ICOT Technical Memorandum: TM-0364

TM-0364

ESP入門

近山　隆，石橋弘義

July, 1987

©1987, ICOT

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

ＥＳＰ入門

近山　隆　・　石橋　弘義

（ＩＣＯＴ　第四研究室）

チュートリアルの進め方

【１】設計言語の背景

【２】言語仕様の概要

休憩

【３】プログラム具体例

【４】質疑応答

用語：計算機／ＯＳ／言語

ＰＳＩ： 個人使用向逐次型推論マシン

ＫＬ０： 核言語第０版
　　ＰＳＩの機械語
　　Prolog の主要な機能 ＋α

ＳＩＭＰＯＳ： ＰＳＩのＯＳ
　　個人使用向き
　　多重プロセス
　　マルチ・ウィンドウ機能

ＥＳＰ： ＳＩＭＰＯＳの記述言語
　　論理型＋オブジェクト指向機能

設計上の要請

ＳＩＭＰＯＳの記述 ——→

・大規模システムの記述に適すること
・十分な実行速度・メモリ効率を得られること
・アプリケーション・プログラム（論理型）との整合性がよいこと

ＰＳＩのＯＳ ——→

・ＫＬ０との整合性がよいこと

機能概要

論理型機能：ＫＬ０の機能をほぼそのまま利用

- Prolog の基本機能
- ＫＬ０の持つ拡張（論理型）機能

オブジェクト指向機能：論理型との整合性が問題

「 公理系＝オブジェクト 」 が基本方針

- 多重継承（multiple inheritance）
- 実行速度を重視

プログラムの記述形式

```
class クラス名 has
nature 継承クラス名並び...;
    attribute 属性スロット名並び...;
    component 要素スロット名並び...;
    :メソッド名(引数並び...) :- 本体;
    ...
instance
    attribute 属性スロット名並び...;
    component 要素スロット名並び...;
    :メソッド名(引数並び...) :- 本体;
    ...
local
    述語名(引数並び...) :- 本体;
    ...
end.
```

述語の種類

☆組込述語：ＫＬ０で直接実行

ＫＬ０の組込述語　＝
Prologの組込述語
－ソフトウェアで実現すべき機能
・入出力
・アトム名前表管理
・プログラム・データベース管理
これらはＳＩＭＰＯＳで提供する

☆ローカル述語：Prologと同じ「手続き」指向
ひとつのクラス定義内だけ有効
──→ モジュラリティ

☆メソッド：オブジェクト指向機能を実現
：メソッド名（オブジェクト，他の引数‥）
引数のオブジェクトによって呼出し先が変わる

オブジェクト

☆クラス・オブジェクト

クラスごとにひとつ

#クラス名

おもな役割：

・状態保持の必要のないサブルーチン群

・インスタンス管理用オブジェクト

☆インスタンス・オブジェクト

ひとつのクラスにいくつでも作れる

:new(Class_Obj, Instance_Obj)

定数としては書けない ⟶

論理変数値・スロット値として保持

スロット

オブジェクトの状態保持に用いる

☆スロット値の参照・更新

```
Object!Slot_name
Object!Slot_name := value
```

★属性スロット（attribute）
どこからでも名前で参照可能

★要素スロット
同じクラス定義内からのみ参照可能

☆クラス定義中の記法

```
attribute a, b, c;
component age := 32, weight := 67;
```

スタックとヒープ

☆スロット値の更新は

バックトラック時に戻らない ⟶
変数値などを入れると変なことが起きる

☆構造データを二種類に分類

スタック上の構造：
バックトラック時に解放
変数を含むことができる

ヒープ上の構造：
バックトラック時にも解放されない
変数を含まない

☆スタック⟷ヒープ変換
ソフトウェアで記述
（ファーム・サポート開発中）

継承

☆クラスの親子関係

・定義中の nature で親クラスを指定

・複数の親を指定できる → 多重継承

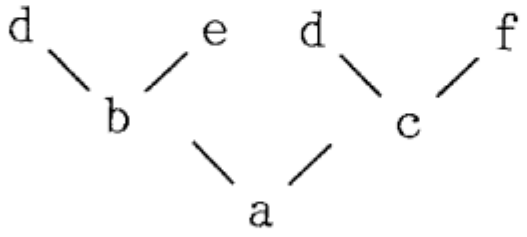
☆親クラスから継承するもの

・スロット
・メソッド
] 全部の合併集合

→ 継承構造は IS-A ネットワーク

継承の順序

```
class a has nature b, c; ... end.
class b has nature d, e; ... end.
class c has nature d, f; ... end.
```



☆継承先祖クラスの順序

a, b, d, e, c, f

スロット・メソッドの継承

☆スロットの継承

・全先祖クラスのスロットの合併集合
　ただし，同じ名前の属性スロットはひとつだけ

☆メソッドの継承

・全先祖クラスのメソッド・クローズの合併集合
・用いる順序は継承の順序

非単調な継承

☆オーバライディング

・継承したメソッド・クローズも「カット」

☆デモン

・論理積として効くようなメソッド

```
before:open(Object, ...) :- B;
      :open(Object, ...) :- P;
after:open(Object, ...) :- A;
```

```
:open(Object, ...) ⟶

  call(B), call(P), call(A)
```

マクロ展開

☆定義の形式

パターン =〉展開形
when 生成ゴール列
where 検査ゴール列
:- 展開条件;

☆ボディでの展開

(1) パターンを展開形で置き換える

(2) パターンのあるゴールの直前に
生成ゴール列を置く

(3) パターンのあるゴールの直後に
検査ゴール列を置く

☆ヘッドでの展開

(1) パターンを展開形で置き換える

(2) ボディの最後に生成ゴール列を置く

(3) ボディの先頭に検査ゴール列を置く

マクロバンクの定義

```
macro_bank マクロバンク名 has
nature 継承マクロバンク名, ...;
    マクロ定義;
    ...
local
    ローカル述語定義;
    ...
end.
```

マクロの使用法

☆ユーザ定義マクロバンクの使用法

class クラス名 with_macro マクロバンク名 has

...

☆標準マクロ：特に指定する必要なし

- ＥＳＰの基本機能 —
  メソッド呼出し，スロット・アクセス，等
- 関数的記法 — 四則演算
- その他の略記 — 比較

バインド・フック

☆変数の具体化時に呼び出されるゴール列を指定

bind_hook(変数，ゴール列)

すでに具体化されていたら → 即座に実行

同じ変数に複数回フック →
フックされた変数同士のユニフィケーション →

具体化時に両方実行（ＡＮＤ）
順序は規定されない

☆逐次マシン上でのコンカレントな実行の記述に便利

現状と将来

☆ＳＩＭＰＯＳ第２版でマクロなど最新機能を提供

☆今後とも拡張・改良の予定

　原則としてアップワード・コンパチブル

ＥＳＰ入門

近山　隆　・　石橋　弘義
（ＩＣＯＴ　第四研究室）

逐次型推論マシンPSIの上で利用できる言語ESPについて解説する。ESPは論理型プログラミング・パラダイムとオブジェクト指向パラダイムの両者を融合した形で提供する言語である。本稿は論理型プログラミング，特にPrologに関する基本的な知識を持つ読者を対象としている。Smalltalk やFlavorsなどのオブジェクト指向言語に関する知識は理解の助けになるが，必須ではない。

１．はじめに

（１）設計目的と開発の歴史

現在のESPはパーソナル型逐次型推論マシンPSIのプログラミング／オペレーティング・システムSIMPOSの記述に用いるのを主目的として，1983年にICOTで設計されたものである。PSIは第五世代プロジェクトのソフトウェア研究のためのツールとして作られた計算機で，その機械語であるKLOは，Prologのもつ本質的な機能をすべてそなえた論理型言語である。PSI上のアプリケーションは，KLOの論理型言語としての機能をフルに活用して作られることになる。また，高度なマンマシン・インタフェースの研究も不可欠であるので，そのオペレーティング・システム SIMPOSはできる限り柔軟に幅広い機能を提供できるものにする必要があった。こうしたアプリケーション・システムから SIMPOS の提供する機能をスムーズに利用できるようにするには，SIMPOS とアプリケーション・プログラムを同じ考え方の言語，つまり論理型言語で統一的に記述することが望ましかった。

当時普及しつつあったPrologシステムの多くは平板なプログラム構造しか持っておらず，なんらかのプログラム構造を持たせたシステムも実験的な範囲に止まっており，実行効率上の配慮が十分になされていないものしかなかった。SIMPOSは個人使用向けのシステムではあるが，マルチプロセス機能を持つ高機能ワークステーションのオペレーティング・システムとして，ある程度大規模なものにならざるをえない。したがって，その記述言語にはなんらかのモジュール構造の導入が不可欠であった。また，実験システムではなく，第五世代プロジェクトのソフトウェア研究用のツールとして実用に供するのが目的であるので，十分な実行効率が得られる言語である必要もあった。このためには従来のPrologシステムと同様の機能では明らかに不十分であり，新たな言語の開発が必要とされたわけである。

当時論理型プログラミングと同様注目を浴びつつあったプログラミング・パラダイムに，Smalltalk-80やFlavorsなどに代表されるオブジェクト指向プログラミングがあった。オブジェクト指向パラダイムは，共通して利用できるプログラム部分の再利用（いわゆる差分プログラミング）が容易であるという長所がある。論理型パラダイムに基礎を置きつつ，このオブジェクト指向パラダイムをできる限り自然な形で取り入れることにより，実用オペレーティング・システム規模のプログラムの開発にも十分なモジュラリティを実現する，という方針で言語 ESPは設計された。

SIMPOSはもっとも低レベルのデバイスの操作から，もっとも高レベルのユーザ・インタフェースに至るまで，すべてESPで記述された。ESPの言語仕様を設定してプログラムの実質的な開発を開始してからでは約１年強，実機上でのデバッグを開始してからでは約８ヶ月後には，FGCS'84 国際会議の会場でデモンストレーションを行えるまでに至った。これだけ迅速な開発が可能であったのは，優秀な開発担当者の努力の成果であることは言を待たないが，言語ESPの持つ諸機能が有効に働いた結果でもある。

