第 7 章

関数型プログラミング

7.1 宣言的表現

★ *副作用のないプログラムは，わかりやすい。*

プログラミングパラダイムにおける宣言的表現を，前章では副作用のないことと記述順序が本質的でないことによって意味付けた。その 1 つの方法として，論理型プログラミングでは，論理の表現に着目した。本章では，「関数」という概念に着目した関数型プログラミングについて述べる。ここでは，関数とは，その定義域の中のある値が入力として与えられたときに，その値域の中の対応する値を出力とするもの，という一般的な数学関数として扱う。

関数という概念は，従来の手続き型言語である FORTRAN, C, Pascal などにも含まれるが，変数による副作用を伴うという意味で数学関数とは異なっている。変数の副作用がある場合は，関数への入力値が同じでも実行順序により出力値が異なるプログラムを作成できるため，プログラムの意味がわかりにくくなる。

本章では，まず始めに関数型プログラミングの概念に基づいた実用的なプログラミング言語である Lisp について述べ，関数型プログラミングの重要な概念については後節で述べる。

7.2 Lisp

★ *関数呼び出しの集まりとして表現されたプログラムは，わかりやすい。*

7.2.1 概 要

記号処理，数式処理，人工知能などの分野で幅広く利用されているリスト処理言語 Lisp (LISt Processor) は，1958 年に米国マサチューセッツ工科大学の J. McCarthy らが開発したものである。その後，言語仕様の少し異なる方言が次々と開発されてきた。代表的なものとして，当初 PDP-10 を対象マシンとして開発後，普及した MacLisp と Interlisp や Lisp マシン向きの ZetaLisp などがある。

このような数多くの方言の存在は，各々の方言を限られた仲間うちだけで利用している間はあまり不都合はなかったが，数式処理や人工知能の分野の拡大と共に，Lisp プログラムの普及に大きな障害となってきた。そこで，1980 年代前半には，言語仕様の標準化の努力がはらわれ，1984 年に MacLisp, Interlisp, ZetaLisp との互換性を考慮した Common Lisp が G. L. Steele らによって開発された。

Lisp が当初から持っている基本的な特徴は次のようなものである。

①リストというデータ構造とそのリストに対する 3 種類の操作（car，cdr，cons）が基本である。

②プログラムは，S 式と呼ばれる形式で表現されたリストの集合で記述される。

③プログラムの意味は，関数として解釈され，ラムダ計算法に基づいて実行される。

さらに，本書で説明をする Common Lisp は，以下の特徴を有し，プログラミング機能の向上が図られている。

①プログラムの移植性

②豊富な言語機能（表現能力）

③処理方式（コンパイラとインタプリタ）に依存しない意味仕様

本節では，プログラム例を中心にこれらのLispの特徴について述べる。

7.2.2 リスト

Lispの基本となるデータ構造はリスト形式である。その例を図7.1に示す。これらの例は，リストの要素がそれぞれ数，記号，リストになっている。Lispのデータの最小単位である記号と数をまとめてアトム（atom）という。リストの一般形は，

(S式　S式　…　S式)

の形をしており，S式とはアトムまたはリストである。評価の対象となるS式は，Common Lispではフォーム（form）と呼ばれ，次の3種類がある。

●数や文字列のように自分自身が値となるもの

●変数名を表す記号でなんらかの値と結びつくもの

●リスト

このようなリストは，コンピュータの内部では図7.2のように2進木表現されることが多い。

リスト・フォームには，意味的には次の3種類がある。

●関数呼出し

●特殊形式

●マクロ呼出し

これらの詳細は後の例の中で説明する。

```
(L1)  点数のリスト
    (70  90  60  80)
(L2)  名前のリスト
    (abe  baba  chiba  dan)
(L3)  名前と点数のリストのリスト
    ((abe  70)  (baba  90)  (chiba  60)  (dan  80))
```

