プログラミング言語

プログラミング言語〈歴史〉（ぷろぐらみんぐげんご〈れきし〉）

プログラミング言語は、プログラム内蔵方式の計算機が発明された1950年代から計算機には必須の**道具**（tool）である。道具といってもハードウェアではなく、目に見えない命令の集まりから構成される。計算機内部では全ての情報が2進数で表現されるため命令も2進数で表現する必要があり、当初は機械語といわれていた。その後、人間には理解することが難解な機械語と英単語を１対１に対応させ、より理解し易くした**アセンブリ言語**（assembly language）が考案され、比較的容易に命令を表現することができるようになった。しかし、アセンブリ言語は､計算機の頭脳にあたる**CPU**（control processing unit、中央処理装置）毎に書き方、文法が異なるため、まだ不便であった。

さらに、1950年代中頃から後期にかけて英語に近い文法で３つの**高水準言語**（high level language）が考案、開発された。一つは、科学技術計算を記述するに向いたFORTRAN、その他、事務処理を記述するに向いたCOBOLと動的処理（後に人工知能分野で多く用いられた）を記述するに向いたLISPである（LISPは当初FORTRANで記述されたこともあり３つの中では最も後発である)。その後、Pascal、Basic、C、Prolog等代表的な言語が考案されるにつれ、計算機処理の記述を**プログラム**（program、算譜)、記述することを**プログラミング**（programming）ということから、これら言語を**プログラミング言語**（programming language）というようになった。プログラミング言語は、人間がコミュニケーションとして用いる日本語､英語､フランス語など**自然言語**(natural language）に対して**人工言語**（artifitial language）といわれる。

1970年代の人工知能研究の高まりによって考案されたプログラミング言語Prologが考案された頃には、プログラミング言語を**手続き型言語**（procedual language)、**関数型言語**（functional language)、**論理型プログラミング言語**（logic programming language）の３つに分類した。手続き型言語の代表としてFORTRAN、COBOL、Pascalがあり、関数型言語にはLISP、ML、Haskell等がある。論理型プログラミング言語の代表としてPrologがある。

1970年代から研究が進んでいた**オブジェクト指向**（object-oriented）という概念（「システム管理論」の用語を参照）に基づいたプログラミング言語としてオブジェクト指向言語Smalltalk-80が1980年に発表された。その後、Self、Javaを代表とする多くの**オブジェクト指向言語**（object-oriented language）が考案、開発された。さらに、複数のCPUを持つ計算機（並列計算機）の処理に向いた**並列言語**（paralell language)、**並行言語**（concurrent language)、アスペクト指向に基づいた言語、**アスペクト指**

向言語（aspect-oriented language）が考案、開発された。これらは、新しい言語を提案すると同時に、新しいパラダイムを提案しているので、既存の手続き型言語に付加する形で提案されることが多く、Concurrent Pascal、Concurrent Smalltalk、Parallel Object-oriented Language、AspectJのように様々な言語が研究の中で提案された。

現在では、インターネットの広まりにより**WWW**（World Wide Web）が構成され、いわゆるホームページがあらゆるところで公開されるようになり、ホームページを記述する言語として**スクリプト言語**(script language)が多く用いられている。スクリプト言語の代表として perl、PHP、Javascript、Rubyがある。

以上のように様々な分野に向いたプログラミング言語が考案され、実用的プログラミング言語に留まらず、研究を進める中で一時的に考案されたプログラミング言語も数多くあり、全体として数万種類にのぼるといわれている。

手続きと関数（てつづきとかんすう）

プログラミング言語を記述する場合、**手続き**（procedure）と**関数**（function）の違いについては本質的な話題であり、研究の分野では様々な議論があった。プログラムは、命令の集まりを実行順序に従って並べて記述するが、各命令が順番に何千、何万個になる可能性がある（実際には何百万個以上になる）ので、特定の機能を実現するグループに分け記述する。これらグループをプログラミング言語では、一般に手続きという。例えば、画面に一文字表示する機能を手続きPutCharacterとして定義したり、キーボードから一文字入力し、プログラムの中に保存する機能を手続きGetCharacterとして定義するということである。手続きは、通常別のプログラム（手続き）から呼出し、その機能を利用する。例えば、以下のように記述する（以下は、架空のプログラミング言語)。

```
Procedure Main ( )
        Call PutCharacter(‘H’)
End Procedure
Procedure PutCharacter (Character c)
  画面に変数cに格納された文字を表示
End Procedure
```

