TM-0134

ESP プログラミング

紺 田 茂 實 (三菱電機)
近 山 隆

September, 1985

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(Extended Self-contained Prolog)は、オブジェクト指向プログラミングを意図したPROLOGベースの論理型プログラミング言語であり、ICOTで開発の逐次型推論マシン(PSI) に適用する。

ＥＳＰは、PSI のオペレーティング・システムを含むシステムプログラムおよびユーザ・プログラムの記述が容易になるよう設計されたシステム記述言語である。

ＥＳＰで書かれたプログラムは、 PSIの機械語であるKL0 にコンパイルされる。
KL0 は拡張機能を有するPROLOG風の言語であり、種々の特徴は、そのままＥＳＰにも適用できる。

KL0 言語の特徴

- ○ コンフィクーション：基本的なパラメータ受渡し機構
- ○ バックトラッキング：基本的な制御構造
- ○ 豊富な組込述語：データ操作，入出力，システム制御

ＥＳＰ言語の主な特徴

- ○ オブジェクト指向型呼出し機構
- ○ スロット代入機構
- ○ クラスと継承機構
- ○ マクロ展開機構

本報告では、既存機種の手続向言語やオペレーティング・システムに一応の知識があり、これからＥＳＰでプログラムを作成しようと考えている人を対象ににしている。従って、ＥＳＰの概念を既存機種の知識のアナロジーとしての理解を主眼にしており、概念規定の厳密さで欠ける面があるので、前もって断わっておきたい。
更に知りたい方は、ＩＣＯＴ発行の他の研究論文(TR)，研究速報(IM)を参照して頂きたい。

ＥＳＰプログラミング

目　次

0 はじめに

本資料は、逐次型推論マシン（ＰＳＩ）のオペレーティング・システムおよびプログラミング・システムであるＳＩＭＰＯＳをプログラム開発・実行環境として、ＥＳＰでユーザ・プログラムを作成するためのプログラミング手引である。

本資料では、ＥＳＰプログラミングを始める上で基本となる考え方、ＥＳＰの言語の特徴およびプログラミング技法を例示・解説しているが、論理型プログラミング言語のＰＲＯＬＯＧについて入門程度の知識があるユーザを対象としている。

ＥＳＰ言語仕様では、ユーザ固有の演算子(Operator)，マクロ(Macro) の定義が可能であるが、通常のプログラミングは暗黙に使える標準のそれ等の範囲で充分であり、それ等の定義の言語仕様の説明は省略している。

尚、ＰＳＩのＫＬＯ組込述語やＳＩＭＰＯＳのウインドウ，ファイルなど入出力を利用するプログラムを作成する際には、下記の関連説明書類を参照願いたい。

関連説明書

ＬＳＰプログラミングについて

・ＫＬＯ組込述語説明書 ， MP7002-1

・ＳＩＭＰＯＳ使用説明書 ， MP3001-1

マシン操作について

・ＳＩＭＰＯＳ操作説明書 ， MP3002-1

・コンソール・プロセッサ(CSP) 操作説明書 ， MP7003-1

1. 概　　要(Abstracts)

1.1 言語の概要(Language summary)

(1) ESPの特徴(ESP characteristics)

ESP(Extended Self-contained Prolog)は、オブジェクト指向プログラミングを意図したPROLOGベースの論理型プログラミング言語である。

ESPは、逐次型推論マシン(PSI)のオペレーティング・システムを含むシステムプログラムおよびユーザ・プログラムの記述が容易になるよう設計されたシステム記述言語である。

ESPで書かれたプログラムは、PSIの機械語であるKL0にコンパイルされる。

KL0は拡張機能を有するPROLOG風の言語であり、種々の特徴は、そのままESPにも適用できる。

KL0言語の特徴

- ○ ユニフィケーション：基本的なパラメータ受渡し機構
- ○ バックトラッキング：基本的な制御構造
- ○ 豊富な組込述語：データ操作，入出力，システム制御

ESP言語の主な特徴

- ○ オブジェクト指向型呼出し機構
- ○ スロット代入機構
- ○ クラスと継承機構
- ○ マクロ展開機構

(2) 構文の表記(Syntax description)

言語構文の記述には、次の拡張を含めたBNFを採用している。

- ○ "X"は、ターミナル・シンボルXを示す。Xが2ヶ連続したダブルクォート(")ならそれは1ヶのダブルクォートを表わす。つまり""""は1ヶのダブルクォートを意味する。
- ○ {X}は、0回以上の任意の回数、Xの繰り返しを示す。
- ○ [X]は、Xまたは空を示す。つまり、Xは随意選択である。

１．２　オブジェクト(Object)

オブジェクト指向プログラミングは、複雑な知識に対する表現を極力単純な要素に分解して、型にはまった枠組みの中で属性などを表現し、状況に応じた変化はスロット(Slot)に保持させようとするフレーム(Frame) の考え方、また、ある状況の表現つまり概念を表現するセマンティック・ネットワーク(Semantic network)や、能動主体，作用対象，動作などを分析してフレームにおけるスロットに値を特定しようとするACTOR モデルなどに基づく方法論とその実践環境とを一体化した仕組みの中に位置付けられる。

オブジェクトとは、問題とその解法を出来るだけ率直に表現・記述するために導入された概念であり、実際の問題領域に登場するものが持つ機能や知識をモデル化した概念的実体(Conceptual object) であると言われている。この定義が最も良く言い現わしていると思われるが、従来の概念からではやはり馴染みにくいので、先ずオブジェクトの考え方から解説する。

実在するもの、架空のものなどを思考の中に捉えた観念上のものがオブジェクトであると考えればよい。架空のもの、例えば童話に出てくる鬼ヶ島の鬼はオブジェクトと考えうる。

実在し、且つソフトウェアの世界で例を挙げるなら、ディスプレィのスクリーン上に現れるウインドウ、ディスク上のファイルなどがオブジェクトである。ウインドウもファイルもソフトウェアが完成すれば、目に見えるか否かの違いはあるが共に実在するものになる。従って、ソフトウェア設計つまりソフトウェアを概念的に捉えようとする思考の中で認識しうるすべてのものがオブジェクトであると考えうる。

すべてのものをオブジェクトと捉える事による利点の一つは、オブジェクトに対してメッセージ（指令とパラメータ）を送りさえすれば、外部から操作しなくてもロボットのように、オブジェクトがメッセージに応えて主体的に動作するものと考えうる点にある。つまり、オブジェクトをそのように構成しようとする考え方がオブジェクト指向である。

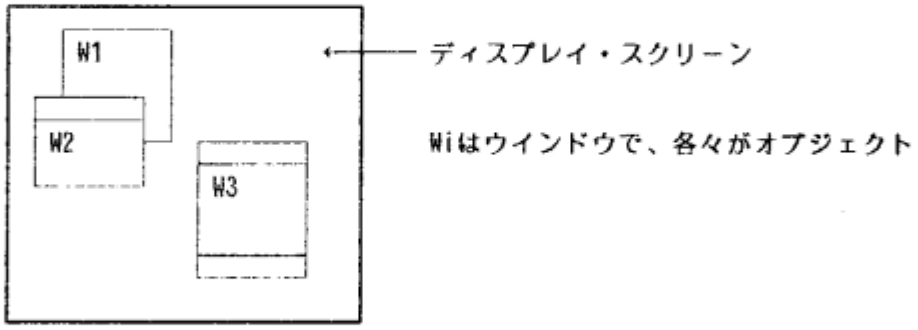


図　１－１　ウインドウ・オブジェクト

ＥＳＰでは、指令を与えればプログラムとして実行され、ある働きを成す主体を指してオブジェクトと呼んでいる。

従来の概念で考えるなら、抽象データ型(Abstract data type)が近く、その典型例のスタック(push,pop で操作されるStack)のイメージであり、次を挙げるなら可成り遠いがタスクに対応させれば理解し易いであろう。但し、オブジェクトは機能の他にオブジェクトの状態情報をオブジェクト内部のスロットに一体化して持つ事が出来る点に特徴がある。

例えば計算機システムのシステム・クロックつまりシステム時刻を管理するオブジェクトに着目すると、刻々と変る時刻を内部状態としてスロットに保持するオブジェクト，何かが開始された時刻をそのままスロットに保持するオブジェクトなどが考えられる。

従来のタスクでは、データベースなどに設けられシステム時刻のデータを更新するタスク，参照するタスクなどが独自に直接アクセスしており内部状態の概念は無い。

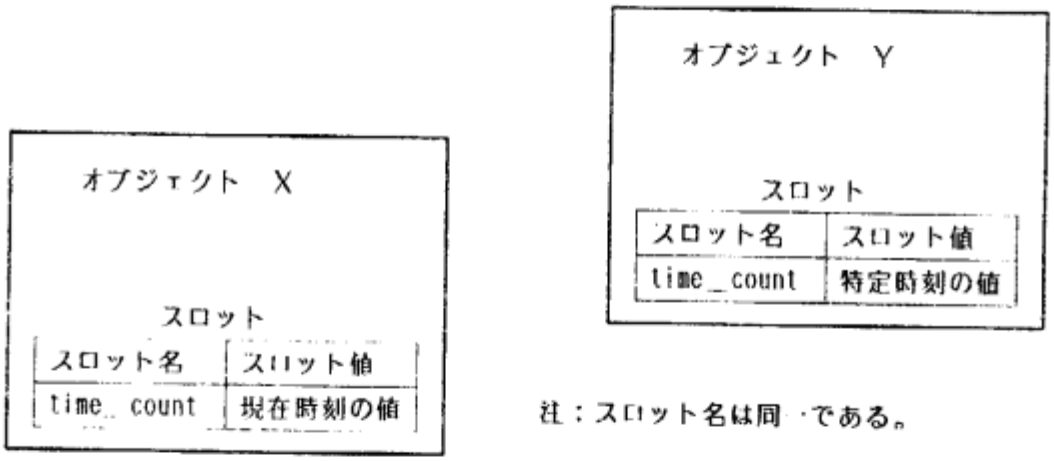


図　1－2　システム・クロックのオブジェクト

ＥＳＰでは、オブジェクトが論理の世界における一つの「公理系」を表している。同じ述語呼出しであっても異なる「公理系」で適用すれば異なった「意味」に取れる事になる。従って、使用する「公理系」つまりオブジェクトを特定出来る機構が設けてある。

オブジェクトのスロットには、スロット名，スロット値があり、スロット値はスロット名を指定する述語で参照される。つまりスロット値は「公理系」の一部を定義している事になる。またスロット値はスロット名を指定した述語により変更する事が出来るので、オブジェクトで表現している「公理系」の変更が可能となる。

１．３　オブジェクトとクラス(Object and class)

前述の例の様に、現在時刻か，特定の時刻かの違い、つまり内部状態が異なる（スロット値が異なる）だけの類似したオブジェクトをまとめてクラスと呼ぶ。

クラスはプログラムの記述単位であり、オブジェクトが有する特性の定義，概念規定が行われる。またクラスはコンパイル単位として扱われる。

クラス自身がまた「公理系」を表すオブジェクトであり、クラスに属するすべてのオブジェクトに共通する事項は個々のオブジェクトに任せずクラス自身のオブジェクトで処置する事が出来る。

クラスには、通常、複数のオブジェクトが属するが、オブジェクトに内部状態が無く、つまり類似したオブジェクトが他に無いなら、単独のオブジェクトでクラスとする事も可能である。

クラス自身のオブジェクトは、クラス・オブジェクトと呼び、クラスに属するオブジェクトをそのクラスのインスタンス(Instance)と呼んで識別する。単にオブジェクトと言う場合は、インスタンスを指していると思ってよい。

従来の概念で考えるなら、インスタンスはシングル・コピー・マルチ・タスクの各々に、クラスは唯一記述されるプログラムに対応させれば理解し易いであろう。但し、クラスはオブジェクトであり、要求があった時点でインスタンスを生成する点に特徴がある。

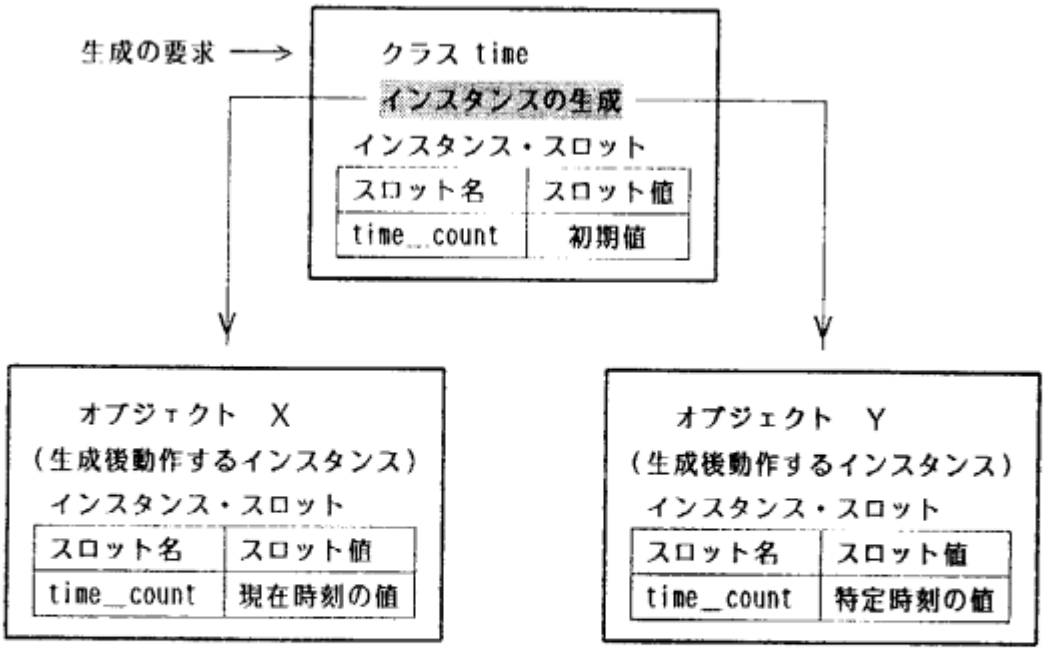


図　1－3　クラス time とそのインスタンス

１．４　クラスと継承(Class and inheritance)

あるクラスは、ある類似したオブジェクトを纏めており、また別のクラスでは、類似した別のオブジェクトを纏めている。

例えば、鬼ヶ島の青鬼，赤鬼の各々をオブジェクトとすれば、クラス　鬼ヶ島の鬼に属し、地獄の青鬼，赤鬼は、クラス　地獄の鬼に属していると考えればよい。

（１）　継承とは(Inheritance)

オブジェクトの構造に着目すると、オブジェクトとして捉えているもの（物・者・もの）が持っている特性のすべてを含んでいなければならない。

例えば、鬼ヶ島の鬼なら角が生え牙もあり、非情で狂暴な性質で、島の外まで暴れ回るであろう。また、地獄の鬼も同じ特性を持つが、但し仕処を離れて暴れ回る事はない。互いに共通点もあるが、また、違いもあるので別のクラスとしている。

この様に共通するものは、別のオブジェクトとして別のクラス例えばクラス　鬼を設け、クラス　鬼ヶ島の鬼，クラス　地獄の鬼がこのクラス　鬼を継承すれば個々の記述が省ける事になる。つまり、部品化と流用の実現である。

従来の概念で考えるなら、新規に作成するプログラムに既成のプログラムをリンケージ・エディタでつなぎ込んで利用しようとするイメージであるが等価ではない。リンケージ・エディトされた既成のプログラムは手続きで呼ばれる外部リブルーチンであるが、クラスの継承では、継承されたクラスは継承するクラスの構成要素である。

クラスを構成する際、オブジェクトに外部サブルーチンの呼び出しを利用させる事も可能であるが注意を要する。

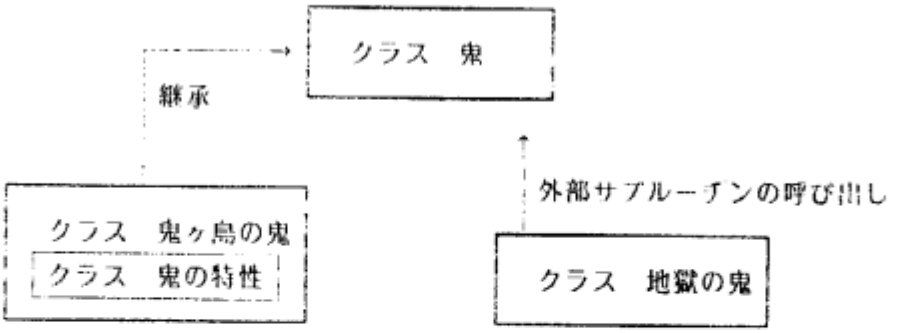


図　１－４　クラスの継承と呼出し

この図から、クラス　地獄の鬼がインスタンスを生成すると角も無くおとなしい大男が現れ、外部サブルーチンの呼び出し後に角が生えて狂暴に成り、クラス　鬼ヶ島の鬼がインスタンスを生成すれば、最初から角を生やした狂暴な鬼が現れる事になる。

（２）　補助クラス(MIXIN class)

