

# ICOT Technical Memorandum: TM-0388

---

TM-0388

## ESP Guide

by  
S. Uchida

September, 1987

©1987, ICOT

**ICOT**

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

(03) 456-3191 ~ 5  
Telex ICOT J32964

---

**Institute for New Generation Computer Technology**

*PSI: Personal Sequential Inference Machine*

## ESP Guide

English Edition

August 1987

*Institute for New Generation Computer Technology*

This manual describes the language specifications of the programming language ESP (Extended Self-contained Prolog) which is available under the operating system SIMPOS (Sequential Inference Machine Programming and Operating System) of the sequential inference machine PSI (Personal Sequential Inference Machine).

## PREFACE

ESP (Extended Self-contained Prolog) is a programming language, developed by introducing the modular programming capability based on an object-oriented paradigm into the logic programming language Prolog, and has the advantages of both logic programming language and object-oriented programming language.

ESP features:

- (1) the advanced modular programming capability based on an object-oriented paradigm, which facilitates modular programming and synthesis and reuse of modules, allowing efficient development of a large program.
- (2) the unification and the backtracking capabilities of Prolog, as well as the class and the inheritance capabilities of the object-oriented paradigm, allowing hierarchical knowledge representations.
- (3) a variety of macro expansion capabilities, allowing concise program description and improving program readability.

ESP, having these features, is a knowledge-representation and system-description language with an advanced description capability and a high applicability to general-purpose use, while Prolog is mainly used for development of a rather small artificial intelligence program. ESP may be used for the following purposes:

- (1) development of artificial intelligence programs, from a small-sized program to a large-sized, practical program such as for machine translation or expert system
- (2) development of system programs such as an operating system or a compiler

This manual describes the language specifications of ESP, and has been written for readers who are familiar with Prolog. The object-oriented paradigm is explained in appendix H "WHAT IS 'OBJECT-ORIENTED'."

- CHAPTER ONE "GENERAL" briefly described the language specifications and features of ESP.
- CHAPTER TWO "CHARACTERS AND REPRESENTATION FORMAT OF LEXICAL ELEMENTS" describes the representation format and function of the characters and lexical elements which are basic units for ESP programming. This chapter and chapter three are the main part of this manual.
- CHAPTER THREE "FORMAT OF CLASS DEFINITION" describes the structure and description format of a class, which is a program module unit of ESP. A class is formed by combining words and lexical elements according to the prescribed rules.
- CHAPTER FOUR "BUILT-IN PREDICATES" briefly explains some main built-in predicates, which are provided by PSI machine instructions, KL0. For details of the entire KL0 built-in predicates, see the KL0 Built-in Predicate Manual.

CHAPTER FIVE "PROGRAMMING TECHNIQUE" describes how to write a good ESP program in terms of ease of reading, ease of debugging, execution speed and memory efficiency. To write an ESP program, you should thoroughly understand the programming technique described in this chapter as well as the language specifications.

## TABLE OF CONTENTS

CHAPTER ONE	GENERAL	1
1.1	Extended Features of ESP	1
1.2	Class Definition	2
1.3	Method Definition and Method Call	6
1.4	Class "class"	10
1.5	Macro	10
CHAPTER TWO	CHARACTERS AND REPRESENTATION FORM OF LEXICAL ELEMENTS	11
2.1	Character Set	11
2.2	Lexical Elements	12
2.3	Term	16
2.4	Precautions on Operator-Applied Terms and compound Terms	23
2.5	Implementation of Terms on PSI	25
CHAPTER THREE	COMPOSITION OF CLASS DEFINITION	28
3.1	General	28
3.2	Inheritance	38
3.3	Method Definition	40
3.4	Method Call	45
3.5	Slot	48
3.6	Class "class"	59
3.7	Macro	62
3.7.1	General	62
3.7.2	System-Defined Macros	65
3.7.3	Users-Defined Macros	72
3.8	Multi-Class Name Space (Package)	80
3.8.1	Class Specification	80
3.8.2	Package Environment and Externally Declared Class	81
3.8.3	Use of Package in SIMPOS	83
3.8.4	Restrictions	86

3.9	ESP Program in PSI	90
CHAPTER FOUR	BUILT-IN PREDICATES	94
4.1	Data Manipulation	96
4.2	Arithmetic Operations	100
4.3	Logical Operations	103
4.4	Comparison operations	105
4.5	Data Type Conversion	110
4.6	Execution Order Control	111
CHAPTER FIVE	PROGRAMMING TECHNIQUE	113
5.1	Writing a clause	113
5.2	Controlling Program Execution	119
5.3	Data Manipulation	123
5.4	Slot	133
5.5	Clause Indexing	134
APPENDIX A	GRAMMAR OF ESP	136
APPENDIX B	TABLE OF STANDARD OPERATORS	142
APPENDIX C	LIST OF STANDARD MACROS	143
APPENDIX D	LIST OF BUILD-IN PREDICATES	145
APPENDIX E	KEYBOARD CODE ENTRY TABLE	148
APPENDIX F		151
APPENDIX G	PRECAUTIONS FOR CONVERSION FROM PROLOG TO ESP	155
APPENDIX H	WHAT IS "OBJECT-ORIENTED"	166

## TERMINOLOGY

### Prolog

A symbol manipulation language developed based on predicate logic. Prolog has several advanced capabilities for describing an artificial intelligence program such as unification and backtracking, and has been widely used as a programming language for artificial intelligence applications, as well as LISP.

### predicate logic

A formal method of logic that is used to conclude whether a given proposition, for example, "Socrates is mortal" or "The sun rises from the west" is true or false from the known true facts (propositions).

### compound term

A formal description of the attribute of a thing or the relation between things. A compound term is a basic description unit in predicate logic, and is described in the form of  $f(a,b,\dots)$ . In Prolog,  $f$  is called a predicate name, and  $a, b, \dots$  are called arguments.

(Example)	on(pen,table)	.....	The pen is on the table.
	cat(michael)	.....	Michael is a cat.

As shown above, the attribute or relation to be notified is written in front of the parenthesized thing (or things).

### clause

A program description unit in Prolog. A Prolog program is written as a set of clauses. A clause is described in the format of  $A:-B,C,\dots$ , where  $A, B$  and  $C$  are compound terms. The number of compound terms on the left side of  $:-$  is limited to one at most, and a clause in this format is called a Horn clause. Clauses are used to describe facts, rules and questions. A Prolog program is executed in this way: a question is given from outside to the program, and an answer is made by repeating inferences based on the facts and rules written in the program.

Facts, rules and questions are described in the following formats:

Fact	human(socrates). This is the abbreviated form of: human(socrates):-.
Rule	mortal(X):-human(X).
Question	:-mortal(socrates).



### head, head part

The left side of :- of a clause.

### body, body part

The right side of :- of a clause.

### goal

Individual compound terms in the body part.

### predicate

In Prolog, a set of clauses, each having the same predicate name and the same number of arguments in its head part is called a predicate.

(Example)	married (taro,hanako)	.....	clause	predicate
	married (jiro,yukiko)	.....	clause	
	parent-of (taro,yoshio)	.....	clause	predicate
	parent-of (taro,nobuo)	.....	clause	

### built-in predicate

A predicate that is already provided in the language processing system of Prolog. Standard predicates that are often used in many kinds of programs and predicates that the user may feel are difficult to describe (because they are complicated to describe or because they would be inefficient if they were described by the user) are provided as built-in predicates. The typical built-in predicates are for I/O operations and arithmetic operations.

### KL0, KL0 built-in predicate

The machine instructions of PSI are called KL0, and their capabilities have been extended based on Prolog. The built-in predicates provided by KL0 are called KL0 built-in predicates. In the ESP manual, the KL0 built-in predicates may be referred to simply as the built-in predicates.

### predicate call

An action to call the corresponding predicate from a goal of the body part of a clause. The corresponding predicate means a predicate that has in the head part the same predicate name and the same number of arguments as that goal.

### unification

An action to make two patterns (structures) identical to each other in terms of the pattern by substituting appropriate values for the variables in the patterns. In Prolog, when a predicate call is performed, unification is performed between the goal (compound term) and the head (compound term) of the clause of the called predicate. For example, `parent-of(taro,X)` and `parent-of(Y,yoshio)` both become `parent-of(taro,yoshi)` by substituting `yoshio` for `X` and `taro` for `Y`, thus the unification succeeds. In the case of `parent-of(X,yoshio)` and `parent-of(Y,nobuo)`, however, the unification fails because `yoshio` and `nobuo` are both constants and the values are not the same.

### backtracking

A solution search method. When there are several selection branches in the course of the path to the solution, the solution is searched for by selecting one branch. If it is not the branch to the solution, then the branch point is returned to and another branch selected. This method is used in execution of a Prolog program: if the predicate called from a goal has more than one clause, the program is executed by selecting these clauses one by one in the order that they appear in the source program. If the execution of a selected clause fails, then the next clause is executed.