SIMPOSの開発がほぼ順調に進行し，アプリケーション・ソフトウェアが作られ始めると，ESP はアプリケーション・プログラムの記述言語としても用いられるようになってきた。PSI

のように大容量のメモリを備え，高速な推論ができる計算機上では，アプリケーション・プログラムも大型化してくる。こうしてプログラムのモジュール化の必要性が高まってくると，ESPの持つモジュール化の機能がアプリケーション・レベルの記述にも役立ち始めたのは，当然ではあるが，当初の予想以上の成果であった。

（2）機能概要

ESPは論理型プログラミングとオブジェクト指向プログラミングの両パラダイムを融合した形で提供する言語である。論理型機能はPSIの機械語であるKLOの持つ機能をそのままの形で提供しているものである。オブジェクト指向機能はKLOに追加されたプリミティブな支援機能を利用しつつ，基本的にはKLOへの翻訳技術によって実現されている。

論理型機能：

KLOの持つ論理型プログラム言語としての機能は基本的にはPrologと同じである，すなわち，ユニフィケーションに基づいた引数渡し，バックトラックによる実行制御などがそれである。したがって，ESPの論理型機能は基本的にはPrologと同じと考えて支障ない。

KLOはPrologの機能に加えて，論理型プログラミングと直接に関係する拡張機能として以下のようなものを備えている。

・変数値の実体化(instantiation)によるプログラム起動機構
　　···· bind_hook
・広域的なバックトラックの制限機構（カットの拡張）
　　···· absolute_cut, relative_cut
・バックトラックによるプログラム起動機構（カットされないもの）
　　···· on_backtrack

このうち特にアプリケーション・プログラムにとって有用なのはbind_hook機能である。

オブジェクト指向機能：

Prologの実行は，与えられた命題の否定に対し実例をあげて反駁する過程(refutation) とみなされる。この際，反駁の基礎となる公理群が論理型プログラムそのものであり，実例をあげるために変数値を具体的に呈示したものが計算結果である。

同じ命題を与えられても，基礎となる公理系が異なれば，反駁の仕方，すなわち計算過程とその結果は異なる。場合によっては反駁の成否も変わってくる。 ESPでは，反駁に用いる公理系を実行時に決まるデータによって決めることができるようになっている。つまり，プログラム上の同じ呼出しを実行しても，データとして与えられた公理系によって呼ばれるプログラムが異なる。ここで公理系すなわちオブジェクトと考えれば，オブジェクト指向プログラミングそのものである。

ESPでも，他の多くのオブジェクト指向言語と同様，オブジェクトは時刻とともに副作用的に値を変えうる状態変数を保持している。状態変数の値は公理系の一部と解釈できるので，その値を変化させることはPrologのassertやretractの機能にあたるものである。

多くのオブジェクト指向言語と同様，ESPも良く似たオブジェクトをひとまとめに定義する手段としてクラスの概念を持つ。また，クラス間に共通する部分をできる限りまとめて共通に記述し，異なる部分だけを抽出して個別に記述する，いわゆる差分プログラミング (differencial programming)を実現するために，クラス間の継承(inheritance) の機構も備えている。ESPの継承機構は，差分プログラミングを徹底しやすいように，複数の親クラスを同時に継承できる多重継承(multiple inheritance)を可能にしている。

２．　プログラムの形式

以下，本稿でプログラムの形式に関する記述の中では，英数字は具体的にその記号，漢字・ひらがな・カタカナはなんらかの具体的な文字列に置き換えられるべきメタ記号を表わす。

ESPのシンタクスは全体としてはDEC10-Prologなどと同様の演算子順位文法によって定義される。ただし標準の演算子定義はDEC10-Prologなどとは若干異なる。

プログラムはクラスという単位ごとにクラス定義の形で記述される。クラス定義の形式は全体としては以下のようなものである。

```
class クラス名 has                          % クラス名の指定
nature 継承クラス名並び...;                 % 継承クラスの指定
    attribute 属性スロット名並び...;        % ┐
    component 要素スロット名並び...;        % │クラス・オブジェクトに
    :メソッド名(引数並び...) :- 本体;       % │関する定義
    ...                                     % ┘
instance
    attribute 属性スロット名並び...;        % ┐
    component 要素スロット名並び...;        % │インスタンス・オブジェクトに
    :メソッド名(引数並び...) :- 本体;       % │関する定義
    ...                                     % ┘
local
    述語名(引数並び...) :- 本体;            % ┐ローカル述語の定義
    ...                                     % ┘
end.
```

ここに現われる"class"や"local"などのキーワードは，実際は予約語ではなく，演算子として定義されているものである。したがって，それらを一般のデータとして書きたい場合は"(class)"のように括弧でくるんで演算子として働かないようにすればよい。

各項目の詳細は項目ごとに後述する。

３．組込述語・ローカル述語・メソッド

ESPの実行の基本単位はPrologと同様ゴールと呼ばれるものである。ESPのゴールは

組込述語
ローカル述語
メソッド

のいずれかの呼出しである。

機械語KLOで直接実行される組込述語や，普通のPrologと同様の意味を持つローカル述語の呼出しは，

述語名(引数並び...)

のような形式をしている。組込述語はどこからでも呼び出すことができるが，ローカル述語はそれを定義しているクラスの定義内からしか呼び出すことができない。これはプログラムのモジュール化を助け，無用な名前の衝突を避けるための機能である。

ローカル述語の定義はクラス定義中のキーワード"local"の後に書く。記法はPrologのクローズと同様である。ただし，クローズの終わりはピリオドではなくセミコロンで表わす。このため演算子セミコロン(;)の優先順位は演算子:-よりも低くなっているので，クローズの中にセミコロンを書くときには全体を括弧で括る必要がある。

実行時に第一引数にオブジェクトを指定し，それによって呼出し先が変わるような述語をメソッド(method)と呼ぶ。メソッドの呼出しは

:メソッド名(オブジェクト，引数並び...)

のように先頭にコロンをつけることによって，組込述語やローカル述語の呼出しと区別する。第一引数のオブジェクトは，後述の記法で直接クラス・オブジェクトを指示する定数を書いてもよいし，プログラム・テキスト上では変数としておき，実行時にオブジェクトがその値として入るようにしておいてもよい。前者の方法は通常のモジュール構造を持つ言語でモジュール名を指定して手続きを呼び出すのと同じである。後者の方法ならばどのようなものを呼び出すかは実行時にはじめて決まることになる。この場合，メッセージの受け手にあたるオブジェクトは呼出しの時点で未定義であってはならず，すでになんらかのオブジェクトに具体化されていなくてはならない。

メソッドの定義はクラス・オブジェクト（後述）に関するもの，インスタンス・オブジェクト（後述）に関するものを，キーワード"instance"でわけてクラス定義中に書く。"instance" の前がクラス，後がインスタンス・オブジェクトに関する定義である。メソッドのクローズの記法もローカル述語の記法と同様であるが，ヘッドの前にコロン(:) をつけてメソッドの定義であることを明示する。

組込述語・ローカル述語・メソッドのいずれも，どれを呼び出すかは単に名前だけではなく引数の個数によっても識別される。たとえば

```
p(a)    p(b)
```

は同じ述語を呼び出すことになるが

```
p(a)    p(a, b)
```

は異なる述語を呼ぶゴールである。

通常のプログラムの書き方としては，クラス定義の外側（自分の親クラスや子クラスも含む）に見せる必要のあるエントリはメソッドとし，それ以外はローカル述語とするのがよい。こうすることによって，内部の実現手段をインタフェース・レベルから隠し，将来変更することがあってもインタフェースを同じに保つことが容易になる。

４．クラスとインスタンス

オブジェクトにはクラス・オブジェクトとインスタンス・オブジェクトの二種類がある。

クラス・オブジェクトはひとつのクラスに対してひとつなので，そのクラス名を用いた

```
#クラス名
```

という記法でESPプログラムのどこからでも参照できる。クラス・オブジェクトは状態を保持する必要のないサブルーチン群をまとめたモジュールを作るときや，インスタンス・オブジェクト群を管理する目的に用いるのが普通である。

インスタンス・オブジェクトはひとつのクラスについていくつでも必要に応じて生成することができる。このため，クラスの名前だけでは特定のインスタンス・オブジェクトを指定することは不可能である。しかし，生成したインスタンス・オブジェクトを論理変数の値，または他のオブジェクトのスロット値（後述）として保持し，後に用いることはできる。

メソッドは原則としてクラス定義にその定義を記述するものであるが，これ以外にいくつかの標準のメソッドが自動的に用意される。たとえば，新しいインスタンス・オブジェクトの生成には，そのインスタンスが属すべきクラスの標準メソッド:new を

```
:new(クラス・オブジェクト，インスタンス・オブジェクト)
```

のように呼び出す。第二引数は生成されたインスタンス・オブジェクトとユニファイされる。たとえばクラスhumanのインスタンスを作るには

```
:new(#human, X)
```

のようにすればよい。呼出しの時点でＸが未定義なら，呼出しの結果としてＸにはクラス human に属する新しいインスタンスが値として入る。

あるオブジェクトがクラス・オブジェクトかインスタンス・オブジェクトかを調べるには標準のメソッド

```
:is_class(オブジェクト)
```

を用いる。このメソッドは引数のオブジェクトがクラス・オブジェクトなら成功，インスタンス・オブジェクトなら失敗する。

クラス・オブジェクトのクラス名を知るためには，標準のメソッド

```
:class_name(クラス・オブジェクト，クラス名)
```

を用いる。このメソッドは第二引数とクラス名のアトムをユニファイする。たとえば

```
:class_name(#human, X)
```

を実行すると，第二引数のＸにはアトムhumanがユニファイされる。このメソッドはクラス・オブジェクトについてのみ標準に定義され，インスタンス・オブジェクトにはない。

５．スロット（slot）

スロットはオブジェクトの動的に変化する状態を保持するために用いる状態変数である。スロット値の参照には，標準のメソッド

:get_slot(オブジェクト，スロット名，値)

を用いる。値が必要な個所に標準のマクロ(後述)を用いて

オブジェクト!スロット名

のように書くこともできる。たとえば

:get_slot(Object, Slot, Value), p(Value)

というゴール列は

p(Object!Slot)

