具体値となる。なぜなら $T_1\sigma = T_2\sigma$ かつ $T_1\sigma' = T_2\sigma'$ だからである。

現在のゴールから質問を開始する;
while　現在ゴールが空でない　**do**
　現在ゴールを $G_1, ..., G_k$ ($k \geq 1$) とする;
　最左部分ゴール G_1 を選ぶ;
　if　G_1に規則が適用可能　**then**
　　最初の適用可能な規則 A :- $B_1, ..., B_j$ ($j \geq 0$)を選ぶ;
　　G_1 とAの最汎単一子を σ とする;
　　新しい現在ゴールは、$B_1\sigma, .., B_j\sigma, G_2\sigma, ..., G_k\sigma$ となる
　else
　　backtrack
　end if
end while;
succeed

図8.12 Prologの制御：図8.10の書き換え

ルは、次のように質問文中の2個の部分ゴールからなる。

```
suffix([a],L), prefix(L,[a,b,c])
```　(8.7)

最左部分ゴール`suffix([a],L)`を選択する。この部分ゴールに適用できる規則を選ぶ。規則中の変数名を書き換えて、ゴール中の変数と混同しないようにする。変数名を`X'`、`Y'`、`Z'`と書き換えると、`suffix`のための規則は、次のようになる。

```
suffix(Y',Z') :- append(X',Y',Z').
```

置換`{Y'→[a],Z'→L}`によって、規則の頭部と選択された部分ゴールが単一化される。規則の部分ゴール`suffix([a],L)`への適用は、次のように疑似コード化できる。

suffix ([*a*], *L*)　**if**　*append* (_1, [*a*], *L*)

ここで [*a*] は`Y'`に、*L* は`Z'`に、そしてPrologの名前 _1 が`X'`にそれぞれ取って代わる。(8.7)の部分ゴール`suffix([a],L)`を条件部`append(_1,[a],L)`で置き換えて、次の新しい現在ゴールを得る。

```
append(_1,[a],L), prefix(L,[a,b,c])
```　(8.8)

事実`append([],Y",Y")`が新しい最左部分ゴール`append(_1,[a],L)`に適用される。なぜなら`[]`が`_1`に、`Y"`が`[a]`と`L`にそれぞれ単一化されるからである。事実は、頭部のみで条件部を持たないので、新しい現在ゴールは次のとうり。

```
prefix([a],[a,b,c])
```　(8.9)

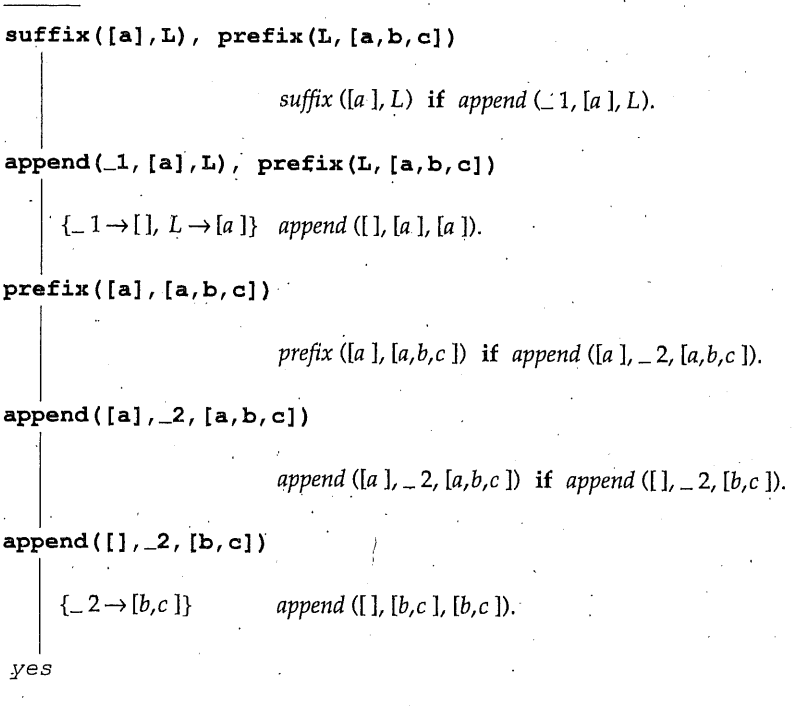


図8.13 バックトラッキングなしで成功する計算

`[a]`によって変数`L`が置換されていることに注目されたい。

その後の計算は、図8.13に示すとうり。 □

Prologの探索木

図8.13に示したゴールの連鎖は、Prologの探索木（Prolog search tree）へと一般化できる。Prologの探索木とは、ゴールのあらゆる可能な解を求める計算を模式化したものである。木の節点毎に、その節点にある最左部分ゴールに適用される規則の子を持つ。子の順序は、規則のデータベース内の順序と同じである[6]。

6 Prologの探索木の定義は、規則が最左部分ゴールに適用されることを仮定している。探索木のより一般的な定義では、規則を適用するのに任意の部分ゴールを選ぶことができる。更により一般的な定義では、規則も任意の順で選んでよい。

Prologの計算は、Prologの探索木を「深さ優先」で調べていく。探索は根から開始され、節点毎に左から右へと子の部分ゴールを調べる。計算は、部分木中で空ゴールを持つ節点に達する毎に *yes* と応答する。