### logical variable

Variables used in Prolog are called logical variables. A logical variable is defined (i.e., has a value) only through a unification operation, and cannot be redefined (i.e., the value cannot be changed) and behaves like a constant until backtracking occurs. When backtracking occurs, the variable is undefined (i.e., the value is released), then it can be redefined with a new value through the next unification.

### side effect

Similar to the side effects of a medicine, it means an unexpected effect that is brought about unintentionally, besides an intentional function provided by the program. Variables used in a procedural language such as FORTRAN or C have side effects. A value substituted for this kind of variable remains unchanged even after the program is ended. Consequently, the internal state of the program is different between when the program is called and when the program is ended. Because of this, when the program is called the second time, the program does not always give the same result as the previous time.

In Prolog, an execution result that is not restored by backtracking is a side effect. Variables of Prolog do not have side effects since they are undefined by backtracking.

### object-oriented, object-oriented paradigm

A new programming method, where the concepts of structured programming and abstract data type in the conventional procedural languages have been expanded further. A program is described on the basis of an object, which is a unified entity of a procedure and data. (They are different entities in conventional languages.)

### class, class object

Objects that have common characteristics and behavior may be defined together as one class. In object-oriented languages, a class is the module unit for the programming. A class definition itself may also be treated as one object, and a class that has been processed into executable format is called a class object.

### inheritance

The capability to include the attribute (i.e., the contents of the definition) of a class (or classes) in the class being currently defined. When defining more than one class, if you define those definitions that are common among the classes as an independent class, you can concisely describe a class definition by including that class through inheritance in the current class definition. Inclusion of a single other class is called single inheritance, and inclusion of multiple other classes is called multiple inheritance.

### instance, instance object

An object generated by a class object. As many instances as necessary are generated by a class object.

### method

A function that an object provides to the outside. An object generally has more than one function, and these functions are identified by the method names. The method of a class object is called a class method, and the method of an instance object is called an instance method.

### method call

An action to send a method name and parameters from one object (the sender) to another object (the receiver) so as to request the receiver to execute the corresponding method. The method name and parameters to be sent are called a message.



## CHAPTER ONE GENERAL

ESP is knowledge-representation and system-description language, developed by introducing the modular programming capability based on an object-oriented paradigm into logic programming language Prolog. This section describes the concepts of ESP.

### 1.1 Extended Features of ESP

Figure 1-1 shows the difference in the programming paradigm between Prolog and ESP.

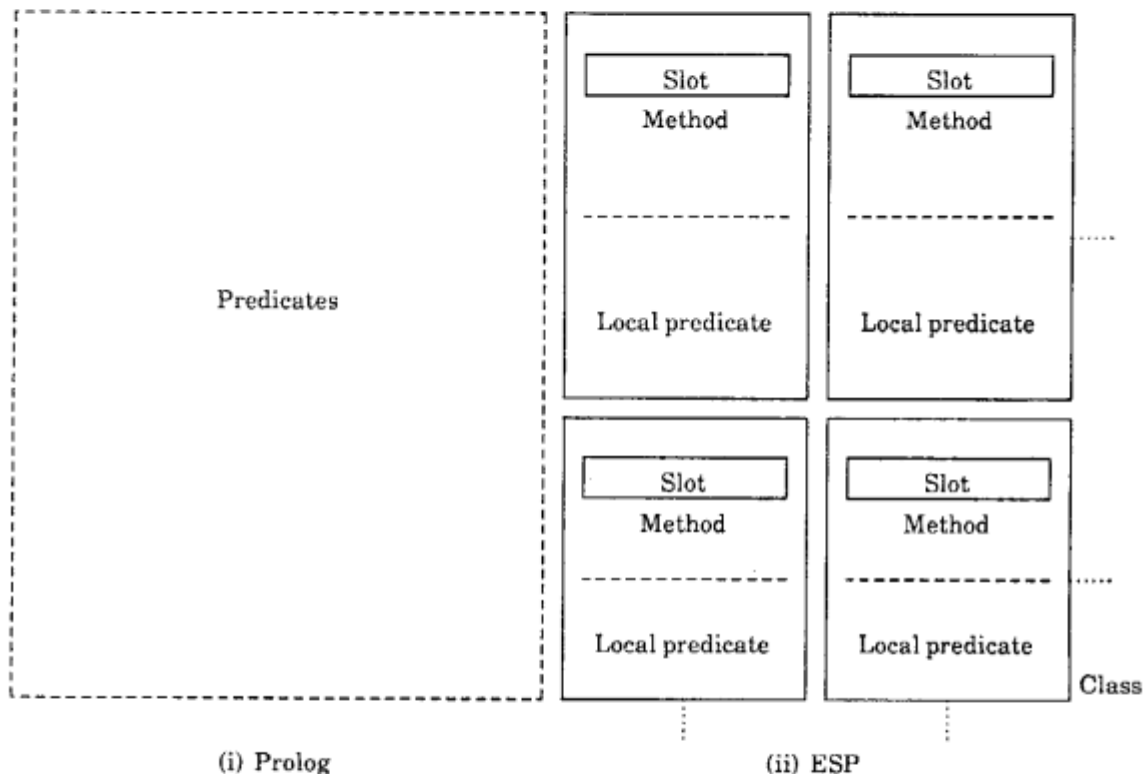


Fig. 1-1 Difference in Programming Paradigm between Prolog and ESP

In Prolog, facts and rules are described in clauses (Horn clauses), therefore, a program written in Prolog is composed of a set of clauses. In other words, a Prolog program is composed of a set of predicates since a group of clauses having the same predicate name and the same number of arguments is referred to as a predicate. Because Prolog does not have as the language specifications a modular programming capability, if you want to expand your program, you do it by adding or modifying the predicates.

On the other hand, an ESP program is written in modules called classes. Because of this, predicates used in ESP are classified into two kinds: a local predicate (a predicate that can be referenced only from within a class) and a method (a predicate that can be referenced from outside a class.) These predicates have basically the same characteristics as the predicates of Prolog. However, some extended features have been added to the method of ESP, such as the demon function.

The following items can be written in the body of an ESP predicate:

Prolog	ESP
built-in predicate	(KL0) built-in predicate
predicate (user-defined)	method (system-defined, user-defined)
	local predicate (user-defined)

In ESP for PSI, KL0 built-in predicates (or simply built-in predicates) mean the predicates provided by the PSI hardware, while in Prolog, built-in predicates mean the system-defined predicates that can be used without user definition. Two kinds of methods are available in ESP: system-defined methods and user-defined methods. The system-defined methods (the methods of SIMPOS system-defined class) provide functions that require considerably complicated processing, such as file I/O functions.

Another extended feature introduced to ESP is a slot. In Prolog, variables, called logic variables, are defined through unification and can be redefined only after their values are released by backtracking. This function is useful to write an artificial intelligence program. However, the logic variables have a disadvantage in that they cannot have side effects. A slot is a variable that can have side effects and can be redefined at any time like those variables used in a procedural programming language such as FORTRAN or C. Also, a slot is not retracted to the previous value by backtracking. From the viewpoint of a logic programming language, a slot is used to change the system of axioms. From the viewpoint of knowledge representation, a slot is used to express a "has\_\_a" relation.

Inheritance is the capability to include the function of a different class in the current class definition. (See Fig. 1- 2.) In ESP, the multiple inheritance function is provided so that more than one class can be included.

When including the functions of other classes through inheritance, if the methods or slots of the same names have been defined in both inheritance destination class and inheritance source class, the inheritance is processed according to the prescribed rules (override, synthesize, etc.). The inheritance function facilitates reuse or synthesis of functions and improves programming efficiency. The inheritance function can also be used to describe an "is\_\_a" relation in knowledge representation, and use of this function, together with a "has\_\_a" representation by a slot, allows concise, hierarchical knowledge representation.

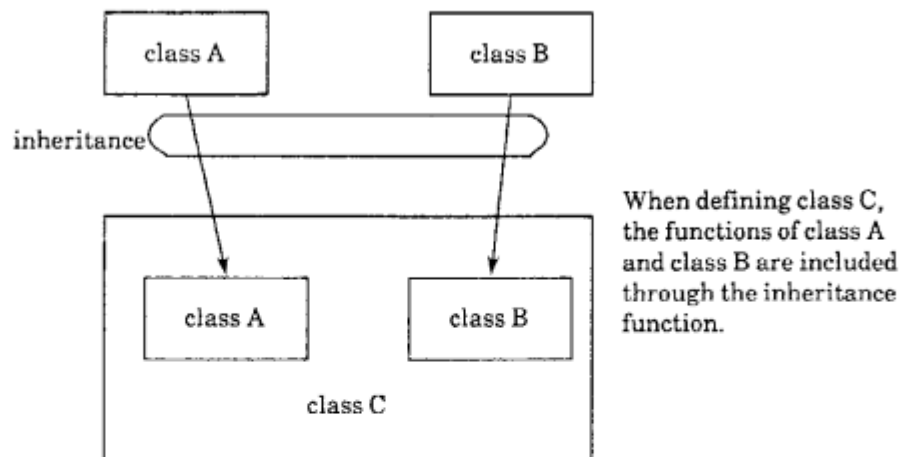


Fig. 1-2 Explanation of Inheritance Function

## 1-2 Class Definition

This section explains the ESP way of class definition. A class definition consists of the following components:

- (1) class name
- (2) inheritance class name
- (3) class slot and class method
- (4) instance slot and instance method
- (5) local predicate

The class name identifies the class and must be unique in the SIMPOS system. A class (or classes) to be inherited is specified by the inheritance class name (or names). Then, slots (variables) and methods of the class and instance are defined. Slots are equivalent to variables used in a procedural language and have side effects. However, slots do not need to be declared for a data type, and thus may have any type of data such as integer number, floating-point number, vector, or list. A method is a function that the object provides to the outside, and can be called from the outside. Sending a method name and parameters as a message to an object causes the object to execute the specified method with the passed parameters and to return the results to the message sender. This message passing capability has been realized by using the paradigm of the unification mechanism of Prolog as is.