と略記することができる。

スロット値の変更には標準のメソッド

:set_slot(オブジェクト，スロット名，値)

を用いる。これを標準のマクロを用いて

オブジェクト!スロット名 := 値

のように書いてもよい。

スロットの値はオブジェクトの定義する公理系の一部と考えられる。":get_slot"という述語の定義がスロットの値を返すようになっているわけである。こう考えると":get_slot"はなんら論理の範囲を逸脱しない通常の述語と考えてよい。":set_slot"が":get_slot"に対するassert/retractの機能を持つメタレベルの述語になっているわけである。

オブジェクトのスロットには属性スロット(attribute)と要素スロット(component)の二種類がある。属性スロットは上記の記法でどこからでも参照，更新ができるが，要素スロットは同じクラス定義の中でしかできない。これは要素スロットの名前がローカル述語の名前と同様，ひとつのクラス定義の中だけからしか見えないように局所化されているためである。

属性スロットについても，自分の親クラスの属性スロット以外は直接参照・変更を行わず，アクセス用のメソッドを設定し，それを通して行うのが標準スタイルである。このスタイルを守ることは，オブジェクトの表現形式の変更の際のプログラムの保守コストの低減につながる。

属性スロットはキーワード"attribute"，要素スロットは"component"を先頭に，その名前をコンマで区切って，クラス・インスタンスそれぞれの定義部の中に書く。終わりはセミコロンである。たとえば

```
attribute a, b, c;
```

はみっつの属性スロットａ，ｂ，ｃを定義している。

スロットの値はオブジェクトが作られたときに整数０に初期化される。この初期値は

```
component age := 32, weight := 67;
```

のように定義中に指定して変更することができる。

スロットの値はいったん設定されるとその設定を行ったプログラム部分をバックトラックしても元には戻らない。これはいろいろな面で有用であるが，このため，スロット中に変数を含みうるような値を直接入れることはできなくなっている。これを許すといわゆるダングリング・ポインタが生ずる恐れがあるからである。したがって，Prologの構造体データのようなものはいったん変数を含まない形式に変換してからでないとスロットに格納できない。この変換のファームウェア・サポートを現在開発中である。

6．継承（inheritance）

クラスの定義中には任意個の親クラスの指定ができる。あるクラスが親クラスを持つとき，そのクラスのクラス／インスタンス・オブジェクトは，そのクラス自身の定義中に記されたものに加え，原則として直接・間接の親クラスの定義に与えられたメソッド／スロットのすべてを合わせもつ。つまり，継承は公理集合の合併集合を作るものである。あるクラスに関する継承関係は木構造をなすことになるが，これはIS-A関係で結合されたセマンティック・ネットワークを構成する。

（1）継承の順序

親子の両クラス，または複数の親クラス同士が同じメソッド（メソッド名も引数個数も同じメソッド）を定義している場合は，子クラスのメソッドはこれらの定義すべてが合成されたものになり，反駁—実行の過程ではどこで定義された公理を適用してもよいことになる。これは継承が公理集合の合併であることの当然の帰結であるが，逐次的実行を行うプログラム言語としては，公理の適用の順序が問題となる。

まず，自分自身を含め，すべての継承クラスにひとつの順序を定める。直接の親クラスは，第一に自分のクラス，以下クラス定義に記された順に順序づける。全継承クラスは，各ノードが直接の親クラスの順に順序づけられた部分木を持つようなひとつの木構造をなすことになる。この木をプリオーダでトラバースしたときに現われる葉の順が継承クラスの順である。ただし，複数回でてくるクラスは最初の一回しか含めない。たとえば

aの親はbとc
bの親はdとe
cの親はdとf
eとfには親クラスがない

とすると，継承の順は

a，b，d，e，c，f

となる。dは二回現われるが，最初の一回（bの親として）だけが問題となる。

（2）スロットの継承

各クラスは継承クラスに定義されたすべてのスロットを持つ。ただし，同じ名前の属性スロットが複数個ある場合は，継承の順で最初に定義されたものだけを持ち，残りは無視する。これを重複除去(duplication elimination)という。属性スロットは初期値が異なること以外は，どのクラスのものを持つと考えても同じである。要素スロットについては重複除去は行わない。

（3）メソッドの継承

同じメソッド（メソッド名，引数個数ともに同じもの）が複数の継承クラスに定義されている場合のバックトラック時の実行順序は，継承の順と同じである。継承順で前にあるクラスに定義されたクローズの実行に失敗したときにのみ，後のクラスに定義されたクローズを試みにいく。

（4）オーバライディング

ここまでに述べた継承の機構は，つねに公理が増えていく方向で，単調性を持っている。単調な継承機構だけで実用規模のプログラムを設計することはたいへん難しい。単調な機能だけということは，プログラムの設計の最初からすべての例外を考えに入れて設計しなければならないからである。

従来のPrologでこの単調性を破る機構として，「カット」が用いられてきた。ESPでは，メソッドを定義するクローズのトップレベルに現われるカットは，単にそのクラス定義中のオルタナティヴ・クローズをカットするだけではなく，継承しの結果生じたオルタナティヴをもカットする仕様になっている。これを用いると，子クラスのメソッド定義クローズ中の決

して failしないあたりににカットを書くことによって，親クラスのメソッド定義を子クラスでの定義に置き換える事ができる。これはSmalltalkなどにみられるオーバライディングの機構と同様のものである。

（5）デモン

オーバライディングの機構だけでは，非常に複雑なコントロールを必要とするプログラムの記述には必ずしも十分でない。この不足を補うため，ESPでは通常の論理和として継承されるメソッド定義のほかに，論理積として継承されるようなメソッドを記述できるようになっている。これをデモンと呼ぶ。

デモンの定義はメソッド・クローズの定義の前にキーワードbeforeまたはafterをつけて行う。こうして定義したもの(before/after demon predicate)はそのメソッドが呼ばれたとき，デモン以外の本来の定義(principal predicate)の実行の，それぞれ直前または直後に行われる。たとえば

```
before:open(Object, ...) :- B;
      :open(Object, ...) :- P;
 after:open(Object, ...) :- A;
```

といった定義があったとすると，このメソッドopenの呼出しは，B，P，Aをこの順で実行することになる。

同じメソッドに対するデモンのクローズがひとつのクラス定義に複数個ある場合は，それらのクローズ全部でひとつの述語を定義するものと考え，この述語がデモンとして呼び出される。

デモンに渡される引数はメソッドに渡される引数そのものである。したがって前デモンで入力引数をチェックしたり，後デモンで計算結果をチェックしたりすることは容易である。デモンは論理積として実行されるので，デモンがfailすればメソッド呼出し全体がfailすることになる。

複数の継承クラスがデモンを定義している場合，やはりその実行順序が問題になる。ESPでは，前デモンは継承の順，後デモンは継承の逆順に実行される。後デモンの実行が逆順になっているのは，前後の両デモンを定義しているクラスが数多くあるとき，両デモンがネストして実行されるようにするためである。

７．マクロ展開機能

Prologは関数型の言語ではないので，数式の値が欲しいときにその場所に式を書くようなことはできない．このため計算のための述語を別途明白に呼び出すことが必要で，プログラムの読みやすさ／書きやすさをやや阻害していた．数式に限らず，一般にデータを記述する個所にその満足すべき性質もいっしょに記述するのは，プログラムの読みやすさ／書きやすさの上で有利である。

ESPはこうした要求を満足するために強力なマクロ展開の機構を備えている。 ESPのマクロは単にマクロの呼出し自体を展開結果を置き換えるだけでなく，それを含むゴールの前後に必要なゴール列を挿入する機能を持つ。これを用いれば，いわゆる「項記述」（term description)の機能を実現することができる。

（１）マクロ定義の形式と展開規則

ESPのマクロ定義は以下のような形式をしている。

```
パターン =〉展開形
        when 生成ゴール列 where 検査ゴール列
        :- 展開条件;
```

この中でパターン，展開形はともに論理変数を含みうるPrologのタームである。 "when" と生成ゴール列，"where"と検査ゴール列はそれぞれオプショナルでPrologのゴール列のようなものである。展開条件も":-"と組でオプショナルであるが，普通のクローズのボディ部と同様の形式で，やはりパターンや展開形などにに現われる変数を含んでいてよい。

クローズ中のゴール，ヘッドやボディの引数またはその一部分がマクロ定義のパターンとマッチし，展開条件の実行も成功すると，その部分はマクロ定義中の展開形に置き換えられる。この際もし生成ゴール列や検査ゴール列があれば

☆展開対象がボディ中のゴールないしはその引数の場合
生成ゴール列は展開対象ないしそれを含むゴールの直前に，検査ゴール列は直後に論理積として（コンマで結合されて）置かれる。

☆展開対象がヘッド引数の場合
生成ゴール列はクローズのボディの最後に，検査ゴール列はクローズ・ボディの先頭（":-" の直後）に置かれる。

たとえば

```
X + Y =〉 Z where add(X, Y, Z);
```

というマクロ定義があったとき，

```
p(X, Y) :- q(X+Y);
r(X, Y, Y+Z) :- s(X, Z);
```

というふたつのクローズはそれぞれ

```
p(X, Y) :- add(X, Y, Z), q(Z);
r(X, Y, W) :- s(X, Z), add(X, Y, Z);
```

のように展開される。同じ名前の変数が出てくれば，必要に応じて名前は付け変えられる。

（２）マクロバンクの定義と使用法

マクロ定義はまとめてマクロバンクとして

```
macro_bank マクロバンク名 has    % マクロバンク名の指定
nature 継承マクロバンク名, ...; % 継承マクロバンクの指定
```

```
    マクロ定義;
    ...
local
    ローカル述語定義;
    ...
end.
```

のように定義する。継承マクロバンク中に定義されたマクロもここで定義されるマクロバンクに継承される。ローカル述語は展開条件の中から呼び出すことができる。

マクロバンクを用いるには，まず上記のようなマクロバンク定義を普通のクラス定義と同様に登録しておき，クラス定義の先頭の部分で

```
class クラス名 with_macro マクロバンク名 hash
    ...
```