図7.1　Lispプログラムの例（リスト形式）

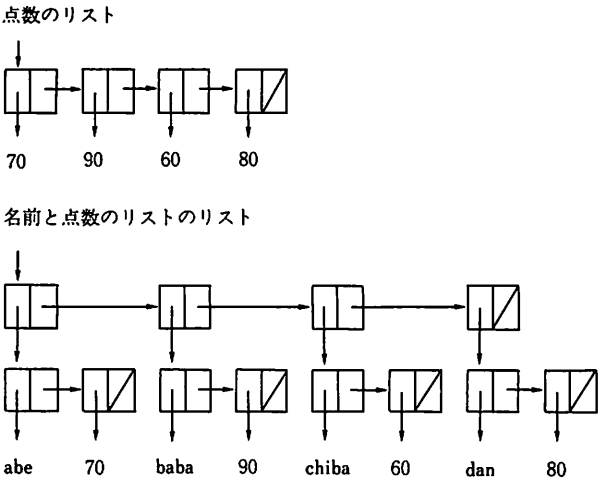

図7.2 Lispのリストのデータ構造（2進木表現）

7.2.3 関数と演算

(1) 関数定義と関数呼出し

Lispのプログラムは，関数を定義し，それを呼び出す形で記述されるので，関数型言語 (functional language) の1つと考えられる。この関数の定義の方法には2種類あり，その1つはラムダ式(lambda expression)を用いるものである。通常，われわれは関数をf(x)と表現することが多いが，これではxを変数とする関数自身を表しているのか，あるいはxにおけるfの値を表しているのかが曖昧である。そこで，Lispの基本となるラムダ計算法(lambda calculus)では，関数自身を表す場合は，λxf(x)のようにλの直後に変数を並べたラムダ式を用いる。Common Lispではこのラムダ式を

(lambda 仮引数リスト 本体)

の形で記述する。もう1つの関数定義の方法は，関数に名前を付けて，

(defun 関数名 仮引数リスト 本体)

の形で記述するものである。

この方法による関数定義の例を図7.3に示す。最初のaverageという関数

```
(L4)  点数リストから平均点を求める
    (defun  average  (marklist)
       (round  (sum  marklist)  (num  marklist)))
(L5)  点数リストの合計を求める
    (defun  sum  (marklist)
       (if  (null  marklist)  0
                    (+ (car  marklist)  (sum  (cdr  marklist)))))
(L6)  点数リストの人数を求める
    (defun  num  (marklist)
       (if  (null  marklist)  0
                        (1+ (num  (cdr  marklist)))))
```

図7.3　Lispプログラムの例（関数定義と関数呼び出し）

は，入力となる仮引数として，点数リスト marklist をもち，それから合計点 sum と人数 num を求め，sum を num で割った値を返すことを示している。関数本体の(round…)，(sum…)，(num…)は，それぞれ round，sum，num という関数の呼び出しである。関数呼出しの一般形は，

(関数名　実引数　…　実引数)

である。

(2)　特殊形式

上記（L5）の sum という関数の本体は，

(if　条件　式1　式2)

の形をしている。これは，手続き型言語では，

if　条件　then　式1　else　式2

に相当するもので，条件が真ならば式1，条件が偽ならば式2の値を返す。このように Common Lisp の言語機能の基本となるものを特殊形式（special form）という。if の他に function，declare，progn，setq など，24種類用意されている。

(3) 再帰関数

上記の例では，条件の(null marklist)は，入力の点数リスト marklist が空リストか否かの判定をするために null という関数を呼び出している。空リストのときは式1すなわち0が関数 sum の値となる。空リストでなければ式2の値となる。式2は，

```
(+  式3  式4)
```

という関数呼び出しの形をしており，式3と式4の値を加算する。式3の car はリストの第1要素を取り出す関数，式4の cdr はリストの第1要素を取り除いた残りのリストを返す関数である。たとえば，仮引数 marklist に対応する入力が上記 (L1) の (70 90 60 80) だったとすると，(car marklist) の値は 70，(cdr marklist) の値は (90 60 80) となる。

ここで式4に注目すると，sum という関数定義の本体の中で sum 自身が呼び出されている。このような関数を再帰関数 (recursive function) という。たとえば，この関数を用いて，

```
(sum '(70 90 60 80))
```

を実行すると，内部で sum が4回呼び出され，その度に式2は次のような式になる。

```
(+ 70 (sum '(90 60 80)))
(+ 90 (sum '(60 80)))
(+ 60 (sum '(80)))
(+ 80 (sum '( )))
```

sum の4回目の呼び出しでは入力が空リストになるので，式1の0が返される。結局，点数リストの各点数は，右から順に

```
(70+(90+(60+(80+0))))
```

という計算が行われ，300 が求まる。なお，sum の実引数の「'」はそれに続くオブジェクトを評価しないで渡すことを意味し，特殊形式 quote の省略形である。すなわち，'f は(quote f)のことである。