Figure 1-3 shows a typical description format of a class definition.

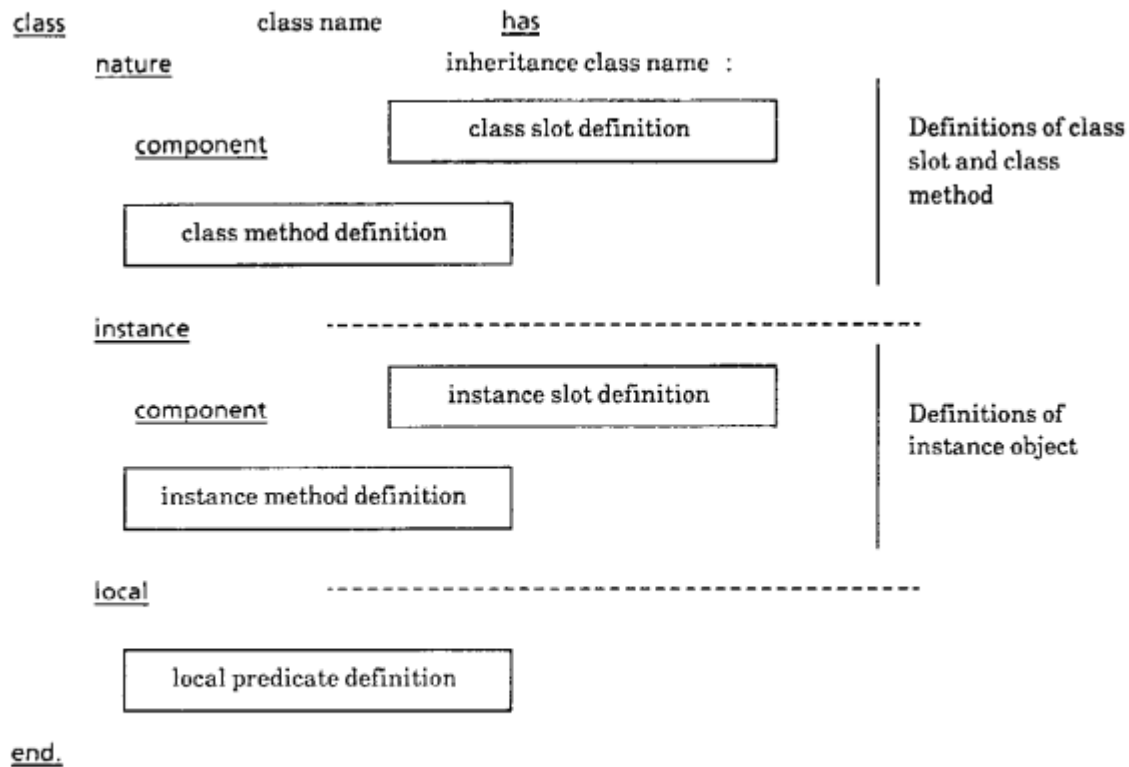


Fig. 1-3 Typical Class Definition Form

In the figure above, the underlined words are the keywords (operators) prescribed in the ESP language specifications. A class definition starts with the operator "class" and ends with the operator "end". The operator "nature" specifies the class name (or names) of a class (or classes) to the inherited. ESP supports the multiple inheritance function: more than one class can be inherited. The slot and the method defined before the operator "instance" are called the class slot and the class method respectively. The slot and the method defined after the operator "instance" are called the instance slot and the instance method respectively. A local predicate is written after the operator "local", which is a predicate to be used locally within the class.

Figure 1-4 shows a program example.



```

class my__window has
  nature standard__io__window;
  :create__my__window(CLASS, Window) :-
    :create(CLASS, [size(900, 250) ,
      position(100, 100)], Window),
    :activate(Window) ;

instance
  component
    user __name,
    group __name;
  :message1(Window, String0, String1) :-
    write1(Window, string0),
    write1(Window, String1),
    user __name (Window) ;
  :get __group__name (W, W!group__name) ;

local
  user __name(Window) :-
    :putc (Window, #":"),
    write1(Window,
      Window!user__name),
    :write (Window, key#1f),
    :write (Window, key#cr);
  write1(Window, String) :-
    string (String, Size, __),
    :putb (Window, String, Size) ;
end.

```

Fig. 1-4 ESP Program Example of Class Definition

The source program of a class definition is compiled, determined for the inheritance, and converted to a format that can be executed as a class object. When determining the inheritance, those slots and method that are defined in the class(es) to be inherited are all included. However, if a method in the class to be inherited has the same name as one in the current class definition, the method in the current class definition takes precedence. (The actual inheritance processing for slots and methods is more complicated. For details, see chapter three.)

When a class object receives an instance generation request (:new method call), the class object generates the instance objects by making copies of the instance object definition part in the memory. (See Fig. 1-5.)

Since multiple instance object share one instance method part (and the local predicate part), only the part that corresponds to the instance slot part is actually generated in the memory (heap area). The instance slot part in the memory is a one- dimensional array (vector).

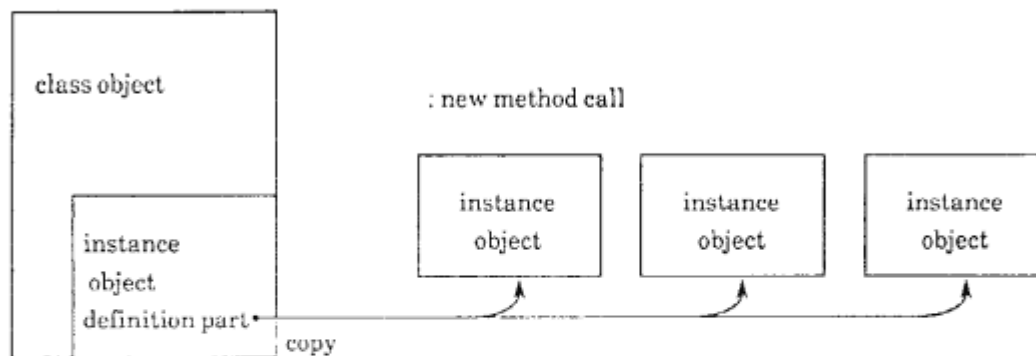


Fig. 1-5 Generation of Instance Object from Class Object

The use of a class slot/class method is explained next. You may want to manage instance objects (referred to as instances hereafter) generated from the same class. For example, when you generate instances from a class where a limited number of resources are defined, the number of instances to be generated is limited, therefore, you need to check instance generation. Or, if you want to collect the generated instances to reuse them, you need to manage the generated instances. This function may be realized by making an object that executes this function. However, the smarter solution is to make the class object do it. The class slot/class method is used for this purpose. :new is a method to request a class to generate instances. The :new method is a class method (which the system internally generates and adds to all classes.)

### 1.3 Method Definition and Method Call

A method is defined in the following format:

```
:method__name(Object,arg__1,...,arg__n)
```

```
:-goal__1,goal__2,...,goal__n;
```

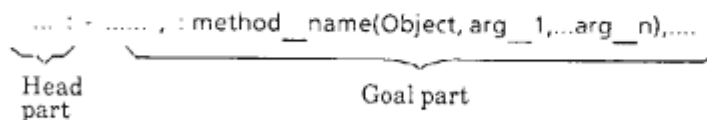
where,

method__name :	Method name
Object :	Variable to receive an object
arg__1,...,arg__n:	Arguments
goal__1,...,goal__n:	A string of goals. A local predicate, a built-in predicate, or a method can be specified.

The method definition has been realized by extending the predicate format of Prolog. The following extensions have been made.

- Put a colon in front of a predicate name. A predicate name preceded by a colon is interpreted as a method name, discriminated from a local predicate. Also, the entire predicate is interpreted as a method definition.
- The first argument in the head is used as a variable to receive an object (an instance object or a class object).
- The second and the following arguments in the head are used as the parameters for data transfer with the method caller.