のように用いるマクロバンクを指定する。こうすることによりこのクラス定義内のクローズ中では，指定されたマクロバンクに定義したマクロの展開が行われる。

たまたまマクロ定義と同じ形式のタームを書く必要が生じたときは，マクロの展開を抑制する必要がある。トップレベルだけ抑制したいときは

```
`(ターム)
```

のように，タームの内側まで展開を抑制したいときは

```
``(ターム)
```

のように記述する。

（3）標準マクロ

ESPでは加減乗除やスロットのアクセスなどのESP基本機能は標準のマクロバンクであるesp_macro_expanderに定義されている。このマクロバンクは特に指定しなくても自動的に用いられる。これを用いてESPのプログラム中には，たとえば

```
count_down(0) :- !;
count_down(N) :- count_down(N-1).
```

といった記述ができる。

8. バインド・フック (bind_hook)

Prologでは原則的にはゴールの実行順は実行結果を左右しない。この性質を意図的に破る機構としては「カット」があるが，これ以外にも，たとえば算術演算などはオペランドの値が演算時に決まっていなくてはならないなどの制約がある。こうした制約を回避するためには「この変数の値が決まった時点でこのプログラムを動かして欲しい」といった，データドリヴンの記述ができると便利である。

KLOではこうした要求に応えるため，"bind_hook"と呼ばれる機構を用意している。この機構は ESPから直接利用することができる。たとえば

```
..., bind_hook(X, p(X, Y)), ...
```

のような記述は，Xに値が入った時点で呼出し"p(X, Y)"を実行するように指定しているものである。もしこの時点ですでにXの値が決まっていれば，"p(X, Y)"は即座に実行される。

同じ変数に複数回フックをかけると，その変数が具体化されたときには両方が起動される。たとえば

```
..., bind_hook(X, p(X)), bind_hook(X, q(X)), ...
```

というゴール列は

```
..., bind_hook(X, (p(X), q(X))), ...
```

または

```
..., bind_hook(X, (q(X), p(X))), ...
```

と同じである。このように多重にフックがかかった場合，その実行順は規定されない。

フックのかかった変数同士をユニファイすると，ユニファイ結果の変数には両方のフックがかかった状態になる。たとえば

```
..., bind_hook(X, p(X)), bind_hook(Y, q(Y)), X = Y, ...
```

というゴール列は

```
..., bind_hook(X, p(X)), bind_hook(X, q(X)), X = Y, ...
```

と同じことになる。

バインド・フックは逐次実行Prologにコンカレントな動作をするコルーチンの記述を可能にする。これを用いたプログラミングはGHCのようなAND並列Prologのプログラミング・スタイルと共通点が多い。

9. ESPの現状と将来

現在SIMPOSは第二版がリリースされており，ESPの処理系もマクロ機能など初期の版よりかなり機能追加したものが提供されている。今後もさらに大規模で複雑なシステムの開発を容易にするため，さまざまな機能拡張を予定している。今後の拡張は原則としてアップワード・コンパチブルなものになる予定である。

ＥＳＰプログラム例

１．オブジェクトの使い方

・２つの二進木のプログラム
　スタック・ベクタ版とオブジェクト版
　前者は メモリをより消費する．

・標準入出力による 出力ルーチン

２．クラスと継承

・ペンギンの生物分類

３．クラス構成とデモンの働き

・ウインドウのクラス standard_io_window の構成（継承クラス）

・メソッド :set_size/3 の動作

４．ユーザ定義マクロの例

・マクロ・バンク定義と実行例
　addl "++" と vector_element "@"

５．バインド・フックの使用

・素数の生成

1. オブジェクトの使い方

(1) スタック・ベクタ版の二進木プログラム

```
% Binary Tree  (Stack Vector version)
%   An empty tree node is [].
%   A non-empty node is [Key, Data, Left, Right].

class binary_tree1 has

    :add(_, Otree, Key, Data, Ntree) :- !,
        insert(Otree, Key, Data, Ntree);

    :get_contents(_, Tree, List) :- !,
        get(Tree, List, []);

 local

    insert([], Key, Data, Ntree) :- !,
        Ntree = [Key, Data, [], []];
    insert([Key0,D0,L0,R0], Key, Data, Ntree) :-
        Key0 > Key, !,
        insert(L0, Key, Data, Left),
        Ntree = [Key0, D0, Left, R0];
    insert([Key0,D0,L0,R0], Key, Data, Ntree) :-
        Key0 < Key, !,
        insert(R0, Key, Data, Right),
        Ntree = [Key0, D0, L0, Right];
    insert([Key0,D0,L0,R0], Key, Data, Ntree) :- !,
        Ntree = [Key0, Data, L0, R0];
        % Update when the same key already exists.

    get([], List, List) :- !;
    get([Key,Data,Left,Right], List, List0) :- !,
        get(Left, List, List1),
        List1 = [[Key,Data] | List2],
        get(Right, List2, List0);

end.
```

```
?- B = #binary_tree1.
 B= #binary_tree1

?- :add(B,[],10,a,T1).
 T1= [10,a,[],[]]

?- :add(B,T1,5,b,T2).
 T2= [10,a,[5,b,[],[]],[]]

?- :add(B,T2,20,c,T3).
 T3= [10,a,[5,b,[],[]],[20,c,[],[]]]

?- :add(B,T3,0,d,T4).
 T4= [10,a,[5,b,[0,d,[],[]],[]],[20,c,[],[]]]

?- :get_contents(B,T4,L1).
 L1= [[0,d],[5,b],[10,a],[20,c]]

?- :add(B,T4,0,e,T5).
 T5= [10,a,[5,b,[0,e,[],[]],[]],[20,c,[],[]]]

?- :get_contents(B,T5,L2).
 L2= [[0,e],[5,b],[10,a],[20,c]]
```

```
?- create(#binary_tree2,B).
 B= $binary_tree2

?- :add(B,10,a), :get_contents(B,L1).
 L1= [[10,a]]

?- :add(B,5,b),  :get_contents(B,L2).
 L2= [[5,b],[10,a]]

?- :add(B,20,c), :get_contents(B,L3).
 L3= [[5,b],[10,a],[20,c]]

?- :add(B,0,d), :get_contents(B,L4).
 L4= [[0,d],[5,b],[10,a],[20,c]]

?- :add(B,0,e).

?- :get_contents(B,L5).
 L5= [[0,e],[5,b],[10,a],[20,c]]
```

（2） オブジェクト版の二進木プログラム

```
% Binary Tree  (Object version)
%   A tree node is an object
%   that has four slots: key, data, left, right.
%   An empty node is what key value is '$empty'.

class binary_tree2 has

 instance
   attribute     key := '$empty',
                 data,
                 left,
                 right;

     :add(Obj, Key, Data) :-
         :is_empty(Obj), !,
         set_values(Key, Data, Obj),
         create_branch(Obj);
     :add(Obj, Key, Data) :- !,
         insert(Obj!key, Key, Data, Obj);

     :get_contents(Obj, List) :- !,
         :get(Obj, List, []);

     :get(Obj, List, List) :-
         :is_empty(Obj), !;
     :get(Obj, List, List0) :- !,
         :get(Obj!left, List, List1),
         get_values(Key,Data,Obj),
         List1 = [[Key,Data) | List2],
         :get(Obj!right, List2, List0);

     :is_empty(Obj) :- !, Obj!key = '$empty';

 local

     insert(Key0, Key, Data, Obj) :-
         Key0 > Key, !,
         :add(Obj!left, Key, Data);
     insert(Key0, Key, Data, Obj) :-
         Key0 < Key, !,
         :add(Obj!right, Key, Data);
     insert(Key0, Key, Data, Obj) :- !,
         set_values(Key, Data, Obj);
         % Update when the same key already exists.

     set_values(Key, Data, Obj) :- !,
         Obj!key  := Key,
         Obj!data := Data;


     get_values(Obj!key, Obj!data, Obj) :- !;

     create_branch(Obj) :- !,
         :class_object(Obj, Class),
         :new(Class, Left),
         :new(Class, Right),
         Obj!left  := Left,
         Obj!right := Right;

end.
```

（3） 出力ルーチン

```
                %%  Output Binary Tree  %%

      %  The argument "IO" is standard_io object.

      class binary_tree has

        nature binary_tree2;

       instance

          :map_tree(Obj, Method, Args) :-
              :is_empty(Obj), !;
          :map_tree(Obj, Method, Args) :- !,
              :map_tree(Obj!left, Method, Args),
              :refute(Obj, Method, Args),
              :map_tree(Obj!right, Method, Args);

          :output(Obj,IO) :- !,
              :putf(IO, "Key= ¥q, Data= ¥q¥n",
                       [Obj!key,Obj!data] );

      end.



?- :new(#binary_tree,B).
 B= $binary_tree

?- :add(B,10,a),
    :add(B,5,b),
    :add(B,20,c),
    :add(B,0,d),
    :add(B,0,e).

?- :get_contents(B,L).
 L= [[0,e],[5,b],[10,a],[20,c]]


?- :create(#standard_io_window,
         [size(300,500)], Window),
    :activate(Window).
 Window= $standard_io_window

?- :map_tree(B, output, [Window] ).


?- :create(#standard_output_file,
             File, "btree.txt").
 File= $standard_output_file

?- :map_tree(B, output, [File] ).          Key= 0, Data= e
                                           Key= 5, Data= b
                                           Key= 10, Data= a
                                           Key= 20, Data= c
?- :close_output(File).
```

2. クラスと継承

```
%% ペンギンの生物分類 %%

class creature has                    %% 生物
    instance
      attribute live := live;          %  生死を記録
      :is_live(Creature) :- !, Creature!live = live;
      :die(Creature) :- !, Creature!live := dead;
end.

class animal has                      %% 動物
    nature creature, locomotive;
end.

class bird has                        %% 鳥
    nature animal, with_wings, flying;
end.

class pengin has                      %% ペンギン
    nature bird, swimming;
    instance
      :can_fly(Pengin) :- !, fail;     %  飛べない（例外）
end.


class locomotive has                  %% 移動性のある
      :can_move(Object);
end.

class with_wings has                  %% 羽を持った
    nature flying;
    instance
      attribute left_wing,
                right_wing,
                tail_wing;
end.

class flying has                      %% 飛べる
    nature locomotive;
    instance
      :can_fly(Object);
end.


class fish has                        %% 魚
    nature animal, swimming;
end.

class swimming has                    %% 泳げる
    nature locomotive;
    instance
      :can_swimming(Object);
end.
```