上記（L6）の点数リストの人数を求める関数 num も sum とほぼ同じ構造である。(1+ 実引数) は，(+ 1 実引数) の簡略形式であり，

```
(num '(70 90 60 80))
```

を実行すると，

```
(1+(1+(1+(1+0))))
```

という計算が行われ，4 が求まる。

(4) 仮引数リスト

関数定義における仮引数リストはラムダリスト (lambda-list) と呼ばれ，構文は次の形式をしている。

```
( {変数}*
  [&optional {変数 | (変数 [初期値 [判定変数]])}*]
  [&rest 変数]
  [&key {変数 | ({変数 | (キーワード 変数)} [初期値 [判定変数]])}*]
  [&aug {変数 | (変数 [初期値])}*])
```

ここで，構文の表記法として，{ }*は「0 個以上の繰返し」，[ ] は「あってもなくてもよい」，| は「または」を表す。このような付加機能を用いた例を 図 7.4 に示す。

関数 sumnum では，先の関数 sum と関数 num で別々に求めていた合計点と人数をまとめて求めている。ただし，計算方法を少し変えて，左から順に加算

```
(L7)  点数リストの合計と人数を一つの関数で求める
    (defun  sumnum  (marklist  &optional  (s 0)  (n 0))
      (if  (null marklist)  (list s n)
           (sumnum  (cdr marklist)  (+ s (car marklist))  (1+  n))))
(L8)  点数リストの平均点を求める
    (defun  average  (marklist  &aux  (work  (sumnum  marklist)))
      (round (car work)  (cadr work)))
```

図 7.4　Lisp プログラムの例（仮引数リストの付加機能）

する方式にしている。そのために，途中の計算結果を覚えておく変数 s と n を導入している。仮引数リストの&optional は，それ以降の第 2 仮引数の s と第 3 仮引数の n に対応する実引数があってもなくてもよいことを示している。実引数がない場合は初期値指定の 0 がデフォルト値として使用される。

関数 sumnum の本体の list は，実引数をリスト要素とするリストを返す関数である。(list s n)はリスト(s n)を返す。たとえば，この関数を用いて，

```
(sumnum '(70 90 60 80))
```

を実行すると，内部で sumnum が再帰的に 4 回呼び出され，その度に次のようになる。

```
(sumnum '(90 60 80)(+ 0 70)(1+ 0))
(sumnum '(60 80)(+ 70 90)(1+ 1))
(sumnum '(80)(+ 160 60)(1+ 2))
(sumnum '( )(+ 220 80)(1+ 3))
```

sum の 4 回目の呼び出しでは 1 番目の実引数が空リストになるので，2 番目の実引数 300 と 3 番目の実引数 4 を要素とするリスト (300 4) が返される。結局，点数リストの各点数は，左から順に

```
((((0+70)+90)+60)+80)
```

という計算が行われ，300 が求まる。人数も同様に，

```
((((0+1)+1)+1)+1)
```

という計算が行われ，4 が求まる。

次に (L8) の関数 average は，sumnum を用いて (L4) を再定義したものである。仮引数リストの&aug は，補助変数 (auxiliary variable) として work を宣言し，その初期値は関数呼出し (sumnum marklist) の値となることを示している。したがって，

```
(average '(70 90 60 80))
```

が実行されると，まず (sumnum '(70 90 60 80)) が評価され，その値 (300 4)

が work に代入される。次に関数本体が評価され，(round 300 4)の値 75 が返される。関数 cadr は合成関数で，(cadr work) は (car (cdr work)) と同じ処理を意味し，work の第 2 要素の 4 を返す。

その他，ラムダリストの構文の中の&rest は，仮引数リストの仮引数と対応しない残りの実引数は，リストにして&rest の次の仮引数に対応付けられる。&key は，それ以降の仮引数はキーワードを介して実引数と対応付けられることを示す。たとえば，

```
(defun foo (a &key b c) (list a b c))
```

では，(foo 1 :b 2 :c 3)も(foo 1 :c 3 :b 2)も同じ値(1 2 3)を返す。

(5) 手続き的記述

Lisp は，基本構造は関数型言語の形態であり，適用的 (applicative) であるが，実際には，逐次実行を前提にした文指向の手続き型言語的な制御構造や副作用を伴う変数の存在をも許している。このような機能を用いた sumnum と average のプログラム例を **図 7.5** に示す。