ここでは、手続きMainから手続きPutCharacterを呼出し、「1文字画面に表示する」という機能を処理している。手続きMainから手続きPutCharacterを呼出す前と後を比較すると計算機の「状態」が変化していることが分かる。つまり、計算機のディスプレイに文字が表示されていない「状態」から文字'H'が表示されている「状態」への変化である。ある手続きを呼出すことにより「状態」が変化することがある。このような手続きの性質を**状態あり**（state full、ステートフル）という。「状態」を手続き自身が持っていると考えることができるので、状態ありという。

一方、逆の性質、**状態なし**（state less、ステートレス）について考えてみよう。以下の例題を考える。

```
Procedure Main ( )
        Integer i =3j
        i = Call Increment(i);
End Procedure
Integer Procedure Increment(Integer r)
```

```
        r = r + 1;
        Return r;
End Procedure
```

ここでは、手続き Main から手続き Increment を呼出し、手続き Main 内にある変数の値を 1 増加する処理をしている。結果として、変数 i の値が 1 増加しているが、手続き Increment の機能としては、変数 i の値を変化させることはないので（変数 i への代入によって値が変化している)、「状態」は変化しないことを示す。または手続き Increment は状態を持たないことを示す。そして、手続き Increment は状態なしという。状態なしの手続きは、数学での関数（function）と同じ性質であることが分かる。数学では、関数を以下のように定義する。

入力 (x) ⇒ 関数 (f) ⇒出力 (y)

ある入力 x に対して出力 y となる場合

と書く。関数　においてどのような処理をするかは「隠されて」入力と出力に注目する。関数自身はブラックボックスとして扱う。この場合、入力側の状態と出力側の状態は変化しない。状態なしの手続きは関数と考えることができ、関数を中心に考えるプログラミング言語を関数型言語という。状態なしのいわゆる**純粋な関数**（pure function）のみを対象とする言語は、**純粋な関数型言語**（pure functional language）という。純粋な関数型言語には、**参照の透明性**（referential transparency）や**型安全**（type safe）等の性質があり、数学的に証明することができる。C 言語や PHP 等のプログラミング言語では、手続きの代わりに全て関数として定義する。その他、手続きを Procedure、関数を Function として区別し定義する言語 Pascal もある。

以上のように、プログラミング言語ではプログラム断片である手続き、関数に対して状態あり（ステートフル)、状態なし（ステートレス）という性質が存在するので、十分に理解することが必要である。この概念は非常に重要であり、情報学の分野の中でもインターネット通信分野に応用されている。

人工知能に向いたプログラミング言語（じんこうちのうにむいたぷろぐらみんぐげんご）

1980 年代に世界的な人工知能研究の高まりに従ってプログラミング言語の研究分野では、いつくかの言語が考案された。日本においても第五世代コンピュータ開発機構を設立し盛んに人工知能の研究が進められた。人工知能研究の目標は、その名の通り「人間の知能を計算機上で実現すること」であった。代表的言語は、LISP と Prolog である。

LISP（LISt Processing）は、計算機の創世記の 1950 年代後半に MIT のジョン・マッカーシにより考案されたプログラミング言語であり、動的処理に強い。文法に相当する構文規則はなく「括弧」の対応によりプログラミング言語の要素を記述する。LISP は、処理系を実現することが簡単であったため、様々な亜種が存在し実際にはどのくらいの数の種類が存在するかはわからない程である。数学や論理学の一分野である「計算論」で使われる関数記述法のλ計算を基にしており、λ計算をそのまま記述することができるという特徴を持つ。例えば、階乗を求める関数 factorial は、以下のように定義

することができる（アンダーラインの部分は、入力した部分）。

```
>(define (factorial x)
   (if (= n 0) 1
    (* x (factorial (- n 1)))))
>(factorial 13)
 6227020800
```

さらに、λ式を定義すると以下のようになる。

```
>(define id
   (lambda (x) x))
>(id 10)
10
```

人工知能分野で多く使われるもう一つの代表的な言語がPrologである。Prologは、PROgramming in LOGicから名づけられた。1972年頃にカルメラウワーとコワルスキーによって考案された。通常、変数への代入はx = 3 のように記述し、片方向の代入と考えることができるが、双方向の代入を示す x ⇔ y（xとyの値を一致させる）概念、**ユニフィケーション**（単一化）が中心の処理系である。例えば、以下のように記述することができる。

```
?- (X,23,Z)=(34,Y,83).
X =34,
Y=23,
Z=83.
```

