ICOT Technical Memorandum: TM-0146

TM-0146

ESPプログラミング・ヒント集

上田尚純、近藤誠一
(三菱電機)

近山　隆

November, 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プログラミング・ヒント集

はじめに

SIMPOSのシステム記述言語ＥＳＰ(Extended Self_contained Prolog)でプログラミングする際のいくつかのヒント，注意事項等について記述してある。ＥＳＰは論理型言語 Prolog をベースとしたオブジェクト指向言語であり，通常の手続型言語と相当異なった面を持つ言語である。また，実行速度，所要メモリ量の点でオーバヘッドの大きな言語であり，プログラム作成にあたってこれらオーバヘッドを最小にするよう努める必要がある。

本説明書は，Prolog，ＥＳＰの言語仕様を一通り理解したプログラマを対象に，本格的なプログラムの作成にあたって心得ておくべきＥＳＰの特徴，プログラミング技法を紹介するためのものである。次のような項目について述べてある。

（ｉ） ＥＳＰのいくつかの特徴
（ii） 基本的機能のプログラミング手法
（iii） 効率よいプログラムを作成するためのプログラミング手法

（ii），（iii）はいわばＥＳＰプログラミングでの定石とも言える技法であり，これらを充分にマスタしておく事が肝心である。

目　　次

1. ＥＳＰの特徴について

1.1 プログラムの静的／動的な特徴

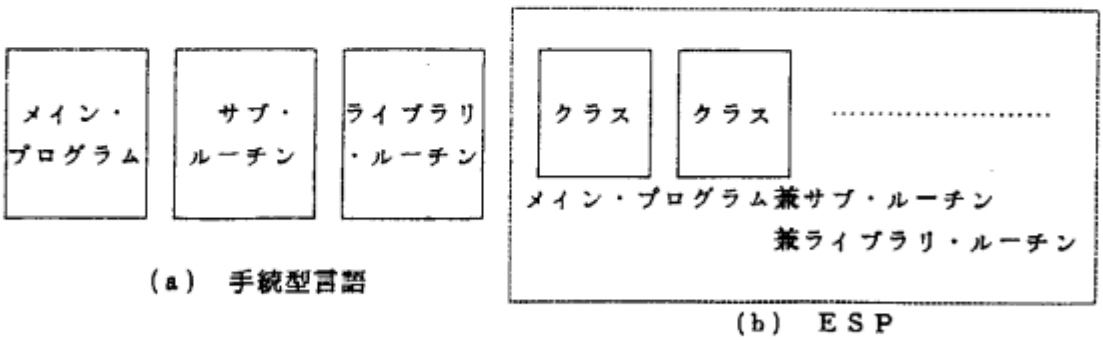
ＥＳＰで記述されたプログラムは，FORTRAN，PASCAL 等の手続型言語と比較してプログラミング・スタイル，プログラム実行時の振舞い，プログラムのシステムによる管理方法等の面で，かなり異なった面を持つ。これらは主としてＥＳＰがオブジェクト指向言語である事から来ており，プログラマの立場からは便利な点が多い。これら特徴的な点について説明する。

(1) プログラム構造及びプログラム間インタフェースの均一性

ＥＳＰでは全てのプログラムをクラスという統一した枠組で記述する。また，クラスとして定義されたプログラム（オブジェクト）間のインタフェースはメソッド・コール（メソッド呼出し）という，やはり統一したインタフェースで記述される。これにより，ＥＳＰプログラムはメイン・プログラム，サブ・ルーチン，ライブラリ・ルーチンといった差異はなく，いずれにしても使用できる。すなわち，メイン・プログラムとして作成したものも，サブ・ルーチンとして他のプログラムから呼出したり，あるいはライブラリ・ルーチンとして使用したりする事が自由にできる。

手続型言語では，メイン・プログラム，サブ・ルーチン，ライブラリ・ルーチン等は一般的に相互に異なる属性を持ち，ソース・プログラム上での記述形式，ロード・モジュールの形式，システム内での管理方法等で多少異なる場合が多い。従ってメイン・プログラムとして記述したものをサブ・ルーチンとして使用するのは不可能であり，あるいはライブラリ・ルーチンの作成は特別な注意や手続きが必要であったりする。

ＥＳＰでは，クラスとして作成されたプログラムは，ライブラリと称されるシステム管理機能により統一的に管理され，均一な扱いを受ける。全てのクラスはライブラリに登録され，プログラマは登録されたクラスを自由に自分のプログラムから呼び出して使用できる。SIMPOSを構成しているクラスもライブラリに登録されており，論理的に正しい使い方さえすれば，プログラマはシステム機能を自由に使用できる。これ故，ＥＳＰで統一して記述されている SIMPOS は，その全機能をユーザに提供しているオープンなシステムとなっている。前述したように，プログラム間インタフェースはメソッド・コールで統一されており，ユーザ定義プログラムを利用するのもシステム機能を利用するのもソース・プログラム上での記述形式は同じでよい。



(a) 手続型言語　　(b) ＥＳＰ

図１－１　手続型言語とＥＳＰのプログラムの種類の比較

(2) 各種機能のオブジェクトとしての提供

FORTRAN, COBOL等の手続型言語では，データ定義機能，入出力機能等，プログラム記述に必要となる機能を言語仕様の枠組の内で提供している。より新しい手続型言語Cでは，入出力機能はライブラリ・ルーチンとして，言語仕様の枠組の外で提供している。言語仕様からはずして外付きで機能を提供する事で，機能の変更，追加等をより容易に行なう事ができ，言語としては機能拡張が容易となる。

ＥＳＰでは，この思想を徹底させている。言語仕様レベルではクラス定義を記述する機能だけを提供している。副作用つきの構造型データ，入出力機能等は，オブジェクトとして言語の枠外で提供されている。これらオブジェクトはやはりＥＳＰでクラスとして定義されており，システム作成者から提供される。オブジェクトとして提供される代表的な機能として次のようなものがある。

- 副作用付き構造型データ（プール）
  - 配列
  - リスト
  - ハッシュ表
  - その他

- 入出力機能
  - ウィンド操作
  - ファイル操作
  - 書式付出力
  - その他

- プロセス機能
  - プロセスの生成，制御
  - プロセス間の同期，交信機能（ストリーム）

これらシステムから提供される機能以外に，ユーザがこれらを拡張したり，あるいはユーザが定義した機能を使用する事も可能である。前述したように，クラスは全て均一の構造を持つため，ユーザが定義したクラスは自動的にプログラムから使用できる事になる。

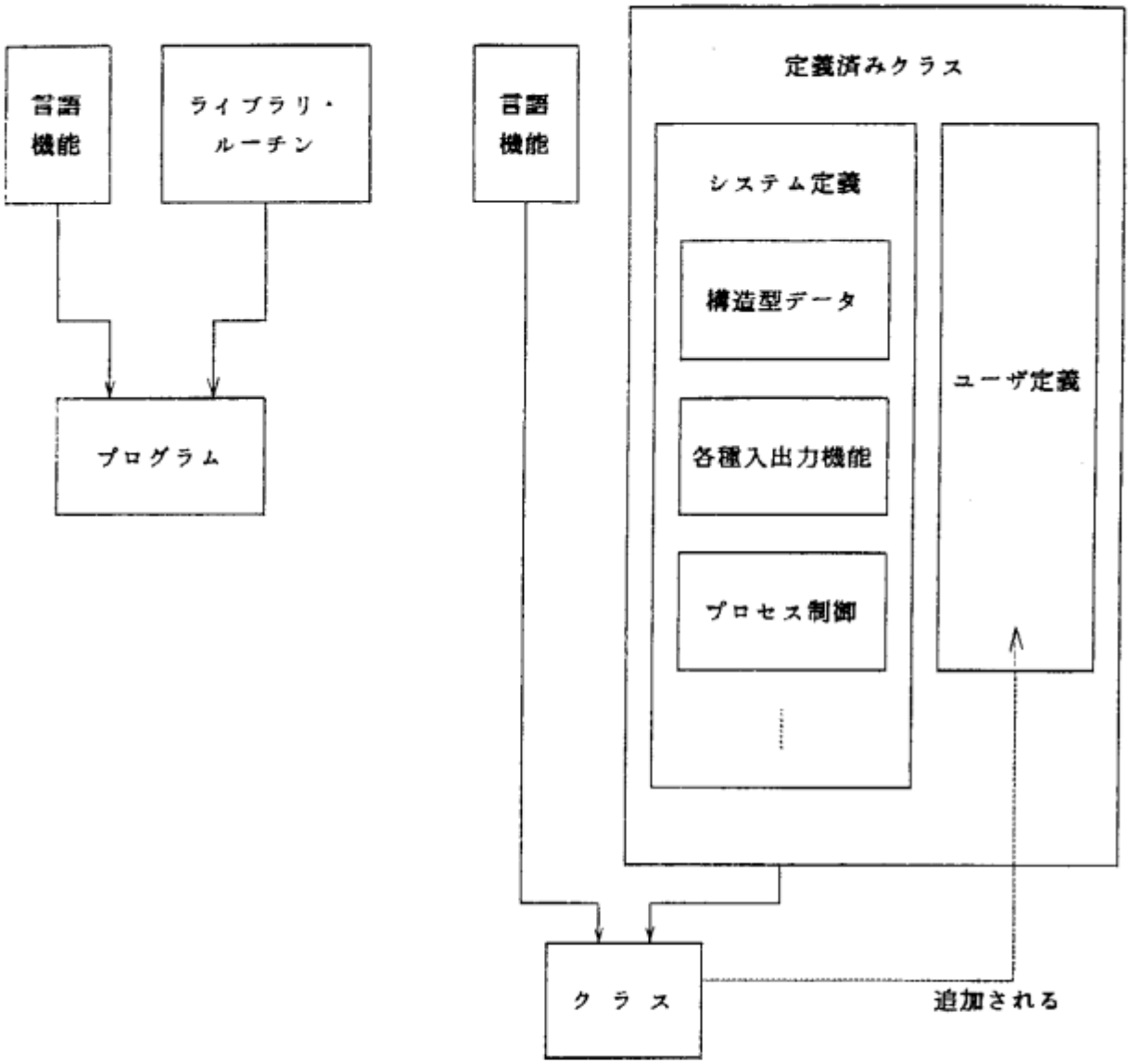


図１－２　プログラム記述時の使用機能の比較

(3)　メモリ領域の動的使用

手続型言語ではデータで使用するメモリ領域は静的な使用が基本である。プログラムで使用するデータの領域や型はソース・プログラム上にデータ定義文として記述しておく必要がある。