3. クラス構成とデモンの働き

%% クラス standard_io_window の継承クラス %%

```
standard_io_window                  % 標準ウインドウ
 ├ as_standard_io_window            % 標準入出力機能
 |   ├                              %   例えば，del キー機能の付加
 |   ├ as_standard_input            % 標準入力機能
 |   |   └ v2_as_esp_parser         % パーサ
 |   └ as_standard_output           % 標準出力機能
 |       ├ with_formatted_output    % 書式出力機能 :putf/3
 |       └ v2_as_esp_unparser       % アンパーサ
 ├ labeled_sash                     % ウインドウ枠
 |   ├ with_margin                  % マージン
 |   ├ with_label                   % ラベル
 |   └ with_border                  % ボーダ
 ├ as_scroll                        % スクロール機能
 ├ as_output                        % 文字出力機能
 ├ as_graphics                      % グラフ出力機能
 ├ as_markers                       % マーカー
 ├ reshape_notifier                 % ウインドウのサイズ変更
 ├ as_input                         % 入力機能
 ├ as_mouse_input                   % マウス入力機能
*└ inferior_window                  % 基本ウインドウ
     ├ as_inferior                  % 子ウインドウ機能
     └ basic_window                 % 基礎のウインドウ
         |                          %   全ユーザ・ウインドウは必ず継承
         ├ external                 % プロセス間コミニュケーション
         └ bare_window              % ベア・ウインドウ
             ├ with_default_mouse   % マウス機能
             ├ as_inside            % インサイド操作機能
             ├ graphic_primitive    % グラフィック・プリミティブ
             └ with_display         % ビットマップ・ハンドラとのインタフェイス
```

(*) ウインドウをテンポラリにしたい時は
クラス inferior_window の代わりに，次のクラスを継承する．

```
temporary_window                    % テンポラリ・ウインドウ
 ├ as_temporary_window
 |   ├ with_mouse_sense
 |   └ as_popup_window
 └ basic_window
```

(クラス定義)

```
class  standard_io_window  has
  nature  as_standard_io_window,
          labeled_sash,
          as_scroll,
          as_output,
          as_graphics,
          as_markers,
          reshape_notifier,
          as_input,
          as_mouse_input,
          inferior_window;
end.
```

4. ユーザ定義マクロの例

ESP プログラムは，以下のものを複数個並べて記述できる．

- クラス定義 -- class end.
- マクロ・バンク定義 -- macro_bank end.
- 演算子宣言 -- add_operator(演算子，タイプ，順位).
- include 文 -- include("ファイル名").

```
     %%  Operator Definition  %%

add_operator(++, fx,  600).
add_operator(@,  yfx, 600).


     %%  Macro Definition  %%

macro_bank  my_macro  has

  ++X => X+1    :- integer(X), !;
  ++X => Z      when  increment(X,Z);

  Vect@N => E   when  vector_element(Vect,N,E);

end.


     %%  Class Definition Using Macro Bank  %%

%    class class_name
%        with_macro macro_bank_name  has
%
%      ...
%
%    end.
```

```
?- N= 2.
 N= 2

?- M is ++N.
 M= 3

?- E = [a,b,c]@1.
 E= b



?- :new(#add1, Macro).
 Macro= $add1

?- H1= p(X), B1= ``( f(++X) ),
     :expand_clause(Macro, H1, B1, NH1, NB1).
 H1= p(X), B1= f(++X),
 NH1= p(X), NB1= increment(X,A),f(A), X ?

?- B2= ``( f(++N) ),
     :expand_clause(Macro, head, B2, head, NB2).
 B2= f(++2), NB2= f(3)

?- H3= ``( p(X,++X) ), B3= f(X),
     :expand_clause(Macro, H3, B3, NH3, NB3).
 H3= p(X,++X), B3= f(X),
 NH3= p(X,A), NB3= f(X),increment(X,A)

?- H4= ``( p(X,++N) ), B4= f(X),
     :expand_clause(Macro, H4, B4, NH4, NB4).
 H4= p(X,++2), B4= f(X),
 NH4= p(X,3), NB4= f(X)

?- B5= ``( f([a,b,c]@1) ),
     :expand_clause(Macro, head, B5, head, NB5).
 B5= f(a(b,c)@1),
 NB5= vector_element(a(b,c),1,B), f(B)

?- B6= ``( Z6=[a,b,c]@(++X) ),
     :expand_clause(Macro, head, B6, head, NB6).
 B6=  ( Z6=a(b,c)@ (++X) ),
 NB6= increment(X,B),
        vector_element(a(b,c),B,C),
        unify(Z6,C),
 Z6 ?
```

5. バインド・フックの使用

```
% Lazy Prime Number Generator

class primes has

    :primes(_, Ps) :-
        naturals(2, Ns), sieve(Ns, Ps, Ps);

local

    naturals(N, [N|Ns]) :-
        bind_hook(Ns, naturals(N+1, Ns));

    sieve([N|Ns], Ps, [N|Pt]) :- prime(N, Ps), !,
        bind_hook(Pt, sieve(Ns, Ps, Pt));
    sieve([_|Ns], Ps, Pt) :- Pt ¥== [], !,
        sieve(Ns, Ps, Pt);
    sieve(_,_,_) :- !;

    prime(N, [P|_]) :- P*P > N, !;
    prime(N, [P|Ps]) :-
        N mod P =¥= 0, prime(N, Ps);

end.
```

```
?- :primes(#primes, [X1,X2,X3] ).
 X1= 2, X2= 3, X3= 5

?- :primes(#primes, [Y1,Y2|T] ).
 T ?, Y1= 2, Y2= 3

?- T = [Y3,Y4|T1].
 T1 ?, Y3= 5, Y4= 7
```

ＥＳＰプログラム例

１．クラスと多重継承　　-- ペンギン

２．オブジェクト指向プログラミング　-- 二進木

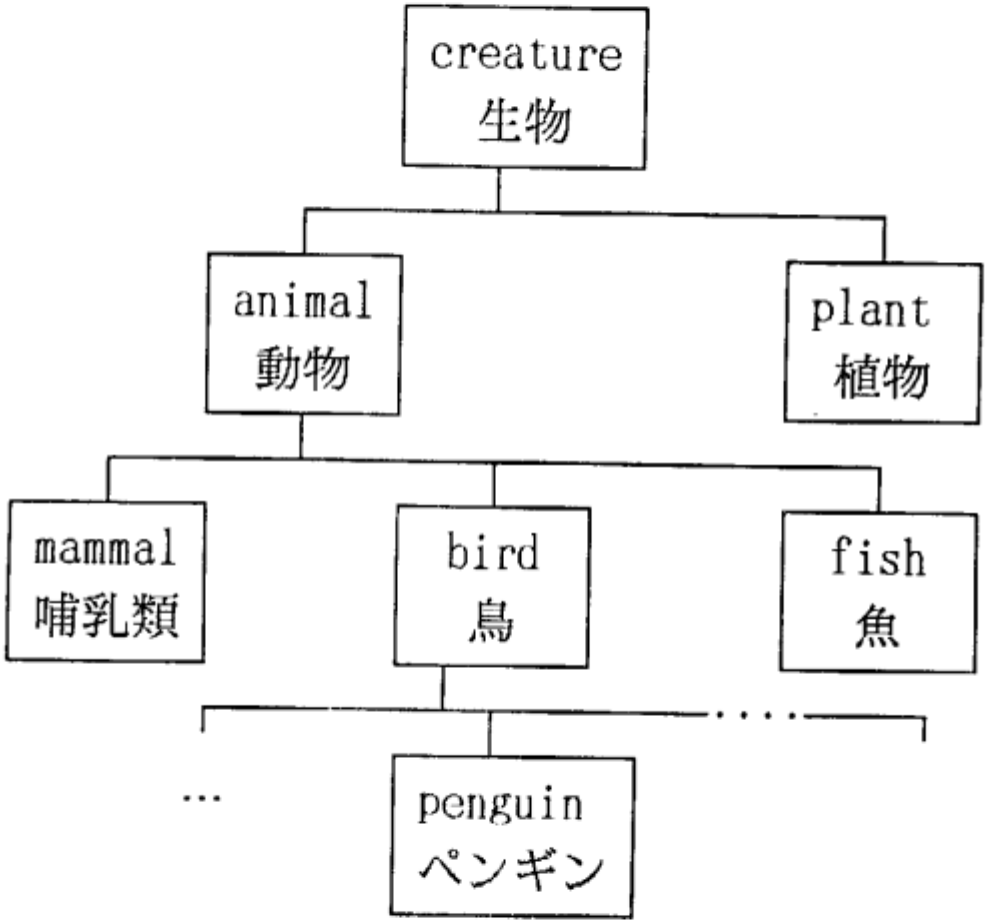
３．ユーザ定義マクロ　　-- add1 など

４．バインド・フック　　-- 素数生成

ＩＣＯＴ　第四研究室

石 橋　弘　義

ペンギンの生物分類


creature
生物
animal
動物
plant
植物
mammal
哺乳類
bird
鳥
fish
魚
…
penguin
ペンギン

分類をクラスで表現

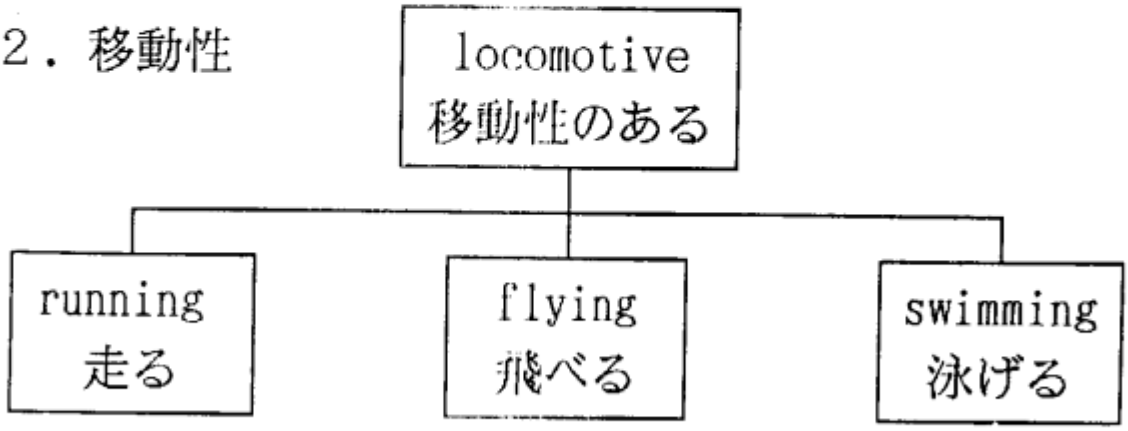
```
class  creature  has
end.

class  animal  has
  nature  creature;
end.

class  bird  has
  nature  animal;
end.

class  fish  has
  nature  animal;
end.

class  penguin  has
  nature  bird;
       %  bird, animal, creature;
end.
```