□例題8.8

次の質問に対するPrologの探索木の部分を図8.14に示す。

```
?- suffix([b],L), prefix(L,[a,b,c]).
     L = [a,b]
```

suffixには1個の規則しかないので、根の子は1つだけである。例題8.7で採用した方法だと、根の子は次のゴールを持つ。

```
append(_1,[b],L), prefix(L,[a,b,c])
```　(8.10)

appendの2個の規則は、いずれも(8.10)の最左部分ゴールに適用可能である。したがって(8.10)の節点は、2つの子を持つ。Prologは、規則をデータベースに現れる順に試していく。したがって**append(_1,[b],L)**と**append([],Y',Y')**を単一化して、次の置換を得る。

```
{_1 → [], L → [b]}
```

appendの最初の規則は事実で条件部がないため、(8.10)から導かれる新しいゴールは次のようになる。

```
prefix([b],[a,b,c])
```　(8.11)

prefixには規則が1つしかなく、それを適用すると次の新しいゴールが得られる。

```
append([b],_2,[a,b,c])
```　(8.12)

さて、ここで問題が起きる。この部分ゴールは**append**のどちらの規則とも単一化しない。そこでPrologは、まだ試されていない規則を持つ最も近いゴールへとバックトラックする。そのようなゴールは(8.10)である。もう一度書いておく。

```
append(_1,[b],L), prefix(L,[a,b,c])
```　(8.10)

appendの最初の規則は成功しなかったので、Prologは2番目の規則を試す。2番目の規則の適当な具体値は次のとうり。

```
append([_3|_4], [b], [_3|_5]) :- append(_4,[b],_5).
```

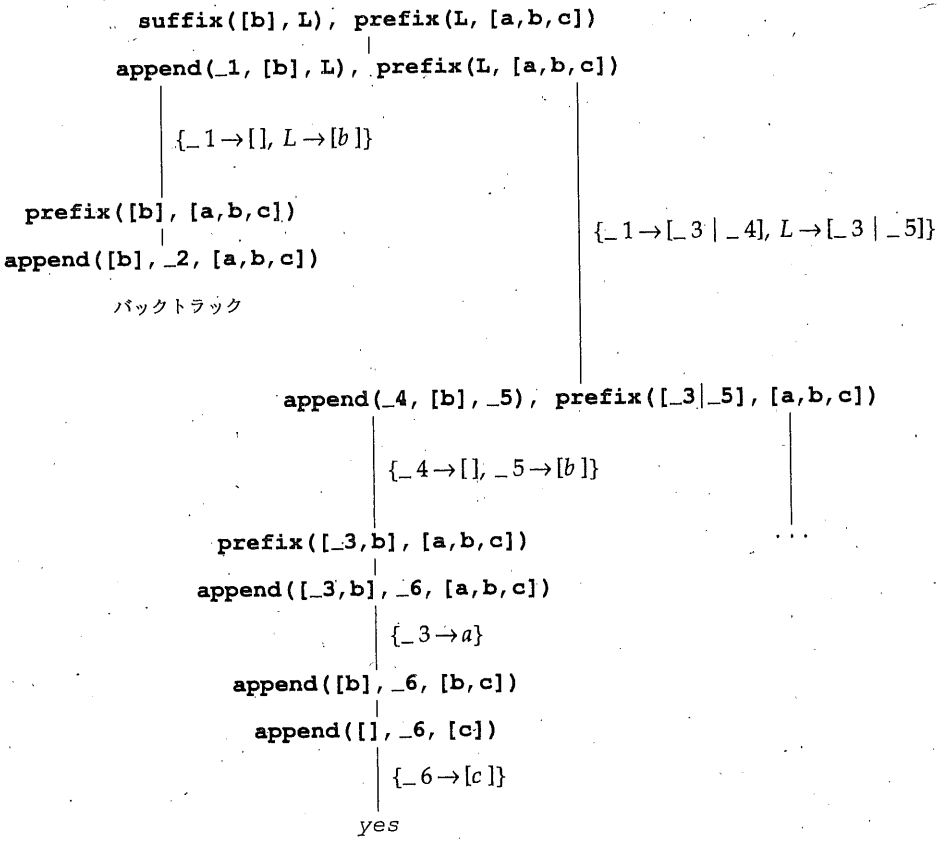


図8.14 *yes*と応答するまでのPrologの探索木の部分

(8.10)から次の置換によって、新しいゴールが形成される。

```
{_1 → [_3|_4], L → [_3|_5]}
```

新ゴールは次のとうり。

```
append(_4,[b],_5), prefix([_3|_5],[a,b,c])
```
(8.13)

今回計算は、図8.14のようにバックトラッキングなしで順調に進み、*yes*との応答を得る。 □

図8.15は、Prologの制御の最終版である。手続き*visit*は、新しい部分ゴールを試すために自分自身を再帰的に呼び出している。探索木に則っていうと、1つの再帰呼び出しは節点の子を1つ訪れるのに対応している。再帰呼び出しは、次の2つの理由のいずれかによって停止する。