クラスの継承はセマンティック・ネットワークのIS_A 階層に相当する。このIS_A により多重継承したクラスも可能であるが、中間に HAS_A 階層を実現する補助クラスを設け、そのクラスを継承する場合もある。ＥＳＰでは、あるクラスのスロットに別のクラスのインスタンス・オブジェクトを入れる事で HAS_A を実現する事が出来る。

例えば、角は、鬼のほか牛やユニコーン（架空の一角獣）にも在るので、クラス　角を設けるが、その直接継承は好ましくない。　次に、クラス　鬼に補助クラスを経由してクラス　角の特性を継承させる例で、直接継承を避けたい理由を示す。

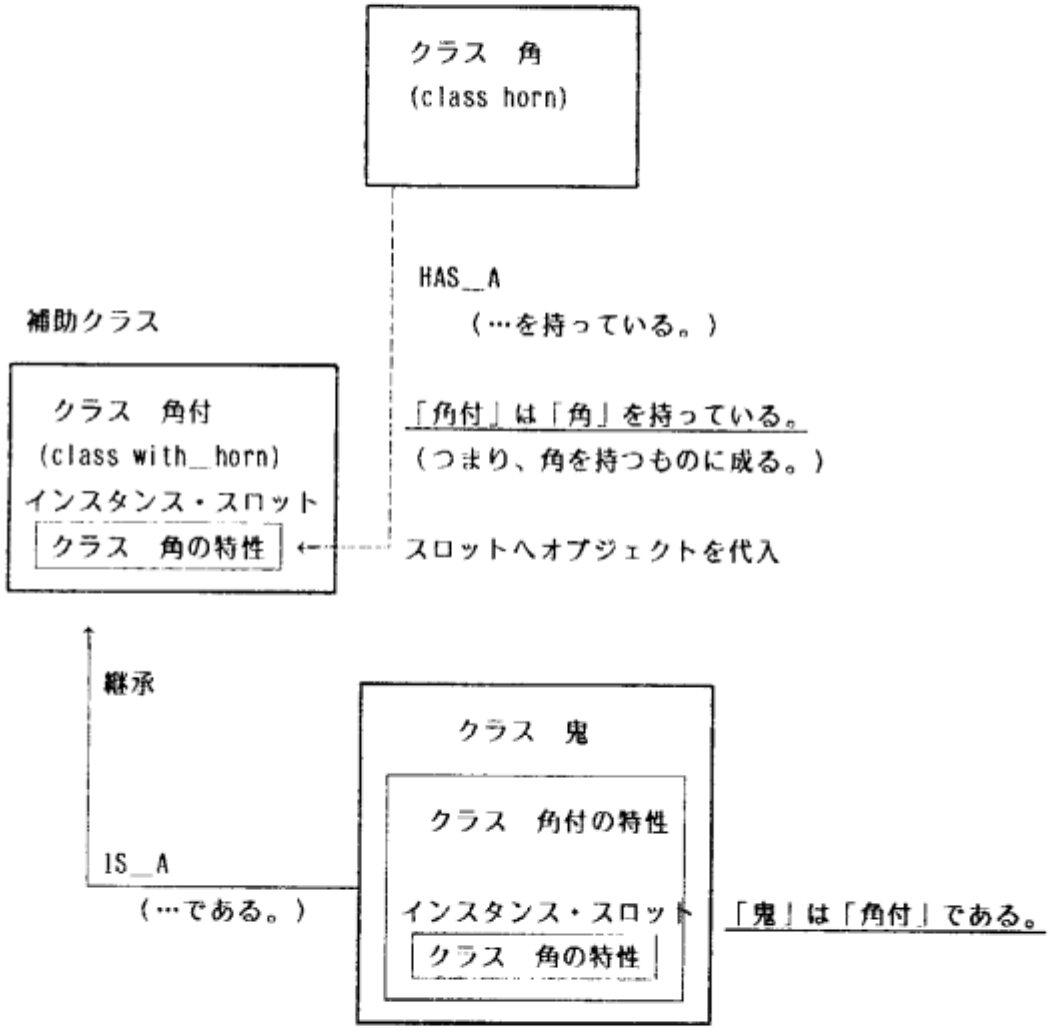


図　１－５　補助クラスの役割

補助クラス　角付を継承する事により、クラス　角を直接継承する場合に起きる「鬼」は「角」であるを避ける事が出来る。

2. 字句の構造 (Lexical structure)

ＥＳＰでプログラムを作成する際の、字句の構造について示す。

2．1 文字(Characters)

ＥＳＰのプログラムで使用出来る文字を次に示す。

SYNTAX

〈英小文字〉 ::=
"a" | "b" | "c" | "d" | "e" | "f" | "g"
| "h" | "i" | "j" | "k" | "l" | "m" | "n"
| "o" | "p" | "q" | "r" | "s" | "t" | "u"
| "v" | "w" | "x" | "y" | "z" | 〈漢字文字〉

〈英大文字〉 ::=
"A" | "B" | "C" | "D" | "E" | "F" | "G"
| "H" | "I" | "J" | "K" | "L" | "M" | "N"
| "O" | "P" | "Q" | "R" | "S" | "T" | "U"
| "V" | "W" | "X" | "Y" | "Z"

〈数字〉 ::=
"0" | "1" | "2" | "3" | "4"
| "5" | "6" | "7" | "8" | "9"

〈特殊文字〉 ::=
"!" | "@" | "#" | "$" | "%" | "&" | "*" | "+"
| "-" | "=" | "~" | "^" | "?" | "/" | "\" | "."
| "<" | ">"

〈書式用文字〉 ::=
" " | 〈改行文字〉 | 〈Tab code〉

〈区切り文字〉 ::=
"(" | ")" | "{" | "}" | "[" | "]"
| "," | ":" | """ | "'" | "|"

2．2　字句の要素(Lexical element)

ＦＳＰのプログラムは、名前(Name)，数(Number)，文字列(Character string)，変数(Variable)，区切り子(Delimiter) などの字句の要素で構成される。

SYNTAX

〈字句の要素〉 ::=
　　〈名前〉｜〈定数〉
　　｜〈文字列〉｜〈変数〉
　　｜〈区切り子〉

◇　字句の要素の例

```
icot        123        "abc"        X        ,
```

「書式用文字」やコメントは、隣り合った字句の要素を分ける。
任意の数の「書式用文字」やコメントは、特に断わらない限り、プログラムの意味を変えずに、字句の要素間に挿入出来る。

コメントは、パーセント文字(%) で始まり「改行文字」で終わる。コメントは「改行文字」以外の任意の文字で記述される。

◇　コメントの例

```
% This is a comment.
```

「書式用文字」は、後述する「クォーテッド・ネーム」と「文字列」を除く、その他の字句の要素内には許されない。なお、コメントもまた字句の要素内には許されない。

２．２．１　名前(Name)

ＥＳＰのプログラムでは、すべて名前でアクセスされる。

SYNTAX

〈名前〉::=
　　〈英小文字〉{〈英小文字〉|〈英大文字〉
　　|〈数字〉| "_" }
　|〈特殊文字〉{〈特殊文字〉}
　|〈クォーテッド・ネーム〉

〈クォーテッド・ネーム〉::=
　　"'"{〈ネーム・ストリング文字〉}"'"

〈ネーム・ストリング文字〉::=
　　〈アポストロフィ(')以外の文字〉|"''"

◇　名前の例

```
icot      psi      name_with_underscore      'A Qouted Name'
```

名前は、名前に使える文字から成ったネーム・ストリングと呼ばれる文字並びで出来ている。

クォーテッド・ネームでは、両端のアポストロフィ(')を除き、また内側で繰り返したそれ(')を１ヶとして扱う。

名前は、後述される述語名(3.5参照),クラス名(4.2参照),スロット名(4.6.1参照)及び名前データ(6.2.1参照)の記述に使用される。

２．２．２　定数(Number)

整数(Integer),浮動小数点数(Floating-point number) の２タイプがある。

SYNTAX

〈定数〉::= 〈整数〉 | 〈浮動小数点数〉

〈整数〉::= 〈数字の並び〉

〈浮動小数点数〉::= 〈数字の並び〉"." 〈数字の並び〉
[〈指数部〉]

〈数字の並び〉::= 〈数字〉 {〈数字〉}

〈指数部〉::= "^" ["+" | "-"] 〈数字の並び〉

◇　定数の例

1985　　3.14159265　　51.2^8

字句の要素は、正の定数のみである。
負の定数は、前置演算子(Prefix operator) の”－”と正の定数で構成する複合構造(Complex structure) で現す事が出来る。

２．２．３　文字列(Character string)

SYNTAX

〈文字列〉::= """ {〈ストリング文字〉} """

〈ストリング文字〉::= 〈ダブルクォート以外の文字〉 | """" ""

◇　文字列の例

"This is a string"　　"ABC" is a string."

２．２．４　変数(Variable)

SYNTAX

〈変数〉::= 〈先頭文字〉
　　{〈継続文字〉}

〈先頭文字〉::= 〈英大文字〉| "_"

〈継続文字〉::= 〈英小文字〉|〈英大文字〉
　　|〈数字〉| "_"

◇　変数の例

X　　_　　R2D2　　_a_variable

文字並びが同一の変数は、ある範囲（詳細は４章以降）例えばクローズ定義内では、同一の論理変数である。ただし下線の文字（_）のみで表す変数は、いかなる時にも独自の論理変数であり、匿名変数(Anonymous variable)と呼ばれる。

２．２．５　区切り子(Delimiter)

SYNTAX

〈区切り子〉::= 〈区切り文字〉

◇　区切り子の例

icot_,_psi　　_"_This is string"_

区切りは１文字でなされ、区切り文字は、構文の構造を規定する際に特定され、区切り箇所に応じて使い分けられる。

3．　構文の構造 (Syntactical structure)

ＥＳＰのプログラムは、タームで表す一つ以上の定義からなっている。タームは構文規則に則った字句の要素の一つ以上で構成される。

ＥＳＰの構文規則は、ベクタ(Vector)，ストリング，複合タームや線形のリスト構造などの特殊な表記を伴った、演算子優先型の文法である。

3．1　ターム(Term)

SYNTAX

〈ターム〉::=
　　〈アトミック・リテラル〉|〈変数〉
　　|〈ベクタ〉|〈ストリング〉
　　|〈複合ターム〉|〈演算子適用〉|〈リスト〉
　　|〈クラス・オブジェクト〉
　　|"("〈ターム〉")"

◇　タームの例

icot　　R2D2　　{a,b,c}　　"a string"　　f(X,Y)

X+3　　[a,f(X),3]　　#class_name　　(3*X+Y)

3．2　アトミック・リテラル(Atomic literal)

SYNTAX

〈アトミック・リテラル〉::=
　　〈名前〉|〈定数〉

◇　アトミック・リテラルの例

icot　　1985　　3.14159

アトミック・リテラルは、それ自身の素関数子(Principal functor) と呼ばれる事があり、この場合の引数の個数(Arity) は０である。

３．３　ベクタ(Vector)

SYNTAX

〈ベクタ〉::= 〈空ベクタ〉 | 〈空でないベクタ〉

〈空ベクタ〉::= "{}"

〈空でないベクタ〉::= "{" 〈ターム・リスト〉"}"

〈ターム・リスト〉::= 〈ターム〉{ "," 〈ターム〉 }

◇　ベクタの例

{}　　{a,b,c}　　{f,X,Y}

空でないベクタの各要素は、ターム・リストでの各タームである。各要素は、０から初めて、左から右へと、要素数－１までの数で指される。

空でないベクタの最初の要素(index 0) は、ベクタの素要素(Principal element) と呼ばれる。それが名前の時は、ベクタの素関数子(Principal functor) と呼ばれる事がある。

素要素が名前の時は、複合タームの形式を意味している。この場合の要素数－１の数は、ベクタの引数の個数(Arity) と呼ばれる。

3．4　ストリング(String)

１ビット，８ビット，１６ビットの整数を要素とする３種類のストリングがある。それぞれビット・ストリング，バイト・ストリング，ダブルバイト・ストリングと呼ばれる。

ダブルバイト・ストリングは、各文字を整数値(JISコード) で示す事で「文字列」の形式を保持する。従って、「文字列」(2.2.3参照) はダブルバイト・ストリングの略記表現とも言える。

SYNTAX

〈ストリング〉::= 〈文字列〉
　　| 〈ストリング・タイプ指定子〉
　　":" 〈ストリング要素〉

〈ストリング・タイプ指定子〉::=
　　"bits" | "bytes" | "double_bytes"

〈ストリング要素〉::= "{}" | "{" 〈整数リスト〉 "}"

〈整数リスト〉::= 〈整数〉{"," 〈整数〉}

◇　ストリングの例

```
"a string"          bits:{1,0,0,1,1}
```

ストリング要素の表現に使える整数の値を次に示す。

① ビット・ストリング：0 か 1
② バイト・ストリンプ：0 から 255
③ ダブルバイト・ストリング：0 から 65535

これらの整数は、左から右へ、0から要素数－１までの数で指されるストリングの要素である。

3．5　複合ターム(Compound term)

SYNTAX

〈複合ターム〉::= 〈関数子〉"(" 〈アーギュメント・リスト〉")"

〈関数子〉::= 〈名前〉

〈アーギュメント・リスト〉::=
　　〈アーギュメント〉{","〈アーギュメント〉}

〈アーギュメント〉::= 〈ターム〉

◇　複合タームの例

f(X,Y)　　g(M,N+3)

複合タームは、素要素が名前であるベクタの略記表現である。ベクタの長さは、複合タームのアーギュメント数＋１である。その素要素(index 0)は、関数子の名前で、それに続く要素(index 1以降) は複合タームのアーギュメントである。
index 1 の要素は第一アーギュメント，index 2 は第二アーギュメントの様に続く。
例えば、"f(X,Y)"の形の複合タームは、素要素が"f" で引数の数が２のベクタ　"{f,X,Y}" と解釈される。

複合タームは通常、後述されるクローズ・ヘッド(4.5.1参照),述語呼出し(4.5.2参照) の記述に使用され、その関数子の名前は述語名と呼ばれる。

３．６　演算子適用(Operator application)

SYNTAX

〈演算子適用〉::= 〈前置演算子の適用〉
　　| 〈後置演算子の適用〉
　　| 〈中置演算子の適用〉

〈前置演算子の適用〉::= 〈前置演算子〉〈ターム〉

〈後置演算子の適用〉::= 〈ターム〉〈後置演算子〉

〈中置演算子の適用〉::= 〈ターム〉〈中置演算子〉〈ターム〉

〈前置演算子〉::= "class" | "instance" | "local" | "before"
　　| "after" | "nature" | "component" | "attribute" | ":-"
　　| "?-" | "mode" | "public" | "~" | "cspy" | "\+" | "spy"
　　| "nospy" | "+" | "-" | "!" | ":" | "$"

〈後置演算子〉::= ":"

〈中置演算子〉::= "has" | "instance" | "local" | ";" | ":-"
　　| "-->" | "=>" | "where" | "when" | "->" | "',' " | "="
　　| "is" | "=.." | "==" | "\==" | "@<" | "@>" | "@=<"
　　| "@>=" | "=:=" | "=\=" | "<" | ">" | "=<" | ">="
　　| ":=" | "+" | "-" | "/\" | "\/" | "xor"
　　| "*" | "/" | "<<" | ">>" | "mod" | "div"
　　| "^" | "!" | ":" | "$"

ここに定義の各演算子は、優先順位の下から順に並んでいる。

◇　演算子適用の例

f(X) ; g(Y)　　X+3　　-15.3　　p(X) :- q(X),r(X)　　Y = -X

演算子適用もまた、ベクタの略記表現である。前置または後置演算子の適用は、その演算子を素要素(index 0)とする２要素のベクタを表している。

中置演算子の適用は、その演算子を素要素とする３要素のベクタを表しており、左側のタームは第二要素(index 1)，右側のタームは第三要素(index 2)である。

演算子適用のp(X):- q(X),r(X)は、ベクタ{(:-),p(X),{(,),p(X),r(X)}} の略記表現である。このベクタ表現の素要素の中置演算子":-"は、次に続く要素間の「区切り子」としての"," を中置演算子と見立てた優先順位より低いので、四則演算の優先順位の処置，(X+Y)/Z と同様に括弧で括る必要がある。また、第三要素（内蔵されたベクタ）の先頭の"," も同様の処置である。

ベクタの各要素の複合タームによる略記表現をベクタの表現に直せば、p(X):- q(X),r(X)は、三重のベクタ{(:-),{p,X},{(,),{q,X},{r,X}}}の略記表現である事が判る。

注：ＳＰでは演算子のユーザ定義も演算子定義(Operator definition) により可能であるが、本書では省略する。

３．７　リスト(list)

SYNTAX

〈リスト〉::= 〈空リスト〉｜〈空でないリスト〉

〈空リスト〉::= "[]"

〈空でないリスト〉::=
"[" 〈ターム・リスト〉["|" 〈ターム〉] "]"

〈ターム・リスト〉::= 〈ターム〉{ "," 〈ターム〉}

◇　リストの例

[]　　[a,f(X),3]　　[a,b,c | d]　　[Head | Tail]

リストもまた、ベクタの略記表現である。リスト" [X | Y] "はベクタ " { Y,X } " を現すが、要素の逆転に注意を要する。

リスト" [a,b,c | d] "は、ベクタ "{{{d,c},b},a}"の略記表現である。

4．　クラスの構造 (Class structure)

クラスは、類似したオブジェクトのグループに共通する性質を定義する。ここで、2つのオブジェクトがスロット値のみ異なっている時、類似していると言う。

クラスに属するオブジェクトは、クラスのインスタンス(Instance)と呼ばれる。

ここでは、クラスの構造の定義つまりプログラム構造の規定について説明する。

4．1　クラス・オブジェクト(Class object)

クラス自身もオブジェクトであり、通常のインスタンス・オブジェクトと区別するため、クラス・オブジェクトと呼ばれる。

クラス・オブジェクトはクラス名を用いた定数(Constant)で記述する事が出来る。クラス・オブジェクトは、明示的なリテラルの表現であり、また直接アクセス可能な唯一の静的なオブジェクトである。一方のインスタンス・オブジェクトは、常に動的に生成され、ある変数に結合(bind)されたり、他のオブジェクトのスロットに代入(1.4(2) 参照) される事もある。

SYNTAX

〈クラス・オブジェクト〉::= "#" 〈クラス名〉

〈クラス名〉::= 〈名前〉

◇　クラス・オブジェクトの例

```
#window        #with_horn
```

4．2　クラス定義(Class definitions)

クラス自身のスロットと述語ならびにクラスのインスタンスのスロットと述語の定義等、ＥＳＰの実行可能プログラムがこのクラス定義により記述される。

SYNTAX

〈クラス定義〉::=
　　"class" 〈クラス名〉
　　[〈マクロ・パンク宣言〉]
　　"has"
　　[〈継承定義〉";"]
　　[〈クラス・スロット定義〉";"]
　　[〈クラス・クローズ定義〉";"]
　　["instance"
　　{ 〈インスタンス・スロット定義〉";" }
　　{ 〈インスタンス・クローズ定義〉";" }]
　　["local"
　　{ 〈ローカル・クローズ定義〉";" }]
　　"end" "."