A method call is described in the body of the method and the local predicate. The description format is the same as that of the head part of the above method definition format:



When a method call is issued at program execution, it is processed in the following sequence:

- the method name and the parameters are passed to the object specified by the first argument (Object), and
- the object executes the method specified by the passed method name, using the passed parameters, then
- the execution results of the method are returned to the output arguments and the object processing is terminated.

This sequence of processing for a method call is actually accomplished using the unification mechanism of Prolog as is. That is, an "object + method name" is used instead of the predicate name at unification between the goal and the clause head of Prolog. (See Fig. 1-6.)

The differences between the predicate call of Prolog and the method call of ESP are as follows:

- The method to be referenced by an ESP method call is determined by the object (specified by the first argument), the method name and the number of arguments, while the predicate to be referenced by a Prolog predicate call is determined by the predicate name and the number of arguments.
- The first argument for a method call specifies the object to which the message is to be sent.

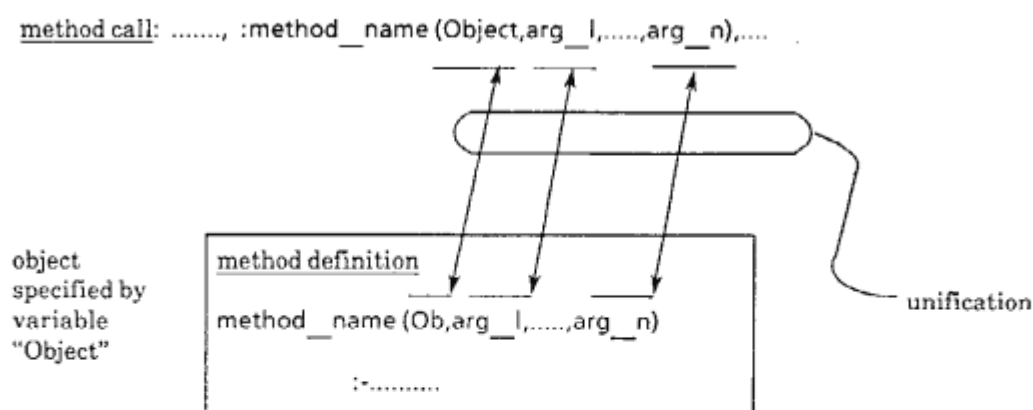


Fig. 1-6 Method Call Processing in ESP

There are no other differences except the above. Unification is performed in the same way, and backtracking at failure is also performed in the same way.

Specification of variable "Object" at a method call is performed as follows. First, a class object is specified as #class name. For example, the class object of class "window" is specified as #window. On the other hand, an instance object has no name, thus cannot be specified by a name. When an instance is generated by :new method or an equivalent method, the class object returns the identifier for the instance object (the pointer to the instance object generated in the memory). This identifier may be used later to specify that instance object. (See Fig. 1-7.)

```

:create(#fdd__volume,Volume),
    % Call class method :create of class object
    % fdd__volume and request to generate an instance.
    % Class method :create internally generates the
    % instance by using :new method and returns the
    % pointer to variable Volume.

:retrieve(#directory,Device,">device>fdd1"),
    % Request class object directory to generate an
    % instance. Class method :retrieve internally
    % generates the instance by using :new method and
    % returns the pointer to variable Device.

:mount(Volume,Device,0),
    % Call :mount method of the instance (of class
    % fdd__volume) specified by variable Volume.

:link(Volume,">FD1"),
    % Call :link method of the instance (of class
    % fdd__volume) specified by variable Volume.

:make(#fdd__binary__file,File,"FD1>MYFILE",20),
    % Call class method :make of class object
    % fdd__binary__file, generate the instance, and
    % receive the pointer in variable File.

:close(File),
    % Call :close method of the instance (of class
    % fdd__binary__file) specified by variable File.

```

Fig. 1-7 Specification Example of Class Object and Instance Object

## 1.4 Class "class"

A class named "class" has been provided as a system-defined class, and is implicitly inherited to all classes. In class "class", the common functions (methods) that all classes should have are defined, so that the user does not have to define these functions by himself at definition of a class. These functions include:

- class method (:new) that generates an instance object
- class/instance method that reference (:get\_\_slot) and set (:set\_\_slot) the value of a slot
- class/instance method (:refute) that dynamically specifies a method name to call the method

For details of the functions provided by class "class", see chapter three section "Class 'class'".

## 1.5 Macro

One of the features of ESP is a strong macro expansion function. The macro expansion function works for a certain character string pattern appearing in an ESP program, and the string is automatically expanded to the prescribed pattern. The macro expansion function allows concise program description and improves program readability. Besides the system-defined macros, the user can define his own macros. For details of the macros, see chapter three, section "Macro".

## CHAPTER TWO

### CHARACTERS AND REPRESENTATION FORM OF LEXICAL ELEMENTS

This chapter describes individual lexical elements that make up an ESP program. The first part describes the kinds of characters that you can use to write an ESP program. The second part explains the representation format of the lexical elements that are the basic components of an ESP program.

In this chapter, the explanations are made without strict definitions for ease of understanding. For the formal descriptions, see appendix A.

#### 2.1 Character Set

An ESP program, similar to a program written in an ordinary programming language, is composed by writing characters such as "a", "b" and "c". For ESP programming, you can use the following characters. These characters are all interpreted as different characters, including the difference between the uppercase and the lowercase letters.

##### ◆ Alphanumeric Characters

- Digits (0, 1, ..., 9)
- Lowercase letters (a, b, ..., z, and kanji characters)
- Uppercase letters (A, B, ..., Z, and "\_" (underscore))

In ESP, the kanji character set (including hiragana, katakana and Greek letters) belongs to the lowercase character set, and can be used not only as character string data but also as basic program units as well as the lowercase alphabet. For example, you can use kanji characters for describing an atom (explained later) such as a class name or a predicate name and for describing a logical variable name.

##### ◆ Special Characters

@, #, \$, %, ^, &, +, -, =, ~, (accent grave), ?, /, \ (back-slash), . (period), : (colon), <, >, \*

##### ◆ Delimiting Characters

(, ), [, ], {, }, !, :, ,(comma), ;(semicolon)

Besides these characters, you can also use three special-use characters (" and ' and %) and three formatting character codes (space, new-line, and tab).

An ESP program may be written with these characters.

## 2.2 Lexical Elements

An ESP program is formed with lexical elements that you make by arranging the characters described in the previous section according to the rules explained below. There are five kinds of lexical elements:

Lexical element	Example		
atom	esp	データ	知識
integer number, floating-point number	123	1.0	-3
character string	"string"	"文字列"	
variable	X	__var	
delimiter	}	(	

### (1) Atom

Atoms correspond to the names of various kinds of entities used in ESP and play a very important role in ESP programming. Class names, predicate names, and various kinds of keywords of ESP are all atoms. In ESP, an atom is notated as described below. Those atoms that are dynamically generated by a built-in predicate (explained later) will not have a notational name unless otherwise given explicitly.

#### ◆ String of alphanumeric characters starting with a lowercase

Examples: esp name\_\_with\_\_underline

esProlog kIO サクラサク 情報 りんご

ク3合Waせ

#### ◆ String of special characters

Examples: :- ?- +

Note that a period (.) and a circumflex (^), when used conforming to the notational rules for a floating-point number (explained later), are not interpreted as an atom. Also, a period, when used alone, indicates the end of a term. However, consecutive use of periods like "=.." is valid for an atom.



◆ String of characters enclosed in single quotes (J)

Example: '\$\$%var'

Note that to include a single quote as part of the string, write two single quote consecutively.

Example: 'describe' 'name'

An atom notated by enclosing a string in single quotes is called a quoted atom.

◆ Four symbols (! : , ;)

Note that these symbols are not an atom when they are used for the special purposes described later (that is, when they are used as a delimiter for a list or a vector.)

◆ [ ] (null list)

(2) Integer number, floating-point number

In ESP, two kinds of numbers can be used: integer numbers and floating-point numbers.\*

Example: 0986 -3 3.14159265  
51.2^8 1.0^-3

You can perform arithmetic operations and logical operations on these numbers by using appropriate built-in predicates. The built-in predicates for double-word operations (which are applied to a pair of two words of integer numbers) are also available. (See chapter five.) However, you cannot directly perform an arithmetic or logical operation on a mixture of integer numbers and floating-point numbers: you have to change the data types to either type before performing the operation.

---

\* PSI expresses these numbers in 32 bits.