1. ゴール G が空となり、**succeed**に達した。
2. G の再左部分ゴールに適用する規則がもはやなく、駆動が終了する。

バックトラッキングは、後者の場合に相当する。そこではいかなる規則も G に適用されず、制御は呼び出し側に戻る。

ゴール順の解への影響

質問中の部分ゴールの順序は、質問に応えるためのPrologの探索木に影響を与える。その理由は、常に最左部分ゴールに規則が適用されるからである。

例題8.7あるいは図8.13で、次の質問に対する解 `L=[a]`は、バックトラッキングなしで生成される。しかし別解を求めようとすると、無限の計算に陥ることに注目されたい。

```
procedure visit (G);
begin
    if 現在ゴール G が空でない then
        G を G1, ..., Gk (k≥1)とする;
        最左部分ゴール G1 を選ぶ;
        for i := 1 to 規則の数 do
            規則 i を A :- B1, ..., Bj (j≥0)とする;
            if 規則 i が G1 に適用できる then
                G1 と A の最汎単一子を σ とする;
                G' を B1σ, ..., Bjσ, G2σ, ..., Gkσ とする;
                visit (G')
            end if
        end for
    else
        succeed
    end if
    { 呼び出し側へ戻ることで、バックトラックする }
end visit
```

図8.15 再帰手続きによるPrologの制御

```
?- suffix([a],L), prefix(L,[a,b,c]).
     L = [a];
     [無限の計算へ陥る]
```

最左部分ゴール

```
?- suffix([a],L).
     L = [a];
     L = [_1,a];
     L = [_1,_2,a];
     ...
```

は無限の解を持つが、最初のものだけが**prefix(L,[a,b,c])**を充足するのである。

換言すれば、次のゴール

```
suffix([a],L), prefix(L,[a,b,c])
```

に対するPrologの探索木は、図8.13に示したようにバックトラッキングなしで到達できる*yes*節点を1つだけ持つ。それ以上の解を求めようとすると、残りの無限の木を不毛に探索することになってしまう。

それとは対照的に、次の質問

```
?- prefix(X,[a,b,c]), suffix([a],X).
     X = [a];
     no
```

は、有限のPrologの探索木を持つ。図8.16には、*yes*節点を導くまでの部分を示す。図8.13ではバックトラッキングなしで、そして図8.16ではバックトラッキングありで同一の解*L=[a]*に到達している。このようにゴールの順序を変更することは、Prologの探索木に変化をもたらす。

規則順の解の探索への影響

データベース中の規則の順序は、Prologの探索木の節点の子の順序を決定する。したがって規則順によっても、Prologの計算が解に到達する順序は変化する。

図8.17の探索木は、次の規則順に基づいている。

```
append([], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).

appen2([H|X], Y, [H|Z]) :- appen2(X,Y,Z).
appen2([], Y, Y).
```

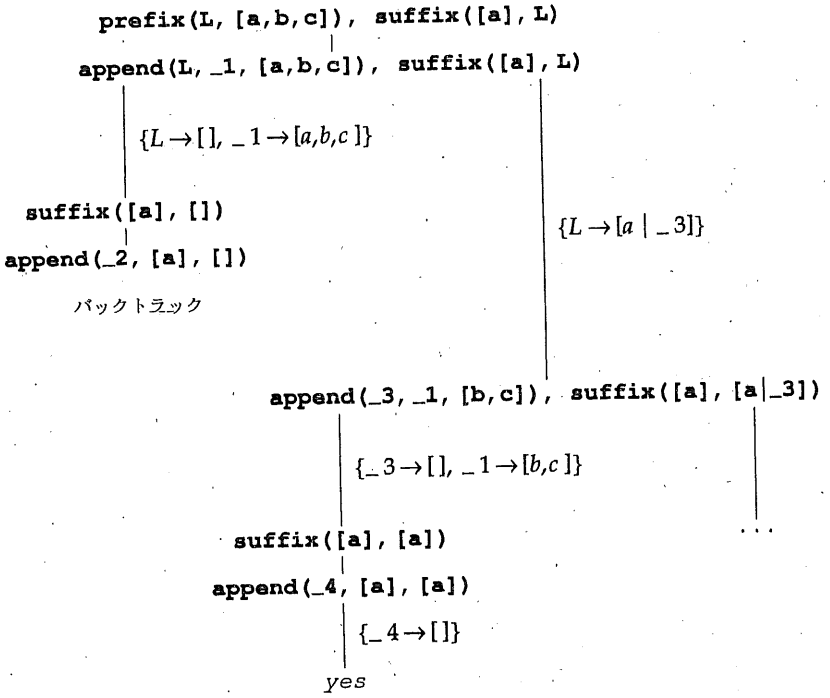


図8.16 ゴール順の変更の効果。図8.13と比較されたい

`appen2(X,[c],Z)`の探索木は、`append(X,[c],Z)`のそれの鏡像である。不幸にして`appen2(X,[c],Z)`のPrologの計算は、次のように無限の経路の深みにはまってしまうので、解に到達できない。

```
?- appen2(X, [c], Z).
     [無限の計算へ陥る]
```

一方`append(X,[c],Z)`の計算は、次のように次々と解を求める。

```
?- append(X, [c], Z).
     X = []
     Z = [c] ;

     X = [_1]
     Z = [_1,c] ;
     ...
```