◇　クラス定義の例

例題 1　Listの正の数字の並びを整数値に変換してInteger に返すプログラムである。例えば[1,9,8,5] を変換して1985を返す。

```
class     decoder   has
:decode(Class,List,Integer):- decode(List,0,Integer);   ---- ①
local
decode([],N,N):- !;                                     ---- ②
decode([ D | Ln],No,I):- decode(Ln,No*10+D,I);
end.
```

類似するオブジェクトは無く（内部状態を持たないオブジェクトであり従ってスロットも使用しない）、クラス・オブジェクトで動作する。①はクラス・クローズ定義であり、②はローカル・クローズ定義である。

マクロ・バンク宣言は、クラス定義の中で展開すべきマクロを指定する。

継承定義は、クラスで継承する他のクラスを指定する。

スロット定義及びクローズ定義は、クラスのテンプレート(template)を定義する。クラスのテンプレートは、そのクラスを継承する他のクラスへ、継承されるべきスロット，述語を定義している。

4．3　マクロ・バンク宣言(Macro bank declaration)

SYNTAX

〈マクロ・バンク宣言〉::= "with_macro"
　　〈マクロ・バンク名〉{ "," 〈マクロ・バンク名〉}
〈マクロ・バンク名〉::= 〈名前〉

マクロ・バンク宣言で指定するマクロ・バンク内のユーザ定義のマクロが使用出来る。通常は、暗黙に使える標準マクロの範囲で使用する。

4．4　継承定義(Nature definition)

継承定義は、クラスで継承するクラス類の範囲，順序を定義する。

SYNTAX

〈継承定義〉::=
　　"nature"
　　〈スーパ・クラス名〉{ "," 〈スーパ・クラス名〉}

〈スーパ・クラス名〉::= 〈クラス名〉 | "*"

継承されるクラス類の範囲は、継承定義で指定されるクラスが継承するクラスのすべてを含んでいる。

継承されるクラス類の順序は、継承定義で最初に現れたクラスが継承するクラスが、次に現れるクラスより、先に継承される。

2度継承されるクラスがあれば、最初の継承のみ有効で残りは無視される。

スーパ・クラス名に"*" を指定すると、この定義を行っているクラス自身が継承されるクラスとなる。

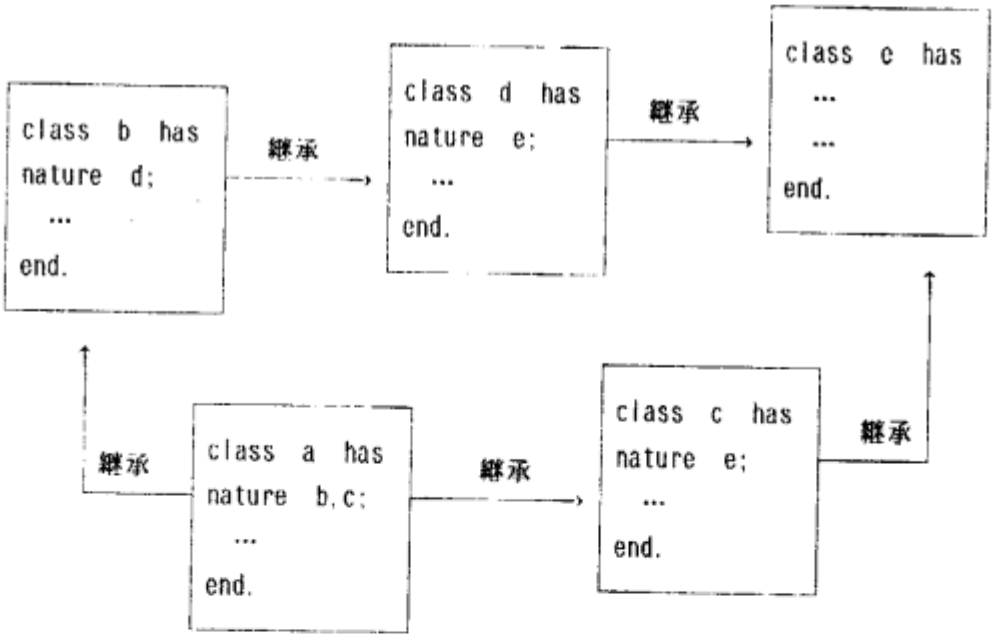
◇　継承定義の例

クラス　鬼(1.4 (2)の例) がクラス　角付を継承する。

```
class     demon     has
nature    with _horn;
  ...
  ...
end.
```

クラス dが他のクラス bに継承されており、またクラス aが継承定義でクラス bを指定しているとすれば、クラス dはクラス aに継承される事になる。

継承されるクラスの定義は、継承するクラスの定義の以前に与えておかなければならない。これは継承のループ化の防止であり、クラス aがクラス bを継承すればクラス bは直接．間接にもクラス aを継承出来ない。



クラス aの継承の順序は b→ d→ e→ cとなる。

図　4－1　継承関係と順序

注：スーパ・クラスを親クラス、継承するクラスを子供クラスと呼ぶ事がある。

４．５　クローズ，述語，メソッド(Clauses,Predicates and Methods)

クローズはクラス定義内のクローズ定義で定義される。
クローズ・タイプ（ 4.5.1に後述）が同じで関数子，引数の数が同じであるヘッドのクローズの組が述語を構成する。
述語は、ローカル述語の定義と、また継承したクラスで定義される述語と共に、メソッドを定義する。

４．５．１　クローズ(Clauses)

SYNTAX

〈クラス・クローズ定義〉::= 〈メソッド・クローズ定義〉

〈インスタンス・クローズ定義〉::= 〈メソッド・クローズ定義〉

〈ローカル・クローズ定義〉::= 〈クローズ定義〉

〈メソッド・クローズ定義〉::=
　　[〈デモン・タイプ〉] ":"〈クローズ定義〉

〈クローズ定義〉::= 〈ヘッド〉[":-"〈ボディ〉]

〈デモン・タイプ〉::= "before" | "after"

〈ヘッド〉::= 〈ターム〉

〈ボディ〉::= 〈ゴール列〉
　　| "(" 〈ゴール列〉{ ";" 〈ゴール列〉} ")"

〈ゴール列〉::= 〈ゴール〉{ "," 〈ゴール〉}

〈ゴール〉::= 〈メソッド呼出し〉 | 〈述語呼出し〉

メソッド呼出し，述語呼出しは 4.5.2以降に後述される。
ヘッド及びゴールに使うタームは複合タームに限られ、その素関数子(Principal f

unctor) は述語名である。即ち、"p" や"f(X,Y)"の様なタームがヘッド，ボディに使え、"1985"，"[a,b,c]" や({a}) は使用出来ない。

クローズは、クラス・クローズ(4.2の例題 1の①)，インスタンス・クローズ及びローカル・クローズ(4.2の例題 1の②) の３種である。
クラス・クローズは、クラス自身の特性を記述するクラス・メソッドを定義する。
インスタンス・クローズは、クラスのインスタンスの特性を記述するインスタンス・メソッドを定義する。
ローカル・クローズは、ローカル述語を定義する。

メソッドを定義するクローズには、次の３っのタイプがある。
クローズ・タイプ識別子"before"を先頭につけたbefore demonクローズ，"after"を先頭につけた after demonクローズ，クローズ・タイプ識別子の着かないprimaryクローズとである。

◇ インスタンス・クローズの例

例題 2　第２引数の整数を先頭に符号付の文字(JISコード) の並びに変換し、リストで第３引数に返すプログラムである。

```
class     encoder   has
instance
component   sign ;   --------  ①
:encode(Object,Integer,[Object!sign | Li]):- Object!sign:= #"+",  ②
          encode(Object,Integer,1,[],Li);
local
encode (Object,I,N,Li,Lj):- I<0,Object!sign:= #"-",  ---------  ③
          encode(Object,-I,N,Li,Lj);
encode(Object,  ,0,L,L):- !;
encode(Object,I,_,Lo,Li):- I/10=Q,I mod 10=D,
          encode(Object,Q,Q,[D+16#"2330" | Lo],Li);
end.
```

②がインスタンス・クローズであり、下線の部分①，③は、それぞれスロットの定義と操作であるが 4.6で後述される。

４．５．２　述語(Predicates)

述語は、ＥＳＰの実行可能な形式のプログラムであり、クローズ・タイプが同じで且つ素関数子，引数の数が同じヘッドを持ったクローズの組で構成される。

クラス・クローズから成る述語はクラス述語、インスタンス・クローズから成る述語はインスタンス述語と呼ばれ、素関数子，引数の数が同じヘッドを持ったクローズでも、クローズ・タイプによりbefore demon述語，primary述語，after demon述語の３種の定義が可能である。

クラス述語とインスタンス述語は、クラスのテンプレート部であり、継承定義をしていれば、その継承されるクラスのテンプレートからそれらの述語を集めてメソッドを構成する。従ってクラス述語とインスタンス述語は、メソッド呼出し(Method call)でのみ呼出す事が出来る。

ローカル・クローズは、ローカル述語を構成する。次に示す述語呼出しで，同じクラス内のローカル述語を直接呼出す事が出来る。但し、他のクラスのローカル述語を呼出す事は出来ない。

（１）　述語呼出し(Predicate calls)

SYNTAX

〈述語呼出し〉::= 〈ターム〉

◇　述語呼び出しの例

```
decode(list,0,Integer)    ---------- (4.2の例題 1)

encode(Object,Integer,1,[],Li)    ---- (4.5.1の例題 2)
```

４．５．３　メソッド(Method)

（１）　デモン結合(Demon combination)

メソッドは、クラス・メソッド，インスタンス・メソッドに拘らず、メソッド名，引数の数が同じであれば、継承したクラスのインスタンス述語なども含み、同一メソッドである。

メソッドは、次の３種のデモン結合で構成される。

① 継承の順序に従い、継承されるクラスで定義されているbefore demon述語すべての呼出しの AND結合。

② 継承の順序に従い、継承されるクラスで定義されているprimary 述語すべての呼出しのOR結合。

③ 継承の順序の逆順に、継承されるクラスで定義されているafter demon 述語すべての呼び出しの AND結合。

同じ引数がデモン結合の述語のすべてに渡されるなら、種々の処置が可能となる。例えば、before demon述語である引数の値を決め、primary 述語にその情報を渡す事も可能である。また、primary 述語に複数の解があるとき、after demon 述語で１ヶに限定する事も出来る。

（２）　メソッド呼出し(Method calls)

SYNTAX

〈メソッド呼出し〉::=
　　〈通常メソッド呼出し〉
　　|〈クラス・メソッド呼出し〉
　　|〈インスタンス・メソッド呼出し〉

〈通常メソッド呼出し〉::= ":" 〈ターム〉

〈クラス・メソッド呼出し〉::=〈クラス名〉":" 〈ターム〉

〈インスタンス・メソッド呼出し〉::= ":" 〈クラス名〉":" 〈ターム〉

述語呼出しが同じクラス内のローカル述語を対象としているのに対して、メソッド呼び出しは、主に他のクラス（継承するか否かに拘らず）内のメソッドを対象としている点に注意を要する。

◇　メソッド呼出しの例

(a) 通常メソッド呼出し

```
:refresh(Window)                :decode(#decoder,list,Integer) ---- ①
```

(b) クラス・メソッド呼出し

```
basic_window:create(Window_class, New_window)
```

(c) インスタンス・メソッド呼出し

```
basic_window:refresh(Window)
```

通常メソッド呼出しの①では、 4.2の例題 1のクラス・メソッドが呼出される。

通常メソッド呼出しでは、実際に呼ばれるメソッドは呼出しの第１引数により動的に決定される。この第１引数は、クラス・オブジェクトまたは、あるクラスのインスタンス・オブジェクトでなければならない。実行時の呼出し時点でオブジェクトであればよく、直前にオブジェクトとなる変数であっても良い。

通常メソッド呼出しの第１引数がクラス・オブジェクトなら、そのクラス・メソッドが呼ばれ、またあるクラスのインスタンスなら、そのクラスのインスタンス・メソッドが呼ばれる。

クラス・メソッド呼出しでは、明示的に特定したクラス(上例の basic_window) のクラス・メソッド(:create(W,N)) が呼出される。このメソッド呼出しの第１引数はクラス・オブジェクトでなければならない。また、呼びし側のクラスは、クラス・メソッドが呼出されるクラス(basic_window) を継承していなければならない。

インスタンス・メソッド呼出しでは、明示的に特定したクラス(上例の basic_window) のインスタンス・メソッド(:refresh(W))が呼出される。このメソッド呼出しの第１引数はインスタンス・オブジェクトでなければならない。また、呼びし側のクラスは、インスタンス・メソッドが呼出されるクラス(basic_window) を継承していなければならない。インスタンス・オブジェクトは、動的に生成されるものであり、実行の直前にオブジェクトとなる変数で表す。

いずれの場合も、呼出されるメソッドは、メソッド名，引数の個数が呼出し側と一致するメソッドのみである。

（３）　メソッド呼出しの利用法(Usage of method calls)

３種のメソッド呼出しの利用法を次に示す。

あるクラスのクラス定義を行うと、継承定義で指定したクラス類からクラス述語，インスタンス述語を集めて、このクラスのメソッドが出来（ 4.5.3(1) 参照）、これらのメソッドが他のクラスから呼出されて動作する事になる。