これに対して，ＥＳＰではデータ領域は動的に使用するのが基本である。LISP，PROLOGといった人工知能用言語では再帰的なプログラムの記述，リストの操作記述といったものを容易に記述でき，これが言語としての記述能力の高さを支える機能の一部となっているが，これら機能の実現にはデータ領域の動的使用が不可欠である。ＥＳＰは PRO

LOG の機能を内合している訳であり，メモリ領域の動的使用は当然と言える。更に，ＥＳＰはオブジェクト指向言語であり，これらはメモリ領域の動的使用を必須なものとしている。オブジェクト指向言語で記述されたプログラムの実行では，クラスから動的にオブジェクトをインスタンスとして生成していく事になり，オブジェクトの領域が動的に確保されていく事になる。

以上のように，ＥＳＰでは動的なメモリ領域の使用が基本であり，これがＥＳＰの強力な言語機能を支えていると言える。反面，メモリ領域を動的に確保するため実行速度が低下する事，使用済みメモリ領域を回収するための，ガーベッジ・コレクション(GC)が必要な事等のマイナス面もあるが，これらはハードウェア，ソフトウェア面の技術進歩にわり年々緩和されてきているのが実情である。

(4) 動的リンケージ

手続型言語ではプログラム間の結合は静的である。リンケージ・エディタと称されるユーティリティ・プログラムを使用して，複数のコンパイル単位モジュールを１ケの実行プログラムとして結合する。これにより，モジュール間の結合（リンケージ）はプログラムの実行に先だち，静的に行なわれた事になる。実行プログラムの一部モジュールを変更したい場合，再リンクが必要となる。

ＥＳＰでは，これに対して，オブジェクト間のリンケージは動的（すなわち，実行時に行なわれる）である。オブジェクトは動的に生成される事，メッセージを送る相手を動的に指定できる事等の理由により，ＥＳＰのようなオブジェクト指向言語はモジュール間（すなわちオブジェクト間）のリンケージは動的である事が必然となる。プログラマはクラス単位でオブジェクト・コードを生成しておけばよく，手続型言語におけるリンケージ・エディタは存在しない。（厳密に言えば，ＥＳＰでは部分的に存在するが，ユーザは意識しなくてよい）ＥＳＰにおいてクラス定義を変更したい場合，あるいはクラスを新規に追加したい場合，そのクラス定義をコンパイルしてシステムに追加するだけでよい。

このように，モジュール間のリンケージが動的になされる事は，プログラムの記述やプログラム管理の面で種々の便宜を得る事ができ，オブジェクト指向言語の利点の１つである。反面，プログラム実行速度は低下するが，前述したように技術的に緩和されつつある。

(5) プログラム管理システム（ライブラリ）の存在

従来の手続型言語をベースとするシステムでは，ソース・プログラムやオブジェクト・モジュール，ロード・モジュール等はファイル・システムの機能を利用して管理しており，またソース／オブジェクト／ロード・モジュール相互間の関係づけを管理するシステムは存在しない。

ＥＳＰでは，クラス継承機能や動的リンケージ機能を実現するために，システム中に存在する全クラスのソース・プログラム，オブジェクト・コードを管理するシステムが必要となる。SIMPOSはこれをライブラリと呼んでいる。ライブラリではクラス定義のソース・プログラム，オブジェクト・モジュール(クラス・オブジェクト)を一括管理し，クラスの登録，削除等をユーザの要求に応じて実行する。

1.2 変数の特性

(1) ＥＳＰの変数

ＥＳＰは Prolog にオブジェクト機能を導入した言語であり，変数として次の２つがある。

- Prologの論理変数
- スロット (attribute, component)

論理変数は，ユニフィケーションにより値を持つ事，バック・トラックにより値が解放される事等，いわゆる副作用を持たない変数である。これに対して，スロットは手続型言語でいう所の変数と同じ性質を持つ，いわゆる物理変数であり，値の変更を任意に行なってよく，またバックトラックで値が戻される事もない。すなわち，スロットは副作用を持つ変数である。

論理変数とスロットに共通している事は，いずれもデータ・タイプを持たない変数であり，任意のデータ・タイプの値を持てる事である。整数，浮動小数点数等は勿論，文字列，配列，リスト等の構造体もその値として持てる。（ただし，スロットでは，スタック・ベクタ等のスタック上で保持されるデータ及び未定義の変数は値として持てない）これは，ＥＳＰ（及びProlog）の強力なプログラム記述を支える機能の１つとなっている。　スロットを参照する場合は，オブジェクトとスロット名の対で指定する必要がある。あるクラスのインスタンス・オブジェクトは一般に複数個生成され，各インスタンスは同一名称のスロットを持つ事になる。このため，どのオブジェクトのスロットかを指定せねばならない。

表１－１に論理変数，スロット，手続型言語の変数の比較表を示す。

表１－１　論理変数，スロット，手続型言語の変数の比較

変数 / 比較項目	ＥＳＰ		手続型言語の変数
	論理変数	スロット	
変数の定義	定義文なし クローズの引数中に直接記述	定義文で行なう データ・タイプの指定なし	定義文で行なう データ・タイプの指定要
データ・タイプ	任意のデータ・タイプ	任意のデータ・タイプ（但しスタック・ベクタは不可）	定義文で定義したデータ・タイプの値のみ持つ
初期値	未定義状態 値のユニファイが可能な状態	整数　0	コンパイラに依存
値の代入	ユニフィケーション	代入文	
副作用	なし バックトラックで値を解放	あり	
変数の指定方法	変数名	オブジェクト＋変数名	変数名

(2) 属性スロットと要素スロット

属性スロット(attribute slot)と要素スロット(component slot)の差異について述べる。差異の１つは，属性スロットは手続型言語のグローバル変数に相当し，クラス定義の外から参照する事が可能であるのに対して，要素スロットはローカル変数に相当し，クラス定義の外からは参照できない。

属性スロット　：グローバル変数
要素スロット　：ローカル変数

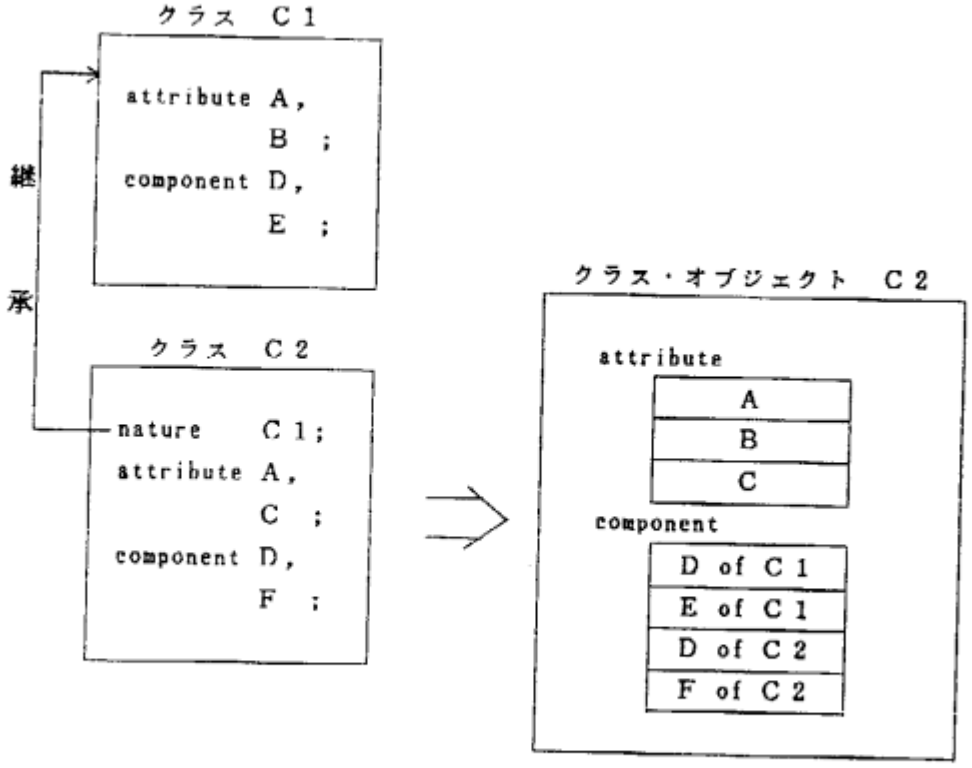
もう１つの差異は，上記の差異とも関係するが，クラス継承時の取扱いの差異である。継承するクラス定義と継承されるクラス定義の間で同じ名称の変数があった場合は，クラス・オブジェクトでは次のようになる。

属性スロット　：縮退して１つの変数（両クラスで共用）
要素スロット　：縮退せず，別々の変数のまま

以下の（例）で具体的に示す。（例）において，クラス定義Ｃ１の要素スロットＤとクラス定義Ｃ２の要素スロットＤとは全く別のものとして扱かわれる。（ＥＳＰコンパイラでは異なる名称の要素スロットとして出力する）