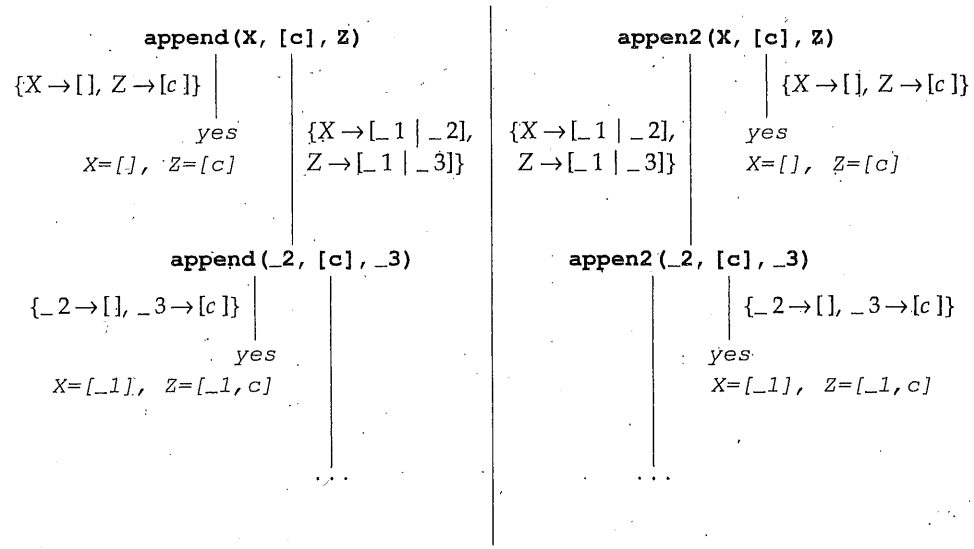


図8.17 規則の順序は、節点の子の順序を決定する

出現検査問題

効率上の理由で、X と T を単一化する前に変数 X が項 T 中に現れるか否かの検査を、Prologは行わない。そのような検査は、出現検査（occurs check）と呼ばれる。もし X が T の中に現れるならば、X と T の単一化によって無限の計算が引き起こされてしまう。たとえば次のとうり。

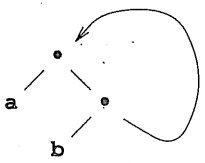
```
?- append([], E, [a,b|E]).
     E = [a,b,a,b,a,b,a,b,a,b,a,b,a,b,a,b,...
```

`append([],E,[a,b|E])`と`append([],Y,Y)`を単一化するためには、変数`Y`を`E`と`E`を含む項`[a,b|E]`の両者と単一化しなければならないのである。

Prologは`E`が`[a,b|E]`中に出現するか否かを検査しないので、`E`を`[a,b|E]`で置換しようとすると、次のようになってしまう。

```
E = [a,b|E] = [a,b,a,b|E] = [a,b,a,b,a,b|E] = ...
```

Prologの処理系によっては、ある変数とその変数を含む項を単一化する際に、次のような回帰項を構成するものもある。



8.6　カット

インフォーマルにいうと、カットは、Prologの探索木の未探索部分を刈り込む、あるいは切り落とす。したがってカットは、不毛な探索やバックトラッキングを削除して計算の効率を向上させるのに使用される。またカットは、ホーン節が行えない否定の実現にも使用される。

カットの使用には賛否両論がある。それというのもカットは、純粋論理からはずれるものだからである。8.5節で見たようにPrologの制御は、無限の計算に陥ることがあるけれども、評価の順序を変えることによって防ぐことができる。純粋論理は順序独立であるので、Prologは純粋論理の近似にすぎないということができる。カットは、Prologをますます純粋論理から遠いものにしてしまうのである。これによって、Prologのプログラムは手続き的に読まなければならないものとなる。つまり計算の順序を読まなくてはならなくなるのである。

!と記述されるカット（cut）は、規則の条件部に現れる。次の規則

$$B \text{ :- } C_1, ..., C_{j+1}, !, C_{j+1}, ..., C_k$$

が計算時に適用されると、カットによって制御は$C_{j-1}, ..., C_1, B$を通じてバックトラッキングされ、カットより右の残余の規則は考慮されない。カットがプログラミングでどのように応用されるかを見る前に、このことの影響をまず調べてみよう。

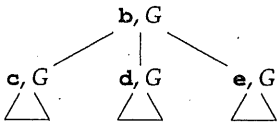
第1条件としてのカット

カットが最初の条件となるB :- !, C. の形式の規則を考えてみよう。ゴールCが失敗すると、制御はBを通過してバックトラックされ、Bについてのその他の規則は考慮されない。このようにカットは、Cが失敗すればBも失敗するという効果をもたらす。

Prologの探索木上でのカットの効果を見るために、`b`についての次の規則を考えよう。

```
b :- c.
b :- d.
b :- e.
```

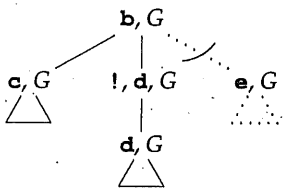
`b`について3つの規則があるので、`b`を最初の部分ゴールとして持つ節点はそれぞれ3個の子を持つ。ある節点の条件が`b`, Gであるとしよう。ここでGは別の部分ゴールを表す。この時この節点を根とする部分木は、次の形をとる。