ここで，prog は手続き的記述に用いられるマクロであり，上記の例は次のようなマクロ呼出しの形をしている。

```
(L9)  手続き的記述の例
    (defun sumnum (marklist)
      (prog (rest sum num)
            (cond ( (null marklist)  (return (list 0 0))))
            (setq rest (sumnum (cdr marklist)))
            (setq sum (+ (car marklist)  (car rest)))
            (setq num (+ 1 (cadr rest)))
            (return (list sum num))))
    (defun  average (marklist)
      (prog (work)
            (setq work (sumnum marklist))
            (return (round (car work)  (cadr work)))))
```

図 7.5　Lisp プログラムの手続き的記述の例

```
(prog (局所変数の並び)
      文の並び)
```

各々の文はフォームの形をしており，前から順に評価されるが，結果の値は無視される。

関数 sumnum では，局所変数として rest，sum，num を導入し，最初の文で入力が空リストの時の例外処理を記述している。cond は if に似ているが，マクロであり，次のようなマクロ呼出しの形で用いられる。

```
(cond (条件1 式1)
      (条件2 式2)
        …
      (条件n 式n))
```

if と異なり，条件を任意の個数だけ指定でき，前から順に条件を評価して最初に真となった条件に対応する式が実行される。この例では，入力が空リストの時は (0 0) という値を返して prog のブロックから抜け出すために，式1のところで return というマクロ呼出しが使われている。入力が空リストでない場合は2番目以降の文が順に評価される。setq は代入のための特殊形式であり，2番目の文では，点数リストの1番目の要素を除いた残りの点数リストに関数 sumnum を適用した結果が局所変数 rest に代入される。3番目の文では，点数リストの最初の要素の点数と rest の第1要素，すなわち残りの点数の合計が加算され，局所変数 sum に代入される。4番目の文では，同様にして合計の人数を局所変数 num に代入している。最後の文では，(sum num) を値として返してブロックから脱出している。

関数 average も同様である。このような手続き的記述機能としては，この他にも種々の条件付き実行や繰返し実行のための構文が用意されている。

このような手続き的記法が関数呼出しを主体とした本来の Lisp の記法に比べてわかりやすいか否かは一概には言えない。読者の判断に委ねたい。

7.2.4 リスト処理

本章では，これまで点数リストから平均点を求めるプログラム例を用いて，

Lispの関数型言語としての特徴を中心に説明してきた。以下では，Lisp言語開発の本来の目的であったリスト処理機能について例を用いて述べる。

図7.6の例は，上記（L3）の名前と点数のリストのリストを点の高い順に並べるプログラムである。並べ替えのアルゴリズムは，挿入ソート法という簡単な方法を用いている。関数nmsortはリストの第1要素を除いた残りのリストをまず並べ替えて，その後，関数insertを用いて第1要素を適切な位置に挿入する再帰関数になっている。

したがって，実際にはリスト要素を右端から順に取り出して点数の大きい順に並べているので，関数nmsortの出力は以下のようになる。

```
( )
((dan 80))
((dan 80) (chiba 60))
((baba 90) (dan 80) (chiba 60))
((baba 90) (dan 80) (abe 70) (chiba 60))
```

ここで，よく利用されるリスト処理関数をまとめると次のようになる。

①基本関数：car, cdr, cons
②合成関数：cadr, cadar,…
③操作関数：list, append

基本関数は図7.7に示すような関係がある。合成関数はcarおよびcdrを組み合わせたものであり，(cadar p)は(car (cdr (car p)))と同じである。リストを組み立てる関数としては，consが第1引数を第2引数のリストの先頭の要

```
(L10)  名前と点数のリストを点の高い順に並べる
  (defun  nmsort  (nmlist)
   (if  (null  nmlist)  ()
                    (insert  (car  nmlist)  (nmsort  (cdr  nmlist)))))
  (defun  insert  (student  sortlist)
   (cond  ((null  sortlist)  (list  student))
          ((< (cadr  student)  (cadar  sortlist))
           (cons  (car  sortlist)  (insert  student  (cdr  sortlist))))
          (t (cons  student  sortlist))))
```