### ◆ Integer number

A decimal number is expressed as a string of digits or a string of digits preceded by "-".\*

Example: 1 25 -3 1986 0

An integer number other than a decimal number may also be expressed using macros (explained later).\*\*

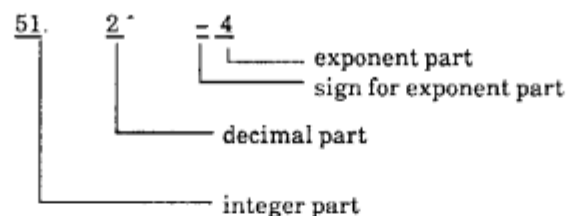
Example:  $\frac{16\#}{\text{radix}}$   $\frac{\text{"FF10"}}{\text{value}}$   $8\#\text{"71"}$

The radix is expressed by a positive decimal number. The value is expressed using ten digits from 0 to 9 and letters from a to z (or A to Z). Letter a (or A) represents decimal number 10, letter b (or B) represents 11, and so on. The uppercase and the lowercase letters are not discriminated. Up to what digit or letter can be used depends on the radix used.

### ◆ Floating-point number

ESP can handle floating-point numbers.

Example: 1.0 -2.1 1.41421356  
51.2<sup>-8</sup> (namely 51.2 x 10<sup>-8</sup>)  
1.0<sup>-3</sup> (namely 1.0 x 10<sup>-3</sup>)



---

\* +1 means a compound term +(1)={+,1} (explained later), different from integer 1.

\*\* SIMPOS provides the notation from binary number to 36 nary number as Macro.

The integer part is a string of digits or a string of digits preceded by "-". The decimal part and the exponent part are a string of digits. The decimal part may be omitted, but the decimal point (namely a period) cannot be omitted. The sign for the exponent part is either "-" or "+". The plus sign "+" may be omitted. If the exponent part is zero, then the symbol ^ and the following part may be omitted.

Because the expression of a floating-point number is a lexical element, no formatting characters such as a space must be inserted before and/or after the decimal point ".", the symbol ^, and the exponent part sign.

Invalid examples: 1. 0 1.0 ^ 3 2.0^ 3 2^3

### (3) Character string

A string of characters enclosed in double quotes (") is called a character string. You can include a double quote in the string by writing two double quotes consecutively.

Example: "flower" "" "Michael" "" is a cat"  
"漢字を用いることもできます。"

### (4) Variable

Variables are expressed as a string of alphanumeric characters starting with an uppercase letter (or an underscore "-"). Kanji characters may be used as a variable name if preceded by an underscore.

Example: X - RS232C Variable\_\_1  
KonDo \_\_変数1

Those variables whose name is expressed with the same string of characters are interpreted as the same logical variable within that clause definition. However, a variable expressed only with a single underscore is interpreted as a different variable even if it appears at more than one place in the clause definition, and a variable of this type is called an anonymous variable.

### (5) Delimiter

A delimiting character is interpreted as a delimiter.

Example: [Welcome,psi] { This,is,a,pen }  
          ↑          ↑ ↑ ↑ ↑ ↑ ↑ ↑

Delimiting is done by one delimiting character and is used for clarification of a syntactical structure.

## (6) Comment

A string of characters starting with a percent symbol (%) up to the new-line is interpreted as a comment and does not have any effect on program execution. However, a percent symbol in a quoted atom (a string enclosed in single quotes) or in a character string (a string enclosed in double quotes) does not indicate the beginning of a comment.

## 2.3 Term

An ESP program is composed of structures called terms, the basic units of a program. Elements such as integer numbers and variables, and structural data such as strings, vectors, compound terms and linear lists are generically called terms. The structural data may have another term as its element. That is, a term has a hierarchical structure (or a recursive structure). In fact, an ESP program itself is one large, complex term structured from a combination of various terms.

The term include the following entities:

<u>Term</u>	<u>Example</u>
atomic literal	esp    123    4.5
logical variable	MITSUBISHI <u>var</u>
string	"String"
vector	{a,b,c}
compound term	f(x,1)
operator-applied term	X = Y + 1
list	[1,a,x]

### (1) Atomic literal

An atomic literal is an atom, an integer number or a floating-point number.

Example:    esp   123   4.5   6.7^-8

### (2) String

A string is a structure formed by arranging integer numbers in one dimension. A string is mainly used as a character string whose elements are character codes, but it can also be used as an array of integer numbers. A character string is a kind of string and is expressed as follows:

Example:    "String" "文字列"

There are four kinds of bit widths of a string element: 1 bit, 8 bits, 16 bits and 32 bits, which are called bit string, byte string, double-byte string and word string respectively. The character strings enclosed in double quotes in the examples above are a 16-bit string and have as the elements the character codes corresponding to the particular characters. Strings other than a character string can be expressed using a macro as follows:

#### ◆ Bit string

0 or 1 can be written as an element.

Example: `bits#{1,0,1}`

#### ◆ Byte string

An integer number between 0 and 255 (i.e., non-negative integer that can be expressed in 8 bits) can be written as an element. A character string preceded by `ascii#` is a byte string whose elements are 8-bit ASCII codes corresponding to the individual characters of that character string.

Example: `bytes#{0,100,16#"F0"}`  
`ascii#"abc"`

#### ◆ Double-byte string

An integer between 0 and 65535 (i.e., non-negative integer that can be expressed in 16 bits) can be written as an element.

Example: `double__bytes#{0,1000,16#"AB00"}`

#### ◆ Word string

An integer between  $-2^{31}$  and  $2^{31}$  (i.e., signed integer that can be expressed in 32 bits) can be written as an element.

Example: `words#{0,1000,-1,16#"FFFF0000"}`

Each element of these strings must have been determined by the time of compiling and cannot be left as a variable.

The individual string elements are numbered 0, 1, 2, ... from left to right sequentially, and a particular element can be specified by that number. Thus, you can take out a specified string element as follows:

`p(X,Y) :- S = "abcdefgh"`  
`string__element(S,X,Y); ... ①`

In the above example ①, `string__element` is a built-in predicate that takes out the element at the position specified by `X` within the string specified by `S` and unifies it with `Y`. If you call the above predicate with `p(2,A)`, the character code (integer) of the letter "c" is unified with variable `A`.

### (3) Vector

In ESP, you can use a `vector**` (one-dimensional array) that is structural data formed by arranging terms in one dimension.

Example: `{0,1} { } {b,x} {c,{d,{e}}}`

A vector is expressed by enclosing one or more terms with `{` and `}`, and each term must be delimited by a comma if more than one. `{ }` is a null vector (a vector that has no elements). The individual vector elements are numbered 0, 1, 2, ... from left to right sequentially, and a particular element can be specified by that number.

```
p(X,Y):-    V={ a,b,c,d,e,f},
           vector__element(V,X,Y);
```

`vector__element` is a built-in predicate that takes out the element at the position specified by `X` within the vector specified by `V` and unifies it with `Y`. If you call the above predicate with `p(2,A)`, the atom, `c`, is unified with variable `A`.

---

\* If you want to compose a string dynamically at execution time, use built-in predicates. (See Chapter five.)

\*\* In PSI, a vector is realized using a stack vector.

#### (4) Compound term

A compound term is expressed as follows:

Functor (Arguments)

Example:  $f(X)$   $date(july,1,1959)$   
 $f(g(X), \{3,4\}, "string")$

The functor is an atom. The arguments are a string of terms, each delimited by a comma. (It is not allowed to make a compound term that has no elements.) A compound term is a special representation of a vector whose first element is an atom and whose number of elements is more than one. A compound term is internally the same as a vector of that type. A compound term is used for writing the header part of a predicate, etc.

Example: The compound term representation (left) is internally the same as the vector representation (right).

$f(X)$	$\{f,X\}$
$date(july,1,1959)$	$\{date,july,1,1959\}$
$f(g(X),\{3,4\}, "string")$	$\{f,\{g,X\},\{3,4\}, "string"\}$

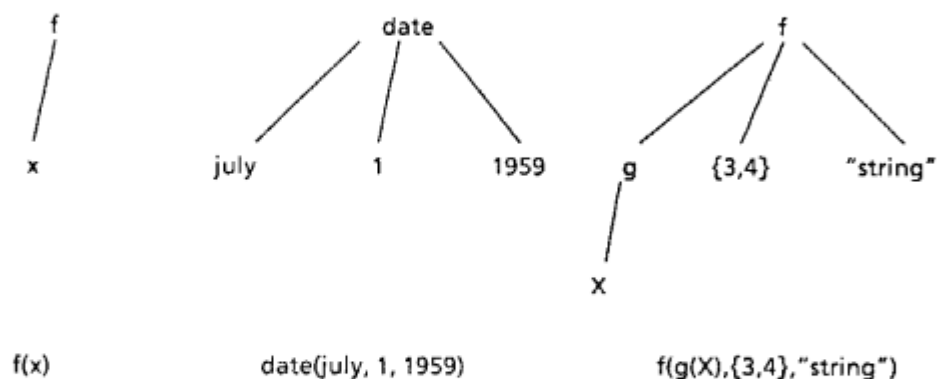
Therefore, unification of  $\{X,1\}$  and  $f(Y)$  will succeed, and  $X$  is unified with  $f$ , and  $Y$  with  $1$ .

The following vectors do not have the corresponding compound term representations:

$\{X,Y\}$	$\{1,2\}$	...	The first element is not an atom.
$\{atom\}$		...	The number of elements is not more than one.