生物の性質を記述

```
class  creature  has
 instance
  attribute  live := live;
             % 生死を記録 (live, dead)

  % 生死の確認
  :is_live(Creature) :- !,
       Creature!live = live;
  % 死を記録
  :die(Creature) :- !,
       Creature!live := dead;

end.
```

1．生物分類

2．移動性




```
class  locomotive  has
 instance
  :can_move(Object);
end.
```

```
class  flying  has
  nature locomotive;
 instance
  :can_fly(Object);
end.
```

```
class  swimming  has
  nature  locomotive;
 instance
  :can_swim(Object);
end.
```

移動性を追加　⇒ 多重継承

```
class  animal  has
  nature  creature, locomotive;
end.

class  bird  has
  nature  animal, flying;
end.

class  penguin  has
  nature  bird, swimming;
 instance
  :can_fly(Pengin) :- !, fail;
    % 飛べない（例外）　：オーバーライド
end.
```

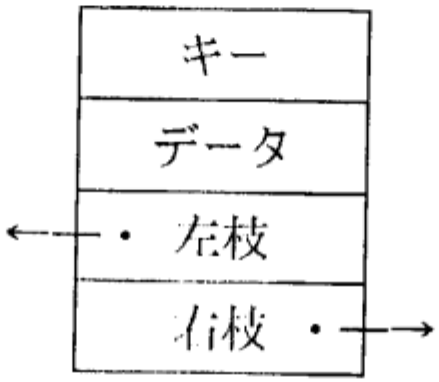
〈 実行 〉

```
%  ペンギンは飛ぶか？
?- :new(#penguin, Penguin),
   :can_fly(Penguin).
no

%  鳥は飛ぶか？
?- :new(#bird, Bird),
   :can_fly(Bird).
yes

%    一匹のペンギンが生れて死んだ.
%      前のペンギンは生きているか？
?- :new(#penguin, Penguin1),
   :die(Penguin1),
   :is_live(Penguin).
yes      % ２匹のペンギンを
         % インスタンス・オブジェクトで区別
```

二進木プログラム

木（tree）は，ノード（node）で構成される．

ノード：



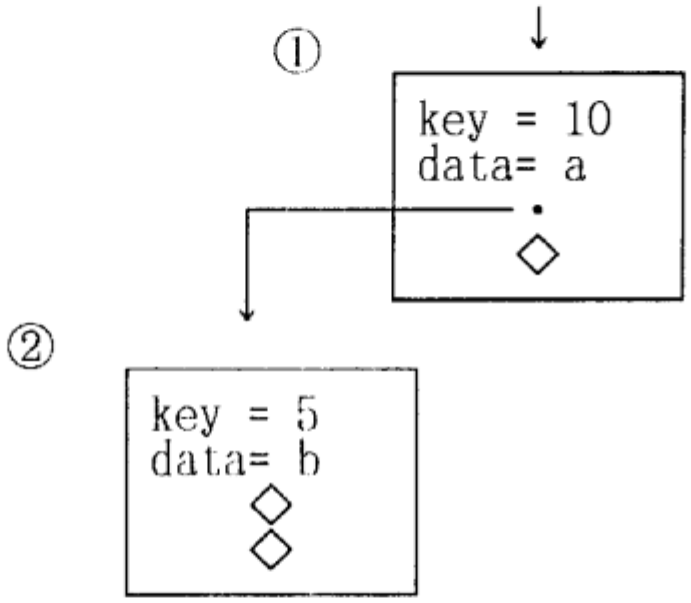
キー値でソートして，データを格納する．

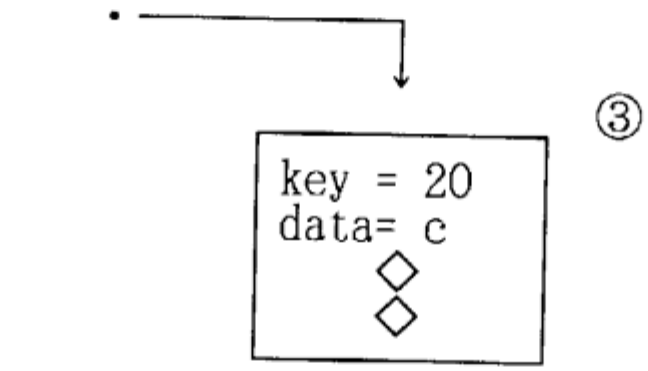
1. 左枝には自分より小さいキー値のデータを，
   右枝には自分より大きいキー値のデータを格納
2. 新しいキー値のデータは必ず枝の先端に追加

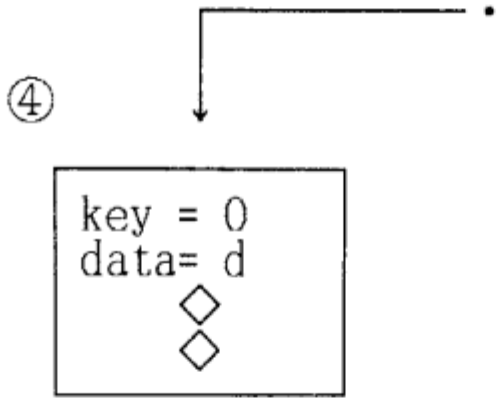
例．　キーとデータのペア

(10,a)，　(5,b)，　(20,c)，　(0,d)

を順に挿入する．




③
key = 20
data= c

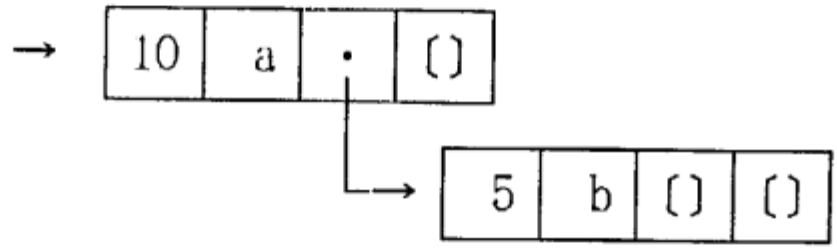

④
key = 0
data= d

node のデータ構造

Node = { Key, Data, Left, Right }

空-tree は〔〕(アトム) で表す.

例. Root = {10, a, Left, 〔〕}
Left = {5, b, 〔〕, 〔〕}



スタック・ベクタ

1. 長さ固定の一次元配列. (⇔ リスト)
2. {a,b,c} は Prolog の '{}'(a,b,c) に似ている.

```
class  binary_tree1  has
   :add(_, Otree, Key, Data, Ntree) :- !,
          insert(Otree, Key, Data, Ntree);
   :get_contents(_, Tree, List) :- !,
          get(Tree, List, []);
 local
          . . .
end.
```

〈 メソッド 〉

:add(_, Old_tree, Key, Data, ^New_tree)

(Key,Data) を追加した新-tree を返す.
Tree の初期値は [] とする.

:get_contents(_, Tree, ^List)

二進木 Tree の内容を取り出す.
結果は {Key,Data} のペアのリスト.

〈 実行結果 〉

```
?- B = #binary_tree1.
 B= #binary_tree1

?- :add(B, [], 10, a, T1).
 T1= {10,a,[],[]}

?- :add(B, T1, 5, b, T2).
 T2= {10,a,{5,b,[],[]},[]}

?- :add(B, T2, 20, c, T3).
 T3= {10,a,{5,b,[],[]},{20,c,[],[]}}

?- :add(B, T3, 0, d, T4).
 T4= {10,a,{5,b,{0,d,[],[]},[]},{20,c,[],[]}}

?- :get_contents(B,T4,L1).
 L1= [{0,d},{5,b},{10,a},{20,c}]
```

```
%  空-tree の時
insert([], Key, Data, Ntree) :- !,
        Ntree = {Key, Data, [], []};

%  左に挿入
insert({KeyO,DO,LO,RO}, Key, Data, Ntree) :-
        KeyO 〉 Key, !,
        insert(LO, Key, Data, Left),
        Ntree = {KeyO, DO, Left, RO};

%  右に挿入
insert({KeyO,DO,LO,RO}, Key, Data, Ntree) :-
        KeyO 〈 Key, !,
        insert(RO, Key, Data, Right),
        Ntree = {KeyO, DO, LO, Right};

%  キー値が同じなら，データを更新
insert({KeyO,DO,LO,RO}, Key, Data, Ntree) :- !,
        Ntree = {KeyO, Data, LO, RO};
```

```
%   メソッド定義
:get_contents(_, Tree, List) :- !,
        get(Tree, List, []);

    %   空-tree
    get([], List, List) :- !;

    %   左, 自分, 右の順に tree をたどる.
    get({Key,Data,Left,Right}, List, List0) :- !,
         get(Left, List, List1),
         List1 = [{Key,Data} | List2],
         get(Right, List2, List0);
```

〈 述語 〉

get(Tree, ^List, Tail)

Tree の内容を ListとTail の差分リストとして返す.