自クラスを含めて４っのクラスからクラス・メソッド（インスタンス・メソッドの場合も同様であり省略）を構成する例を次図に示す。

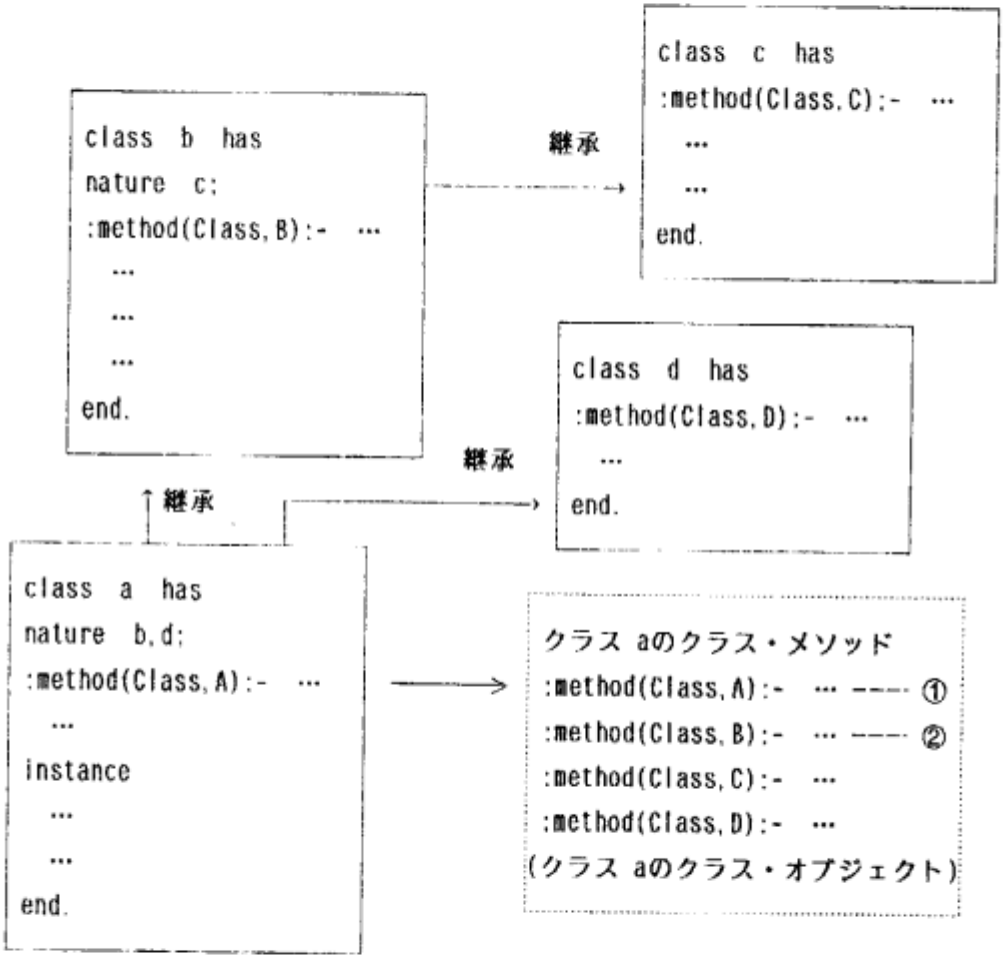


図　４－２　メソッドの構成

メソッド名，引数の個数が同じでも、異なるクラスのクラス述語であれば、異なる働きを持っていると考えられるが、通常メソッド呼出し(:method($a,X))では常に先頭の①(:method(Class,A)) が呼出されて実行される。その実行が何にかの理由で失敗(fail)しない限り②以降が呼出される事はない。従って、②以降を呼出す手段がクラス・メソッド呼出しであり、b:method($a,Y)で②を呼出す事が出来る。

４．６　スロット (Slots)

スロットは、クラス・オブジェクト，インスタンス・オブジェクトの内部状態を保持する入れもの（容器）の様なものであり、それぞれクラス・スロット，インスタンス・スロットと呼ばれる。スロットはスロット名で識別され、スロット名毎に必ずスロット値を持ち、論理変数の未定義に相当する状態は無い。

同一スロットが種々の条件に応じて変化する値をスロット値として保持する事が出来る。つまり、スロットに既に値が保持されていても、従来型言語の変数と同様に、スロット名を指定した代入のプログラムにより更新が可能である。この代入の動作は、決定的であり、論理変数の様にバックトラックで元の状態に戻る事は起き得ない。

また、プログラム内ではスロット名を指定して定数（スロット値）相当に扱う事が出来る。

４．６．１　スロット定義(Slot definition)

SYNTAX

〈クラス・スロット定義〉::= 〈スロット定義〉

〈インスタンス・スロット定義〉::= 〈スロット定義〉

〈スロット定義〉::= 〈スロット・タイプ〉〈スロット定義項目〉
　　{ "," 〈スロット定義項目〉}

〈スロット・タイプ〉::= "attribute" | "component"

〈スロット定義項目〉::= 〈スロット名類〉［〈スロット初期化〉］
　　| "(" 〈スロット名類〉［〈スロット初期化〉］":-"
　　〈スロット初期化コード〉")"

〈スロット名類〉::= 〈スロット名〉
　　| "(" 〈スロット名〉{ "," 〈スロット名〉} ")"

〈スロット初期化〉::= ":="〈ターム〉| "is"〈クラス名〉

〈スロット初期化コード〉::= 〈ゴール列〉

スロットには、クラス自身に属するクラス・スロットとクラスのインスタンスに属するインスタンス・スロットの２種がある。

また、スロット・タイプとしてクラス・スロット，インスタンス・スロット夫々に、プログラムの何処からでもアクセス出来るattributeスロットと同じクラス内からのみアクセス出来るcomponentスロットの２形態がある。

attribute スロット，component スロットは、それぞれ従来型言語に於けるCOMMON変数，ローカルな変数に相当すると思えばよい。

クラス・オブジェクトは、クラス自身のクラス定義内のスロット定義で指定したスロットを含み、継承定義で指定したクラスのテンプレートに定義されている次のスロットを持つ事になる。

① すべての componentスロット

② ユニークな attributeスロットのすべて（同一スロット名があれば、継承順序の最初の１ヶを選び、残りは無視）

スロット定義では、同じスロット名を componentスロットと attributeスロットの両方に与えてはならない。

各スロットのスロット値は、「スロット初期化」で特定される値に初期化される。尚、「スロット初期化コード」が定義されていれば、その初期化以前に実行される。「スロット初期化」，「スロット初期化コード」などが省略されている場合、スロット値は"0" に初期化される。

この初期化の動作は、各クラスに対して暗黙に定義されるクラス述語 "new"の after demonで行われる。

クラス・スロットの「スロット初期化コード」は、KLO 組込述語のみ許される。

「スロット初期化コード」が ":=〈ターム〉" (4.5.1の例題 2の①参照) なら、そのタームが初期値である。

「スロット初期化コード」が "is〈クラス名〉" の形式は、インスタンス・スロットにのみに適用が許され、特定されるクラスのインスタンス・オブジェクトが初期値となる。

◇ スロット初期化の例

```
some _slot:=10
 other_slot:= "Character string"          ----------
(other _slot:= String:- String="Character string") ---- 等価である。
 another_slot is some  class  _name
```

４．６．２　スロットのアクセス・メソッド(Slot access method)

"get_slot","set_slot" の両メソッドがクラス・スロット，インスタンス・スロットのアクセス用に、クラス・メソッド，インスタンス・メソッドとして、夫々次の様に暗黙に定義される。

(1) クラス・メソッド
:get_slot(Class, class_slot_name, Value)
:set_slot(Class, class_slot_name, Value)

(2) インスタンス・メソッド
:get_slot(Object,instance_slot_name, Value)
:set_slot(Object,instance_slot_name, Value)

これ等のメソッドの第１引数は、 4.5.3(2) に示した様に、クラス・オブジェクトまたはインスタンス・オブジェクトでなければならない。第２引数はスロット名であり、第３引数は、スロット値に対応する。

"get_slot" はスロット値と第３引数とのユニファイを行い、"set_slot" は第３引数をスロットに代入する。

component スロットに対するこれ等のメソッドは、同じクラス定義内でのみ使用可能である。

尚、通常のスロット・アクセスでは、標準マクロによる次の略記表現を利用すればよい。

◇　通常のスロット・アクセスの例

①　"get_slot"
#class_name!slot_name = X　　変数 Xとスロット値がユニファイされる。
Y =Object!instance_slot_name　　変数 Yとスロット値がユニファイされる。
注：変数が未定義なら、実行後スロット値を得る。

②　"set_slot"
#class_name!slot_name:= A　　スロット値が変数 Aの値となる。
Object!instance_slot_name:= B　スロット値が変数 Bの値となる。
注：「スロット初期化コード」が ":=〈ターム〉" の形式に同じ。

5. マ ク ロ (Macro)

マクロ定義(Macro definition)により、ユーザ固有のマクロの定義も可能であるが通常は、暗黙に使える標準マクロの範囲を利用する。従ってマクロ定義の仕様は省略する。

5.1 標準マクロ(Standard macro)

標準マクロとは、演算子適用(3.5参照) によるスロット・アクセス，定数，算術演算，ビットの論理操作などの略記表現である。

［ＳＰのプログラム中に現れる特定のパターンに対しては、マクロ展開機能が働き、自動的にあらかじめ定められたパターンに展開される。

◇ 標準マクロの展開例

Object!Slot:= Value　－＞　:set__slot(Object,Slot,Value)　(4.6.2参照)

X+Y　－＞　KL0 組込述語のadd(X,Y,M)への展開とX+Y をM に置換える。
注： Mは、クローズ内でユニークな変数名となる様自動的に設定される。

X-Y=Z　－＞　KL0 組込述語のsubtract(X,Y,Z) への展開。

標準マクロとしては、以下のようなものが定義されている。

（１） 定数値の表記のためのマクロ

底#"文字並び"

文字並びを底で指定した基数に基づく整数値に変換した「整数」に展開する。底としては 2から36までが許される。底が10を越える時、英文字がA,B,C の順に文字として使用可能となる。

◇ ［例］ 16#"A000" －＞ 40960

#"文字"

文字の文字コード(JISコード) の整数値に変換した「整数」に展開する。

◇ ［例］ #"A" －＞ 9025

control#"文字"
meta#"文字"
control_meta#"文字"

文字のキーと，コントロール・キー，メタ・キー，その両者を同時に押した場合にキーボードから入力されるコードの整数値に変換した「整数」に展開される。

key#名前

名前で指定される特殊なキーの入力に対応するコードの整数値に変換した「整数」に展開される。名前は以下のいずれかである。

abort, help, bell, bs, cr, del, esc,
lf, tab, up, down, left, right

keypad#"文字"

キーボード右端のキーパッドの入力に対応するコードに変換した「整数」に展開される。文字としては、0から9までの数字，"."，"."，"−"である。

ascii#"文字並び"

ASCII コードによる「文字列」を表わす。展開結果は8ビットの「ストリング」になる。

string#"文字並び"

単に「文字列」で書いた場合と同様、JIS コードによる16ビットの「ストリング」になる。このマクロは開発用のクロスシステムとの整合性を保つために実機上でも定義されているもので、通常実機上で用いる必要はない。

mouse#クリック

マウスのクリック入力に対応するアトムを表す。クリックとしては、l，ll，m，mm，r，rrが、それぞれ左一回、左二回等に対応しており、アトムとしては'mouse#l'等になる。

（2） 算術演算

2引数の算術演算マクロ・オペレータとしては以下のものが用意されている。

+, −, *, /, mod	加算，減算，乗算，除算，剰余
<<, >>	左シフト，右シフト
/\, \/	ビットごとの論理積，論理和

１引数の算術演算マクロ・オペレータとしては以下のものが用意されている。

~ ビットごとの反転

（３） ＥＳＰ実行機構に関わるマクロ

ＥＳＰプログラムの実行のメカニズムに直接かかわるマクロとして以下のようなものがある。

＃クラス名

クラス名で指定したクラスのクラス・オブジェクトの表記に用いる。

：メソッド名（引数並び…）

ＥＳＰのメソッド呼び出しに展開される。

クラス名：メソッド名（引数並び…）

クラス名で指定したクラスのクラス・メソッドの呼び出しに展開される。子供クラスによってオーバーライドされた親クラスのクラス・メソッドを呼び出す場合に用いる。

：クラス名：メソッド名（引数並び…）

クラス名で指定したクラスのインスタンス・メソッドの呼び出しに展開される。子供クラスによってオーバーライドされた親クラスのインスタンス・メソッドを呼び出す場合に用いる。

（４） マクロ展開の抑制

マクロ展開は原則として自動的に行われてしまうが、特別な記法を用いると、この展開を抑制することができる。

‘‘（ターム）

ターム中のマクロ展開はすべて抑制され、タームそのものになる。

‘（ターム）

ターム中のマクロ展開のうちトップレベルの展開のみが抑制される。より深いレベルに現れるマクロは展開の対象となる。

5．2　マクロ展開(MACRO expansion)

マクロ展開とは、コンパイルの際プログラム中のマクロをマクロ毎に定められた述語呼出しなどの形式、つまり実行可能なプログラムに変換する事を言う。

マクロの書かれている位置がクローズのヘッドかボディによりマクロ展開される位置が異なる。

（1）　ボディのマクロ展開

ボディのタームにマクロの利用を認識すると、直ちにマクロ展開をそのタームの位置で行う。そのタームのすべてのマクロ展開を終え、そのタームに未だ実体が残っていればマクロ展開の後に AND結合で続けた後、次のタームへと進む処理を繰り返す。

◇　ボディのマクロ展開の例

(a)
```
decode([ D | Ln],No,I):-  decode(Ln,No*10+D,I);        (4.2の例題 1)
 ↓                                  ↓
decode(L,No,I):- cons(D,Ln,L),multiply(No,10,M),add(M,D,N),
                                  decode(Ln,N,I);
```

(b) 上例と同じ機能のクローズでマクロ展開後にタームの実体が残らない例

```
decode(L,No,I):- cons(D,Ln,L),No*10+D=Nn,decode(Ln,Nn,I);
 ↓                                  ↓
decode(L,No,I):- cons(D,Ln,L),multiply(No,10,M),add(M,D,Nn),
                                  decode(Ln,Nn,I);
```

（２）　ヘッドのマクロ展開

ヘッドのターム内でマクロの利用を認識すると、ボディの最後のタームの後、つまりクローズの区切り子(.) の前にマクロ展開する。

◇　ヘッドのマクロ展開の例

(a) 第１引数，第２引数の差分を第３引数で返す例

```
defference(X,Y,X-Y):- X>Y,!;   ------ ①  ( X>Y はボディのマクロ)
defference(X,Y,Y-X);   ---------- ②
      ↓
defference(X,Y,M):- less_than(Y,X),!,subtract(X,Y,M);   ------ ③
defference(X,Y,M):- subtract(Y,X,M);   ---------- ④
```

注：①が③に、②が④にマクロ展開される。

6. プログラミング技法(Proguramming technique)

ＦＳＰプログラミング技法を、従来型言語に於ける技法と対比しながら、紹介する。従って、手続き的側面が主となるが、ＦＳＰプログラミング技法の例題ではクローズ単位の論理的側面も取り挙げる事とし、分りやすさを優先に次ぎの範囲としている。

① 部分的な例題

単純化のためローカル述語で示す。（メソッドとローカル述語では、他のクラスから呼出されるか同じクラス内から呼出されるかにより、クローズ・ヘッドの記述形式の違いの外に、パラメータの妥当性チェックなどボディの記述に差異がある。）

② クラス単位の例題

簡単な機能のクラス・オブジェクトで示す。（複雑な多重継承のクラスは省略）

③ 例題内のメソッド利用

例題では、SIMPOSが提供するメソッドの一部を、その都度メソッドの解説をして利用する。

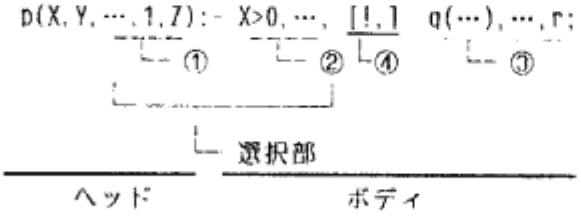
④ 引数を入出力双方向性(Reversible)の記述では複雑になるので、いずれかに限定した引数を用いた述語で記述している。

6.1 プログラムの構造

ＦＳＰプログラムでは、ＰＳＩのパターン・マッチング，ユニフィケイション，バックトラックなど KLOのマシン・レベルで、述語の選択，変数の値定義，変数の値設定の解除・再試行など非決定的なプログラムの制御が行われる為、プログラムの制御構造が目に見える形で現れ難い。

先ず、プログラムの実体と成るクローズの特徴と記述の要点から始め、次に従来型言語に於けるプログラムの３つの基本構造つまり、順次処理，選択岐処理，繰返し処理に対応したＦＳＰプログラムとしての述語記述の要点を示す。

クローズは 4.5.1の定義に基づき記述されるが、記述の注意事項をクローズの一般形式で示す。



①：一致条件
②：比較条件のゴール列
③：実行本体のゴール列
④：選択岐のカット

選択部には、当該クローズの実行本体のゴール列を実行すべきか否かの判定条件を記述する。従って述語呼出し側の変数の値により選択部が成立するクローズのみ、その実行本体のゴール列が実行される事になる。

選択部が「空」なら、ヘッドの一致（述語名と引き数の個数が同じ、且つユニファイ成立）により、実行本体のゴール列が無条件に実行される。

述語名と引き数の個数が同じで、選択部，実行本体のゴール列を異にする複数のクローズの組により述語を構成して、意図するプログラムの機能を実現する。従って、述語呼出し側の変数の値により述語のいづれが実行されるかは未定の状態であり、その各クローズは選択岐の構造である。

例えば、述語呼出し側が…,p(A,B, …,N,C),q(C), …で述語p を呼出し、また変数の値により決まるいづれか１つのみを選択して実行するものとする。

```
p(X,Y,…,1,Z):- X>0,… ,[!,]X*Y …=Z;
p(X,Y,…,2,Z):- Y ＼==0,[!,]X/Y …=Z;
p(X,0,… ,__,Z):- X*X …=Z;
```

変数A が正で、変数N が1 ならば、述語p の１行目が実行される。このクローズの"!,"（カット）が省略されていたなら、 2,3行目のクローズが選択岐として残存のまま呼出し側へ戻り、次の述語呼出しq(C)以降が何かの理由で失敗(fail)すれば、バックトラックして述語p の３行目のクローズが実行されるので、意図に反する事になる。従って、条件成立の直後に"!,"を入れて残りの選択岐を削除する必要がある。