ここで2番目の規則にカットが、次のように挿入されたとしよう。

```
b :- !, d.
```

このカットによって規則`b:-e`は削除され、考慮の対象外となる。新しいPrologの探索木は次のとうり。（破線部分は、比較のために残してあるだけで、新しい探索木の一部ではない。）



より細かく述べると、`!,d,`Gの中の最初の部分ゴールであるカットはすぐに充足され、新しいゴール`d,`Gを残す。しかしながらバックトラッキングに際してカットは、3番目の規則`b:-e`を考慮からはずすという副作用をもたらすのである。

□例題 8.9

次の規則のデータベースは、図8.18(a)に示すPrologの探索木のために作られている。

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
d.
e.
```

規則`a(X)`は、次のように2つの解を持つ。

```
?- a(X).
     X = 1 ;
     X = 2 ;
     no
```

もし規則`b:-c`を`b:-!,c`と変えたならば、Prologの探索木は、図8.18(b)のようになる。すると質問`a(X)`の解は、次のように1個となる。

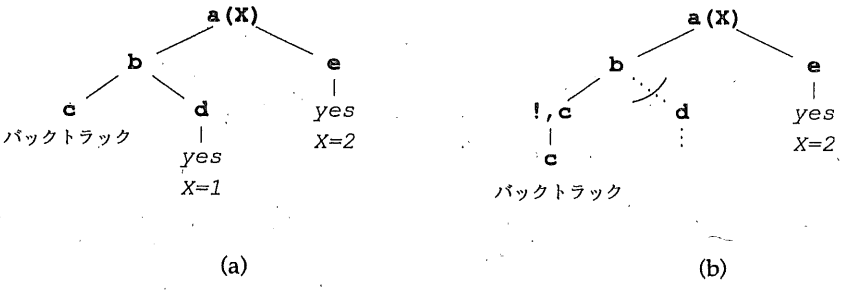


図8.18 カットの効果

```
?- a(X).
     X = 2 ;
     no
```

□

カットの効果

前述のように、計算中に次の規則

$$B :- C_1, ..., C_{j-1}, !, C_{j+1}, ..., C_k$$

が適用されると、カットによって制御は、$C_{j-1}, ..., C_1, B$ を通してバックトラックされ、残余の規則は考慮されない。

次の例題では、推測と確認規則の途中にカットが挿入された場合について考える。8.4節で議論したように、推測と確認規則の右辺は *guess* (*S*)、*verify* (*S*)の形式であり、*guess* (*S*)は *verify* (*S*)を充足するものが見つかるまで解の候補を生成する。したがってカットをその間に、次のように挿入することによって

conclusion (*S*) :- *guess* (*S*), ! , *verify* (*S*)

最初の推測以外は削除される。

□例題 8.10

図8.19の探索木は、図8.20の規則に基づいている。図8.19(a)に描かれた計算は、次の一連のゴールから始まっている。

`a(Z)`	開始ゴール
`b(Z)`	*a* (*X*) :- *b* (*X*). による
`g(Z),v(Z)`	*b* (*X*) :- *g* (*X*), *v* (*X*). による

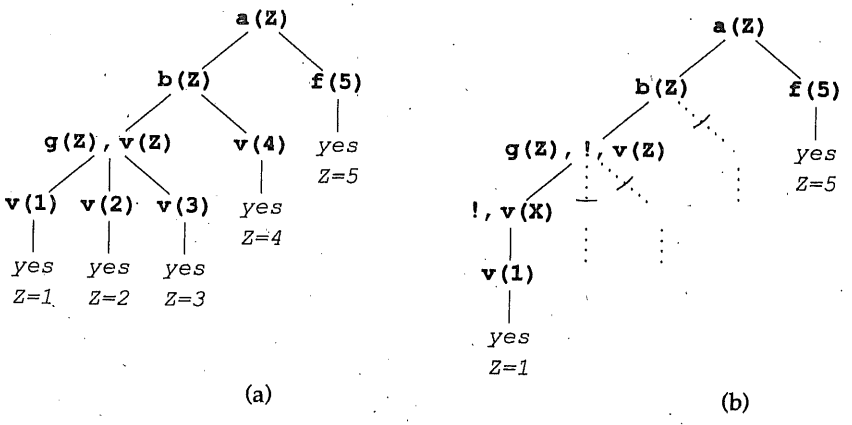


図8.19 カットによって削除される解

部分ゴール`g(Z)`は、`Z`のために値`1`、`2`、`3`を生成する。これらの値それぞれから、もとのゴール`a(Z)`の解が導かれることになる。

図8.19(a)中の4番目の解 *Z=4*は、バックトラッキングによって次のような`b`の別の規則が試されることによって得られる。

```
b(X) :- X=4, v(X).
```

最後の解 *Z=5* は、もとのゴール`a(Z)`までずっとバックトラッキングして、次のゴールを試すことによって得られる。

```
a(X) :- f(X).
```

ここで図8.20(b)のデータベースを考えてみよう。`b`の最初の規則中にカットが、次のように挿入されている。

```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), v(X).
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).
```

(a)