スロットを属性スロットとして定義するか要素スロットとして定義するかは，上記の差異を考慮して決定する必要がある。

（例）



attribute Aはクラス・オブジェクトＣ２作成時，１個に縮退する。

クラスＣ１におけるスロットＤへのアクセスは，無条件にクラスＣ１で定義したcomponent Ｄへのアクセスとなる。同様に，クラスＣ２におけるスロットＤへのアクセスは，無条件にクラスＣ２内のcomponent Ｄへのアクセスとなる。

（注）・ 他のクラスのオブジェクト内の属性スロットを参照し，そのスロットと同じ名称が要素スロットとして自クラス定義内で定義されている場合は，ＥＳＰコンパイラは無条件にその要素スロットへの参照とみなす。従って，他のクラスのオブジェクトの属性スロットを参照する場合において，その属性スロットと同じ名称の要素スロットを自クラス定義内で定義してはいけない。

・ 手続型言語では，他プログラム中の(グローバル定義されている)変数を参照する場合，自プログラム内でその変数名を外部参照宣言しておく必要がある。（宣言がないと未定義変数となる。）これに対してＥＳＰでは外部参照の宣言は不要な言語仕様となっている。

・ 属性スロットはクラス定義の外からアクセスできるが，情報隠蔽というオブジェクト指向言語の特徴をこわす機能となっている。従って，スロット参照用のメソッドを被参照側オブジェクトで定義し，参照側オブジェクトはそのメソッドを使用してスロットにアクセスする方法が望ましい。

1. 3　メソッド・コール

メソッド・コールは，次の３つのＡＮＤ結合により構成される。

(1)　before demon述語のＡＮＤ結合

全ての継承されたクラスのbefore demonを継承順にＡＮＤで並べたもの。

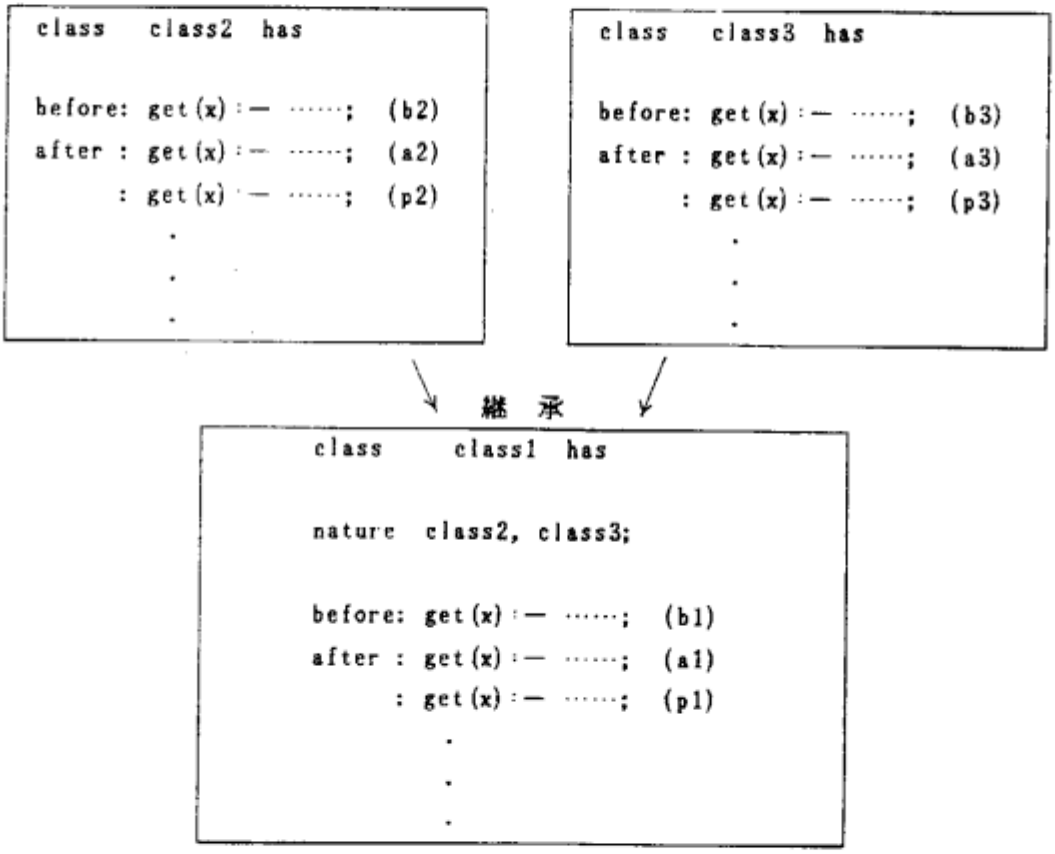
(2)　primary述語のＯＲ結合

全ての継承されたクラスのprimaryを継承順にＯＲで並べたもの。

(3)　after demon述語のＡＮＤ結合

全ての継承されたクラスのafter demonを継承の逆順にＡＮＤで並べたもの。

［例1. 1］



(a1) (p1)……などは，各クラス内の述語（同一述語名，同一個数の引数を持つクローズ群）を指す。

上の例の場合，クラス｀class1´でのメソッド｀: get(x) ´の構成は次のようになる。

```
: get(x): -
        b1.b2.b3  (p1;p2;p3;fail), a3.a2.a1;
before demon      primaryの      after demon
のＡＮＤ          ＯＲ           のＡＮＤ
```

以下の条件を満足するときは最適化が行なわれる。

① prmary述語がひとつの場合（すなわち，継承クラスに同一メソッドがなく，しかもデーモンがない場合。多くの場合は，この条件にあてはまる。）
上記のようなメソッドは生成されず，そのprimaryが直接，メソッドとなる。

② before及びprimaryがない場合（すなわち，afterのみの場合）は，単にfailする。
（afterのみのメソッドは許されない事になる。）

(注意事項)

・ ローカル述語（すなわちPrologの述語）コールの場合は，コンパイラにより呼出しのためのリンケージがつけられるにの対し，メソッド・コールの場合は動的にリンケージをつける。（理由は，一般にメッセージの送り先きは実行時に決まるため）この動的リンケージのため，クラス・オブジェクト毎に用意されているメソッド表（メソッド名と，コードの対応表）を探索する。この分だけメソッド・コールはローカル述語コールより余分に処理時間がかかる。(現状ではメソッド・コールはローカル述語コールと較べて倍ほど遅い)

・ メソッド表の探索を経由して到達するメソッド述語はＫＬＯレベルではローカル述語と同じ形式である。従って，ユニフィケーション，バック・トラック，cut ，等の機能はメソッド述語とローカル述語とはほとんど差異はない。但し，次の事に注意が必要である。

  ・ primary内の！（カット）はコンパイラによりすべてrelative－cut(1)に変換される。したがって，ＯＲクローズ内の他のprimaryにも！（カット）が及ぶ。しかし，①のように最適化した場合も不都合が生じないように，method－callはすべてレベル２上がる。demon内の！（カット）は通常の１レベルカットであがるので，同一クラス内のオルタナティブをカットするのみである。
  ・ before demonはＡＮＤ結合であるので，failした場合，primaryは実行されない。また，途中の述語中に引数にユニファィするようなものがあった場合，その結果は，後の述語呼び出しに引き渡される。

1.4 継　承

継承の定義部では，継承されるクラス及びその順序を記述する。

［例1.2］

```
nature      class1  ,   class2;
nature      class1  , ＊  class2;
```

継承順は，以下のようにして決定される。

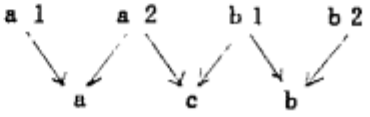
(1) 継承の定義がない場合は，自クラスそれ自体が唯一の継承されるクラスとなる。

(2) 継承の定義がある場合，自クラス，及びそのクラスが継承しているクラスが，継承クラスとなる。記述の順が継承順となる。自身のクラスは＇＊＇で表わす。省略時は，先頭にあるものとみなされる。

(3) 重複がある場合，最初に現われたものが有効となる。

［例1.3］

クラスaがクラスa1，a2をクラスaがクラスa2，b1を，クラスbがb1，b2を継承しているものとする。



(1)
```
class    ex1    has
    nature    a, b;
```
このとき継承順は，

ex1 → a → a1 → a2 → b → b1 → b2

（a→a1→a2：aの継承，b→b1→b2：bの継承）

となる。

(2)
```
class    ex2    has
    nature    a, c;
```
このとき継承順は，

ex2 → a → a1 → a2 → c → b1

となる。

(3)
```
class    ex3    has
    nature    a, ＊ c;
```
このとき継承順は，

a → a1 → a2 → ex3 → c → b1

となる。

以上のように，depth－first（深さ優先）の順序となる。

2. 基本機能の記述方法

Prologあるいは ＥＳＰといった言語は手続型言語とかなり記述形式が異なった言語であるが，手続型言語と同様な機能の記述は可能である。ある程度以上の規模を持つシステム・プログラムを作成するには，手続き的なプログラム記述が楽であるし，また必須でもある。以下で，各種の手続き的な基本機能を実現するための手法について説明する。

2.1 基本事項について

(1) クローズの標準形式

Prologのクローズの標準形式は，次のような形式となるのが一般的である。

$$\underbrace{\underbrace{P : -C_1, C_2, \cdots, C_m}_{\text{Head}}}_{\text{選択部}}, \underbrace{[!], \underbrace{G_1, G_2, \cdots, G_n}_{\text{実行ゴール列}}}_{\text{実行部}}$$

（条件ゴール列）

クローズは選択部と実行部とからなり，選択部はヘッドと条件ゴール列（条件ゴールは空の場合もある）とで構成される。選択部が成功すると（ヘッドの場合はユニフィケーションの成功，条件ゴール列は各ゴールの成功），実行部のゴール列の実行に移る。選択が決定的でよいのなら，選択部の次に ！(cut)を入れる。