A tree structure representation is often used to show the structure of a compound term, in which a functor is represented as a parent and arguments as children.

Example:



## (5) Operator-applied term

If the functor of a compound term could be written as an operator, the program would become more readable. For example, when representing the addition of  $X$  and  $Y$ , if it could be written

as  $X + Y$

instead of  $+(X, Y)$

it would be more readable. In ESP, similar to a usual programming language, you can use arithmetic operators. In this case,  $X + Y$ , for example, is internally converted to the same structure as  $+(X, Y)$ .

In ESP, to realize this representation of operator-applied terms, operator precedence grammar is used for syntax analysis.

There are three kinds of operators: infix, prefix and postfix. Infix operators are used to represent a compound term having two arguments, and prefix and postfix operators are used to represent a compound term having one argument.

- ◆ Infix operator ... A compound term is represented by the first argument (term), an infix operator, and the second argument (term) in that order.

Example:  $X + Y$        $1 + 2 * 3$   
                   $\uparrow$                     $\uparrow \uparrow$

- ◆ Prefix operator ... A compound term is represented by a prefix operator and an argument (term) in that order.

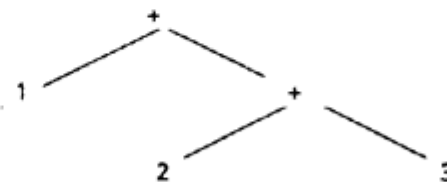
Example:  $-1$        $+3$   
                   $\uparrow$                     $\uparrow$

- ◆ Postfix operator ... A compound term is represented by an argument (term) and a postfix operator in that order.

Example:  $p;$   
                   $\uparrow$

Individual operators have been given a priority, which indicates the degree of operation conjunction. An operator of a less priority has a greater degree of conjunction. For example, the expression  $1 + 2 * 3$  should normally be processed with the multiplication  $2 * 3$  first and then the addition.

$1 + (2 * 3)$   
 $+(1, *(2, 3))$   
 $\{+, 1, \{*, 2, 3\}\}$





To indicate that operator \* has a greater degree of conjunction than operator +, in ESP, operator \* has given priority 400 and operator + has given priority 500.

Besides the priority, the position and the degree of conjunction are specified using the type of operator. For example,

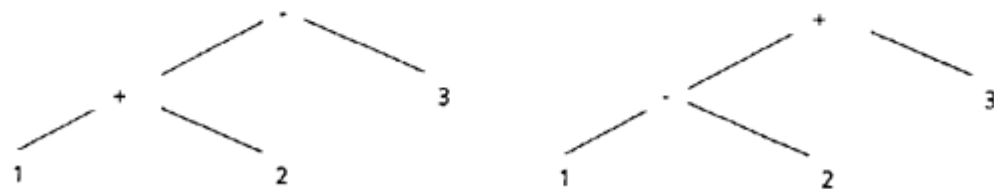
$$1 + 2 - 3$$

$$1 - 2 + 3$$

the former expression should normally be processed with the addition first while the latter with the subtraction first. That is, the order of operation for addition and subtraction is determined by the operator position but not by the kind, therefore, it cannot be determined only by the priority.

$$(1 + 2) - 3$$

$$(1 - 2) + 3$$



If you represent a calculation expression using a tree structure as shown above, you can specify that an operator of the same priority is permitted to be placed at the left branch and prohibited to be placed at the right branch.

ESP has the following seven types of operators:

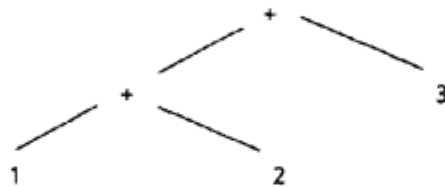
- ◆ Infix operator:        xfx    sfy    yfx
- ◆ Prefix operator:     fx    fy
- ◆ Postfix operator:    xf    yf

f indicates an operator, and x and y indicate arguments. x and y specify the degree of conjunction. If the argument is an operator-applied term, argument x of operator f can only have an operator whose priority is less than that of operator f. Argument y of operator f can only have an operator whose priority is equal to or less than that of operator f.

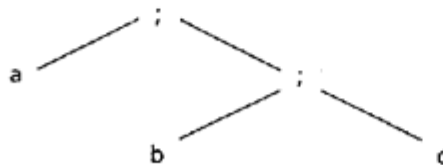
The above examples of addition (+) and subtraction (-) are of operator type yfx because an operator of the same priority is permitted on the left side and prohibited on the right side.

Therefore, operator "+" (yfx type) and operator ";" (xfy type) are parsed into different structures as shown below.

$$\begin{aligned}
 1 + 2 + 3 &= (1 + 2) + 3 &= +(+ (1,2), 3) \\
 &= \{+, \{+, 1, 2\}, 3\}
 \end{aligned}$$



$$\begin{aligned}
 a;b;c &= a;(b;c) = ';(a, '(b,c)) \\
 &= \{(:), a, \{(:), b, c\}\}
 \end{aligned}$$



In ESP, the operators listed in appendix B have already been defined.

\* SIMPOS allows the user to add new operators or modify the standard operators.

## (6) List

ESP can use a linear list similar to Prolog.

Example:    [1,2,3]  
               [1,f(x),{1,2}, [a,b]]  
               [1,2 /Tail]  
               [X|Y]

[ ] is a null list, and in ESP it is defined as an atom. A non- null list consists of the **car part** and the **cdr part**: the car part is the first element and the cdr part is a list of the remaining elements.

The car part of [1,2,3] is 1, and the cdr part is 2,3 . A list may be represented in one of two ways: (1) specify all elements and delimit each of them by a comma or (2) specify some of the elements (each delimited by a comma) followed by " " followed by a list of the remaining elements. The following lists have the same meaning:

```
[1, 2, 3]
[1, 2 | [3]]
[1 | [2, 3]]
[1 | [2 | 3]]
```

You may specify a variable on the right side of “|”.

```
P([1,2 | T],T);
```

If you call the above predicate with  $P(L, [ ])$ , then  $L$  is unified with  $[1,2 | [ ]]$ , that is,  $[1,2]$ . If you call it with  $P(L, [3,4])$ , then  $L$  is unified with  $[1,2 | 3,4]$ , that is,  $[1,2,3,4]$ .

## 2.4 Precautions on Operator-Applied Terms and compound Terms

The representations of an operator-applied term, a compound term and a vector described in the previous section are rather complicated and not easy to understand. This section describes the precautions when writing these terms and vectors.

### (1) Delimiter “,” has a special use

A comma (,) used to delimit each element of a term, list or vector is not an operator. If you want to write an operator having the priority of 1000 or more as an element, you must enclose it in parentheses.

A comma in  $[x,y,z]$  is a delimiter.

A comma in  $a(b,c)$  is a delimiter.

A comma in  $a((b,c))$  is an operator (atom).

A comma in  $\{1,2,3\}$  is a delimiter.

A comma in  $p:-q,r$  is an operator (atom).

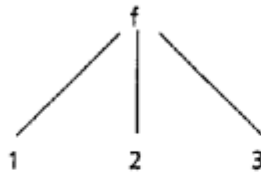
$f(p:-q,r)$  results in a grammatical error because the priority of the operator  $:-$  is 1100. It should be written as either

$f((p:-q),r)$  or  $f(p:(-q),r)$ . However, they have different meanings. The former is a one-argument compound term whose element is  $(p:-q)$ , while the latter is a two-argument compound term whose elements are  $(p:-q)$  and  $r$ .

$\{+,-,=\}$  does not result in an error. The individual elements are an atom.

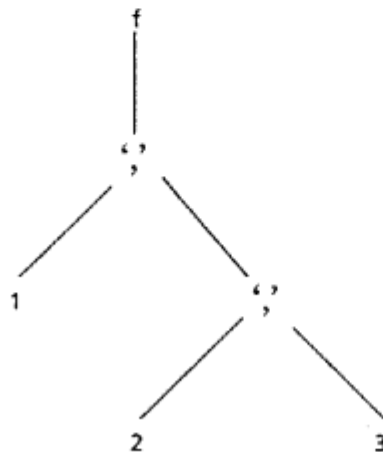
The following two representations have different meanings:

$$f(1, 2, 3) = \{f, 1, 2, 3\}$$



Three-argument compound term  
where  $f$  is the functor

$$f(1, 2, 3) = \{f, \{ (., 1, \{ (., 2, 3) \} ) \}$$



One-argument compound term  
where  $f$  is the functor

## (2) An operator is interpreted just as an atom

An operator is interpreted as an atom if the priority is less than that of the adjacent operator. If not so, enclose the operator in parentheses. The priority of an operator can be changed only by enclosing it in parentheses. Note that enclosing an operator in single quotes (') does not change the priority of that operator.

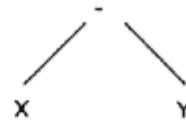
$X = +$  is correct (because the priority of  $+$  is 500 and the priority of  $=$  is 700.)