```
a(X) :- b(X).
a(X) :- f(X).
b(X) :- g(X), !, v(X).
b(X) :- X = 4, v(X).
g(1).
g(2).
g(3).
v(X).
f(5).
```

(b)

図8.20 第3の規則にカットを挿入

```
b(X) :- g(X), !, v(X).
```

カットの挿入によって、Prologの探索木は図8.19(a)から図8.19(b)のように変化する。このカットによって制御は、`g(X)`と`b(X)`を通じてバックトラッキングされ、図8.19(b)に示すように`g`と`b`の残りの規則は考慮されなくなる。ゴール`a(Z)`は、2つの解のみを持つようになった。 □

Prologのカットは誤解をされることが多いが、それなりの理由があろう。図8.19(b)の探索木から質問`a(Z)`は、次のように2つの解を持つ。

```
?- a(Z).
     Z = 1 ;
     Z = 5 ;
     no
```

この応答から、`a(2)`、`a(3)`、`a(4)`は充足不能であると思いがちであるが、実は図8.21の探索木は、`a(2)`、`a(3)`、`a(4)`への応答として*yes*に到達する。

質問`a(2)`と`a(3)`は、図8.21(a-b)に示すようにバックトラッキングなしで応答*yes*を導く。バックトラッキングがないので、カットは*yes*に到達するのを妨げない。

最後に`a(4)`を考える。図8.21(c)に示す計算はカットに到達しない。なぜなら`g(4)`が充足不能だからである。したがってカットはどのような効果も生じず、計算は*yes*に到達する。

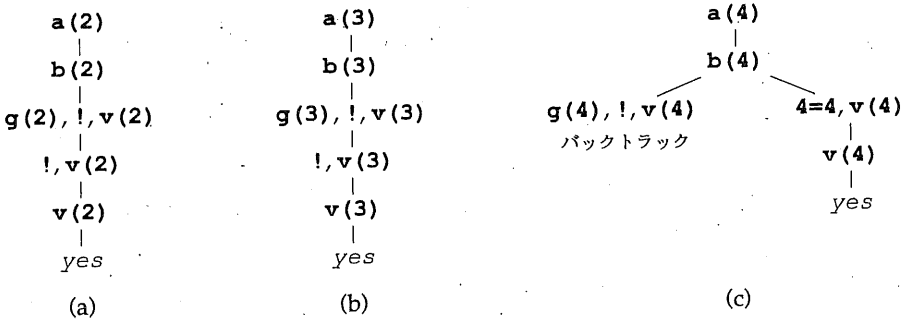


図8.21 図8.20(b)の規則に基づいたPrologの探索木

カットのプログラミングでの応用

カットの比較的良性な使用は、Prologの探索木のうち解に到達しない部分を刈り込むことである。そのようなカットはグリーン・カット（green cut）と呼ばれるが、それは解を変えることなく効率を向上させるからである。グリーンでないカットはレッド（red）と呼ばれる。

バックトラッキングを制限することで、カットはプログラムのメモリ要求を減少させる。カットがなければ、計算全体が成功するかバックトラッキングが起きるまで、あらゆる規則の適用と単一化を記録しておかなければならない。

□例題 8.11

グリーン・カットの例として、347ページの図8.6の二分探索木の規則を考えてみよう。規則を再び書き出しておく。

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

これらの3つの規則は、排他的である。なぜなら`K=N`、`K<N`、`K>N`のうち1つしか同時に真となり得ないからである。したがって次のように2番目の規則にカットを挿入するのは、グリーン・カットである。

```
member(K, node(K,_,_)).
member(K, node(N,S,_)) :- K < N, !, member(K, S).
member(K, node(N,_,T)) :- K > N, member(K, T).
```

このカットには`K<N`である時だけ達するので、カットに到達したならば3番目の規則が適用されることはない。このカットによって`K<N`であるが`K`が木の中にないので`member(K,S)`は失敗する場合に、プログラムは効率的となる。カットがなければPrologはバックトラックし、3番目の規則を試して、右の部分木中に`K`を探そうとする。カットはこの不毛な努力をなくす働きをする。　□

次の例題中の`lookup`関係と`install`関係は、Prologで書かれたコンパイラで使用されているものである。コンパイラは二分探索木を用いているが、簡単にするためにこの例では、キーと値の対のリスト中でキーを探すのに線形探索を用いる[7]。

7 8.2節の単純化した項の構文では、`(K,V)`の形の項を許していないが、Prologでは記述可能である。項`(K,V)`は、第1要素が`K`、第2要素が`V`であるような対である。

□例題8.12

lookupの規則と**install**のそれとの唯一の相違は、**lookup**では、次のように最初の規則にカットが挿入されている点である。

```
lookup(K,V, [(K,W)|_]) :- !, V = W.
lookup(K,V, [_|Z]) :- lookup(K,V,Z).
install(K,V, [(K,W)|_]) :- V = W.
install(K,V, [_|Z]) :- install(K,V,Z).
```

関係**lookup**は、キーと値の対の表に情報を挿入するのに使用できる。この表は、変数で終わる開リストとして保持される。