Prologではこの標準形式以外の形式で記述する事も自由であるが，上記の形式は言わば Prolog でプログラムを記述する際の`定石´であり，このパターンのクローズの組合わせでプログラムを記述する事により，分かり易く且つ効率のよいプログラムの実現が可能となる。 ！ を２ヶ以上含むクローズは，以下に示すように標準形式の組合わせで容易に実現できる。

```
P : −C1, C2, !, C3, C4, !, G1, G2

P : −C1, C2, !, Q
Q : −C3, C4, ., G1, G2
```

なお， ！ の代表的な使い方として次のようなものがある。

(i) クローズの選択を一意にする。
(ii) 非決定的(non-deterministic) な述語の実行を一意にする。
(iii) failと組合せて否定を実現する。

これらの用途の具体例は以下で示されている。

(注)　クローズ内 OR を使用する場合，その OR 内で ！ を使用する事はよくある。この際はクローズ全体で見れば複数ヶの ！ が使用される場合が出てくるが，これは標準形式の組合せの短縮記述形式と考えられ，むしろ分かりやすくなるのでさしつかえない。

$P : - (C_1, !, G_2; C_2, !, G_2, ; G_3), C_3, !, G_4;$

$P : - Q, C_3, !, G_4;$
$Q : - C_1, !, G_1;$
$Q : - C_2, !, G_2;$
$Q : - G_3;$

(2)　クローズ(述語)の決定的終了について

効率のよいプログラムを作成する上で大事となる，クローズの決定的終了について説明する。

クローズが何の選択枝（アルタネーティブ）も残さずに終了する場合，クローズは決定的に終了したと呼ぶ。別の言葉でいうと，クローズのボディ部の最後のゴールの実行を終えてクローズを抜け出る時点で，クローズ自体及びボディ部のいずれのゴールに対しても選択枝が残っていない場合である。より具体的には，クローズのボディ部の最後のゴール終了時点で次の(ｉ)，(ii)の条件を共に満たしている場合である。（以下で，プレディケートとは同じ述語名と同じ個数の引数を持つクローズ群を指す。）

(ｉ)　クローズがプレディケートの最後に実行するクローズである。
次の２つのケースがある。

（ケース１）　クローズがプレディケートの最後の位置にある。

```
P : - … ;
P : - … ;
P : - … ; ←このクローズを実行
Q : - … ;
```

（ケース２）　クローズがプレディケートの末尾でなくても，！によりそれ以降のクローズが cutされた。

```
P : - …          ;
P : - …, !, …    ; ←このクローズを実行
P : - …          ;
Q : - …          ;
```

(ii)　ボディ部の全てのゴールに対して選択枝が残っていない。
次のようなケースがある。

（ケース１）　P : －Q, R, S ;
Pが(ｉ)の条件を満たし，且つQ，R，Sのいずれにも選択枝が残っていなければ，決定的に終了する。

（ケース２）　Ｐ：－Ｑ，Ｒ，！，Ｓ，　；
Ｓが選択枝を残していなければ，決定的に終了する。　Ｑ，Ｒの選択枝は！で cutされる。

(ii)については，決定的終了という定義を recursiveに使用して，次のように言い換える事ができる。

クローズ自体に選択枝がなく，且つボディ部の全てのゴールが決定的に終了した場合，クローズは決定的に終了する。

上記(i)，(ii)を満足してクローズが終了した場合，クローズは決定的に終了する事になるが，プレディケートも決定的に終了した事になる。従って両者は同じ事になる。（プレディケートＰにおいて，ｎ番目のクローズＰｎが決定的に終了したとする。クローズＰｎが呼ばれるのは，クローズＰ１～Ｐｎ-1に関連する全ての選択枝を実行した後である。従って，Ｐｎ に関連する選択枝が残っていなければ，プレディケートＰに関する選択枝はない事が保証されている。）
以下に例を挙げて説明する。次のような例を考える。

[例2. 1]

```
P1 : －Q  ;
P2 : －!, R, S      ;

P3 ;
R1 : －!, T      ;
R2 ;

S1 : －U  ;
S2      ;

T1 : －U  ;
T2      ;

U1      ;
U2      ;

Q1  ;
Q2  ;
```

Ｐｎのｎは各クローズを識別するために便宜上つけた数字であり，クローズとしての述語名はＰである。他も同じである。↑は各プレディケートにおける実行位置を示す記号として使用してある。

P_2においてゴールSの実行が終了した時点での，クローズP_2の決定的終了について考える。P_2のボディ部に！があるため，！以降のゴールR，Sについて選択枝があるかどうかを考えればよい。Rについては，R_1が実行されているが，ゴール部に！があるので，Tが決定的に終了しておれば，R_1も決定的に終了となる。Tは，T_2が決定的に終了している。よって，R_1も決定的終了となる。Sは，S_2が決定的終了となっている。以上により，P_2は決定的に終了した時になる。

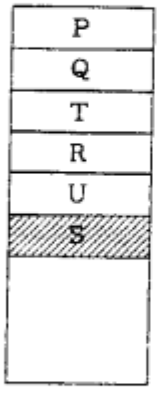
クローズの決定的終了が重要なのは，プログラム実行時にシステム側の言語処理系で最適化処理を施こす事により，メモリ使用量の節減がはかれるからである。Prologの実行処理はスタックを使用してなされ，ゴールからの述語呼出し毎に，呼出されたプレディケートに対してスタック上に作業領域が確保される。この作業領域は，そのプレディケートが決定的に終了した時点で解放が可能となる。

例をあげて説明する。次の例において，クローズPのゴールSをこれから実行しようとしている所とする。ゴール実行までに呼ばれた述語のうち，プレディケートT，R，Uは決定的に終了しているため，これらの領域は既に消去され，スタックが有効に利用されている事がわかる。このように，決定的に終了したプレディケートのスタック内作業領域は，決定的に終了した時点でたたまれていく。従って，各プレディケート（クローズ）を，プレディケート（クローズ）単位で，できるだけ決定的に記述する事は，プログラム実行時のオーバヘッドを少なくするために重要である。

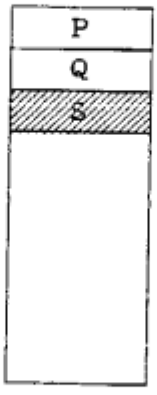
［例2. 2］

```
P: −Q, R, S:
          ↑
Q: −T   ;
Q   ;
R: −U   ;
R↑  ;
T↑  ;
U↑  ;
```

クローズPのゴールSをこれから実行しようとしている所とする。



（i） 最適なしの場合



（ii） 最適化を行なう場合
T，R，Uは決定的に終了しているので，対応する領域は消されている。

(3) ＴＲＯ (Tail Recursion Optimization)

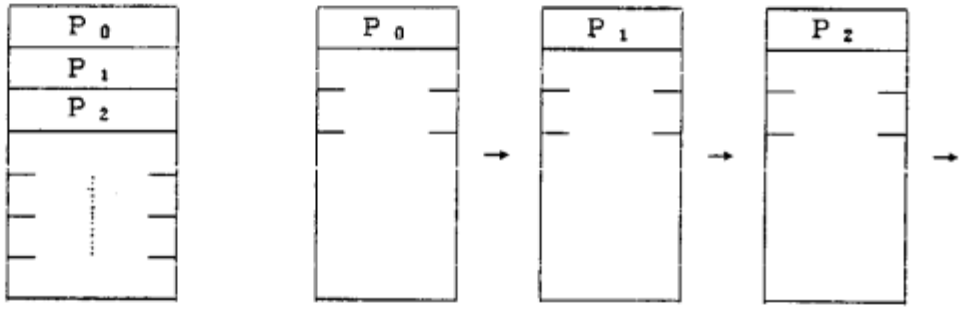
Prologではクローズのボディ部から自分自身を呼び出せる。これを再帰呼出し (recursion あるいは recursive call) と呼ぶが，このうち，自分自身を呼び出すゴールがボディ部の最後にあるものを末尾再帰呼出し (Tail Recursion) と称する。この末尾再帰呼出しにおいて，そのクローズ（プレディケート）が決定的に終了する場合，そのクローズのスタック領域を解放できるため，メモリ量を節約できる。これを特にＴＲＯ (Tail Recursion Optimization) と呼ぶ。

［例2. 3］

ＴＲＯの例　　Ｐが末尾再帰呼出しである。

```
P;
P: −R, S, P;
```

この例において，Ｒ，Ｓに選択枝がなければ，ＴＲＯの適用が可能となる。



（ｉ）　ＴＲＯを行なわない場合
再帰呼出し毎にスタック領域は延びる。

（ii）　ＴＲＯを行なう場合
スタック領域は延びない。

PrologではＴＲＯを利用して，手続型言語におけるループ制御を効率よく実現できる。Prologにおいては非常に利用度の高い手法である。

2.2 実行制御の実現方法

Prologでは GO TO, CALL, IF THEN ELSE, DO, WHILE 等の実行制御命令は明白には存在しない。しかし，Prologのバックトラック，cut ，再帰呼出し等の機能を組合せる事で，これら実行制御機能を実現できる。

実現手法は次の通りである。

実行制御機能	Prolog での実現方法
無条件分岐 (GO TO) ……	な　し
サブルーチン呼出し (CALL) ……	ゴール呼出しがそのまま対応
条件分岐 (IF) ……	バック・トラックと cutを利用して実現
多方向分岐 (CASE)	
繰返し (DO, WHILE等) ……	バック・トラック，cut ，再帰呼出しを利用して実現

GO TO 文に相当する記述は Prolog ではできない。構造化プログラミングでは GO TO文はその使用が好ましくないと排斥されている機能であり，GO TO 機能がなくてもプログラム作成は可能である。Prologでは GO TO文がないため，構造化プログラミングにそった良いプログラムしか記述できない。