$X = '='$  results in an error. It should be written as  $X = (=)$ .

## (3) Precautions for prefix operators

If you want to put a prefix operator in front of a parenthesized entity, you should insert one or more formatting characters (e.g., space) between them. Otherwise, the prefix operator is interpreted as an atom and becomes the functor of a compound term.

$$-(X, Y) = \{-, X, Y\}$$



$$-(X, Y) = \{-, \{(.), X, Y\}\}$$



## 2.5 Implementation of Terms on PSI

The terms described in section 2.3 may be implemented in many ways. This section describes, from among the internal representations on PSI, only those related to the user program and the precautions for their usage.

### (1) Integer number

PSI provides 32-bit integer numbers. It also supports those built-in predicates that are used for 64-bit double-length arithmetic operations to a combination of two integer numbers.

A negative number can be written by preceding a string of digits with a hyphen (-). A string of digits directly preceded by a hyphen, "-10" for example, is read in as a negative integer number at parsing. However, if a space is inserted between the hyphen and the digit string, "- 10" for example, it is read in as a compound term of " ' ' (10)", then converted to a negative integer number by a macro. Similarly, a radix- of-n notation, "16'A0" for example, is read in as an integer number at parsing. However, "16#"A0"" is read in as a compound term, then converted into an integer number by a macro.

(2) Floating-point number

A floating-point number is implemented by a total of 32 bits: 8 bits for the exponent part, and 24 bits for the data part.

(3) Vector

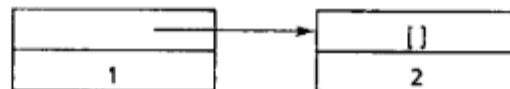
Vectors, compound terms and operator-applied terms are all implemented using a stack vector that is a one-dimensional structure having a fixed number of elements. Therefore, unification can be performed between these entities.

$$\{+1,X\} = +(1,X) = 1 + X$$

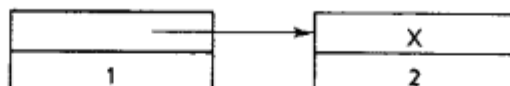
(4) List

A list is composed of a stack vector having two elements. The second element of the two elements is the car part of the list, and the first element of the two elements is the cdr part.

$$[1, 2] = \{[[], 2], 1\}$$



$$[1, 2 | X] = \{[X, 2], 1\}$$



[ ] is an atom that indicates the end of a list. Since a list has the structure shown above, the following relation holds:

$$f(a) = \{f,a\} = [a | f]$$

Therefore, note that unification of  $f(X)$  and  $[1 | Y]$  succeeds and  $X$  is unified with  $1$ , and  $f$  with  $Y$ .

(5) String

The length of each element of a string may be 1, 8, 16 or 32 bits. Because the data part length of one word of memory in PSI is 32 bits, in an 8-bit string, for example, the four elements are packed in one word. Like this, a string can be used not only for a character string but also to pack integer data in less bits.

## CHAPTER THREE COMPOSITION OF CLASS DEFINITION

The ESP language accomplishes modular programming by adding an object-oriented function to logic programming language Prolog. Modules in ESP correspond to “classes” in an object-oriented language. This chapter describes definition of a class, the basic unit of an ESP program.

The format of a class definition is outlined below.

```
class class-name with__macro macro-name ] has
[ nature inheritance-class-definition ]
[ class-slot-definition
  class-method-definition ]
[ instance
  instance-slot-definition
  instance-method-definition ]
[ local
  local-predicate-definition ]
end.
```

As shown above, an ESP program starts with the keyword “class” and ends with the keyword “end”.

Those parts enclosed in square brackets [ and ] may be omitted. Hereinafter, square brackets [ and ] indicate that the part enclosed in [ and ] may be omitted unless these symbols are confused with those used for a list.

### 3.1 General

#### (1) Defining a predicate

A simple example is shown below.

```
class with__append has
  :append (Class, X, Y, Z) :-
    append(X, Y, Z);
local
  append ([], X, X) ;
  append ([W | X], Y, [W | Z]) :-
    append(X, Y, Z) ;
end.
```

①

②



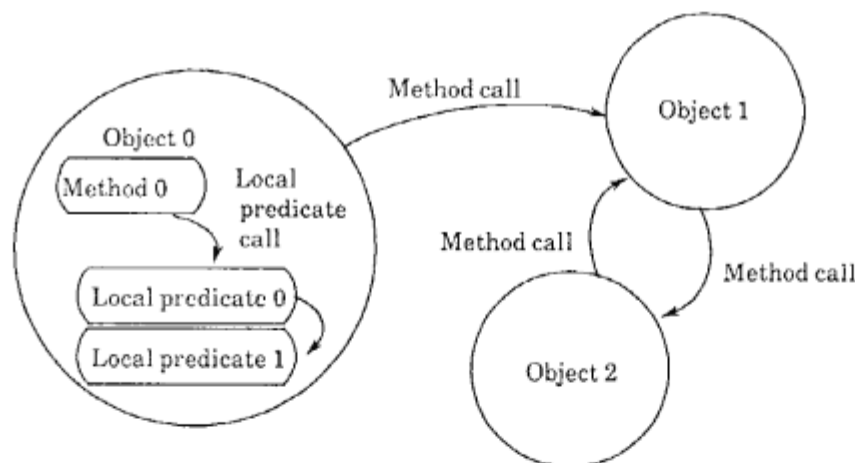
This is a description example of 'append', the most basic among the list processing of Prolog. The part indicated by ② is written according to the same notational conventions for clauses and predicates as those used in an ordinary Prolog, except that a clause ends with a semicolon, not a period. A predicate as shown at ② is called a local predicate. A local predicate is formed by writing, between the keywords "local" and "end", a series of clauses that have the same name and the same number of arguments.

A local predicate is literally a predicate that can be referenced only within the class locally. Therefore, it cannot be called from other classes.

Since a local predicate can be called only from within the class, the function cannot be used from outside the class unless it is interfaced with the outside. A function called "method" is used to accomplish this interface.

A method can be called not only from within the class where the method is defined, but also from outside the class.

ESP is an object-oriented language. That is, a combination of data and procedures, called an object, is processed as a unit, and exchanging messages between objects is always carried out by using a method.



A method is defined as follows:

```
:append (Object,X,Y,Z) [ :- body ] ;
```

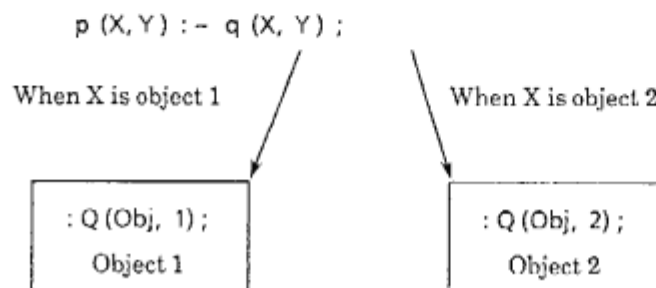
colon    method name    ↑    arguments

An object is passed to this argument.

You can call a defined method by writing the body part of a clause as follows:

head :- string-of-goals,			
:append (Object, [ 1, 2 ], [ 3 ], X), string-of-goals;			
colon	method	object	arguments
	name		




The first argument of a method call specifies the object for the method to be called. The object needs to have been determined only by the run time. That is, the object may be written as a logical variable at class definition, and the variable may be determined to be a method to be called dynamically at run time.



Upon execution of predicate p, if the value of variable X is object 1, then the method <1> is executed and variable Y is unified with 1. If the value of variable X is object 2, then variable Y is unified with 2. In this way, a method to be called may be determined dynamically at run time.

There are two kinds of methods (class and instance), which are related to a class object and an instance object respectively.

class with `__member` has

<pre>:member (Class, X, List) :-     :new (Class, Inst__object), *     :member (Inst__object, X, List) ;</pre>		Class method definition
<pre>instance : :member (Obj, X, List) :-     member (X, List) ;</pre>		Instance method definition
<pre>local member (X, [X __]) ; member (X, [__ L]) :- member (X, L) ;</pre>		Local predicate definition
<pre>end.</pre>		

As shown above, there are three kinds of predicates:

- (1) class method
- (2) instance method
- (3) local predicate

Head unification and backtracking are performed for these predicates, similar to those predicates used in Prolog.

The first argument of a method call specifies an object. One class object, and only one class object, can exist in a class and can be directly specified by writing “#class-name”. As many instance objects as necessary can be generated by class method “new”, which is a method automatically provided to any classes.

---

\* Class method “new” is a special method to be automatically provided to any ESP class, and generates instance objects from a class object.

#class-name	class-object
:new(#class-name, instance-object)	
(class object)	

Class method "member" of class "with\_\_member" in the above example may be called by the following:

```
:member(#with__member, 1, [0,1,2])
```

#with\_\_member denotes a class object. Method "member" is called, and the first argument (logical variable "Class") is unified with the class object (#with\_\_member). Then, class method "new" generates an instance object, and "member" of the instance method is called.