図7.6　Lispプログラムのリスト処理の例

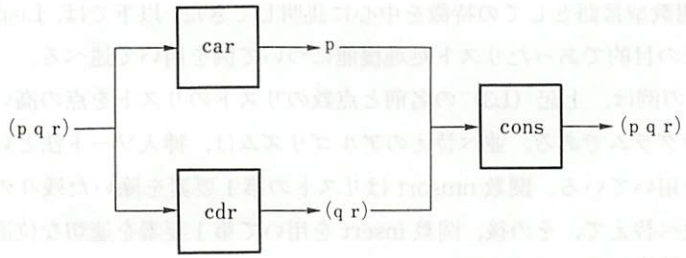

図7.7　リスト処理の基本関数

素として加えるのに対して，list は，任意の個数の引数をすべてリスト要素としたリストを生成する。append は，第1引数のリストと第2引数のリストを1つのリストに統合する。各々の例を以下に示す。

```
(cons '(p q) '(r s)) → ((p q) r s)
(list '(p q) '(r s)) → ((p q) (r s))
(append '(p q) '(r s)) → (p q r s)
```

図7.6のプログラム例でこれらの機能を確認して欲しい。なお，2行目の( )は空リストのときに空リストを返すという意味の定数で，nil と書いてもよい。関数 insert の定義の中の<は，比較演算の関数である。t は，常に真であることを意味する定数で，この例では，cond というマクロ呼出しの最初の2つの条件が共に偽のときに必ず実行する処理を指定するのに用いている。

最後に，名前と点数のリストを読み込み，関数 nmsort を用いて成績順に並べ替えた結果を出力するプログラムを図7.8に示す。

```
(L11)  成績順に出力する
(defun  listout  ()
   (print  "Input  a  list  of  (name  marks)  :  ")
   (setq  inlist  (read))
   (setq  outlist  (nmsort  inlist))
   (print  "A  sorted  list  of  students  :  ")
   (print  outlist))
```

図7.8　Lisp プログラム（入出力処理の記述例）

7.2.5 LispとPrologの比較

人工知能分野の基本言語として比較的よく用いられているLispとPrologの類似点と相違点について述べる。

(1) プログラミング機能

まず計算処理やリスト処理などのプログラミング機能についてみてみよう。本書では，同じ例題を用いているので，以下のプログラムについて，比べてみる。

- 算術計算処理（平均点の計算）：図6.5と図7.4，図7.5
- リスト処理（点数順のソート）：図6.6と図7.6，図7.8

主な類似点と相違点は次のようなものである。

① アルゴリズムの記述はほぼ同じである。

② プログラム構造は，関数型の言語であるLispが関数定義という枠組みを持ち，関数呼出しによって処理が記述されているのに対し，論理型言語のPrologはホーン節という述語を平坦に並べたものである。

③ パラメータ機構は，Lispは関数の引数はすべて入力となり，関数呼出しで返された値を関数自身が出力として持つのに対し，Prologでは引数が入力にも出力にもなる代わりに計算結果を返す必要があるときにはそれも引数にする必要がある。たとえば平均値を求めるaverageは，Lispでは入力用の引数が1つあるだけだが，Prologでは出力用の第2引数を持つ。

④ 例外処理のような場合分け処理は，Prologの方が簡単に記述できる場合が多い。たとえば入力が空リストのときの処理は，Lispでは関数定義本体の最初でifを用いて記述しているが，Prologでは節の左辺の引数部分のユニフィケーション機能（パターンマッチング機能）を用いる。

⑤ リスト処理の基本機能は，Lispでは基本関数を用いて(car marklist)，(cdr marklist)，(cons student marklist)などと記述するが，Prologでは[Student | Marklist]というように簡単に記述できる。

(2) 推論機能

LispとPrologの大きな違いは推論機能である。Prologではユニフィケーシ

ョンとバックトラッキングによる縦型探索機能がすでに備わっているが，Lispでは推論機能を自分で作りこむ必要がある。それだけ自由度が大きいが手間もかかる。興味のある読者は，図6.2の家系図から祖父を求めるPrologプログラムあるいは図6.8の飛行ルートを選択するPrologプログラムをLispで記述してみて欲しい。

7.3 関数型プログラミングの特徴

★

副作用のない関数だけで構成されたプログラムは，その意味が文脈で規定され，実行過程には依存しないので，わかりやすい。

7.3.1 参照透明性

関数型プログラミングの大きな特徴は，副作用のない関数だけでプログラムを構成することにより，そのプログラムの意味が文脈によって規定され，実行過程には依存しないことである。このような性質は参照透明性（referencial transparency）といわれる。