サブルーチン呼出し (CALL)については，クローズのボディ部からのゴール呼出しはCALLと同じである。P: －Q，RはP: －call(Q)，call(R)の意味であり，CALL機能はそれと意識されずに使用されている事になる。但し，手続型言語のCALL文と異なり，ヘッド部のユニフィケーションが成功した場合のみ呼び出したクローズの本体部が実行される事，バックトラックがなされる事等の差異はある。

条件分岐，多方向分岐，繰返しの実現方法は以下の通りである。

(1) 条件分岐 (IF)

条件分岐は cutとバックトラック機能を使用して実現する。以下に例を示す。なお，Prologでは，これら条件分岐を実行するプレディケートを独立に定義し，それを必要とするボディ部から呼び出す形式とする事が必要な事に注意の事。（以下の(1)でこれを示している。）

[例2.4]

	手続型言語	Prolog
(1)	if P then S_1 else S_2.	R: － … , Q, … Q: －P, !, S_1 ; Q: －S_2 ;

S_2 の前に！を入れてもよいが，省略しても同じ。以下の例も同様

	手続型言語		Prolog
(2)	if P_1 then S_1 else if P_2 then S_2 else if P_3 then S_3 else S_4	→	Q: $-P_1$, !, S_1 ; Q: $-P_2$, !, S_2 ; Q: $-P_3$, !, S_3 ; Q: $-S_4$;
(3)	if P_1 then if P_{11} then S_{11} else S_{12} else if P_2 then if P_{21} then S_{21} else S_{22} else S_3.	→	Q: $-P_1$, !, Q_1 ; Q: $-P_2$, !, Q_2 ; Q: $-S_3$ Q_1: $-P_{11}$, !, S_{11} ; Q_1: $-S_{12}$; Q_2: $-P_{21}$, !, S_{21} ; Q_2: $-S_{22}$;

(2) 多方向分岐(CASE)

多方向分岐は条件分岐(IF)の組合せでも実現できるが，Prologではヘッド部のユニフィケーション機能を利用してスマートに実現できる。

[例2.5]

手続型言語	Prolog
case X is when C_1 → S_1 when C_2 → S_2 when C_3 → S_3 when others → S_4 (C_1, C_2, C_3は定数)	Q: − … , P(X, …), … P(C_1, …): −!, S_1 ; P(C_2, …): −!, S_2 ; P(C_3, …): −!, S_3 ; P(___, …): $-S_4$;

Prolog のこの形式を使用して多方向分岐を記述すると，後述する clause_indexing と称される最適化機構により，高速で多方向分岐を実行する事ができる。

(3) 繰返し

繰返しの代表的な例として，2つの場合がある。1つは FORTRANの DO ループに相当するもので，指定した回数を繰返し実行するものである。もう1つは，ファイル操作におけるレコードの read/write のように，ある終了条件を検出するまで繰返しを行なうものである。

繰り返しをPrologで表現する場合，一般に２つの方法が考えられる。

(1) Tail recursionを使用して再帰的に行なう。

[例2.6]

```
DO    I＝K    to    N

END
```

```
……, P (N－K＋1, K, ……) ……
P (0, _, ……) : －!;
P (I, K, ……) : －
        P (I－1, K＋1, ……) ;
```

(2) オルタナティブを用いてfailで回す。

failで回す場合(1)と比較して，

利　点：スタックが伸びない

欠　点：自由なcontrolが難しい。結果を返しにくい。

という特性を持ち，これを考慮して選択することが望ましい。

基本的には，(1)を使用し，大きいループ（例えば一番外側）は(2)を使用するといった考え方が一般的と思われる。

[例2.7]

ＤＯループ

指定した制御変数をＩからＪまでＫ刻みで増加させながら，実行する。

（その１）　Tail recursionを使用

```
r: －  ……  , φ (1, 10, 3, ……)
p (I, J, _, ……) : －I＞J, !;
p (IO, J, K, …) : －
```

（その２）　failによる繰り返し

```
r: －  ……  , p (1, 10, 3, ……) , ……
p (I, J, K, ……) : －for (I, J, K) , S1 , S2, …, fail;

p (_, _, _, ……) ;                    実行本体部
for (I, J, K) ;
for (I, J, K) : －for (I＋J, J＋K) ;
```

［例2. 8］

終了条件検出までのループ

（その１）　tail recursionを使用。

```
p: - stop, !;
```

終了条件

```
p: - S1, S2, ……, p;
```

実行本体部

（その２）　failによる繰り返し

```
p: - repeat, S1, S2, ……, stop, !;
```

実行本体部　終了判定

2.3 構造体データの操作

ＥＳＰは言語仕様としてはリスト，配列，レコード，文字列等の構造型データのデータ定義機能を提供していないが，次のような機能を利用して，実質的に強力な構造データの操作を記述できる。

(ⅰ) クローズの引数として直接記述する。
文字列，リスト，複合項，ベクタ（スタック・ベクタ）

(ⅱ) 構造データ操作用ＫＬ０組込述語の使用
文字列，ベクタ（スタック・ベクタ，ヒープ・ベクタ）

(ⅲ) POOLサブ・システムの利用

上記(ⅰ)の機能でかなりのケースをカバーできるが，これらは副作用を持たないため，副作用を持つ構造データを利用したい場合は(ⅱ)又は(ⅲ)の機能を利用する。ＰＳＩにおいては，リスト及び複合項(コンパウンド・ターム)はベクタ（一次元配列）を使用して表現しており，(ⅱ)の機能により(ⅰ)で示した構造データは全て実現できる。なお，二次元以上の配列を扱かう機能はＫＬ０組込述語ではサポートしていないため，ベクタを組合せてソフトウェアにより実現する必要がある。なお，(ⅱ)の機能の詳細については，ＫＬ０組込述語説明書を参照の事。

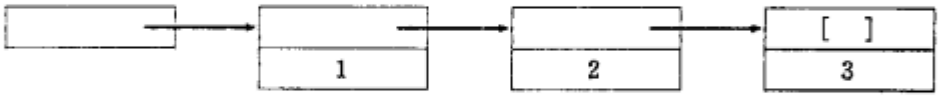
(ⅲ)の機能は，種々の構造データ操作機能を提供するクラス群よりなっており，ＥＳＰの記述機能を補完するものであり，実質的にはＥＳＰ言語の拡張機能と考えられる。POOLサブ・システムは配列，リスト，スタック，セット等の構造データをオブジェクトとして提供する。すなわち，従来の手続型言語での構造データが単なるデータ領域だけを提供するのに対して，オブジェクトでは位置指定によるデータの格納，取出し以外にも，内容を指定してのデータの取出し，格納データの情報の問合せ，ハッシュによる高速のデータ格納，取出し等の，高度の操作機能を提供しており，大幅にプログラムの生産性を高める事ができる。更に，ＥＳＰが提供する継承機能を利用して，POOLが提供するオブジェクトに機能を追加したり，修正して，プログラムが望む構造データ・オブジェクトを作成する事もできる。なお，POOLサブ・システムは内部的に(ⅱ)の機能を利用して実現されている。(ⅱ)のＫＬ０組込述語を直接使用する場合と比較して，POOLサブ・システムを使用すると一般に実行速度は低下するので，この事を念頭において使用する事。

POOLサブシステムが提供する機能の詳細は，SIMPOS使用説明書を参照の事。

2.4　リスト操作

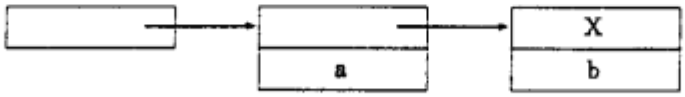
ＳＩＭＰＯＳでは，リストは，２要素のスタック・ベクタの連結で実現している。

［例2.9］　　［1，2，3］　　{{{[　]，3}，2}，1}．



注　［　］はアトム番号０のアトム

［a，b｜X］　　{{X，b}，a}



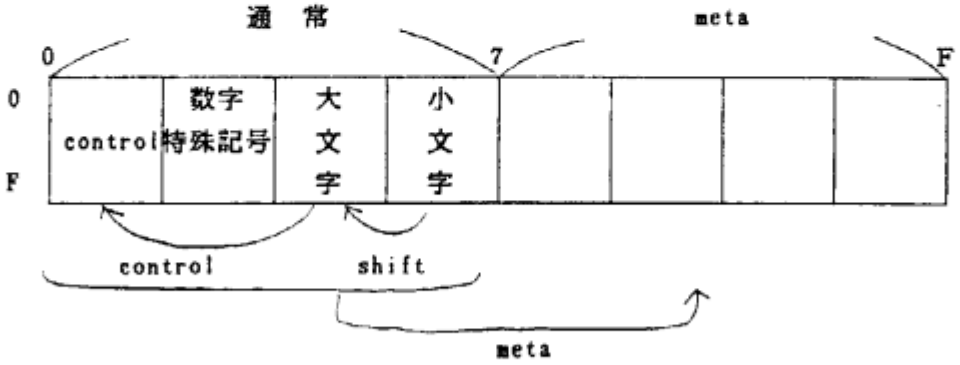
したがって，ＥＳＰのリスト操作は，リストがスタック・ベクタとユニファイできることを除いて，ＤＥＣ－10　prologと同様である。
リスト操作については，既に種々の文献があるので，ここでは省略する。

2.5　入出力操作

ＥＳＰの入出力は，すべてＳＩＭＰＯＳが提供するオブジェクトへのメソッドで実現される。以下のリソースに対する入出力が，用意されている。

(1)　ビット・マップ・ディスプレイ
　ウィンドウ・サブシステムの提供するメソッドにより，ウィンドウの生成，削除，文字り表示，描画などが可能である。

(2)　キーボード
　キーボードよりキャラクターのコードを１文字ごとに読み込む。
　コードの配置は大まかに以下のようになっている。



キャラクタはマクロで表現できる。

［例2.10］

・#" a" , control#" a" , meta#" a" , control_meta#" a"