クローズが選択岐の残存なく終了する場合、そのクローズは決定的に終了したと呼ばれ、バックトラックによる無意味な再試行の防止，メモリー使用の効率化に有効である。

６．１．１　順次処理

選択肢処理，繰返し処理を含まない単純な処理の AND結合（「区切り子」が ","）を順次に実行する。

従来型言語では、代入文の並びが代表例である。

ＣＳＰでは、クローズ・ボディのタームが失敗(fail)すれば、そのクローズも失敗してバックトラックが起こり、他のクローズと共に選択肢処理，繰返し処理の振舞いをする可能性がある。従って、従来型言語での代入文の並びに似た記述（次の例題 3）であっても、タームの失敗が起こり得るので注意を要する。

◇　順次処理の例

例題 3　第１引数，第２引数の２っの数の和と積を求めて、それぞれ第３引数，第４引数に返す述語とする。

```
calcurate(X,Y,Sum,Mul):- X+Y=Sum,X*Y=Mul;
```

変数 Sumまたは Mulが何らかの理由により定義済で、且つ演算結果と不一致なら失敗する。

例題 4　３引数の変数を各辺とする三角形の成否を判定する述語とする。

```
triangle(A,B,C):- A>0,B>0,C>0,A+B>C,B+C>A,A+C>B;
```

triangle(3,4,5) で呼出せば成功し、triangle(1,2,4) で呼出せばボディの４番目のタームで失敗(fail)する。

６．１．２　選択岐処理

ある条件に対応する処理（実行本体のゴール列）を選択して実行する。

従来型言語では、一分岐のIF文（(IF THEN ELSE も含む），複数分岐のCASE文(COMPUTED GOTOも含む）などがある。

ＥＳＰでは、IF文，CASE文相当を共に、(a) IF文に相当する判定条件のクローズのOR並列による述語，(b) クローズ・ボディで判定条件ごとの処理のOR結合記述，(c) 前記の(a),(b) の併用、で実現する事が出来る。

注：クローズ・ボディでのOR結合は、左括弧"("，右括弧")" の間に、１つの選択岐処理毎に「区切り子」";" で別けて記述しなければならない。尚、括弧内でのカット"!" は括弧外には影響しない。

◇　選択岐処理の例

例題 5　第１から第３引数の変数を３辺とする三角形の形状を判定し、正三角形なら３，二等辺三角形なら２，その他は１を第４引数に返す述語とする。但し、既に 6.1.1の例題 4のtriangle(A,B,C) で成立した３つの変数で呼出すものとする。

述語呼出し側 … ,triangle(A,B,C),shape(A,B,C,Result),… 但し、A,B,C は整数値

(a) IF文に相当する判定条件のクローズのOR並列による述語の解答例

```
shape( N, N, N,3):- !;          % 正三角形
shape( N, N,__,2):- !;          % 二等辺三角形
shape( N,  , N,2):- !;          % 二等辺三角形
shape(__ , N, N,2):- !;         % 二等辺三角形
shape(__ ,__ ,__,1);            % その他の三角形
```

注：一致条件ならば、クローズ・ヘッドのみで処置が可能である。

(b) クローズ・ボディで判定条件ごとの処理のOR結合記述の解答例

```
shape(A,B,C,R):- (A=:=B,B=:=C,!,R=3;                          % 正三角形
                  A=:=B,!,R=2;B=:=C,!,R=2;A=:=C,!,R=2;         % 二等辺三角形
                  R=1);                                        % その他の三角形
```

注：演算子"=:=" は、整数に対する等価の判定を行う。

この(a),(b) は当然同じ機能であるが、プログラムの読み易さや、前述のマシン・レベルのサポートを考慮すれば、(a) の記述が望ましい。尚(a) は実行速度の点でも(b) より約25% 速い。

6．1．3 繰返し処理

ＩＳＰでは、再帰呼出し(Recusive call) またはバックトラックの種々の応用により実現する。先ず、再帰呼出し，バックトラックの応用法に関する一般形式を示す。

◆ 再帰呼出し

```
p(…,I,I):- !;   または   p(…,S,T):- S>I,… ,!;      % 終了判定
   └①                          └②
p(…,S,T):- q(…),…,p( …,S+1,I) [,r(…)] ;      % 繰返し処理
   └③              └④          └⑤
```

①：終了判定の一致条件　　③，⑤：繰返し処理のゴール列

②：終了判定の比較条件　　④：再帰呼出し

引数を更新した再帰呼出し④が１行目の述語p で成立するまで、２行目の述語p が繰返される。

再帰呼出しの後に記述する⑤が「空」ならば、つまり再帰呼出し④が終端となる場合、終端再帰(tail recursion)呼出しと呼び、メモリー使用の効率化に有効である。

◆ バックトラック

1) 述語呼出し側が終了条件をTargetに与えて、p(…,Target)で呼び出す場合。

```
p(…,Z):- ……,Z>=…,!;          % 繰返し処理のゴール列①が終了判定②
          └─ ① └─ ②              の成立まで失敗しながら繰返す。
p(…,Z):- p(…,Z);               % 常に残存する選択肢
```

１行目のクローズが失敗するのでゴール列の内、副作用のあるターム（例えばスロットに代入など）以外はすべてUNDO（ユニファイ解除など）され、処理が無効と成る。従って、副作用のないタームは最終に成立した時の処理のみが有効と成る。

2) 述語呼出し側が終了条件をTargetに，他の引数に終了条件以外を与えて、例えば、p(…,not_target,Target)で呼出す場合。

```
p(…,Z,Z):- !;  または  p(…,Z,Y):- Z>Y,…,!;     % 終了判定の成立まで
                                                  失敗を繰返す。
p(…  ,  ,Y):- ……  ,Z-…  ,p(…,Z,Y);            % 常に残存する選択肢で繰り
               └─ ①                               返し処理のゴール列①
```

繰返しのゴール列は残存する選択肢側にあり、上の1)より引数が１つ増えるが、副作用のないタームもUNDOされずに繰り返す事が出来る。（後述の例題08参照）

繰返し処理には、(a) 所定の回数までの繰返し，(b) 期待値を得るまでの繰返し，(c) 終り（無くなる）までの繰返し、の３種がある。

(a) 所定の回数までの繰返し

積んだカードを上から順に１枚取り、Ｎ枚取るまで繰返す動作と言える。（Ｎ枚になったら終了）

従来型言語では、DO文,FOR文が相当する。

ＥＳＰでは、再帰呼出し(Recursive call)の先頭の述語でＮに到達を判定する。特に終端再帰(tail recursion)による実現が望ましい。

(b) 期待値を得るまでの繰返し

積んだカードを上から順に１枚取り、スペードのエースを引くまで繰返す動作と言える。（枚数と無関係に終了）

従来型言語では、通常IF文で判定し分岐で繰り返させる。

ＥＳＰでは、終了条件のタームを含むクローズが成功すれば終了し、そのクローズの失敗(fail)により起こるバックトラックの繰返しが望ましい。

(c) 終り（無くなる）までの繰返し

積んだカードを上から順に１枚取り、無くなるまで繰返す動作と言える。（取れなくなったら終了）

従来型言語では、IF文で「空」を判定し分岐で繰返させる。

ＥＳＰでは、(a) 同様、再帰呼出し(Recursive call)の先頭の述語で「空」を判定する。特に終端再帰による実現が望ましい。

◇ (a) 所定の回数までの繰返しの例

例題 6　第１引数の変数 Nの階乗N!(=1*2*3*…*N) を求め第２引数に返す述語とする。

述語呼出し側　　… ,factorial(N,Result),　　但し、N は整数値

解答例①

```
factorial(N,R):- fact(N,1,R);                  % 初期値の設定
fact(0,R,R):- !;                               % 繰返しの終了判定
fact(N,No,R):- fact(N-1,N*No,R);               % Tail recursion
```

変数 Nの減算により N回の再帰呼出しを行う。また、算法はN!=1*N*(N-1)*(N-2)* …*3*2*1である。終了判定の成立時、算出結果を第３引数にユニファイ。

解答例②

```
factorial(N,R):- fact(1,N+1,1,R);              % 初期値の設定
fact(N,N,R,R):- !;                             % 繰返しの終了判定
fact(N,M,No,R):- fact(N+1,M,N*No,R);           % Tail recursion
```

１からの加算により N回の再帰呼出しを行う。また、算法はN!=1*1*2*3* …*(N-1)*Nである。終了判定の成立時、算出結果を第４引数にユニファイ。

解答例③

```
factorial(0,1):- !;                              % 繰返しの終了判定
factorial(N,R):- factorial(N-1,Ri),R=N*Ri;       % Left recursive call
```

変数 Nの減算により N-1回の再帰呼出しを行う。また、算法はN!=1*2*3* … *(N-1)*Nである。繰返し回数が解答例①, ②より 1回少なく、算法も計算式通りであるが、再帰呼出しの後に計算処理が来るためスタック領域が延びたままに成るので、複雑な計算処理の際は注意を要する。

注：この例題は、N!=1*2*3*4* … *N , N!=1*N*(N-1)*(N-2)* … *2*1 のように、どちらからでも算出可能な数少ない例の１つである。通常は、いずれか一方のみが適用可能の場合が多いので、この例題に惑わされない事。

例題 7　Fibonacci 数列の N番目を求める述語とする。尚、Fibonacci 数列は、N=0 から 1,1,2,3,5,8,13,21, …と前項，前々項の和で構成されている。

```
述語呼出し側        … ,fibonacci(N,Result),          但し、N>0
```

解答例①

```
fibonacci(N,R):- fibo(N,R,_,_);      % 述語の置換え
fibo(1,1,0,1):- !;                   % 繰返しの終了判定
fibo(N,Rm+Rn,N-1,Rm):- fibo(N-1,Rm,Mn,Rn);
```

変数 Nの減算により N-1回の再帰呼出しを行う。前項，前々項の値を同時に求めている。

解答例②

```
fibonacci(0,1):- !;                          % N-2 の時のみ使用する終了判定
fibonacci(1,1):- !;                          % 繰返しの終了判定
fibonacci(N,R1+R2):- fibonacci(N-1,R1),fibonacci(N-2,R2);
```

変数 Nの減算により再帰呼出しを行うが、前項，前々項の値を独立の再帰呼出しで求める為、 N番目のfibonacci 数-1回の再帰呼出しを行う。例えば、N=17なら2583回の再帰呼出しが行われる。

注：この例題の解答例は、共に終端再帰呼出しの様に見えるが、ヘッドにマクロ記述が在るので、終端再帰呼出しではない。(5.2(2) を参照)

◇ (b) 期待値を得るまでの繰返しの例

例題 8　第１引数で指定するウインドウから１文字づつのキー入力を、"CR"を読むまで繰返し、入力文字の並びをリストで第２引数に返す述語とする。尚、第１引数はウインドウのインスタンスである。
述語呼出し側　　… ,from_keyin(W,List),

```
from_keyin(W,L):- keyin(W,_,key#cr,[],[ H | L ]);
keyin(W,C,C,L,L):- !;  ------ ②  └──────────────── ①：終了条件
keyin(W,_,C,Lo,L):- :read(W,Ch),keyin(W,Ch,C,[ Ch | Lo ],L); ---- ③
```

:read(W,Ch) は、ウインドウのインスタンス・メソッドであり、ウインドウ・オブジェクト(W) がキーボード操作の１文字を入力して第２引数のChに返す。
keyin の②は終了条件に一致するまで失敗(fail)して、keyin の③へ進み再帰呼出しで②から繰返す。（前述のバックトラックの一般形式の2)に相当）
リストには、キーボード操作の順に右から左に並んでいる。例えば "1","2","3","4","5","cr"とキーボード操作するなら、リストは[5,4,3,2,1] となる。

◇ (c) 終り（無くなる）までの繰返しの例

例題 9　第１引数のリストの各要素の順を逆転したリストにして第２引数に返す述語とする。
述語呼出し側　　… ,reverse(lo,ln),

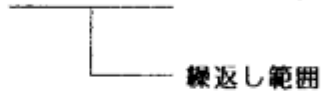
```
reverse(Lo,List):- reverse(Lo,[],List);          % 初期値の設定
reverse([],L,L):- !;                             % 繰返しの終了判定
reverse([H | T],Li,L):- reverse(T,[ H | Li],L); % Tail recursion
```

入力側のリストが「空」(Empty) に成るまで繰返す。[5,4,3,2,1] は [1,2,3,4,5]に代わる。尚、 4.2の例のdecode, 4.5.1の例のencode共に、この技法で記述している。
注：述語名は同じreverse であるが、引数の個数が異なるので 1行目と2,3 行目は別の述語である。

◇　従来型言語のDO文,FOR文，流の繰返し技法の例

例題10　I=1,2,3,…Max まで何かの処理を繰返す。

述語呼出し側　　… ,for(1,I,Max), … ,… ,… ,endfor(I,Max),




```
for(M,M,M):- !;  ---------------- ①
for(I,I,M);     ---------------- ②
for(Io,I,M):- for(Io+1,I,M); ---- ③
endfor(M,M);    ---------------- ④
```

for の②は常に成功し、endforの④が失敗(fail)するとバックトラックして、③へ進み再帰呼出しで②が成功して、繰返し範囲が実行される。

呼出し側で変数名を変え（例えば I→J ）同じ forを、別の処理の繰返しに何度でも利用出来る汎用性がこの技法にはある。

但し、繰返し範囲のタームが述語呼出しで残存する選択岐(Alternative) を持ちながら残りをカット(Cut) 出来ない（してはいけない）場合には、この技法が使えないので注意を要する。

また、この繰返しはendforの④により終了するが、①は終了後の残存する選択岐である③をカットするためのクローズである。（呼出し側のendforの直後で"!" すれば①は不要となるが、呼出し側のfor の上流のすべての残存する選択岐がカットされる事になる。）

６．２　データの記述

オブジェクト指向プログラミングのＥＳＰでは、従来型言語の様に、READ,WRITE文，INPUT,PRINT 文を使ってシステムの外部との入出力を直接を行う必要はない。つまり、入出力を司るSIMPOSのオブジェクトにメッセージを送れば（メソッド呼出しで実現）、入出力動作が実施される。従って、プログラムを作成する際、SIMPOSのオブジェクトと受渡するデータ，つまりプログラム内部で使用するデータについて考慮すればよい。

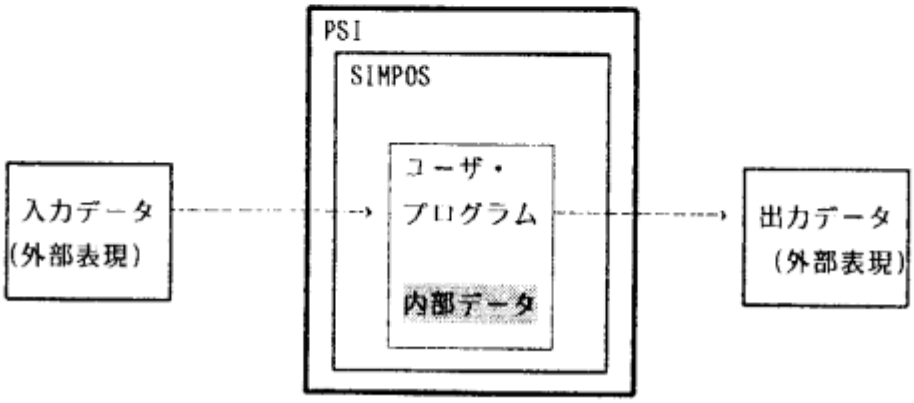


図　６－１　データの範囲

データは、一般にプログラムで参照・操作出来る情報のすべてを指している。

「ＥＳＰでプログラムを作成する場合も、プログラム内でプログラムとしての述語名，各タームなど述語自身を操作する事が可能であるが、ここでは、最初からデータとして扱う様に意図している通常の使用範囲に限定する。また、その様なデータの記述要領（ＥＳＰ表記）とデータの操作法を解説する。

データは、それが持つ意味の類別および、操作単位の違いにより、幾つかのデータ・タイプに分類される。先ず、操作単位の分類から示す。

(a)　アトミック・リテラル

１つの記述内容それ自身が１つの意味を表わす字句である。つまり単独のデータである。従って、その記述内容を分割した操作は出来ない。これに属するデータ・タイプは「名前」，「整数」，「浮動小数点数」である。

(b)　構造体

１つの記述内容それ自身が１つの意味を表わすタームであると共に、個々に意味を持つターム，字句などの集りとして、個々にに操作する事が可能である。つまり、複数のデータを各々要素とする構造体データと言う事が出来る。これに属するデータ・

タイプは「ベクタ」，「ストリング」であり、2.と3.で述べた略記表現がある。

(C) オブジェクト

オブジェクトは、データと見做して操作出来るので一種のデータ・タイプである。例えば、複数インスタンスを構造体データの要素とし、それを任意に使い分け出来る。

6．2．1 「名前」データ

名前のデータは、 2.2.1に示した表記法で記述され、プログラム内部では、その名前を意味する内部データとして取扱う事が出来る。但し、飽くまでも内部データであり外部表現形式ではない。