関数の数学的概念については，関数を写像と見る場合と計算規則と見る場合がある。前者の場合，関数fは，集合Pから集合Qへの写像を表し，集合Pの要素xが集合Qの要素f(x)に対応することを意味する。後者の場合は，入力xを与えられて，出力f(x)を生成するとみなせる。写像の概念の方が計算規則の概念より広いが，ここでは，計算パラダイムの観点から後者の計算規則の立場をとる。

この場合，関数型プログラムの意味論に形式性を与えるのにλ-計算法（lambda calculus）が用いられる。これは，もともとは関数の一般的な数学的性質を形式的に論じるために，A. Churchらによって1940年頃に導入されたものである。すでに，7.2.3項で述べたように，関数自身を表すときには，λxf(x)というλ-記法を用いる。そして，関数型プログラミングの基本である関数作用，すなわち関数fを値aに作用させることをf(a)またはfaと記述する。新たに関数を定義するときは，関数本体に対応する式Mを用い，その式の中に

現れるxを変数とみなして，λxM(x)と記述できる。この操作をλ-抽象(lambda abstraction) またはλ-束縛 (lambda binding) という。

7.3.2 仕様記述言語

関数には，手続き型プログラミングのような計算手順とか計算過程の状態という概念がないので，仕様とプログラムを区別して考える必要がない。関数は数値データなどと同格のものであり，数学的な計算操作により，変換や合成ができるので，各々の関数は，ある計算の抽象レベルを表現した仕様とみなすことができる。したがって，抽象度の高い仕様からコンピュータ処理向きの実用的なプログラムを導出する作業を人手によらず，数学的規則の適用により機械的に行える。そして，手続き型言語では非常にやっかいなプログラムの検証を，形式的仕様に対してのみ行えばよいという大きな利点がある。

このような関数型パラダイムの性質は，ソフトウェア工学の重要なテーマの1つである形式的仕様記述と仕様からのプログラム自動生成に適している。形式的仕様から実用的なプログラムを導出する技術はプログラム変換 (program transformation) といわれる。この技法は，1977年にエジンバラ大学のR.M. Burstallらによってその有用性が示されて以来，多くの研究がなされている。基本的な変換規則として，新しい関数の定義，関数の引数を具体値（定数）で置き換える具体化 (instantiation)，関数呼び出し（関数の左辺）にあたる部分を関数本体 (関数の右辺) で置き換える展開 (unfolding)，その逆に関数の右辺にあたる部分をその関数の左辺で置き換える畳み込み(folding)，関数の右辺の一部を変数で置き換える抽象化(abstraction)，あるいは結合則や交換則などの数学的法則の関数右辺への適用，などがあり，これらを用いて実用的なプログラムを導出する。

筆者は，手続き型パラダイムの構造化技法の研究の中で，まず最初に徹底した構造化プログラムを作成し，その検証終了後に最適化コマンドを用いてソースレベルのプログラム変換を行う2段階プログラミング法（文献(31)：two-stage programming) を実現した経験がある。この時，変換規則を基本的なものに限定すると，規則の適用が容易になるが最適化の効果も限定される。一方，ノウハウ的な細かい変換規則をライブラリにたくさん登録すると，規則の適用が複雑になるというジレンマに陥ってしまった。妥協案として，会話型システ

ムにして，変換の前後の等価性の検証のみを自動化した。関数型プログラミングの世界では，プログラム変換の対象となる仕様が形式的であるため，手続き型言語に比べてプログラム自動生成の可能性は高い。

7.3.3 プログラミング言語

関数型言語は数多く開発されているが，特徴的な機能としては以下のようなものがある。

①関数の引数に関数を指定したり，出力として関数を返すような高階関数を扱える。
②参照透明性の性質のため，並行処理を導入しやすい。
③遅延評価 (lazy evaluation) 方式により，無限長リストのようなデータを扱える。

関数型言語は，これまで理論面の研究が先行してきたが，実用面からの検討も活発に行われている。上記のような機能に加えて，関数の定義域と値域に対応する型に関する強い型付けや型を1つに限定しないで複数の型を許すポリモーフィズムの導入，より高速の実行方式と関連した関数型プログラム向きマシンなどが研究されている。言語仕様の意味記述への適用などの実用実績も挙げられている。