上図からわかるように，

control#" a" ＝control#" A"
meta#" a" ≠meta#" A"
control_meta#" a" ＝control_meta#" A"
control#" 1" ……エラー，meta#" 1" ……O．K．
control# cr ……エラー，meta# cr ……O．K．

controlキーは，以下のとき使用できる

・英大文字，英小文字
・" @" , " [" , "] " , " \" , " ^" , " _"

・pfキー

pf#1，pf#2，……

・キーパッド

keypad#" 0" , keypad#" 1" , ……
keypad#" , " , keypad#" −" , keypad#" ." ……

・その他 （ ）内の数字はコード（10進数）

key#bell	(7) ,	key# bs	(8) ,
key#tab	(9) ,	key# lf	(10) ,
key# cr	(13) ,	key# esc	(27) ,
key# del	(127) ,		
key# help	(136) ,	key#about	(137) ,
key# up	(143) ,	key#down	(144) ,
key#left	(145) ,	key#right	(146) ,
key#enter	(255) ,		

各々，メタキー付（meta#del，meta#leftなど）が用意されている。
メタキー付は各々のコードと16進数80との OR をとったものである。

(3) マウス

マウス・クリックは，ウィンドウ・サブシステムの提供するメソッドにより，読み取られる。以下の6種類が存在する。

mouse#l , mouse#ll
mouse#m , mouse#mm
mouse#r , mouse#rr

また，現在のマウスの位置を読みとるメソッド，マウスのフォントを変更するメソッドが用意されている。

(4) シリアル・プリンタ
レーザ・プリンタ
２種類のプリンタが接続できる。命令は，各々，クラスprinter，クラスlaser_printerのメソッドを使用する。

(5) 固定ディスク
フロッピーディスク
ＭＴ
38MByteの固定ディスクが２台，フロッピーが２台，標準装備されている。これらは，すべて，ファイル・サブシステムが提供するメソッドにより，運用される。

上記の他に，RS232Cのハンドラが用意されているので，他システムとの接続を行なうことができる。

3. 効率よいプログラムを記述するための技法

ＥＳＰは記述能力は大きいが，反面オーバヘッドも大きい言語であり，実行速度面，メモリ容量面，更にはデバッグ面で効率のよいプログラムを作成するようプログラマ（特にシステム・プログラマ）は努力する必要がある。以下で効率よいプログラムを記述するための技法，読み易くデバッグもし易いプログラミング・スタイル等について紹介する。

3.1 プログラミング・スタイル

(1) 述語の決定的な記述

述語を決定的(deterministic)に記述する事により，次のような多くの利点が得られる。

- 実行速度が速い。
- 使用するメモリ容量を節約できる。
- デバッグが容易となる。

従って，述語単位でできる限り決定的に記述する事は大事である。

〔例　題〕　同じ仕事Ｐ(Ｘ)を与えられた回数だけ行なう述語 do_p(N, X) を作れ。

〔悪い例〕
```
do_p(0, _)  ;                          …… a
do_p(N, X) :－P(X), do_p(N-1, X);      …… b
```

この記法では，ｂのクローズを，第１引数が０の場合には選択しないということが明示されていないので，選択枝として残る。

〔良い例〕
```
do_p(0, _)  : －  !  ;                          …… c
do_p(N, X) :－〔!, 〕P(X), do_p(N-1, X);        …… d
```

ｃが選択されて実行されると，！によりｄは選択枝でなくなる。
なお，ｄの！はあってもなくても効果は同じである。

Prologでは非決定的に述語を記述する事により，同じ述語を双方向に使用する事ができる。しかし，前述したように，効率が悪い。このため，決定的に記述した述語と非決定的に記述した述語の両方を用意し，必要に応じて使い分けるとよい場合がある。

[例3.1]

述語 member は
```
member(X, [X | _]);
member(X, [_ | L])  :－member(X, L);
```

と定義すればリストの要素かどうかの判定にも，リスト要素のバックトラックに

よる生成にも使える。しかし，判定に用いる場合には余分な選択枝が残り，使いにくい。

判定用と生成用は分けて定義するとよい。

```
判定用  : member(X, [X | _]): -!  ;
          member(X, [_ | L]): -member(X, L);

生成用  : one_of(X, [X | _]);
          one_of(X, [_ | L]): -one_of(X, L);
```

(2) クローズ標準形による統一した記述

クローズ標準形は2章で説明したように次のような形式となる。

```
head  : -条件ゴール列[ , !], 実行ゴール列

        選 択 部          実行部
```

このように，

(i) 各クローズを選択部と実行部に区分ける。
(ii) 各クローズでの!は高々1つ

という形式に統一してクローズを記述する事により，プログラムを見通しのよいものにする事ができる。

(3) クローズ内 OR による記述

IF～THEN～ELSE形式の条件分岐の記述が必要となる場合が多い。クローズ内 OR で記述する事により，次のような利点が得られる。

(i) 読み易い
(ii) 記述が楽（述語名を考える必要がない）
(iii) IF～の部分がある最適化条件を満たしているとコンパイラによる最適化により実行が高速化される。

ただし，次の事に注意。

- IFが多重にネストす場合は，そのままクローズ内 OR のネストで対応すると見にくくなる。適当に別クローズに分割する事が必要である。
- ヘッドのユニフィケーションで渡される変数の値によって処理を区分ける場合は，ヘッドのユニフィケーションを条件判定に使用した方が処理が速い。

[例3. 2]

渡された値が0ならQを，0以外ならRを実行する述語Pを作れ。

(クローズ内OR)

```
P(X): -(X==0, !, Q; R)
```

(ヘッド・ユニフィケーション)

```
P(0): -!, Q ;
P(_): -R ;
```

この場合，後者の方が実行速度が速い。

(4) 他クラスのオブジェクトのスロットの直接参照の禁止

スロットには属性スロット(attribute slot)と要素スロット(component slot)とがあり，このうち，属性スロットに対しては，他クラスのオブジェクトのものでもObject!slot_name の形式で直接参照できる。しかし，被参照側クラスのスロット定義を変更した場合，参照側クラスの参照ステートメントの変更も必要となり，これはオブジェクトの情報隠蔽の原則を破る用法となっている。

次のようにするのがよい。

- 被参照側クラスにおいて，被参照スロットの値をアクセス(取出し / 書込み)するメソッドを定義する。
- 参照側はそのメソッドを使用する事で目的とするスロットにアクセスする。

すなわち，メソッド・コールという統一した手法により他クラスのオブジェクトのスロットにアクセスするようにする事で，上記問題点を回避する。(実行速度の面では損をするが止むを得ない。)

3.2 実行高速化の手法

(1) 計算量の少ないアルゴリズムの採用

Prologはかなり複雑なアルゴリズムも簡単にかける場合が多いが，計算回数がどれ位か意識せずにプログラムを書いてしまう危険性が高い。同じ目的を達成するためのプログラムによる記述方法は何通りかあるのが普通であるが，計算量のオーダが異なってくるような非効率なプログラムを作ってしまわないよう注意が必要である。特に，単純で繰返し実行する部分については，計算の手間をきちんと見積って最善のアルゴリズムを使うようにする事。

〔例3.3〕 与えられたリストを反転する述語 reverseを作れ。

〔悪い例〕
```
reverse([ ], [ ]): －！  ;
reverse([H | T], R) : －reverse(T, RT),
append ([ ], L, L): －！;
append ([W | X], Y, [W | Z]): －append(X, Y, Z);
```

このアルゴリズムでは，

- 長さｎのリストを第１引数にとる append の手間はｎ＋１（呼出回数の合計）
- 長さｎのリストを第１引数にとるreverse の手間は
  - ｎ＝０ →１
  - ｎ＞０ →長さｎ－１の reverseの手間＋
    長さｎ－１の append の手間＋１

オーダとして，第１引数の長さの２乗に比例する。

〔良い例〕
```
reverse(L, R): －reconc(L, [ ], R);
reconc([ ], R, R); －！;
reconc([W | X], Y, Z): －reconc(X, [W | Y], Z);
```

このアルゴリズムでは，

- 長さｎのリストを第１引数にとる reconc の手間はｎ＋１
- 長さｎのリストを第１引数にとるreverseの手間は，長さｎのreconの手間＋１

オーダとして，第１引数の長さに比例する。

こういう基本的な部分のアルゴリズムが悪いと，他をどう工夫しても効率は悪くなる。

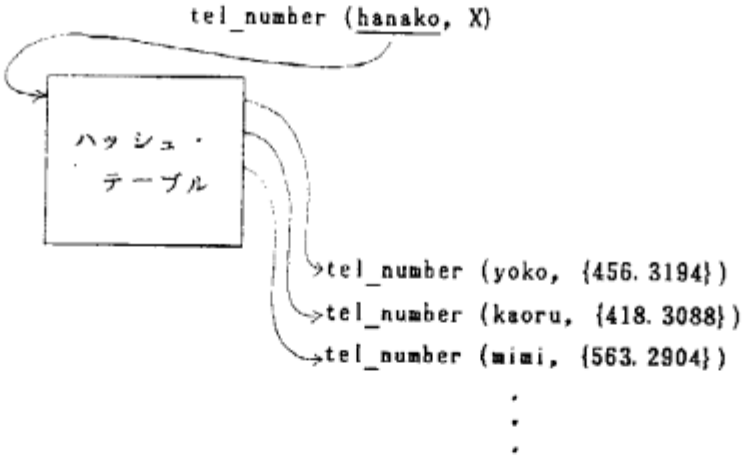
(2) clause indexing

次のような例を考えて見る。人名と電話番号を対にしたデータベースがあるとして，ある特定の人の電話番号を取り出したいとする。プログラムは次のようになる。

```
tel_number (hanako, X)
```

```
tel_number (yoko, {456.3194})
tel_number (kaoru, {418.3088})
tel_number (mimi, {563.2904})
        .
        .
        .
```