```
?- lookup(p,72,D).
     D = [(p,72)|_1] ;
     no
```

（別解を求めるのに、セミコロンをタイプしている。応答*no*は、これ以外に解が存在しないことを示している。）　2個の異なる値を、同じキーに挿入しようとすると次のように失敗する。

```
?- lookup(p,72,D), lookup(p,73,D).
     no
```

次の質問は、対を挿入するのと値を求めるのの両方に関係**lookup**を用いている。

```
?- lookup(1,58,D), lookup(p,72,D), lookup(p,Y,D).
     D = [(1,58),(p,72)|_1]
     Y = 72 ;
     no
```

カットを持たない関係**install**は、異なった働きをする。次の質問は、無限個の解を持つ。

```
?- install(p,72,D).
     D = [(p,72)|_1] ;
     D = [_2,(p,72)|_3] ;
     D = [_2,_4,(p,72)|_5]
     yes
```

更に関係**install**は、次のように同一のキーに異なる値を挿入することも許す。

```
?- install(p,72,D), install(p,73,D).
     D = [(p,72),(p,73)|_1] ;
```

カットの役割は、lookupの規則を次のように書き換えることにより明かとなろう。

```
lookup(K,V,L) :- L = [(K,W)|_], !, V = W.
lookup(K,V,L) :- L = [_|Z], lookup(K,V,Z).
```

ここでDが新しい変数ならば、次の一連のゴールを得る。

```
lookup(p,72,D).                 開始ゴール
D=[(p,_1)|_2], !, 72=_1         第1規則を適用
!, 72=_1                        Dを単一化
72=_1
yes                             72が_1と単一化される
```

カットによって、lookupの2番目の規則の考慮が除外されるので、これ以上の解を求めなくなる。

次の質問を考えてみよう。

```
?- lookup(p,72,D), lookup(p,73,D).
    no
```

今見たように部分ゴールlookup(p,72,D)は、Dをリスト[(p,72)|_2]へと束縛する。したがって、次の一連のゴールが得られる。

```
lookup(p,73,[(p,72)|_2])
[(p,72)|_2]=[(p,_3)|_4], !, 72=_3      第1規則を適用
!, 73=72                               リストを単一化
73=72
```

カットによってlookupの2番目の規則の考慮が妨げられるので、ここで失敗が確定する。□

失敗による否定

Prologのnot演算子は、次の規則によって実現されている。

```
not(X) :- X, !, fail.
not(_).
```

インフォーマルには、最初の規則はnotの引数Xを充足しようとする。ゴールXが成功するとカットに、そしてfailに到達する。構成子は失敗を強制し、カットが2番目の規則の考慮を妨げる。他方もしゴールXが失敗すれば、2番目の規則が成功する。なぜなら_は、どのような項とも単一化されるからである。

notのこの規則により、次の応答

```
?- X = 2, not(X = 1).
     X = 2
```

と、次の応答

```
?- not(X = 1), X = 2.
     no
```

との違いが説明できよう。

最初の質問では、部分ゴールX=2により X と 2 が単一化され（図8.22(a)参照）、not(2=1)を現在ゴールとして残す。notの最初の規則により、次の新しいゴールが形成される。

```
2=1, !, fail
```

2=1は失敗するので、カットには到達しない。そこで図8.22(a)にはカットが現れていない。次にnotの２番目の規則が試され、ゴールnot(2=1)は成功する。

２番目の質問の探索木を図8.22(b)に示す。notの第１規則は、次の現在ゴールを形成する。

```
X=1 !, fail, X=2
```

部分ゴールX=1が成功し、カットが充足されてfailに到達する。カットにより、これ以上notの規則の考慮が排除されるので、全体の計算が失敗する。部分ゴールX=2には決して到達しないことに注目されたい。

一般に、notは変数を持たない項に適用するのが安全である。なぜならそのような項は、単一化によって変化する変数を持たないからである。

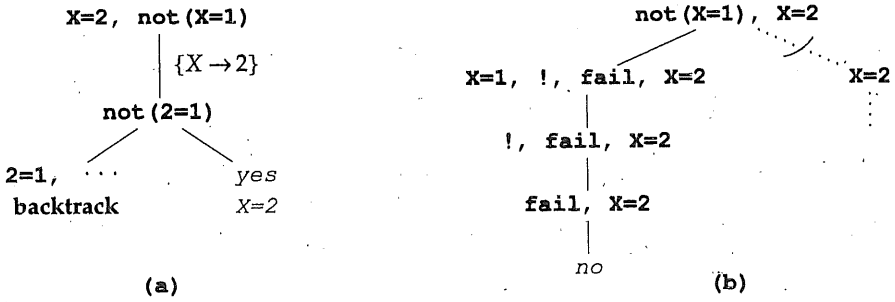


図8.22 失敗による否定を示すPrologの探索木

演習問題

8.1　次の関係が与えられている。

`father(X,Y)`	`X`は`Y`の父親である
`mother(X,Y)`	`X`は`Y`の母親である
`female(X)`	`X`は女性である。
`male(X)`	`X`は男性である

次の関係を定義せよ。

a) sibling（兄弟姉妹）

b) sister（姉妹）

c) grandson（男孫）

d) first cousin（従兄弟）

e) descendant（子孫）

8.2　**append**関係のみを用いて、次を求める質問を構成せよ。

a) リストの3番目の要素。

b) リストの最後の要素。

c) リストの最後の要素を除いたすべて。

d) 与えられたリストが、同一の部分リストを3つ繋ぎ合わせたものであるかどうかの判定。

e) リスト`Y`が、要素`A`をリスト`X`の任意の場所に挿入したものであるかどうかの判定。

8.3　リストが次の性質を持つか否かを判定する関係を定義せよ。

a) 一方が他方の順列となっている。

b) 偶数個の要素を持つ。

c) 2個のリストの併合となっている。

d) 回文となっている。つまり左から読んでも右から読んでも同じリスト。

8.4　次のリスト操作に対応する関係を定義せよ。

a) 隣接する重複要素の2番目以降を削除する。

b) 隣接する重複要素を持たない要素のみを残す。

c) 隣接する重複要素を1つだけ残す。

8.5　二分探索木上に関係**insert**と**delete**を定義して、347ページの例題8.3を完成せよ。

8.6　次の形式の部分ゴールを用いることによって、Prologは算術式を実行できる。

<varible> **is** *<expression>*

式の評価の結果が次のように変数と単一化される時、この部分ゴールは成功する。