◇ 「名前」のデータの使用例

例題11 第１から第３引数の変数を３辺とする三角形の形状を判定し、直角三角形なら"right_angled_triangle"，鋭角三角形なら"acute_angled_triangle"，鈍角三角形なら "obtuse_angled_triangle" を第４引き数に返す述語とする。但し、既に 6.1.1の例題 4のtriangle(A,B,C) で成立した３つの変数で呼出すものとする。

述語呼出し側 …,triangle(A,B,C),shape_check(A,B,C,Name),

```
shape_check(A,B,C,Name):- longest(A,B,C,X,Y,Z),
          shape(Name,X*X,Y*Y+Z*Z);
longest(A,B,C,A,B,C):- A>B,A>C,!;
longest(A,B,C,B,A,C):- B>A,B>C,!;
longest(A,B,C,C,B,A);
shape( right_angled_triangle,N,N):- !;
shape( acute_angled_triangle,N,M):- N<M,!;
shape(obtuse_angled_triangle ,_ ,_);
```

プログラム内部では、述語shape の第１引数に記述されている名前は記述のままの文字並びではなく、その文字並びを意味するアトム(Atom)番号で取扱われる。従って、外部表現形式の文字並びで操作を意図する場合は、SIMPOSのメソッド（simpos使用説明書のトランスデューサの項参照）によりアトムからストリング，リストなどの形式の名前（但し単語間は下線付のまま）に変換する事ができる。

6．2．2 「整数」データ

整数のデータは、2.2.2に示した表記法で記述され、プログラム内で利用する定数値を与える。プログラム内部では、整数値として加減乗除の演算に使用できる。単精度の整数は、-2147483648～2147483647（符号＋31ビット）の範囲である。

◇ 「整数」データの使用例

例題12 第１引数の正整数値の平方根を次の如く近似計算し、小数１桁目で四捨五入した整数値として第２引数に返す述語とする。但し第１引数が負の時は失敗(fail)する。

√N の整数近似値をXとすると、初期値X_0をある値に定め、
$X_{n+1} = (X_n + N/X_n)/2$を順次求めて$X_n \leqq X_{n+1}$に達した時、実数の√N に近い方を選びXとする。
述語呼出し側 … ,square_root(N,Result),

```
square_root(N,R):- N<0,!,fail;              % 負なら選択肢を削除し失敗
square_root(0,0):- !;                       % 0 なら0 を返す
square_root(N,R):-  initial_value(N,Xo),
          square_root(N,Xo,Xo+1,V),
          round(N-V*V,(V+1)*(V+1)-N,V,R);
square_root(N,Xn,Xo,Xo):- Xo=<Xn,!;         % 繰返し終了判定
square_root(N,Xo,_,V):- (Xo+N/Xo)/2=Xn,     % 平方根の近似計算
          square_root(N,Xn,Xo,V);           % 終端再帰呼出し
 initial_value(N,N):- N<100,!;              % 初期値設定の述語
 initial_value(N,N/10):- N<10000,!;
 initial_value(N,N/100):- N<1000000,!;
 initial_value(N,N/1000):- N<100000000,!;
 initial_value(N,N/10000);
round(X,Y,V,V):- X<Y,!;
round(_,_,V,V+1);
```

N=2 で呼出せば、Result=1，N=3 ならResult=2，N=200000000 ならResult=14142，N=300000000 ならResult=17321が求まる。

注：述語 initial_value は近似計算の収斂を速める事が目的であり、使わなくともXo=N-1でもよい。但し N=1対策にsquare_root(1,1):- !;の追加要。

６．２．３　「浮動小数点」データ

浮動小数点のデータは、 2.2.3に示した表記法で記述され、プログラム内で利用する定数値を与える。プログラム内部では、実数値として加減乗除の演算に使用できる。

実数は、符号＋仮数部24ビットの16進正規化であり、有効数字は10進 6桁である。実数の取扱範囲は、10^{-78} ～10^{75}である。

◇　「浮動小数点」データの使用例

例題13　第１引数の実数 Xより指数関数（e^X）を次式より求めて第２引数に返す述語とする。

$$e^X = 1 + x + x^2 / 2! + x^3 / 3! + \cdots + x^n / n! + \cdots$$

述語呼出し側　　… exponent(X,Result).

```
exponent(0.0,1.0):- !;
exponent(1.0,2.71828):- !;
exponent(X,R):- exp(X,2.0,X,1.0,R);          % 初期値の設定
exp(_,_,_,T,R,R):- T< 1.0^-5,!;              % 繰返しの終了判定
exp(X,N,T,Ro,R):- T*X/N=Tn,                  % 漸化式による近似計算
          exp(X,N+1.0,Tn,Ro+Tn,R);           % 終端再帰呼出し
```

漸化式では、$x^n / n! = (x^{n-1} / (n-1)!) * (x/n)$を使用している。

X-1.0 で呼出せば、述語exponentの 2行目によりResult=2.718277 が返り、X=1.3125なら近似計算によりResult=3.715439 が求まる。

例題14　ラジアン(Radian)角の三角関数を次式より求め、実数値で返すクラス・メソッドを持ったクラスとする。

$\sin(X) = X - X^3/3! + X^5/5! - X^7/7! + X^9/9! - \cdots$

メソッド呼出し側　…,:sin(#trigonometric_function,Radian,Result)

```
class  trigonometric_function  has
:sin(Class,R,X):- floating_point(R) R>= 0.0 sin(R,X);
:cos(Class,R,X):- floating_point(R),R>= 0.0 cos(R,X);
:tan(Class,R,Y,Z):- floating_point(R),R>= 0.0 sin(R,Y),cos(R,Z);
local
sin(R,X):- R =<1.570796,!,sin(R,1.0,R,-1.0,R,X);
sin(R,X):- R=< 3.141592,!,3.141592-R=Rn sin(Rn,1.0,Rn,-1.0,Rn,X);
sin(R,-X):- R=<4.712392,!,R-3.141592=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
sin(R,-X):- R=<6.283185,!,6.283185-R=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
sin(R,X):-module(R,Rn),sin(Rn,X);
cos(R,X):- R =< 1.570796,!,1.570796-R=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
cos(R,-X) :- R=< 3.141592,!,R-1.570796=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
cos(R,-X) :- R=< 4.712392,!,4.712392-R=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
cos(R,X):- R=< 6.283185,!,R-4.712392=Rn,sin(Rn,1.0,Rn,-1.0,Rn,X);
cos(R,X):-module(R,Rn),cos(Rn,X);
sin(R,N,T,_,Xo,X):- T<1.0^-5,!,X=Xo; ------------------ ①
sin(R,N,T,Sign,Xo,X):- T=R*R/((N+1.0)*(N+2.0)),fn,
          sin(R,N-2.0,Tn,-Sign,Xo+Sign*Tn,X);
module(R,R):- R=< 6.283185,!;
module(R,Rn):- module(R-6.283185,Rn);
end.
```

各クラス・メソッドでは、次のパラメータの妥当性チェックを行っている。

(a) floating_point(R)は KLO組込述語であり、引数R が浮動小数点なら成功し、それ以外なら失敗(fail)する。

(b)　引数R が負なら失敗(fail)する。

尚、引数X が求まる値以外（整数など）で定義済なら、①により失敗する。

引数R が 2π(360°) を越えても述語moduleにより処置し、三角関数値は正しく求められる。

R=0.785398 (45° に相当) で:sin(R,X) を呼出せば、X=0.707106(=1/√2) が求まる。

６．２．４　構造体データ

（１）　「ベクタ」データ

ベクタは、個々の要素が一列に並んだ一次元の配列(Array) である。尚、ベクタの各要素がまたベクタで有ってもよく、これを繰返せば多次元の配列を実現出来る。

ベクタには、ＥＳＰでプログラムを作成する際の記述形式（ＥＳＰ表記）としての「ベクタ」(3.3を参照）と、構造体のデータの入れものとしてのベクタ（スタック・ベクタ，ヒープ・ベクタで KLO組込述語説明書を参照）の２通りの意味がある。ベクタのデータ構造が同じでも、操作の手段，操作結果の保持性，ＥＳＰ表記等で差異があるので、その特徴を次表に示す。

データ・タイプ	操作の手段	ＥＳＰ表記	操作結果の保持性
スタック・ベクタ	KLO組込述語	「ベクタ」 「複合ターム」 「演算子適用」	副作用なし （バックトラックでUNDOされる。）
	[H | T]を含む述語	「リスト」	
ヒープ・ベクタ	KLO組込述語	なし (操作結果の代入のみ)	副作用あり

（２）　「ストリング」データ

ストリングは、個々の要素が整数値を持つ、一次元の配列のみである。その特徴を上表と同じ形式で次表に示す。

ストリング	KLO組込述語	「文字列」 「ストリング」	副作用あり

一般ユーザでは、これ等のＥＳＰ表記の内、複合ターム，演算子適用はプログラムとしてのクローズの記述に使用するが、データ記述に使用する例は少なく、またリストの使用例は例題 01,02,08,09で何度も示しているので、共に使用例を省略する。

◇　構造体データの使用例

例題15　第１から第３引数の変数を３辺とする三角形の形状を判定し、正三角形なら外部表現形式の"Regular triangle"，二等辺三角形なら"Isoscelens triangle"，その他は"Scalence triangle" をストリングで第４引数に返す述語とする。但し、既に 6.1.1の例題 4のtriangle(A,B,C) で成立した３つの変数で呼出すものとする。

述語呼出し側　…,triangle(A,B,C),shape_message(A,B,C,Sting),

```
shape_message(A,B,C,St):- shape(A,B,C,P),
        V ={" triangle","Scalence","Isoscelens","Regular"},---- ①
        vector_element(V,P,St1),        % P番目の要素の取出し ---- ②
        first(V,St2),                   % 先頭の要素の取出し
        string(St1,Size1,Type),string(St2,Size2,_),  % 文字数入手
         new_string(St,Size1+Size2,Type),      % 入れもの確保
         set_substring(St,0,Size1,St1),                          ③
        transfer(St2,0,Size2,Size1,St);                  ---- ④
shape(N,N,N,3):- !;
shape(N,N,_,2):- !;
shape(N,_,N,2):- !;        ---- % 例題 5(a) に同じ
shape(_,N,N,2):- !;
shape(_,_,_,1);
transfer(_,N,N,_,_):- !;
transfer(S,Po,N,Cn,St):-string_element(S,Po,Code), % 要素の取出し
         set_string_element(St,Cn,Code),        % 要素の代入
        transfer(S,Po+1,N,Cn+1,St);                          ③
```

プログラムは、形状判定に応じてスタック・ベクタから２つの文字並びを取出して結合したストリングを生成している。この例題の問題解決に適した解法ではないが、構造体の要素単位の操作例を示す為と理解願たい。

①は「ベクタ」表記の４要素のスタック・ベクタであり、更に各要素は「文字列」表記のストリングである。

②はスタック・ベクタ，③はストリングを操作する KL0組込述語である。

④で呼出している述語transferは、 set_substring で代替出来るが、ストリングを操作する KL0組込述語string_element, set_string_element の使用例を挙げる為に設けている。

6．3　変数，スロット類

「ＳＰでは、演算処理過程のデータの入れものとして次の５種を使用する事が出来る。

(a) 変数(Variable)
(b) スロット(Slot)
(c) スタック・ベクタ(Stack vector)
(d) ヒープ・ベクタ(Heap vector)
(e) ストリング(String)

これ等の内、(a) の変数，(c) のスタック・ベクタは、演算処理過程で一旦、値を持っても副作用がなく、バックトラックのundoの範囲であれば未定義（値を持たない）の状態に戻される。その他は副作用があり、バックトラックのundoの範囲であっても何ら影響されない。また、従来型言語の変数の様に、値の更新を許すものと更新を禁止しているものがある。これ等の特徴を次表に示す。

特　徴	変　数	スロット	スタック・ベクタ	
入れものの定義	変数名の記述	スロット定義の記述	「ベクタ」「リスト」等の記述	KL0 組込述語の記述
データ・タイプ	全てのデータ・タイプ	スタック・ベクタを除く残りの全てのタイプ	全てのデータ・タイプ	
初期値(初期状態)	な　し（未定義）	スロット定義の「スロット初期化」または"0"	入れものの定義時の値	な　し（未定義）
値の代入（例）	ユニフィケイション（X = defined）	代入マクロ(set_slot)（Object!slot:=Value）	リストのみ[H|T]等	組込述語（first(V,E)）
参照方法（例）	変数名の指定（X >- 0）	参照マクロ(get　slot)（X== Object!slot+Y）	組込述語，但しリストのみ[H|T]を含む述語	
更新可否(更新方法)	不　可	随時可能（代入マクロ）	リストのみ可能（同上）	不　可

特　徴	ヒープ・ベクタ	ストリング	
入れものの定義	KL0 組込述語の記述（例 new_heap_vector(Heap,Size)）	「文字列」「ストリング」の記述	KL0 組込述語の記述（例 new_string…）
データ・タイプ	スタック・ベクタを除く残りの全てのタイプ	ストリングのみ（各要素は整数値のみ）	
初期値（初期状態）	な　し（未定義）	入れものの定義時の値	な　し（未定義）
値の代入（例）	KL0 組込述語(set_vector_element(H,P,E))	不　可（参照のみ可能）	KL0 組込述語（例題14の③）
参照方法（例）	KL0 組込述語(second(Heap,Element))	KL0 組込述語（例題14の③）	
更新可否（更新方法）	随時可能（KL0 組込述語）	不　可	随時可能（KL0組込述語）

変数は、未定義の状態からユニフィケイションにより、全てのデータ・タイプの値を取り得て、定義済の状態と成る。一旦、定義済の状態に成るともはや変数ではなくユニファイで決定したデータ・タイプとして振舞う事に成る。

スロットは、オブジェクトの内部状態を保持し、未定義の状態がなく常に何かの値を持っている。従って、UNDOされる恐れのあるスタック・ベクタの代入は出来ない。

スタック・ベクタは、KL0 組込述語により新たなスタック・ベクタを生成する際に既存のスタック・ベクタからの要素の取出しは可能であるが、既存のスタック・ベクタの内容を更新する事は出来ない。但し、リストのみ更新可能である。

ヒープ・ベクタは、従来型言語の配列の如く、データ領域の確保（いれもの定義），値の代入，参照がKL0 組込述語により随時可能である。

ストリングは、ヒープ・ベクタと同様の操作が可能である。但し、プログラム内で「文字列」，「ストリング」の記述により初期値を与える場合、プログラムの変更に相当する代入・更新は許されない。

６．４　既成プログラムの利用

ＥＳＰでプログラムを作成するに際して、既成のプログラム．例えばSIMPOSを構成する各プログラムやユーザ作成のプログラム等をクラスの継承やメソッド呼出しで利用する事が出来る。

SIMPOSではライブラリ・サブシステムが．既成のクラス類の利用が容易に成る様、すべてのプログラムを管理している。

ここでは、クラスの継承による流用やメソッド呼出しによる他オブジェクトの利用の方法と使い分け方および他からの利用を前提とした部品的な作成法を具体例と共に解説する。

６．４．１　クラス継承による利用

クラス継承の意味は 1.4に、継承定義は 4.4に示されている。従って、ここではセマンティック・ネットワークのIS_A(クラスの継承)．HAS_A(スロットに他オブジェクトを保持）の使い方を具体例で示す。

例題16　ビットマップ・ディスプレイにウインドウを設け、所定の半径の外側の円を描いておき、キーボードより入力する円の半径の比により、転がりの開始位置に内側の円とそのハイポサイクロイド(Hypocycloid) を描くプログラムを作成するものとする。

クラスの構成は、例えば次の様に計画する。

(a) ウインドウは、SIMPOSの完成品クラスwindowのインスタンス・オブジェクトを使用するがキーボードから整数を入力するメソッドを追加した専用のクラスwith_user_windowを作成する。

(b) クラス hypocycloid_drawerは、処理の中心を成すプログラムとして作成し、クラスwith_user_windowを継承した上で、ウインドウを使用する。

(c) 三角関数が必要と成るが、ラジアン角の三角関数を提供するクラス trigonometric_function(6.2.3の例題14) のクラス・メソッドを利用する。