各クローズを先頭からシリアルになめるとすると，クローズがｎ個ある場合は平均ｎ／２個のクローズを調べる必要があり，ｎが大きくなると実行時間も大幅に増大する。このため，clause indexing と称される最適化手法をコンパイラで採用しており，ハッシュ・テーブルを内部的に生成して，クローズの数に対してほぼコンスタントの時間で目的とするクローズを取り出せるようにしている。



現時点のＥＳＰコンパイラ(ＳＩＭＰＯＳ　１．５版)は，次の条件を満たすプレディケートに対して clause_indexingを適用している。

- ローカル述語は第１引数，メソッド・コールは第２引数を対象とする。
- プレディケートにおいて，上記引数が定数であるクローズが連続して４個以上続いており，且つ定数の値が２種以上ある。

プログラム作成時，clause_indexing の対象としたい引数は，ローカル述語なら第１引数に，メソッド・コールなら第２引数とする必要がある。なお，引数として変数を持つクローズが出現すると，そこで clause_indexingの処理は区分けられる。なぜなら，変数は常にユニフィケーションが成功してしまうためである。

clause_indexing による高速化の効果はクローズの数が多い場合は顕著なものとなり，利用できるケースでは積極的に用いるべきである。

（注意）　スタック・ベクタの場合は，その第１要素がインデクシングの対象となる。
したがってリストは、ほとんどの場合適用されないと見てよい。

(3)　if_then_else

クローズ内 OR を使用する事で条件分岐を簡単に記述する事ができる。このクローズ内 OR において，ある条件を満たしていると，特別な最適化処理をコンパイラに依頼する事ができ，実行速度を向上できる。

（条　　件）　P：－ …… ，（C，！，G_1；G_2），……

において，Cの部分に以下の１４種の組込述語のみしか出現しない場合。
（１ヶないし複数ヶ使用していてよい）

atom, atomic, equal, floating_point, identical, integer,
less_than, location, not_equal, not_identical, not_less_than,
number, structure, unbound

これら組込述語の共通点は，組込述語の実行によって，引数である変数が未定義状態から値を持つ状態に変化する事がない事である。このため，内部的にクローズ内 OR に対応する制御ブロックを作成する必要がなくなり，その分実行が高速化される。

（記述形式）　上記条件を満足しているクローズ内 OR に対して，コンパイラは最適化したコードを出力するが，この際プログラマが特別に用意された記述形式を使用してプログラムを記述する事になっている。形式は次の通りである。

P：－ …… ，（C－>G_1；G_2），……

手続型言語で IF 文をネストできるのと同じく，クロード内 OR のif_then_else もネストが可能であり，次のような記述も可能である。

① P：－ ……，（C_1－>G_1；（C_2－>G_2；（C_3－>G_3；G_4））），
② P：－ ……，（C_1－>（C_2－>G_1；G_2）；G_3），……
③ P：－ ……，（C_1－>G_1；
C_2－>G_2；
C_3－>G_3；G_4），……

③のようにかっこを省略した場合は，①と同様になる。
（`；´と`－>´のオペレータの優先順位による。）

(注　意)　(C－>G_1；G_2)において，Cの所に許容されている以外の組込述語，あるいはユーザ定義述語が含まれていた場合，それら述語がfailするとG_2も実行されなくなる。例えば(stack_vector (V.L)，L＜3－>，G_1；G_2)と記述する。ここで，less_thanがfailした場合，G_1は期待通りに実行されるがstack_vectorがfailすると，G_2は実行されない。したがって，条件部に前出の組込述語がひとつも含まれない場合は，ELSE部は決して呼ばれることはない。

このように，if_then_elseは用法を誤ると意図しないプログラム動作を引起こすため，濫用は慎むべき機能であるといえる。

なお，この機能は，SIMPOS1.5版では，コンパイル済みのときのみ使用できる。(インタプリティブ・モードでは使用できない。)

(4) メソッド・コールの回数削減

メソッド・コールはローカル述語呼出しと較べて現状では実行速度は約2倍ほど遅い。このため，メソッド・コールの回数を減らす工夫をすべきである。特にメソッド述語で再帰呼出しを行うことは，極力，避けたい。これを含め，自オブジェクト内のメソッド・コールを使用するケースはよく生じる。この場合も(i)と同じくメソッド・コールをローカル述語コールに変える事で高速化できる。

［例3.4］

```
        ～, self_call(Object, X), ～
    : self_call(Object, X): －～
                  ↓
        ～, self_call(Object, X), ～
    : self_call(Object): －self_call(X, Object);
local
    self_call(X, Object): －　～
```

(5) ユニフィケーションの負荷の軽減

クローズが受けとる引数が構造体の場合，その形式がわかっていても，ヘッド中にそれと同じ形式の構造体を記述する必要はない。

〔例3.5〕　もらってきた引数をそのまま渡して述語q，rを呼ぶ述語pを作れ。
但し引数はいつでもf(X，Y)の形式とする。

〔悪い例〕　p(f(X，Y)): －q(f(X，Y))，r(f(X，Y));

この記述だと次の問題点がある。

- Prologインタプリタでの処理では，pが呼ばれる毎に引数が，f(X，Y)の形式である事を確認し，X，Yの値を持ってきて，改めてf(X，Y)を作り直してq，rに渡すため，かなり手間がかかる。
- pが引数をそのままq，rに渡すという事が，直感的にわかりにくい。

〔良い例〕　p(F): −q(F), r(F)

この記述方法を採用する事で，上記問題点は解消される。

(注)　Fがf(X, Y)の形式である事を確認するようなロジックを入れたい場合は次のようにするとよい。

```
p(F): −check(F, f, 2), q(F), r(F);
check(T, F, A): −integer(A),
                 stack−vector(T, A＋1),
                 first(T, F), !;
check(T, F, A): − {エラー処理を記述} ;
```

また，引数の順序は，

```
p( ,  ,  ,  )
   条件   その他（出力など）
```

にした方が　1)clause indexがかかる　2)余分なユニフィケーションを行わないとうい点で有利であり，また，プログラムが見易くなる。

〔例3. 6〕

(悪い例)

```
                        ～, p(X, 2), ～
p({}, 0): −!;
p({1}, 1): −!;
p({1, 2}, 2): !;
```

(良い例)

```
                        ～, p(2, X), ～
p(0, X): −!, X={};
p(1, X): −!, X={1};
p(2, X): −!, X={1, 2};
```

3.3 メモリ使用の効率化手法

Prolog（従ってＥＳＰも）はプログラム実行の際に相当量のメモリを使用する言語であり，へたなプログラミングをすると多量にメモリを消費し，ＧＣの多発等による実行効率の低下をもたらす。従って，プログラマは Prolog でのメモリの使用方法についてある程度の知識を持ち，非能率なプログラムを作らないよう注意すべきである。以下に不必要にメモリを浪費しないプログラムを作成するための手法について，いくつか述べる。

まず，スタック領域の使い方について説明する。

（スタック領域の使用）

Prolog（及びＥＳＰ）のプログラム実行は4種のスタックを使用して処理される。これらスタックの種類と主な用途は次の通りである。

グローバル・スタック	……	グローバル変数の値の保持
ローカル・スタック	……	ローカル変数の値の保持
コントロール・スタック	…	バック・トラック等の実行制御情報の保持
トレール・スタック	……	バック・トラック時に論理変数の値を undo するための，古い変数値を保持

述語呼出しが行なわれる毎に，これら4種のスタック内に新しい領域が確保されて使用される。これらスタック領域は不要になると解放されるが，その解放のタイミングは次のようになる。

ローカル・スタック コントロール・スタック	……	クローズ（プレディケート）が決定的に終了するか，またはクローズが fail した。
グローバル・スタック トレール・スタック	……	クローズが fail した。

クローズを決定的に記述する事は，プログラム実行速度の面で有利であるばかりでなく，ローカル・スタックやコントロール・スタックの領域も解放されるため，メモリ領域の面でも有利である。グローバル・スタック，トレール・スタックについては，クローズが決定的に終了しただけでは領域は解放されない。一般には，これらスタックの領域の解放はＧＣ(ガーベッジ・コレクション)にまかされる。 Prolog の実行ではグローバル・スタックが最もメモリ領域を使用するのが一般的である。このための対処策を後述してある。

なお，ＴＲＯ(Tail Recursion Optimization) はプログラム実行速度の向上やローカル／コントロール・スタック領域の節約には効果的であるが，グローバル／トレール・スタック領域の節約にはならない事は留意しておく必要がある。

(1) スタック領域の解放

スタックには4種類あり，このうち，ローカル・スタック，コントロール・スタックはクローズが決定的に終了すれば領域が解放されるのに対し，グローバル・スタック，トレールスタックはクローズが決定的に終了しても領域は解放されない。グローバル・スタック，トレール・スタックの領域の解放はガーベッジ・コレクション(ＧＣ)にまかされる。しかし，ＧＣの発生は一般には好ましくなく，できるだけスタック（特にグローバル・スタック）が延びるのを抑制するのがよい。

グローバル・スタックも含めて，各スタックの領域は fail により解放される。これを利用して次のようにすればスタック領域の伸長をさけられる。

（ｉ） 一連の処理を実施し，その結果をヒープ領域に残す。
（ii） 次いで，failを発生させ，一連の処理で使用したスタック領域を解放する。

この方式は，同じ処理を繰返すような場合に特に有効である。システム入出力割込ハンドラでも使用している。入出力割込ハンドラは入出力割込の発生毎に起動される。そのままでは，システム作動中にグローバル・スタックは伸長するばかりであり，いずれＧＣが発生してしまう。この事態を防ぐため，上記の手法によりスタックが延びないように工夫されている。反面， TROを利用した場合と比較して処理速度が落ちるので，用途に応じて選択することが望ましい。

〔例3. 7〕 一般の繰返し処理

```
loop(X)  : -repeat, ·····  , stop__condition(Y), !, fail.;

                          処理を実施し，ループから抜け出す
                          結果をヒープ  終了条件を判定
                          領域に残す。
loop(X)  ;
```