このように、等式が成立するという条件を満たすように「等式の右かつ左からの代入」を同時実行することができる。一般には**パターンマッチ**（pattern match）という。ユニフィケーションを用いると、いわゆる**事実**（fact）を蓄積することができる。

```
father(hiroshi,taro).
father(hiroshi,jiro).
?- father(X,taro).
X=hiroshi.
?-father(hiroshi,Y).
Y=taro.
Y=jiro.
```

taroとjiroの父がhiroshiであるという事実を最初の2行で示している。その後「taroの父はだれか?」という**問い合わせ**(query)に対してhiroshiであるとPrologの処理系が応えている。逆に「hiroshiの子はだれか?」という問い合わせにも応えることができる。以上の記述の単位になっている要素を**述語**（predicate）といい事実であると同時に蓄積されて事実は**知識**（intelligence）を表現していると考える。そのため知識表現に向いたプログラミング言語ともいわれている。Prologは、手続きではなく（事実の）宣言によりプログラムを記述するので、宣言型プログラミング言語または、非手続き型プログラミング言語ともいう。

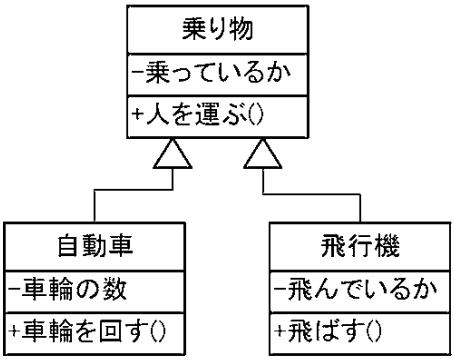
オブジェクト指向言語の機能（おぶじぇくとしこうげんごのきのう）

オブジェクト指向に基づいたプログラミング言語を**オブジェクト指向言語**（object-oriented language）という。オブジェクト指向言語として最初に考案されたのはSmalltalk（現在はsqueakいう）であるが、その後Self、Objective C、C++、Ruby等が考案され、現在ではJavaが多く用いられている。

オブジェクト指向言語は、オブジェクト指向概念に含まれる1. 全てをオブジェクトとして扱う、2. データ（状態）と手続き（挙動の記述、操作、メソッド）をオブジェクトの中に**カプセル化**

(encapuslation、information hiding、情報隠ぺい）する、3. クラスベースとプロトタイプベースがある等の機能を持っている。その他、オブジェクト間の関係を定義する機能、並列・並行処理を定義する機能、動的な型チェックをする機能等、実際にプログラムを記述する際に必要な様々な機能が備わっている。

継承（inheritance）は、クラス間の親子関係を定義する機能である。クラスベースオブジェクト指向言語に備わっている機能であるが、プロトタイプベースオブジェクト指向言語には同様の**委譲**(delegation）という機能がある。



例えば、システム設計でよく用いられるUML（Unified Modeling Language）のクラス図を用いて定義することができる（上図)。四角一つは、クラスを示す。親クラス「乗り物」と2つの子クラス「自動車」、「飛行機」を考える（UMLでは､状態を属性、挙動の記述を操作という)。クラス「乗り物」は､クラス「自動車」とクラス「飛行機」に共通となる「人を運ぶ」という操作をもつ。同時に、「乗っているか」という属性を持つ。子クラスであるクラス「自動車」は「車輪を回す」という操作を持ち、「車輪の数」という属性を持つ。クラス「飛行機」は、独特の「飛ばす」という操作を持ち、「飛んでいるか」を示す属性をもつ。このように、子クラスは親クラスの属性、操作を継承し、さらに独自の属性、操作は自らの中に定義する。結果として親クラスにて属性、操作の共有を行うことができる。

オブジェクト指向言語により記述されたプログラムは、属性、操作をオブジェクト内にカプセル化し、全体としてオブジェクト間の関係は継承等を用いて緩やかに結ばれた関係（loosely coupled relation）を築いている。そのため、依存関係を巧みに変化させることにより様々な関係の動的な変化を記述することができる。