差分リスト

L = [E1,E2,..,En| T] の時
L と T のペアで要素 E1,..,En を表す

オブジェクト版プログラムの方針

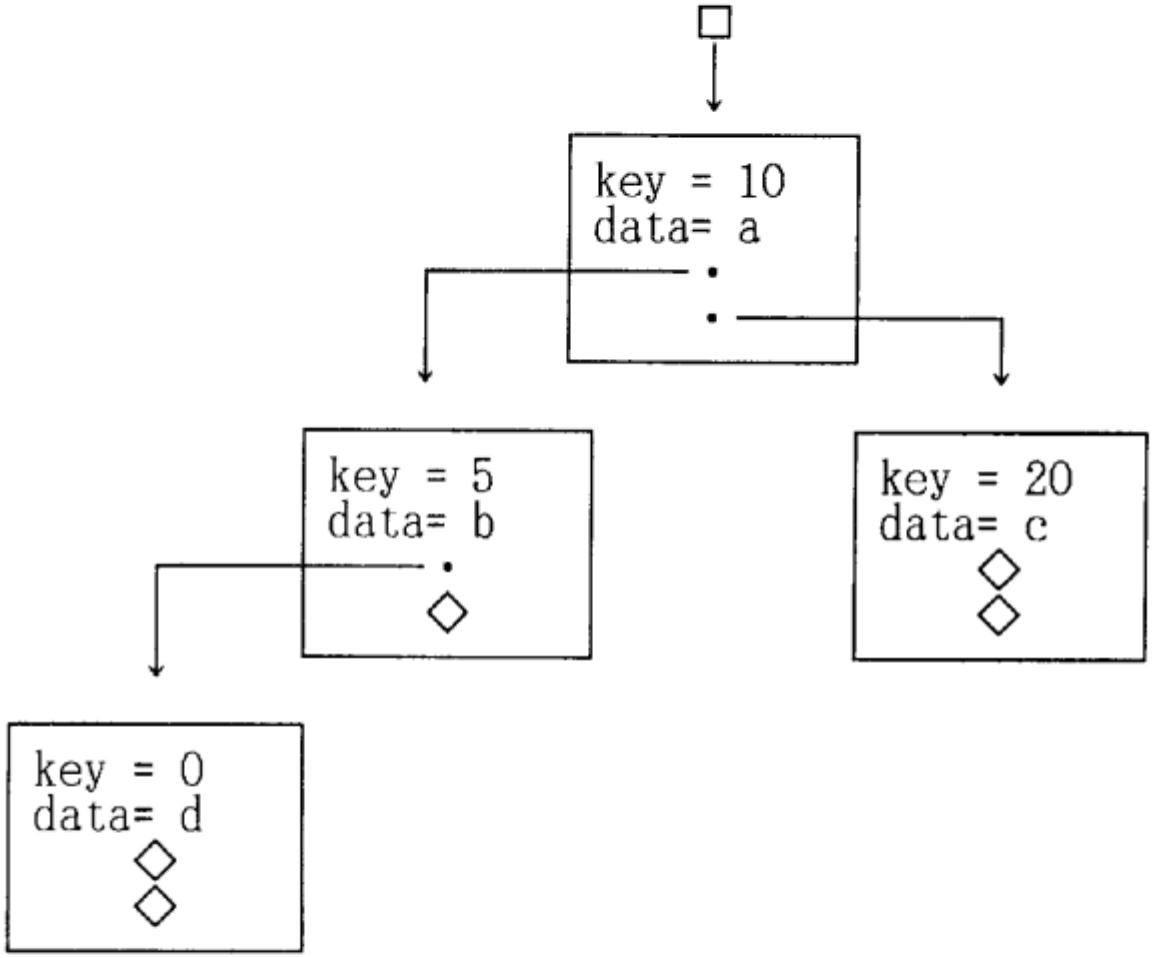
Ⅰ．node のデータ構造

① スタック・ベクタ　{Key,Data,Left,Right}
　　⇒　スロット　key, data, left, right

② クラス・オブジェクト
　　⇒　インスタンス・オブジェクト

Ⅱ．各種操作（:add, :get など）は
　　オブジェクト（各 node）自身に持たせる．

オブジェクト ＝ データ ＋ 操作


key = 10
data= a
key = 5
data= b
key = 20
data= c
key = 0
data= d

```
class  binary_tree2  has
 instance

   attribute     key := '$empty',        % 空-tree
                 data,
                 left,
                 right;

    :add(Obj, Key, Data)
    :get_contents(Obj, List)
    :get(Obj, List, List0)    % サブ-tree の内容を取出す
    :is_empty(Obj)            % 空-tree のチェック

 local
       . . .
end.
```

〈 実行結果 〉

```
?- :new(#binary_tree2,B).
 B= $binary_tree2

?- :add(B,10,a),   :get_contents(B,L1).
 L1= [{10,a}]

?- :add(B,5,b),    :get_contents(B,L2).
 L2= [{5,b},{10,a}]

?- :add(B,20,c),   :get_contents(B,L3).
 L3= [{5,b},{10,a},{20,c}]

?- :add(B,0,d),    :get_contents(B,L4).
 L4= [{0,d},{5,b},{10,a},{20,c}]

?- :add(B,0,e),    :get_contents(B,L5).
 L5= [{0,e},{5,b},{10,a},{20,c}]
```

〈 挿入操作（1）〉

```
%    メソッド定義
:add(Obj, Key, Data) :- !,
        insert(Obj!key, Key, Data, Obj);

    %     左に挿入
    insert(Key0, Key, Data, Obj) :-
          Key0 〉 Key, !,
          :add(Obj!left, Key, Data);

    %    右に挿入
    insert(Key0, Key, Data, Obj) :-
          Key0 〈 Key, !,
          :add(Obj!right, Key, Data);

    %    キー値が同じなら，データを更新
    insert(Key0, Key, Data, Obj) :- !,
          Obj!key := Key, Obj!data := Data;
```

〈 挿入操作（2）〉

```
%    空-tree の時
:add(Obj, Key, Data) :-
        :is_empty(Obj), !,
        set_values(Key, Data, Obj),
        create_branch(Obj);

    %    キー，データのセット
    set_values(Key, Data, Obj) :- !,
         Obj!key  := Key,
         Obj!data := Data;

    %    左右の枝を新しく作る
    create_branch(Obj) :- !,
         :class_object(Obj, Class),
         :new(Class, Left),
         :new(Class, Right),
         Obj!left  := Left,
         Obj!right := Right;
```

〈 内容の取出し 〉

```
%   外部インタフェイス
:get_contents(Obj, List) :- !,
        :get(Obj, List, []);

%   内部メソッド
:get(Obj, List, List) :-
        :is_empty(Obj), !;

:get(Obj, List, List0) :- !,
        :get(Obj!left, List, List1),
        List1 = [{Obj!key, Obj!data} | List2],
        :get(Obj!right, List2, List0);

%   空-tree のチェック
:is_empty(Obj) :- !, Obj!key = '$empty' ;
```

ローカル述語化

〈 なぜ メソッド にしたか？ 〉

× ローカル述語

親オブジェクトが 子-tree すべてを取出す.

○ メソッド

子-tree の取出しは 子-tree に任せる.

⇒ 1. 機能分割，概念整理 をすすめる.

2. 継承による改良(オーバーライド)を可能にする.

オブジェクト版の改良

欠点：　① 空-tree（leaf）では，

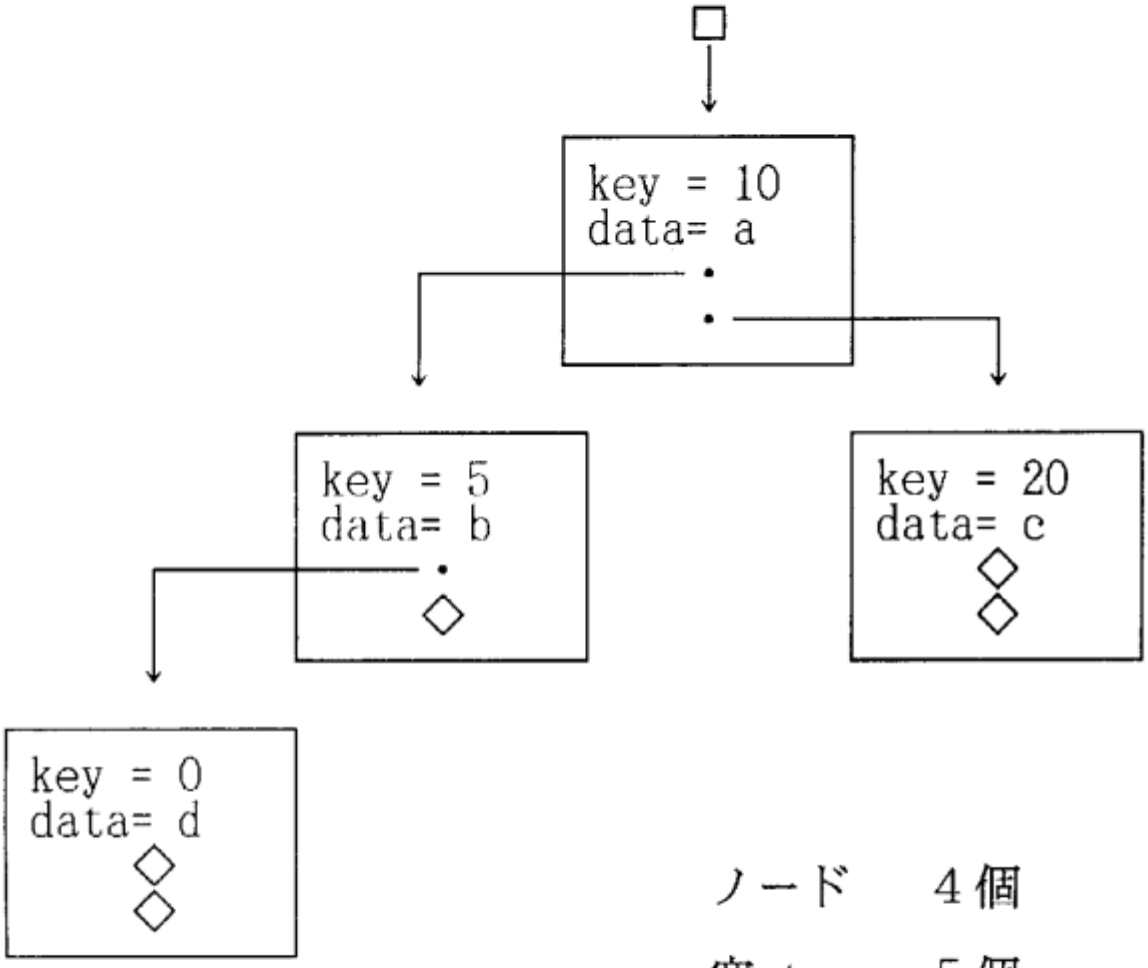
スロット data，left，right はムダ！

② 空-tree を別扱いしなければならない．

⇒　性質の異なるものは 別クラスにする．

クラス empty_tree と binary_tree3

→　特別処理がなくなる．
（空-tree）



ノード　4個

空-tree　5個

```
class binary_tree2 has
 instance
  attribute    key := '$empty',
               data,  left,  right;
  :add(Obj, Key, Data) :-
        :is_empty(Obj), !,
        set_values(Key, Data, Obj),
        create_branch(Obj);
  :add(Obj, Key, Data) :- !,
        insert(Obj!key, Key, Data, Obj);
  :get_contents(Obj, List) :- !,
        :get(Obj, List, []);
  :get(Obj, List, List) :-
        :is_empty(Obj), !;
  :get(Obj, List, List0) :- !,
        :get(Obj!left, List, List1),
       List1 = [{Obj!key,Obj!data} | List2],
        :get(Obj!right, List2, List0);
  :is_empty(Obj) :- !, Obj!key = '$empty';
end.
```