〔例3. 8〕 入出力割込ハンドラ ····· 無限ループ

```
: goal(Handler): -  loop(Handler)
local
  loop(Handler): -
                          :                前処理
                          :
                  : get(STREAM, Object), ····· 割込事象持ち
                          :
                          :                割込み処理
                          :
                  fail  ;
loop(Handler): -loop(Handler);
```

システム立上げ時に : goal が呼ばれ，以下無限ループを続ける。割込事象をストリームより受け取り，１つの処理が終了すると，次の事象発生まで待つ。

(2) ヒープ領域の再利用

ヒープ領域は，一旦領域が確保されると，たとえクローズが fail しても解放されない。ヒープ領域を解放する手段はＧＣだけである。従って，ヒープ領域のむやみな浪費をさける方法は，使用済み領域の再利用である。例えば，入出力バッファをヒープ領域上に確保するような場合，必要となるたびに新たなバッファ領域を取るのでなく，使用済みとなったバッファを自分で管理し，（引数で受け渡す，あるいはスロットで保持する。）次にバッファが必要となると，その使用済みバッファを再利用する。

プログラムでヒープ領域を使用する事になるのは次のような場合である。

(i) new_heap_vector, new_string 等，ヒープ領域上にデータ領域を確保するＫＬ０組込述語を使用した場合。
(ii) POOLの機能を使用した場合。
(iii) ヒープ領域を使用するオブジェクトを使用した場合。

(ii)は(iii)に含まれるが，分けて掲げてある。(ii)，(iii)で注意すべき事は，自分の想像していないような仕方でヒープ領域が多量に使用されている場合がある事である。この点，注意しておく必要がある。

3.4　リスト操作の手法

(1)　表現対象の適合性チェック

リストは

① 同じ種類のものが
② 不定個

並んでいる物を表現するのに使うべきである。リストは非常に汎用性の高いデータ構造であるが，それだけに処理速度，メモリ容量の面でのオーバヘッドも大きい。なんでもかんでもリストにするのは，リストのまずい使用法の典型である。異なる種類の要素がある場合は，種類毎に要素を集めて別々のリストとする。また，要素の個数が決っているような場合はベクタあるいは複合項を使った方がよい。

〔例3.9〕

〔悪い例〕　姓，名，誕生年，月，日を表現するのに

〔takashi, chikayama, july, 24, 1953〕のような表現をとると，

次の良い例と比較して

- 見た目が分かりにくい
- メモリが余分に必要
- 処理速度（ユニフィケーション等）が遅い

等の欠点だらけのデータ構造となる。

〔良い例〕　例えば

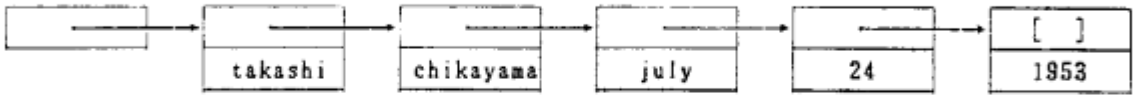
```
birthday (name (takashi, chikayama) , date (july, 24, 1953) )
```

のように意味内容に沿った関数(functor) 名をつけてまとめる。一見複雑そうだが，リストによる表現によりメモリ，処理速度の面で効率がよい。

上記の2例を含め，ベクタの内部構造を以下に示す。

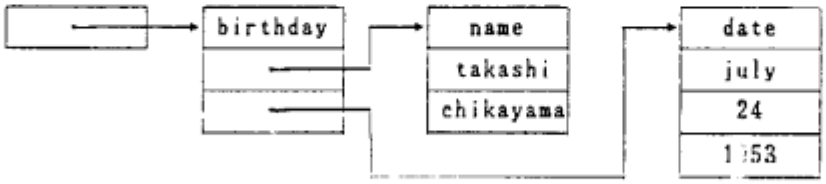
① [Takashi. chikayama. july. 24. 1953]

= { { { { { [ ] . 1953}. 24}. july} , chikayama} , takashi}



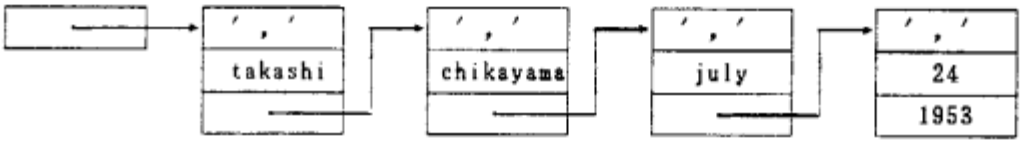
② birthday (name (takashi. chikayama). date (july. 24. 1953))

= {birthday {name. takashi. chikayama}. {date. july. 24. 1953}}

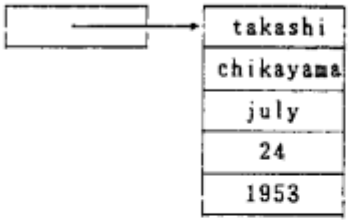


③ (takashi. chikayama. july. 24. 1953)

=′ , ′ (takashi. ′ , ′ (chikayama, ′ , ′ (july, ′ , ′ (24. 1953))))



④ {takashi. chikayama. july. 24. 1953}



(2) リスト処理の高速化

リストは，非常に汎用性の高いデータ構造であるが，基本的には，リニアな構造であるため，ランダムなアクセスには弱い。要素数がｎ個のとき検索にオーダｎかかるため要素すべてに何かランダムに処理を施す場合，n^2 のオーダになることがあり，要素数が多いとき問題となる。

SIMPOSでは，キーにより検索，追加，削除を行なうことができる２－３木（クラス two_three_tree）をサポートしている。２－３木は，スタック・ベクタにより実現されておりオーダlog nで，任意にデータにアクセスすることができる。詳細は，SINPOS説明書のトランスデューサの項を参照。

〔例3.10〕

memberは

```
member (X, [X | _] : - !;
member (X, [_ | L] : -  member (X, L) ;
```

で実現されるが，このとき平均のアクセスは，要素数nに対してn/2となる。特に，存在しないことを確かめるためには必ずn回呼ばれる。このような処理が多い場合は，リストではなく，2－3木を用いる方が望ましい。しかし，リストは，逐次的な処理には有利であるので，用途によって使い分けることが必要である。

ヒープを用いた構造体（POOLサブシステム）についても同様のことが言える。クラスlistは，リニアな構造のため要素数が多いとき不利である。そのためPOOLサブシステムでは，キー・インデクス付，ハッシュ・インデクス付などを用意しているので，それらを使用するとよい。また，関連項目をポインタで結んでおくのもひとつの手段である。

3.5 マクロ展開

マクロ展開は，コンパイル時に行うなうため，実行時の速度向上となる場合がある。

〔例3.11〕

```
①  p: -  new_string(String. 3. 16),
         set_string_element(String, 16#" 1C" ) ,
         set_string_element(String, 16#" E2" ) ,
         set_string_element(String, 16#" F1" ) ,
         q (String) ;
```

⇓

```
②  p: -q (double_bytes:{16#" 1C" , 16#" E2" , 16#" F1" })
```

①では，実行時にstringが生成されるが，②では，コンパイル時にstringが生成される。

マクロ展開を行なう場合，展開した結果を得るために必要となる条件と，チェックするために付加される条件が加えられることがある。

〔例3.12〕

（ケース1）

X＝A＋B　　⇒　　add (A, B, X)

（ケース2）

～, p (A＋B) , ～　　⇒　　～, add (A, B, C) , p (X) , ～

〔例3.13〕　#クラス名という，クラス・オブジェクトを取り出すマクロは条件として，location_element/2という組み込み述語が付加される。

```
～ , : p (#abc , ……) ,  ～

～ , location_element (…) , method_call (…) ,  ～
```

したがって同じクラスに対して複数のクラスメソッドを発行する場合，ありいは，ループの中にクラスメソッドを入れる場合は以下のようにした方が望ましい。（クラス・スロットへのアクセスも同様）

①
```
p (x) : -  : m1 (#abc , …) ,
           : m2 (#abc , …) ,
           Y =  #abc ! slot1;
```

⇓

```
p (x) : -  Obj =#abc , …,
         : m1 ( Obj, …) ,
         : m2 ( Obj, …) ,
         Y =    Obj ! slot1;
```

②
```
      ～ , q (X) ,  ～
```

```
q (X) : -  <終了条件>, !;
q (X0) : -: m1 (#abc , …) ,
           : m2 (#abc , …) ,
            q (x) ,
```

⇓

```
      ～ , Obj =#abc , q (X, Obj ) ,  ～
```

```
q (X, _) : <終了条件>, !;
q (X0,  Obj) : -: m1 ( Obj, …) ,
                : m2 ( Obj, …) ,
                 q (X, Obj );
```

ボディー部に条件付マクロが存在する場合は，その直前に置かれる。しかし，ヘッドに条件付マクロが存在する場合は，ラスト・ゴールの後に，条件が置かれる。

〔例3.14〕

```
a (A+B) : -b (A), c (B);
```

は次のように展開される。

```
a (C) : -b (A), c (B),
        add (A, B, C);
```

そこで，以下のような場合，注意を要する。

```
a (A, B, A+B) : -!;
a (A, B, A-B) : -!;
```

これは，一見，加算，減算はカット（!）の前に行われるように見える。しかし，実際は以下のように展開される。

```
a (A, B, C) : -!   ,   add (A, B, C);
a (A, B, C) : -!   ,   subtruct (A, B, C);
```

したがって，a（1，1，2）はsuccessするが，a（1，1，0）はfailする。もし，両者ともsuccessさせたいときには，以下のように，明確に記述しなければならない。

```
a (A, B, C) : -C=A+B, !;
a (A, B, C) : -C=A-B, !;
```