値渡しと参照渡し（あたいわたしとさんしょうわたし）

手続きまたは関数を定義する場合に値の受渡しに**引数**（argument）と**返り値**（return value）を定義する。特に、引数の部分には、手続きまたは関数を呼出す側と呼出される側で値を受渡す。以下の例で考える（C言語)。

```
#include<stdio.h>
int twice(int x)
{
    return 2*x;
}
int main(void)
{
    int y = twice(2);
    printf(“%d\n”,y);
    return 0;
}
```

この例では、関数mainから関数twiceに値2を渡すと2倍となって返ってくる。

つまり、関数 twice に値を受渡している。C 言語では、引数として指定する値は呼出される側の関数へ受渡される。この呼出し方式を**値渡し**（call by value）という。C 言語では、全ての場合値渡しである。

一方、値渡しではそのままでは実現できない呼出し方式がある。例えば、新しい関数として呼出し側で定義された 2 つの変数の値を交換することを考えよう。以下のように記述したとする。

```
#include <stdio.h>
void swap(int x, int y)
{
    int t = x; x = y; y = t;
}
int main(void)
{
    int a = 34, b = 87;
    printf(“(a,b)=(%d,%d)\n”,a,b);
    swap(a,b);
    printf(“(a,b)=(%d,%d)\n”,a,b);
    return 0;
}
```

この例では、main 関数で定義されている変数 a,b の値 34、87 を交換しようとしているが、交換しない。原因は、呼出し側である main 関数において「値渡し」で受渡しているためである。この問題を解決するためには、**参照渡し**（call by reference）を用いればよい。

```
#include <stdio.h>
void swap2(int *x, int *y)
{
    int t = *x; *x = *y; *y = t;
}
int main(void)
{
    int a = 34, b = 87;
    printf(“(a,b)=(%d,%d)\n”,a,b);
    swap2(&a,&b);
    printf(“(a,b)=(%d,%d)\n”,a,b);
    return 0;
}
```

この例では、関数 swap2 に変数 a、b へのポインタ（参照）を受渡し、呼出された側の関数 swap2 では、ポインタを基に main 関数で定義されている変数の中身を直接参照している。よって、変数 a、b の中身を交換することができる。C 言語では関数の呼出しにおいて値渡しをするため、ポインタ（参照）を用いて明示的に参照渡しを実現することができる。

呼出しの際に、値渡し、参照渡しを使い分ける機能は、ほとんどのプログラミング言語に備わっているが、参照渡しのみを提供するプログラミング言語（FORTRAN 等）もある。

パラドックス（ぱらどっくす）

ある命題について、証明しようと（推論すると）真（正しい、true）とも偽（正しくない、false）ともなってしまう命題を**パラドックス**(paradox)という。例えば、以下の命題について考える。

この 13 文字の文は嘘である非常に単純な（たった 13 文字の）文であるが、難しい性質を持っている。

証明：この文が真（正しい）と仮定するとこの文は嘘であるので偽（正しくない)となる。逆に､この文が偽(正しくない)と仮定するとこの文は真（正しい）となってしまう。つまり、この命題は絶対的に正しいとも正しくないとも証明することができない。正しいとも正しくないと

も証明することができない命題をパラドックスという。パラドックスは、数学では扱わない（ゲーム理論では扱う)。

より正確に定義するために、集合論で紹介される2つのパラドックスを紹介する。

床屋のパラドックス： ある村の住人である床屋の主人は、自分の髭を剃らない「すべての」村人の髭しか剃らないという。では、この床屋の主人の髭は自分で剃るか？

証明：この床屋の主人が自分で自分の髭を剃るとしたら、自分の髭をそらない「すべての」村人の髭しか剃らないという仮定に反してしまう（床屋の主人自身も村人の一人であるため)。逆に、自分の髭を剃らないとしたら、この床屋の主人（自分自身）に髭を剃られることになってしまう。

ラッセルのパラドックス：自分自身を含まない無限集合全体の集合Xは、自分自身を含むか？

証明：集合Xは自分自身を含む(X ∈ X)と仮定すると、Xは自分自身を含まない集合の全体集合であるので、X ∉ Xと結論づけられる。逆にXは、自分自身を含まないと仮定すると、Xの定義（Xは、自分自身を含まない無限集合の全体集合）よりX ∈ Xと結論づけられる。