⇒ binary_tree3

オブジェクト版の改良　（続き）

1．　空-tree を別クラスにする．

2．　？　？　？

この先は 練習問題です．

★　少し工夫が必要

［ ヒント ］　ルート（root）は別クラスにする．

これは，ユーザ・インタフェイスを司る．

〈 出力機能付き二進木 〉

- クラス binary_tree2 を継承
  :add/3, :get_contents/2 を取り込む

- :map_tree は :get/3 と同様の tree-walk

- :refute(Object, Method, Args) - メタ述語
  例 1. :refute(Tree, output, {IO})
  = :output(Tree, IO)
  2. :refute(Tree, add, {7,f})
  = :add(Tree, 7, f)

- 書式付き出力
  :putf(IO, "フォーマット", 〔変数リスト〕)

⇒ binary_tree オブジェクト自身が出力機能を持つ.
しかし, 出力先は知らなくて良い.

```
%    IO は標準I/O オブジェクト

class  binary_tree  has

    nature binary_tree2;

 instance

    :map_tree(Obj, Method, Args) :-
        :is_empty(Obj), !;

    :map_tree(Obj, Method, Args) :- !,
        :map_tree(Obj!left, Method, Args),
        :refute(Obj, Method, Args),
        :map_tree(Obj!right, Method, Args);

    :output(Obj,IO) :- !,
        :putf(IO, "Key= \q, Data= \q\n",
                    [Obj!key,Obj!data] );
end.
```

〈 出力例 〉

ウインドウ

```
?- :create(#standard_io_window,
                [size(300,500)], Window),
   :activate(Window).

Window= $standard_io_window
?- :map_tree(B, output, {Window} ).
```

ファイル

```
?- :create(#standard_output_file,
                File, "btree.txt").

File= $standard_output_file
?- :map_tree(B, output, {File} ).

?- :close_output(File).
```

標準入出力

	ターム	キャラクタ
入力	:gett(IO, Term)	:getc(IO, Code)
出力	:putt(IO, Term)	:putc(IO, Code)

〈 書式付き出力 〉

:putf(IO, "フォーマット")
:putf(IO, "フォーマット", Arg_list)

\q ターム,　　\c 文字コード
\b 空白　　\n 改行

〈 クラス binary_tree2 〉

```
%    左右の枝を新しく作る

create_branch(Obj) :- !,
        :class_object(Obj, Class),    % 継承用
        :new(Class, Left),
        :new(Class, Right),          ⇔ #binary_tree2
        Obj!left  := Left,
        Obj!right := Right;
```

二つのプログラムの比較

キー，データ	スタック版	オブジェクト版
アトム	○	○
数値	○	○
変数	○	×
ターム	○	×
ストリング	○	○
オブジェクト	○	○

メモリ消費	スタックが伸びる	スタックを畳める
性質	サブルーチン的	オブジェクト指向

ＥＳＰプログラム

・クラス定義　　class　名前　has　end.

・マクロバンク定義　　macro_bank　名前　has　end.

・演算子宣言　　add_operator(演算子，タイプ，順位).

・include 文　　include("ファイル名").

これらを複数個並べたもの

マクロバンク定義

```
macro_bank　名前　has
    〔 nature 〈継承クラス並び〉; 〕
    {〈マクロ定義〉}
    〔 local 〔〈ローカル述語定義〉〕〕
end.
```

マクロ定義

```
〈パターン〉 =〉 〈展開形〉
        〔 when  〈生成ゴール列〉〕
        〔 where 〈検査ゴール列〉〕
        〔 :-    〈展開条件〉    〕 ";"
```

例えば，次のように展開される．

```
〈パターン〉 =〉 〈生成ゴール列〉，
                〈展開形〉，
                〈検査ゴール列〉
```

```
    %%  演算子宣言  %%
add_operator(++, fx,  600).
add_operator(@,  yfx, 600).

    %%  マクロバンク定義  %%
macro_bank  my_macro  has
  ++X =〉 X+1    :- integer(X), !;
  ++X =〉 Z      when increment(X,Z);
  Vect@N =〉 E   when vector_element(Vect,N,E);
end.

    %%  クラス定義  %%
%   class  class_name
%       with_macro  macro_bank_name  has
%       ...
%   end.
```

```
++X  =〉  Z  when  increment(X,Z);
```

```
M is ++N  =〉  unify(M, ++N)                    : システム定義
          =〉  increment(N,Z), unify(M,Z)
```

```
++X  =〉  X+1  :- integer(X), !;
```

↓

```
++X  =〉  Z  :- integer(X), !, increment(X,Z)        : システム定義
```

```
M is ++2  =〉  unify(M, ++2)
          =〉  unify(M, 3)
```

```
Vect@N  =〉  E  when  vector_element(Vect,N,E);
```

```
E = {a,b,c}@1  =〉  unify(E, {a,b,c}@1)
               =〉  vector_element({a,b,c}, 1, A),
                                  unify(E,A)
```

```
?- :new(#my_macro, Macro).
?- :expand_clause(Macro,
                  Head, Body, New_head, New_body).
```

展開前		展開後
(Head :- Body)	⇒	(New_head :- New_body)

例1.
```
p(X) :- f(++X);
  ⇒ p(X) :- increment(X,A), f(A);
```

例3.
```
p(X, ++X) :- f(X);
  ⇒ p(X,A) :- f(X), increment(X,A);
```

例6.
```
:- Z = {a,b,c}@(++X);
  ⇒ unify(Z, {a,b,c}@(++X));
  ⇒ increment(X,A), unify(Z, {a,b,c}@A);
  ⇒ increment(X,A),
      vector_element({a,b,c},A,B), unify(Z,B);
```

バインド・フック (bind_hook)

例： .., bind_hook(X, p(X,Y)), q(Z), X = a, ..

変数 X に値が入った時点で
呼出し "p(X,Y)" を実行する.

⇒ .., q(Z), X = a, p(X,Y), ..

```
% Lazy Prime Number Generator

class primes has
    :primes(_, Ps) :-
        naturals(2, Ns), sieve(Ns, Ps, Ps);
local
    naturals(N, [N|Ns]) :-
        bind_hook(Ns, naturals(N+1, Ns));
    sieve([N|Ns], Ps, [N|Pt]) :- prime(N, Ps), !,
        bind_hook(Pt, sieve(Ns, Ps, Pt));
    sieve([_|Ns], Ps, Pt) :- Pt \== [], !,
        sieve(Ns, Ps, Pt);
    sieve(_,_,_) :- !;
    prime(N, [P|_]) :- P*P > N, !;
    prime(N, [P|Ps]) :-
        N mod P =\= 0, prime(N, Ps);
end.
```

```
?- :primes(#primes, [X1,X2,X3] ).
 X1= 2, X2= 3, X3= 5

?- :primes(#primes, [Y1,Y2|T] ).
 Y1= 2, Y2= 3, T ?

?- T = [Y3,Y4|T1].
 Y3= 5, Y4= 7, T1 ?
```

〈 述語の意味 〉

:primes(_, ^Ps)　　-- 素数の生成
　　Ps: 素数のリスト

naturals(N, ^Ns)　　-- 自然数の生成
　　N:　自然数
　　Ns: N 以上の自然数のリスト

sieve(Ns, ^Ps, ?Pt) -- ふるい
　　Pt: まだ得られてない素数
　　つまり, Ps = [P1,P2,..,Pm| Pt]
　　(P1,P2,..,Pm: 既に得られた素数)

prime(N, Ps)　　-- 素数のチェック
　　N:　自然数
　　Ps: 素数のリスト

%% 〔X〕で呼出した時

```
:- :primes(#primes, 〔X〕)
:- naturals(2, Ns1), sieve(Ns1, 〔X〕, 〔X〕)
    clause : naturals(N, 〔N|Ns〕) :-
                bind_hook(Ns, naturals(N+1,Ns))
 →  Ns1= 〔2|Ns2〕,
       bind_hook(Ns2, naturals(3,Ns2))          ‥ hook1
:- hook1(Ns2), sieve(〔2|Ns2〕, 〔X〕, 〔X〕)
    clause-1 : sieve(〔N|Ns〕, Ps, 〔N|Pt〕) :- prime(N, Ps),
                        !, bind_hook(Pt, sieve(Ns,Ps,Pt))
 →  N= 2, Ns= Ns2, Ps=〔X〕=〔N|Pt〕=〔2|Pt〕  ∴ X= 2, Pt= 〔〕
      ⇒  Pt= 〔〕となり, bind_hook は即実行される.
:- hook1(Ns2), sieve(Ns2, 〔2〕, 〔〕)
    clause-1 : 第3引数の unify が失敗
    clause-2 : 第3引数 \== 〔〕が失敗
    clause-3 : 成功
:- hook1(Ns2), X= 2  で終了
```

⇒ 〔X|T〕で呼出した時

∵ Ps=〔X|T〕
=〔2|Pt〕
∴ Pt = T