これらのクラスの相互関係を次図に示す。尚、プログラムの個々の説明は 6.4.3に後述する。

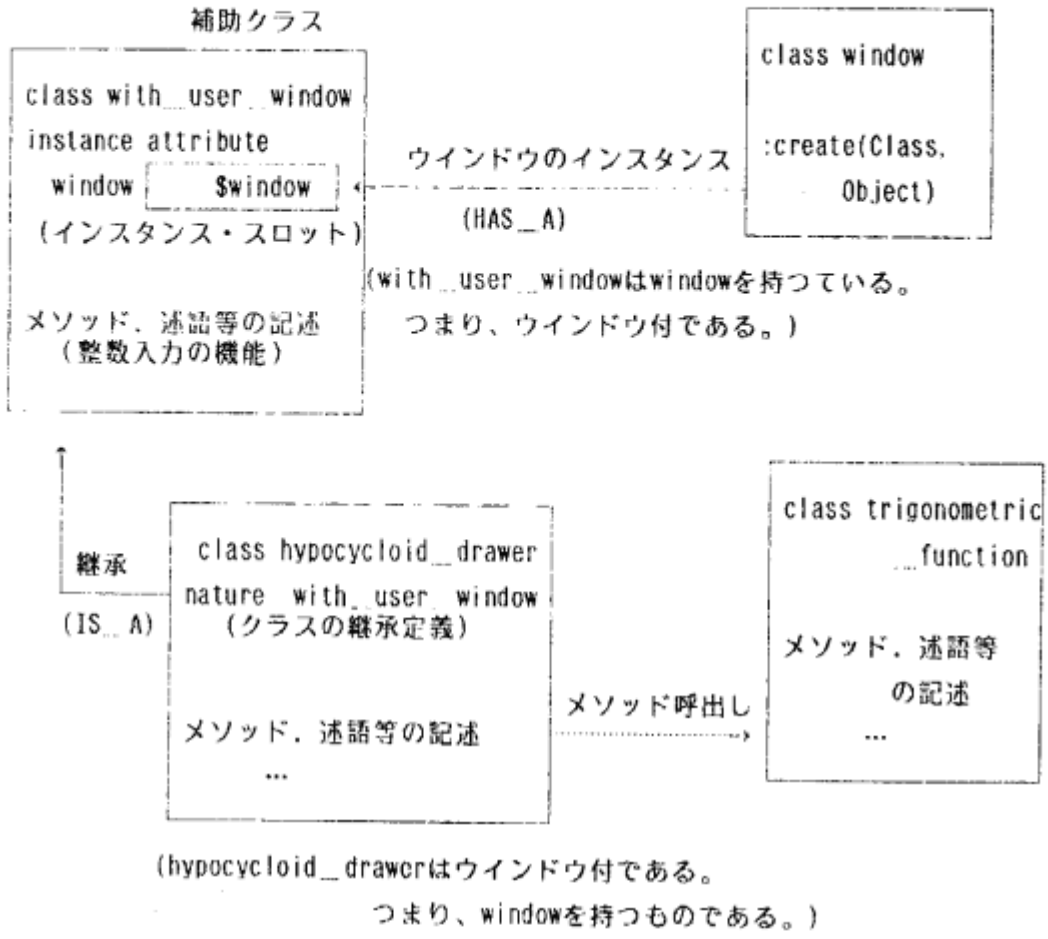


図　6－2　作成クラス類の相互関係

オブジェクト指向プログラミングのIS_A ,HAS_A の意味を厳密に使い分けてプログラミングする事は、複雑な問題解決に有効であろう。

通常の簡単なプログラムでは、IS_A のみで実現しても全体が見通せる事もあり、また意図する動作にも支障ないので、あまり厳密に使い分けない例も起き易い。

例えば上の例で、クラス hypocycloid drawerがクラスwindowとクラス trigonometric_functionを直接に継承しても支障なく動作する。但し、意味的には「hypocycloid _drawerはウインドウであり、三角関数である。」と成り、 hypocycloid_drawerのメソッドで三角関数を求める得る等おかしな事に成るので注意を要する。

注：補助クラスは通常、テンプレートまでで自らのオブジェクトがなく、他が継承の上利用する事を前提としている。

６．４．２　メソッド呼出しによる利用

メソッド呼出しの定義，利用方法等は 4.5.3に示されているが、ＥＳＰのメソッド呼出しは、従来型言語のサブルーチン・コールに相当する。

使いたいメソッドを持つクラスが実行主体となるオブジェクトつまりクラス・オブジェクト，インスタンス・オブジェクトであれば、継承する事なく、第１引数でそのオブジェクトを指定したメソッド呼出しで使いたいメソッドを利用する事が出来る。

前述の例でもクラス trigonometric_functionを継承せずに、メソッド呼出しの利用を意図している。

また、補助クラスwith_user_windowを設けず、クラス hypocycloid_drawerがクラスwindowをメソッド呼出しで利用する事も可能であるが、補助クラスwith_user_windowに追加するウインドウからの整数のキーボード入力のメソッド，述語などは、クラス hypocycloid_drawer自身で保有する必要がある。このクラスとしては意味の異なる機能を持つ事になり好ましくない。欠張り、クラス継承とメソッド呼出しの使い分けが必要である。

６．４．３　例題プログラム（例題16）の説明

前述の 6.4.1に示したハイポサイクロイドを描くプログラムを、作成するクラス毎に説明し、プログラムの記述例を示す。尚、これの使用法と使用例は後述する。

（１）　クラスwith_user_window

(a) クラスwindowのインスタンス・オブジェクトをこのクラスのインスタンス・スロットのwindowに保持する。但し、ウインドウの位置，大きさはクラスwindowのデフォルト値である。

(b) キーボードより整数を入力する次のメソッドを提供する。

:integer_from_keyin(Object,Integer)

１文字毎に整数値としての妥当性チェックを行い、正しければその文字を表示し、整数以外に成るキー操作は無視する。尚、訂正はdel キーにより１文字毎に取消す事が出来る。"+" 符号は省略可能である。

CR入力までの入力文字を整数値に変換して第２引数のInteger に返す。但し、符号のみ入力の時は値 "0"を返す。

符号，整数の入力操作がなく、CR入力のみの場合は、名前のデータ "no_data" を第２引数のInteger に返す。

注：第２引数に返す際、既に異なる値で定義済の場合はメソッドが失敗する。

```
class with_user_window has
instance attribute
          {window:= W:- :create(#window,W)); ------ ①
component   data_sign:= undefined;
:integer_from_keyin(Obj,Integer):- W=Obj!window,
          from_keyin(Obj,W,Integer);
local
from  keyin(Obj,W,I):- keyin(Obj,W,non,key#cr,[],List),
           (Obj!data_sign==undefined,!,Integer=no_data;
            recerse_decode(List,N), Obj!data_sign*N-I,
             Obj!data_sign:= undefined);
keyin(_,W,Target,Target,list,list):- !,:write(W,key#cr),  ------ ②
          :write(W,key#lf);                    % 終了時、改行，復改
keyin(Obj,W,_,Target,Lo,L):- :read(W,Ch),
           check_character(Obj,W,Ch, Obj!data_sign,Lo,L),
          keyin(Obj,W,Ch,Target,Ln,I);
check_character(Obj,W,key#del,_,[ H ! T ],I):- !,:write(W,key#dc!),
           (T=[],!, Obj!data_sign:= undefined;true);
check_character(Obj,W,Ch,Sign,lo,L):- Ch=< #"9",Ch>= #"0",!,
           (Sign==undefined,!, Obj!data  sign:= 1,Ln=[0 ! Lo];
            Ln-lo),
          !Ch- #"0" ! Ln]+I,:write(W,Ch);      % 数値をリストに追加
check_character(Obj,W,#"-",Sign,Lo, [0 ! Lo]):- Sign== undefined,!,
          Obj!data_sign:= -1,:write(W,#"-");
check_character(Obj,W,#"+",Sign,Lo, [0 ! Lo]):- Sign== undefined,!,
          Obj!data_sign:= 1,:write(W,#"+");
check_character(_, _, _, _,L,L);       % 不要な文字は無視
reverse_decode([],0):- !;
reverse_decode( [D ! Ln],No*10+D):- reverse_decode(Ln,No);
end.
```

①は、スロット初期化コードによりウインドウのインスタンスを、継承して使用する他クラスからのアクセスの為、attribute スロットに保持する。
②は、復改を入力するまでバックトラックする、前述 6.1.3の2)の方法である。
尚、:write(W,Ch)は、Chの文字コードを１文字表示するウインドウのメソッドである。

（２）　クラス hypocycloid　drawer

(a) 次の２通の起動方法を提供する。

①外側の円の半径は指定せず、デフォルト値の半径 200ドットで使用する。

:initiate(Object)

②外側の円の半径は 100を越え 360の範囲で任意に指定して使用する。

:intiate(Object,Radius)　　但し、Radiusは整数値の事。

(b) クラスwith_user_windowを継承し、外側の円の半径に応じたサイズのウインドウをスクリーンカレントに設け、ウインドウ内に外側の円を描く。

(c) 半径比（＝外側の円の半径／内側の円の半径）の許容値を示した入力促進の表示を行ない、キー入力待ちとなる。

"Ratio(2,3,..<199)- "　　但し、下線部はRadius-1で可変

(d) 半径比の入力が許容値以外なら再度、入力促進から繰り返す。

(e) 半径比の入力が正しければ、外側の円に内接した内側の円を転がりの開始位置（外側の円の中心を通る縦線の真下）に描く。

(f) ハイポサイクロイドは、外側の円の半径を r_1，内側の円の半径を r_2 とし、外側の円の中心を通常の直交座標の原点に取り、転がりの開始位置と現在位置が外側の円の中心で成す角を θ，現在位置での内側の円の回転角を ϕ として次式より座標点を求める。

$$X = (r_1 - r_2) * \sin(\theta) - r_2 * \sin(\phi - \theta)$$

$$Y = (r_1 - r_2) * \cos(\theta) - r_2 * \cos(\phi - \theta)$$

但し、$\phi = (r_1 / r_2) * \theta$

この座標点を、θ=0.0から 2π(=6.283185) を越えるまで、ある増分（＝Radiusの逆数の実数値）で繰返し求めて、プロットする。

引：ビットマップ・ディスプレイは、縦方向のＹ軸が通常の直交座標とは逆向きであり、プログラム上でＹの算式に手当が必要である。

(g) ハイポリイクロイドを描き終えれば、プログラムは成功して終了する。

```
class  hypocycloid_drawer has
nature  with_user_window;
instance component
          prompt:= "Ratio(2,3,..<    )= ",radius:=200,
          Size_x,Size_y,center_x,center_y;
:initiate(Obj,Radius):- integer(Radius),Radius>100,
          Radius=<360,Obj!radius:= Radius,:initiate(Obj);
:initiate(Obj):-  Obj!size_x:= 2*(Obj!radius+20),
          Obj!size_y:= Obj!size_x+54,
          Obj!center_x:= Obj!radius+14,
          Obj!center_y:= Obj!radius+24,
          :get_number_string(#symbolizer,Obj!radius-1,String), -- ②
          set_substring(Obj!prompt,13,3,String), ----- ③
          W=Obj!window, :set_size(W, Obj!size_x,Obj!size_y),  -- ④
          :set_title(W,"    < Hypocycloid >"),:activate(W),  ---- ⑤
           get_ratio(Obj,W,Ratio),
          hypocycloid(Obj,W,Obj!radius,Ratio);
local
 get_ratio(Obj,W,Ratio):- :clear(W),
          :draw_circle(W, Obj!center_x,Obj!center_y, ----- ⑥
          Obj!radius ,0,0,1,0,or),
          :write_lines(W,Obj!prompt), ----- ⑦
          :integer_from_keyin(Obj,Ratio),  ----- ⑧
          Ratio>=2,Ratio<Obj!radius,!,Obj!radius/Ratio=R2,
           :draw_circle(W, Obj!center_x,Obj!center_y+Obj!radius-R2,
          R2,0,0,1,0,or);
 get_ratio(Obj,W,Ratio):- get_ratio(Obj,W,Ratio);
hypocycloid(Obj,W,Radius,Ratio):-
           integer_to_floating_point(Radius, Real_radius),
           integer_to_floating_point(Ratio,Real_ratio),   ---- ⑨
          cycloid(Obj,W,Real_radius, Real_ratio,
          1.0/Real_radius,0.0);
```

% ウインドウのサイズと中心の座標の設定

```
cycloid(Obj,W,R1,Rt,Inc,Radian):- Radian>= 6.283185,!,;
cycloid(Obj,W,R1,Rt,Inc,Ang1):- R1/Rt-R2,Ang1*(Rt-1.0)=Ang2,
          :sin(#trigonometric  function,Ang1,V1),
          :sin(#trigonometric  function,Ang2,V2),
         floating_point _to  integer((R1-R2)*V1-R2*V2,X), ---- ⑨
          :coc(#trigonometric_function,Ang1,V3),
          :coc(#trigonometric_function,Ang2,V4),
         floating_point _to_ integer((R1-R2)*V3+R2*V4,Y), ---- ⑨
         :set  point  value(W,X+Obj!center_x,Y+Obj!center_y,1), ⑩
         cycloid(Obj,W<R1,Rt,Inc,Ang1+Inc);
end.
```

②は、整数値からストリングを求めるメソッドである。（SIMPOS使用説明書のトランスデューサを参照） 4.5.1例題 2に相当するが、正なら符号は省略する。
③は、ストリングの部分書き換えを行う KLO組込述語である。（ KLO組込述語説明書を参照）この例で注意すべきは、実質的に「文字列」で初期値を与えたストリングの部分書き換えを行なっているがスロット値の変更であり、前述の 6.3で解説したプログラムの変更に該当しない点である。
④～⑦は、リイズの設定，上部のタイトル付け，:activate(W)でウインドウの表示，⑥で円を描き，⑦で文字の並びの表示を行うウインドウのメソッドである。（SIMPOS使用説明書のウインドウの項を参照）
⑧は、継承したwith user_windowのメソッド呼出しである。
⑨は、整数～実数間の変換を行う KLO組込述語である。
⑩は、１点をプロットするウインドウのメソッドである。

６．５　プログラム作成・実行のマシン操作

プログラムをＰＳＩ，SIMPOS上で作成して動作・実行する操作のすべては、ビットマップ・ディスプレイ，マウス及びキーボードを用いた対話形式で行う事ができる。ここでは例題プログラムの使用法，使用例に先立ち、プログラムの作成・実行の手順の概略を紹介するが、SIMPOS操作説明書の該当項を合せて参照願いたい。

尚、SIMPOS操作説明書のイニシャル・プログラム・ローダの項のＩＰＬ手順で、ＰＳＩ，SIMPOSを前もって立上げてあるものとする。

（１）　ログ・インの操作

SIMPOS立上げ後、ユーザ名，パスワードの順にキー入力待ちと成るので、登録してあるユーザ名，パスワードを入力するが、未登録のユーザは共に "me" とCR入力を行えば、一般ユーザとしてのログ・インが完了し、スクリーンは無表示に戻る。

（２）　システム・メニューの操作

マウスの右ボタンを２回連続してクリックすればシステム・メニューが表示され、デバッガ，エディタ，ライブラリアン等のプログラム開発支援ツールをマウスの移動と左ボタンのクリックにより選択して使用する事が出来る。

（３）　エディタからプログラムの作成

システム・メニューでエディタを選択し、任意の大きさのウインドウをマウス操作（左クリック，移動，左クリック）で表示し、クラス毎に、プログラムをキーボード操作で作成の上、ディスクのファイルへキーボード操作で送る手順を繰返す。

例えば、例題16のプログラムの作成では、例題14の三角関数の trigonometric_function, with_user_window, hypocycloid_drawerの３つのクラスを作成する。

（４）　ライブラリアンによるコンパイル

システム・メニューでライブラリアンを選択し、クラスのテンプレート，クラス・オブジェクトの生成を行う。

例えば、上で作成するプログラムでは、クラス継承，メソッド呼出しの有無を考慮し、次の順序でCatalogue,Compile を行う。

① クラス trigonometric_function
② クラスwith_user_window　　注：①，②は逆でも支障なし
③ クラス hypocycloid_drawer

（５）　デバッガからのプログラム実行

システム・メニューでデバッガを選択し、プロンプト":-"に続きキー操作する事により、従来のサブルーチン・コール形式で、プログラムを起動する。

例えば、上例のプログラムでは、次の操作で実行する。

① :- :new(#hypocycloid_drawer,Obj).CR入力により、Obj に#hypocycloid_drawerのインスタンス・オブジェクトを得る。

② :- :initiate(Obj).または、:initiate(Obj,Radius).CR入力により、このオブジェクトが動作し、このオブジェクトのウインドウが表示されキーボード入力待ちとなる。但し、Radiusは整数である。

↓

このオブジェクトに対する入力操作

↓

③ このオブジェクトの動作が終了すると、プロンプト":-"が現れる。引続き変数名を変更し①から別のオブジェクトで繰返すか、または同じオブジェクトに対して②から繰返す事も出来る。

（６）　操作終了の処置

システム・メニューを表示し、デバッガを選択して、"kill"すればデバッバ及びこのオブジェクトのウインドウが消滅して終了する。尚、システム・メニューを表示で、"shut down" を選択すれば、すべてのウンドウが消滅して終了する。

（７）　例題16の使用法と使用例

前述の(5) ②に於けるオブジェクトに対する入力操作と操作結果を次に示す。

(a) 次図の例の様に、< Hypocycloid > のタイトルのウインドウの入力待ちに対して、キーボード操作により半径比として指定の範囲内で任意の値を入力する。

(b) 半径比に応じたサイクロイドを描いて終了する。但し、半径比が指定の範囲外なら、再度入力待ちに戻る。

次図は、ビットマップ・ディスプレイのスクリーンのハード・コピーである。

デバッガのウインドウが右側に出ており、プログラムの起動操作が表示されている。外側の円の半径を指定ものとデホルト値のもので内部状態の異なる２つのオブジェクトで動作させている。その結果が左側の２つのウインドウで、半径比の３，４が入力操作値である。尚、半径比４のウインドウは、ウインドウ・マニピュレータを用いて中央へ移動しているが、元の表示位置は左上のコーナである。