パラドックスの存在は、1930年代に10代のクルトゲーデルにより厳密に証明された（ゲーデルの第二不完全性定理)。その後、数学、論理学ではパラドックスが生じないように様々な工夫がなされ、「証明することができる」命題しか扱わない（ゲーム理論では扱う)。パラドックスが生じない工夫としてラッセルのパラドックスを発見したバートランドラッセルが型理論を考案した。プログラミング言語の分野では型理論を応用して「エラー」を発見するために、型理論を応用したデータ型を考案した。

関数型言語 Standard ML（かんすうがたげんご すたんだーど えむえる）

1970年代よりエジンバラ大学教授ロビン・ミルナー博士により考案された言語が**ML**（Meta Language）である。MLは、定理証明系を記述するために考案されたプログラミング言語である。その後、改良がなされいくつかのバージョンが考案された。エジンバラ大学のEDML、ベル研究所のStandard ML of New Jersey、CAML（Caml、Caml-light、OCaml）等がある。1997年にStandard MLという言語仕様がエジンバラ大学で定義され、現在では標準となっている（以下、Standard ML of New Jerseyでの実行結果を用いて説明)。

Standard MLは、関数型言語であり、以下のように簡単に実行することができる。

```
-1 + 3;
val it = 4 : int
-fun twice x = 2 * x;
val twice = fn: int -> int
```

太字で示したように型を自動的に推論する機能を持っている。この機能を**型推論**（type inference、type reconstruction）という。つまり、

項の型を宣言，定義することなく、処理系が自動的に型を推論する機能

が備わっている。さらに、「MLのプログラムがうまく型付けされれば（principal type を持つ)、プログラムを実行しても

エラーとならない」という命題（**型の健全性**という）が成立することが数学的に証明されている。この性質は、非常に良い性質でプログラムを実行することがなくてもエラーを完全にチェックすることができることを示している（一般の言語では、逆にプログラムを実行することなしに停止するかどうかを完全にチェックする方法がないことが数学的に証明されている)。ML は、型の健全性というよい性質を利用して、さらに、厳密さが要求される定理証明系を記述することができる。

関数型言語であるので、純粋な関数を簡単に定義することができる。

```
-fun decrement x = x – 1;
val decrement = fn: int -> int
-val f = fn x => x;
val f = fn : ‘a -> ‘a
-f 10;
val it = 10 : int
```

fn x => x は、λ式のλ x .x に相当し、λ式も直接記述できることがわかる。また fn x => x は､x に「どんな型」の値も適用することができるので、型推論した結果､「全ての型を示す型 'a を用い 'a -> 'a と推論される。この 'a を多相型(polymorphic type) という。**多相型**を扱うことができるのも Standard ML の利点である。

Standard ML は、関数の引数として関数を受渡すことができる**高階関数**（higher order function）を扱うことができる。

```
-val inc x = x + 1;
val inc = fn: int -> int;
-inc 2;
val it = 3 : int
-map inc [63,26,33,45];
val it = [64,27,34,46] : int list
```

と処理することができ、関数 map は関数 inc を引数として受渡している高階関数である。

Standard ML は、数学や論理学で議論になった様々な要素を含んでおり、証明問題の練習として良い題材としても使うことができ、計算機科学の学生には授業、演習等で多く用いられている。

データ型（でーたがた）

プログラミング言語において、プログラムを実行することなく静的にエラーをチェックするために**データ型**（data type）が導入されている。元々、数学、論理学の分野でパラドックスを回避するために型理論が考案された。パラドックスは、永遠に計算が続き計算が停止しない計算であると考えることができるが、計算機科学ではパラドックスは「必要な計算」である。しか、一般に計算機上でプログラムを実行する場合、プログラムが停止するかどうかを（数学的に）証明することができないため、停止するか（エラーが存在するか）をチェックするためにデータ型を導入している。データ型は、型理論を基に考案された仕組みである。プログラム言語には、データ型を導入しない**型フリー**（type free）の言語も存在するが現在の主流はデータ型があり静的にエラーをチェックする言語である。

データ型は、ものの集まりである集合に対応する。例えば､整数型は整数の集合、関数型は、関数の集合に対応する。集合論でよく知られているように、自分自身を含む無限集合を許すとパラドックスとなることがあり、パラドックスが生じないように**型理論**（type theory）が考案さ