```
?- X is 2, Y is X+1.
     X = 2
     Y = 3
```

isの後の式が束縛されていない変数を含めば、次のようにエラーが発生する。

```
?- Y is X+1, X is 2.
     Error
```

a) 階乗関数に対応する関係を定義せよ。

b) 末尾再帰を使用した階乗関数に対応する関係を定義せよ。

8.7　次の質問に対するPrologの探索木を描け。

```
?- reverse([a,b,c,d],W).
```

ここで**reverse**は、次の規則によって定義される。

a)
```
reverse([],[]).
reverse([A|X],Z) :- reverse(X,Y), append(Y,[A],Z).
```

b)
```
reverse(X,Z) :- rev(X,[],Z).
rev([],Y,Y).
rev([A|X],Y,Z) :- rev(X,[A|Y],Z).
```

8.8　次の規則によって定義される関係**member**を考える。

```
member(M, [M|_]).
member(M, [_|T]) :- member(M, T).
```

次のそれぞれの質問への応答に対応するPrologの探索木の部分を描け。

a)
```
?- member(b, [a,b,c]).
     yes
```

b) `?- member(d, [a,b,c]).`

```
no
```

c) `?- member(b, X).`

```
X = [b|_] ;
X = [_,b|_] ;
X = [_,_,b|_]
yes
```

8.9　演習問題8.8の関係**member**を用いて、次の質問のPrologの探索木を描け。

a) `X = [1,2,3], member(a,X).`

b) `member(a,X), X = [1,2,3].`

8.10 最初の規則中のカットを除いて、次の規則は8.5節のものと等しい。

```
append([], Y, Y) :- !.
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
prefix(X,Z) :- append(X,Y,Z).
suffix(Y,Z) :- append(X,Y,Z).
```

次の質問に対するPrologの探索木を、図8.13と図8.14のものと比較せよ。

a) `suffix([a],L), prefix(L,[a,b,c]).`

b) `suffix([b],L), prefix(L,[a,b,c]).`

参考文献

Programmation en logiqueから採られたPrologとは、Alain ColmerauerとPhillipe Rousselによって1972年に開発されたプログラミング言語の名前である。Kowalski [1988]とCohen[1988]による互いに補いあう論文によって、初期の歴史を知ることができる。この言語の開発は、Algol68の記述言語 W–文法（van Wijngaarden他[1975]）と機械的定理証明のためのRobinson[1965]の導出原理の影響を受けている。

Kowalskiは、次のように述べている。「われわれの初期の発見を振り返って見て一番重要であったのは、計算が演繹によって包含され得ることであった。」彼の初期の例題には、「加算と階乗のような再帰的述語のための計算的に効率的な公理」が含まれている。彼は続けて、「[Colmerauer]にとってリストの連結のホーン節の定義は、論理プログラミングの重要性にとってより特徴的であった。」と回顧している。

Cohenは、Prologの開発がLispに較べてなぜ遅れたかを、次のように理由づけてい

る。(1) 言語の表現力を示す興味深い例題の欠如。(2) 適当な実現の欠如。(3) Lispが利用可能であったこと。しかし次のようにも述べている。「Warrenによってその後開発されたインタプリタとコンパイラが、Prologの受け入れに重要な役割を演じたことを述べておくべきであろう。」Warren[1980]は、Prolog自身がコンパイラの記述にどのように利用できるかを述べている。

Prologのプログラミング技法についてのより詳しい情報は、SterlingとShapiro[1986]やClocksinとMellish[1987]のような教科書に見ることができる。例題8.5と演習問題8.7は、SterlingとShapiro[1986]中の例題から採られている。Cohen[1988]には、Rousselの最初のPrologインタプリタへの貴重な貢献となった差分リストが引用されている。ClarkとTarnlund[1977]は、差分リストについての参照文献となっている。

カットは、メモリ空間を節約するためにColmerauerによって導入された。失敗による否定の取扱いについては、Clark[1978]を参照されたい。

論理プログラミング一般についての議論は、Kowalski[1979a]ないしはHogger[1984]を参照されたい。