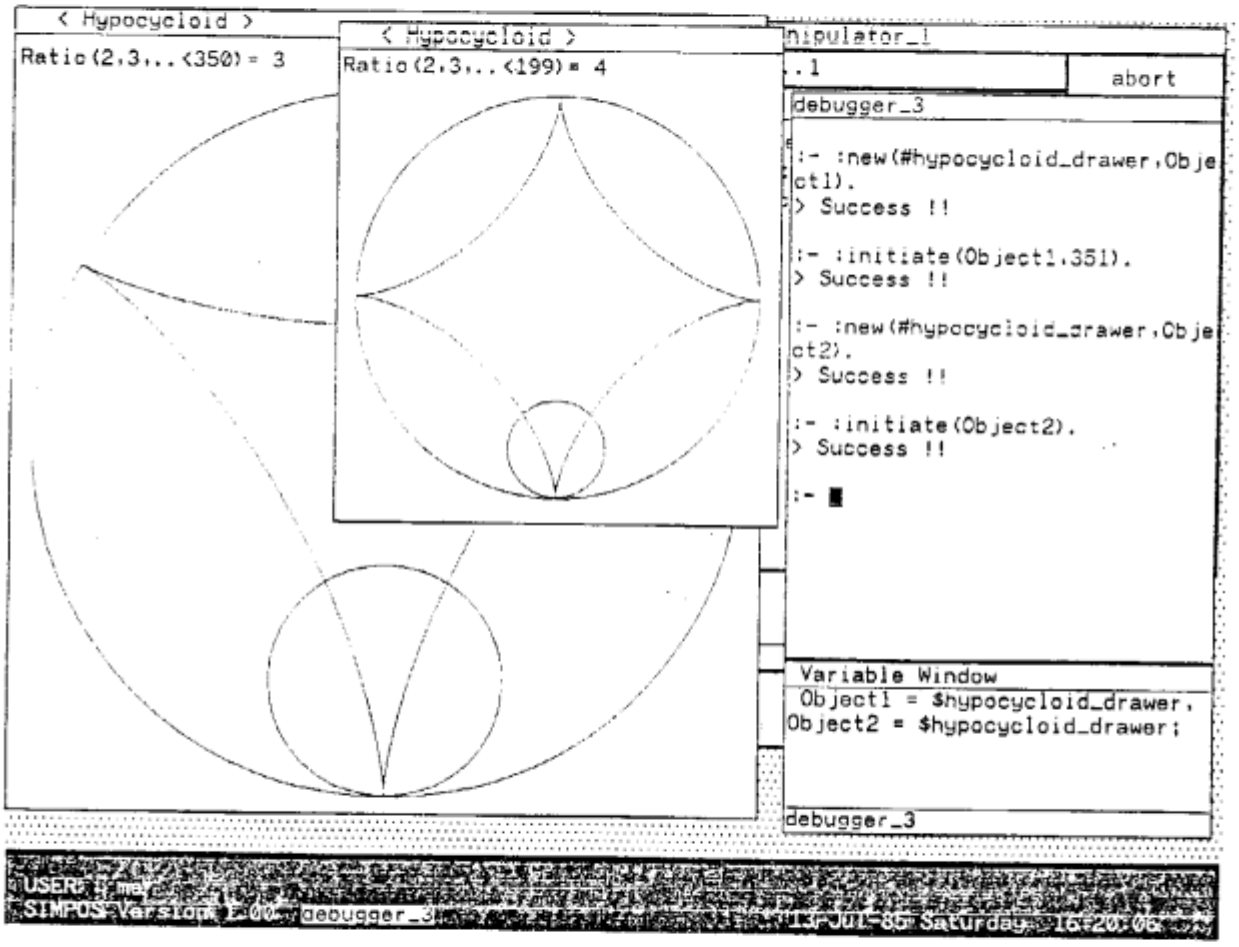


図　６－３　ハイポサイクロイドの使用例

６．６　プロセスの作り方

SIMPOSでは、プログラムの実行をプロセスと呼ばれる実行単位で、インスタンス・オブジェクトを独立に動作させるプログラムの実行制御の機構を提供している。（SIMPOS使用説明書のプロセスを参照）従来の概念では、タスクの実行制御が対応する。

前述 6.5の例題プログラムはサブルーチン形に作成してある為、デバッガからのメソッド呼出しにより実行し、デバッガのプロセス上でのサブルーチンに相当する。従って一旦、例題プログラムを呼出すと動作が終了するまで、そのデバッガの動作が中断している。

（１）　プロセスの要件

独立したプロセスとして動作させ得るプログラムの作成の際には、次の要件を含む必要がある。

(a) SIMPOSのクラスas_program を継承する事。

この継承により、次のクラス・メソッドが自動的に付加される。

```
:create(Class,Object)
```

:create が呼出されるとそのゴールの:new(Class,Object)により、インスタンス・オブジェクトが生成され、そのオブジェクトが第２引数に渡される。

(b) 実行開始のゴール列の呼出し用ヘッドのインスタンス・メソッドは、１引数の次のものとする。引数は当然、インスタンス・オブジェクトである。

```
:goal(Object)
```

（２）　例題プログラムのプロセス作り

前述 6.4.3(2) のクラス hypocycloid_drawerをサブルーチン形からプロセス形に改造する例を示す。

改造範囲は、継承定義にas_program を追加する事及びインスタンス・メソッドの:initiate を、ウインドウ・サイズ等インスタンス生成の前準備の部分はクラス・メソッドに、実行処理部はインスタンス・メソッドに分離する事である。

尚、独立したプロセスである以上、このオブジェクト自体で繰返し動作可能と成る様、一部変更し、例えば次の機能を追加する必要がある。

(a) ハイポサイクロイドを描いた後、ウインドウの２行目に次の表示を行いキー入力待ちとなる。

```
" Any key(End=^Z)? "
```

(b) キー入力がcontrol#"Z" ならウインドウを消し(kill)てプロセスを終了し、その他のキーなら再度ウインドウをクリアし、半径比の入力から繰返す。

クラス hypocycloid_drawerの改造・追加の範囲のみ記述する。

```
class  hypocycloid_drawer has
nature  as_program,with_user_window;
component  default_radius:= 200;
:create(Class,Radius,Obj):- integer(Radius),
          Radius>100,Radius=<360,Class!default _radius:=Radius,
          :create(Class,Obj);
after     :create(Class,Obj):- Obj!radius:=Class!default_ radius,
          Obj!size_x:= 2+(Obj!radius+20),
          Obj!size_y:= Obj!size_x+54,
          Obj!center_x:= Obj!radius+14,
          Obj!center_y:= Obj!radius+24,
          :get_number_string(#symbolizer,Obj!radius-1,String),
          set _substring(Obj!prompt,13,3,String),
          Class!default _radius:= 200;        % 初期値の更新対策
instance component
          prompt:= "Ratio(2.3,..<    )- ",radius:=200,
          Size_ x,Size_y,center_x,center_y;
:goal(Obj):- W=Obj!window, :set _size(W, Obj!size_x,Obj!size_y),
          :set_title(W,"    < Hypocycloid >"),:activate(W),
          operate(Obj,W);
local
operate(Obj,W):- get_ratio(Obj,W,Ratio),
          hypocycloid(Obj,W,Obj!radius,Ratio),
          :write_lines(W,"Any key(End= ^Z)?"),        % 変更・追加
          :read(W,control#"z"),!,:kill(W);
operate(Obj,W):- :operat(Obj,W);
```

前述 6.5 (5)①に於けるオブジェクト生成のメソッド:newは、継承したas_program のメソッド:create(Class,Object) に任せる。

インスタンス・オブジェクト生成の際に、外側の円の半径に基づくオブジェクトの内部状態をインスタンス・スロットに設定する。この為、:create(Class,Object) の after demon(4.5.3(1)参照) 結合で、インスタンスの生成を待って処置している。

追加の述語operate は、control#"z" を入力するまでバックトラックするが、前述 6.1.3の1)の方法である。

（３）　プロセスの起動法

プロセスとして動作しているオブジェクトから、他のオブジェクトを別のプロセスとして起動する事が出来る。（SIMPOS使用説明書のプロセスの項を参照）

ここでは、前述の 6.5(5) デバッガ（プロセスとして動作しているオブジェクト）から (2)の改造済プログラムの起動の例を紹介する。

① :- :create(#hypocycloid_drawer,Radius,Object).または、
:create(#hypocycloid_drawer,Object). 、とCR入力により、内部状態が設定されたインスタンス・オブジェクトを生成する。

② :- :create(#process,Process). CR入力により、プロセスのインスタンス・オブジェクトとが確保される。

③ :- :activate(Process,Object). CR入力により、プロセスのインスタンス・オブジェクトに、上の#hypocycloid_drawerのインスタンス・オブジェクトを渡し、:goal(Object) が呼出されて実行が開始される。

このメソッド:activate は直ちに終了し、プロンプト" :- "が現れデバッガの操作を継続する事が出来る。

この動作の一例を次図に示すが、左側と右側のデバッガとは、独立プロセスである。

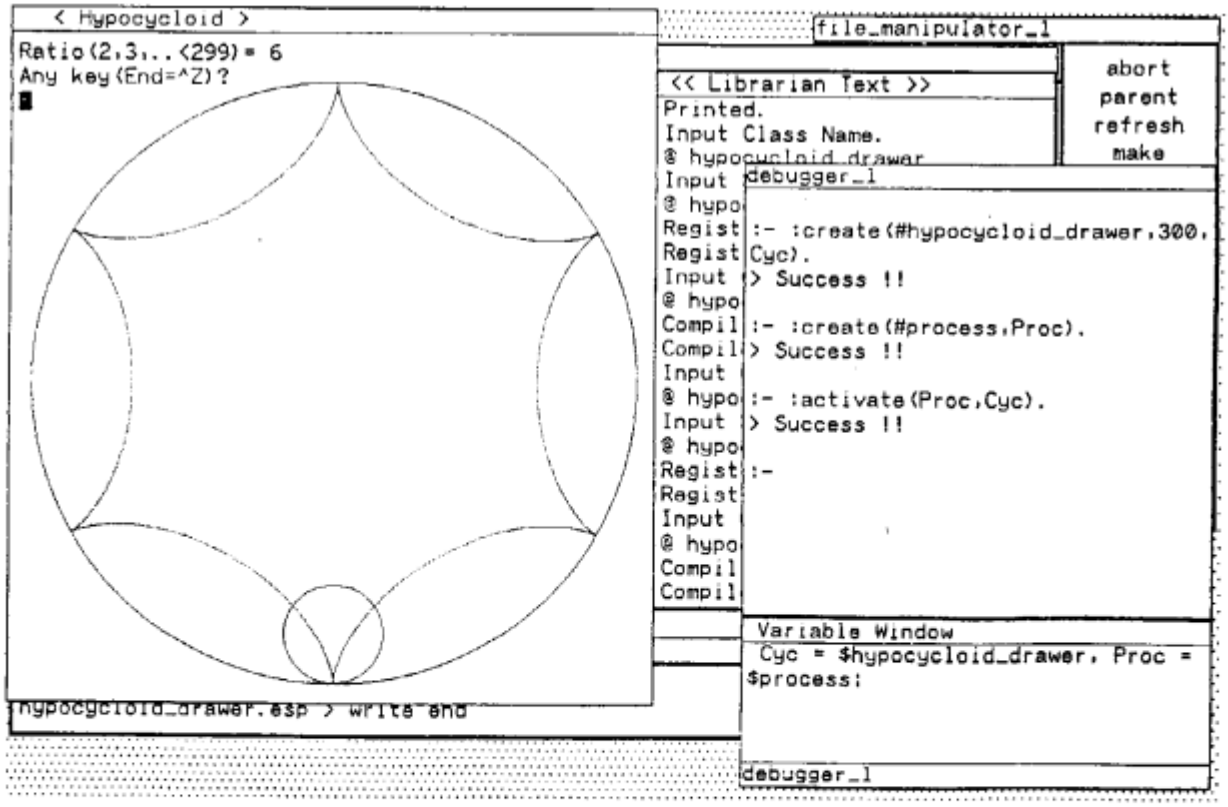


図　６－４　プロセスの起動例

6．7　プロセスの終結とデータ保存

プロセスは、インスタンスの実行単位であり、実行中はインスタンス・スロットに各種のデータを保存して置く事が出来た。但し、:goal(Object) で呼出されたゴール列が、成功．失敗に拘らず、実行終了するとプロセスが終結(terminate) と成り、これまで実行して来たインスタンスも消滅し、このインスタンス・スロットに保存していたデータも消滅する事に成る。

今、ＣＡＩの学習用プログラムを例に取ると、各生徒の進捗に応じて、また各自の時間的都合に合せた細切れの学習を支援する必要がある。つまり、各生徒がログインの都度、その生徒に関する前回までの進捗データ（達成済カリキュラムNo，問題別所要時間，後戻り回数など）を伴って各インスタンスが動作しなければならない。

この様な目的の為、データを継続して保存する方法として次が考えられる。

(1) ファイルに保存する。（ファイルの使用法はSIMPOS使用説明書を参照願たい。）

(2) データが不要に成るまでプロセスを終結させない。

(3) クラス・スロットに保存する。

以上の内、(1) は従来からの一般的な使用法であり、説明は省略する。

(2) はSIMPOSのストリームを利用して他からの再始動を待つ事が出来る。但し、リソースを占有しながら数時間以上待つ恐れもあり、この例での適用は好ましくない。

(3) はプロセスの終結の直前で、各インスタンスがクラス・メソッド（例えば、:savef#cai, …））によりインスタンス・スロットからクラス・スロットにSaveして置き、次回のインスタンス生成時に、クラス・オブジェクトが:create のafter demon でそのインスタンスのインスタンス・スロットにRestore する事で実現出来る。

但し、(2),(3) はシステムを"shut down" するまでの間のみのデータ保存である。

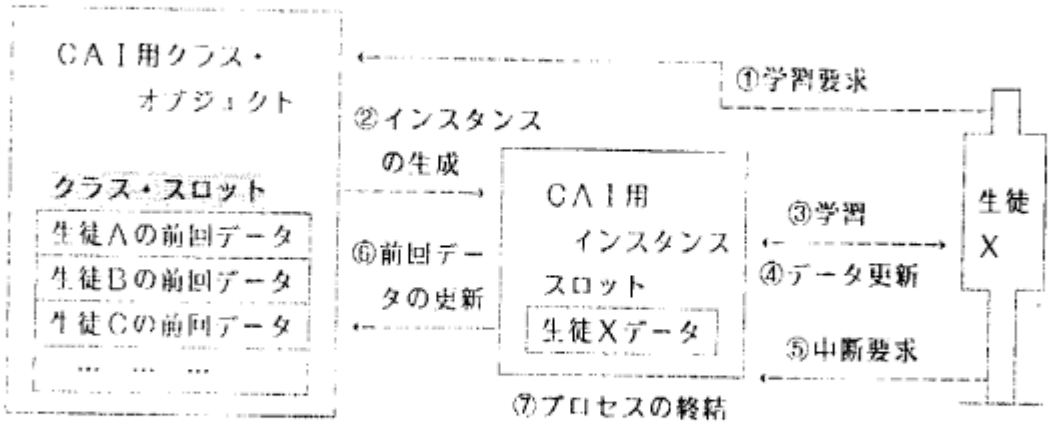


図　6－5　クラス・スロットによるデータ保存

7. おわりに

オブジェクト指向プログラミングに関する解説が、一般の目に触れる機会が多く成ってきている。しかし、人工知能研究の知識表現の手段として生立ちの影響かも知れないが、理解を促進すべき例示は、現実のコンピュータ・プログラムに結び付かないオブジェクト（クラス動物を継承したクラス鳥の如く）と成りやすい。

本報告でも、止むなく鬼を登場させたが、極力コンピュータ・プログラムでの解説に心掛け、読者が各自の現実のプログラミング環境（言語，ＯＳなど）と対応付けて理解される事を期待している。

オブジェクト指向プログラミングは、複雑な問題解決のプログラムに於ても、オブジェクトの独立性が保たれ互いに干渉する不安が無く、ソフトウェアの品質と生産性が高い技法と言えるが、その半面プログラム実行時の処理性が犠牲に成り易い。
ＥＳＰも例外ではなく、従ってインプリメント時のオプティマイズの徹底，ファームウェア化などの処理性改善は勿論の事、言語企画段階からこの面を考慮し、数値，文字（列）などをオブジェクトとせず、従来通りのデータとして扱う事でオブジェクト指向と処理性のトレードオフを計っていると言えよう。

本報告の言語仕様は、下記の研究報告より一般ユーザ向けの範囲を抜粋したものであり、仕様の全容，言語の思想などを知りたい方は下記の研究報告を参照されたい。

◇T.Chikayama,"ESP Referenve Manual", ICOT TR-004, (Feb. 1984)

本報告は、取り敢えず見て頂く為の初版であり、今後とも一層解りやすく、また内容の充実等の改訂，増補を計りたいと考えている。
読者各位が、ご意見，解りにくい箇所や改善箇所のご指摘あるいは、お気付きの点を筆者まで連絡頂ければ幸いである。

索 引

図