れた。プログラミング言語では、さらに複雑な処理を記述するために厳密に数学、論理学での型定義とは異なり、エラーを「できるだけ」静的に見つけるためにデータ型が考案された。しかし、最近では厳密にデータ型を定義し、エラーを全て静的にチェックすることができる**強い型付け**（strongly typed)、厳密さには欠けるがほぼエラーをチェックすることができる**弱い型付け**（weekly typed）の両方が考案されている。プログラムの中で**式**(expression）に型を対応させプログラム全体の型を計算する。強い型付けにおいて矛盾なく型付けをすることができれば、プログラムにエラーが含まれない。最近では、関数型言語に強い型付けをするプログラミング言語の登場（ML､Haskell 等）により、強い型付けをするプログラミング言語が多く提案された。さらにオブジェクト指向言語 Java の登場でクラスを型に対応させ複雑な型（オブジェクト指向言語の機能に対応した型）を定義し、より強力にエラーをチェックすることができる言語もある。

データ型には、プログラムに対して静的にチェックする場合だけでなく、実行時にチェックする動的な型（dynamic typing）もある。動的な型チェックはプログラムの実行効率に関係するためできるだけ少なくする方がよい。オブジェクト指向言語 Java では、基本的に静的に型チェックをするが、一部で動的型チェックをする効率のよい型チェックをしている。

以上のように、最近では計算機の処理の向上に伴って、数学、論理学の分野で厳密に議論されてきた集合論を用いて複雑なプログラムでも正確にチェックすることができるようになってきた。

コンパイラコンパイラ（こんぱいらこんぱいら）

プログラムを計算機上で実行可能な機械語に変換する処理系であるコンパイラ(compiler）に対し、コンパイラ自身を自動生成する処理系をコンパイラコンパイラ（compiler compiler）という。コンパイラコンパイラは、コンパイラを容易に開発するために考案された。1970 年代に様々なコンパイラが開発されたが、処理系そのものは複雑になってきた。コンパイラでおこなわれる処理は、字句解析、構文解析、意味解析、最適化、コード生成の各フェーズがある。コンパイラコンパイラは、前半のフロントエンド（front end）といわれる字句解析、構文解析部分を中心に自動生成する処理系として活発な研究が行われた。

字句解析は、プログラミング言語の最小単位である**字句**（token）を定義し、プログラムの中から字句を切り分ける処理である。コンパイラコンパイラの一部である字句解析生成系（字句解析部を自動生成する処理系）は、正規表現を用いて字句を定義しプログラムから字句を切り分ける処理(字句解析系)を自動生成する。構文解析は、プログラムが文法に適って記述されているかをチェックする。同じく構文解析生成系は自由文脈文法を用いてプログラムの文法を定義し、文法に適っているかどうかをチェックする処理（構文解析系）を自動生成する。

代表的な字句解析生成系には、lex があり様々な言語のバージョンがある。C 言

語のプログラムを生成する lex（最初に考案された）と flex、Standard ML プログラムを生成する ml-lex 等がある。構文解析生成系として yacc、bison は C 言語のプログラムを生成し、ml-yacc は、Standard ML プログラムを生成する。

ガベージコレクション（がべーじこれくしょん）

プログラム中で記憶領域を動的に割り当てる場合は、割り当てた領域の参照を管理し、計算の過程で使う。簡単なデータ構造から複雑なデータ構造等、様々な場面で動的に割り当てた記憶領域を使う。動的に割り当てた記憶領域は、使用しなくなった時に解放しないと限られた記憶領域を使い尽くしてしまう。一般に、記憶領域の「割り当て」、「解放」はプログラム中で明示的に指定しなければならないが、「解放」のタイミングが計算結果に依存したり、「解放」を忘れたりとプログラミング上のミスをすることが多い。その場合にはプログラムの実行時に予期しないエラー（unexpected error）となり、致命的エラーとなる。

以上の理由により LISP をはじめとして動的記憶領域割り付けを多く用いる言語の処理系には、使用しなくなった記憶領域を自動的に解放するガベージコレクション（garbage collection、ごみ集め）という機能が備わっている。ガベージコレクションの方法としては、参照カウント法、マークアンドスイープ法、コンサバティブ・ガベージコレクションなど様々な方法が考案､研究されてきた｡1970 年代、1980 年代の計算機は、記憶装置が大変高価であったため非常に小さな容量しか実装されておらず、プログラミング言語処理系の分野ではガベージコレクションに関する研究が活発であった。日本においてもガベージコレクション専門に研究する研究者も多かった。