In general, since a method accompanies a mechanism that makes the dynamic call possible, the speed of a method call is slower than a local predicate. It would be reasonable that a predicate for a predicate recursive call within a class is made as a local predicate. This point will be discussed in chapter five "PROGRAMMING TECHNIQUES".

The following shows a simple ESP program.

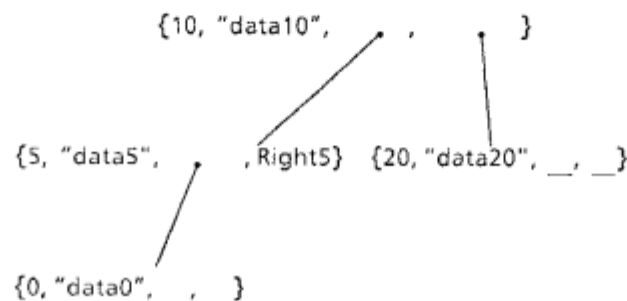
```
class binary__tree1 has
  :add(__, Tree, Key, Data) :-
    insert(Tree, Key, Data);
  :traverse(__, Tree, List) :-
    traverse(Tree, List, []);
local
  insert(Tree, Key, Data) :-
    unbound(Tree),!,
    Tree = {Key, Data, __, __};
  insert({Key0, __, Left, __}, Key, Data) :-
    Key0 > Key, !,
    insert(Left, Key, Data);
  insert({Key0, __, __, __}, Key, __) :-
    Key0 >= Key, !, fail;
  insert({__, __, __, Right}, Key, Data) :-
    insert(Right, Key, Data);

  traverse(T, L, L) :- unbound(T), !;
  traverse({Key, Data, Left, Right}, List, List0) :-
    traverse(Left, List, List1),
    List1 = [{Key, Data}|List2],
    traverse(Right, List2, List0);
end.
```

This program registers given key-and-data pairs in a binary tree structure according to the greater-less relation between keys and lists the data by keys. In this program, a vector is used to express the node of each binary tree.

Structure of each node  
 {Key, Data, Left\_\_node, Right\_\_node}  
 Key: key  
 Data: data  
 Left\_\_node: left child node  
 Right\_\_node: right child node

A node is maintained so that the left branch of the node has only such a node whose key is less than that of the node concerned and the right branch has only such a node whose key is greater. When giving data, you start at the root of the tree and proceed to either left or right branch depending on the greater-less relation\* of key. In this program, if a left or right branch does not exist, it is expressed with an undefined variable. For this, you only need to perform unification when adding a new node.



Suppose that the node whose key is 5 has no right branch and thus Right5 is an undefined variable. When logical variable Treee has as its value the root of this binary tree, if you perform the following method call:

```
:add(#binary__tree1, Tree, 7, "data7")
```

a new node

```
{7, "data7", __, __}
```

is generated and unified with variable Right5.

With this program, if a node of the same key exists above the tree, the method add fails.

Reverse listing of the key-indexed data can be done by searching for the left branch, the node and the right branch in that order.

```
:traverse(#binary__tree1, Tree, List)
```

---

\* For details of the greater-less relation expressed with >, <, >=, and <=, see section 3.6 "System-Defined Macros".

As a result, the following list is unified with List:

```
[ {0, "data0"}, {5, "data5"}, {7, "data7"},
  {10, "data10"}, {20, "data20"}]
```

With this program, a new node is added by performing unification to an undefined logical variable. However, once a logical variable is defined, it is not undefined and thus cannot be redefined until it is backtracked by failure. Because of this, the structure of this program does not allow the overriding of a value or the deletion of a node if the key is the same.

The following program uses an algorithm in which nodes are copied one after another and behaves as if a value is overridden if the key is the same. The structure of each node is the same as:

```
{Key, Data, Left__node, Right__node}
```

except that [ ] is used, instead of an undefined logical variable, to indicate that there is no child node.

With this program, when adding data, any nodes to be passed are all duplicated. Therefore, you can easily achieve the extension to maintenance of all data whose key is the same and deletion of nodes. Since the structure of the tree itself is almost the same as the previous program, method "traverse" may be written in almost the same way.

```
class binary__tree2 has

  :add(__, Old__tree, Key, Data, New__tree) :-
    insert(Old__tree, Key, Data, New-tree);
  :traverse(__, Tree, List) :-
    traverse(Tree, List, []);

local
  insert([], Key, Data, {Key, Data, [], []}) :- !;
  insert({Key0, Data0, L0, R}, Key, Data, {Key0, Data0, L, R}) :-
    Key0 > Key, !,
    insert(L0, Key, Data, L);
  insert({Key0, __, L, R}, Key, Data, {Key, Data, L, R}) :-
    Key0 >= Key, !;
  insert({Key0, Data0, L, R0}, Key, Data, {Key0, Data0, L, R}) :-
    insert(R0, Key, Data, R);

  traverse([], L, L) :- !;
  traverse({Key, Data, Left, Right}, List, List0) :-
    traverse(Left, List, List1),
    List1 = [{Key, Data}] List2,
    traverse(Right, List2, List0);
end.
```

## (2) Inheritance

ESP can use the inheritance function of an object-oriented language. This facilitates inclusion of the function of other classes.

The following class "list\_handler" inherits the function of the class "with\_\_append", thus it becomes a class having the class methods "append" and "reverse". Therefore, you can call the method "append" or "reverse" for the class object of the class "list\_handler".

```
class list_handler has
  nature with__append ;
  :reverse(Class, X, Y) :- reconc(X, [], Y) ;
  local
    reconc([], X, X) :- ! ;
    reconc([W|X], Y, Z) :- reconc(X, [W|Y], Z) ;
  end.
```

## (3) Slot

The nature, characteristic and internal state of an object are stored as a variable called a slot. For example, for the object of a window, the position and the size may be given as a slot. The value of a slot is not lost by backtracking and can be rewritten any time.

A logical variable of Prolog, once defined, is not undefined and thus cannot be redefined until backtracking occurs by failure. Contrarily, when backtracking occurs by failure, the value of a variable is release and cannot be preserved. To solve this problem, ESP supports various kinds of functions to implement side effects. A slot is typical among them. The following shows an example of where a slot is used to maintain the internal state of an object.

```
class lock has
  instance
    component state := unlocked ; ..... ①
    :locked(Lock) :- Lock!state == locked ; ..... ②
    :lock(Lock) :- Lock!state := locked ; ..... ③
    :unlock(Lock) :- Lock!state := unlocked ; ..... ④
  end.
```

The instance object of the class "lock" indicates a particular lock. The lock has either "locked" or "unlocked" state. To indicate this state, in the class "lock", the instance slot "state" is defined and given the initial value of "unlock" as shown by <1>. The predicate shown by <2> means access to the value of that slot and it succeeds or fails depending on the slot state.

Object!slot=name	... Referenceing the slot value
Object!slot=name:=value	... Substituting a value to the slot

There are two kinds of slots (class and instance), which are related to a class object and an instance object. A value is substituted by overriding the previous value. Also, a value is not released by backtracking. On the other hand, a slot cannot store an undefined variable or a vector (as well as a list). The programmer may use either the technique of dragging with an argument using a logical variable or the technique of storing to a slot, depending on the use of the program. A slot is described in detail in section 3.5.

The following shows a simple program using a slot. This is a program in which the previously described class "binary\_\_tree2" has been rewritten using a slot instead of a vector. If it were written using a vector, the following problem would arise:

- If you want to add items with which you can perform overriding for the logical variable of a node, you need to duplicate all those nodes you pass.
- Because the tree itself has the structure of a stack vector that does not have side effects, you cannot release the stack that will be necessary for control at run time, therefore, you cannot handle a large amount of data.

Class "binary\_\_tree2" in the following program uses an object- oriented function. That is, in all instance objects, the child branch is given as a slot value. The following four instance slots are provided:

key	... Key
data	... Data
left	... Instance object indicating the left branch
right	... Instance object indicating the right branch

Since key and data are stored in slots, an undefined variable and a vector (as well as list) are not allowed.

Storage of structural data may seem difficult since a vector and a list cannot be stored in a slot. However, this can easily be implemented using an instance object as shown. Also, since you can issue a method for that object, various kinds of operations can be widely selected.



```

class as __binary__tree__element has
instance
attribute key,
          data,
          left := [],
          right := [] ;
end.

class binary__tree3 has
nature
  as __binary__tree__element ;

  :create (Class, Obj) :-
    :new(Class, Obj),
    Obj!key := '$$root';

instance

  :add(Obj, Key, Data) :-
    Key0 = Obj!key,
    insert(Key0, Key, Data, Obj),
    fail;
  :add( __, __, __ );

  :traverse(Obj, List) :-
    traverse(Obj!left, List, List1),
    traverse(Obj!right, List1, []);
  :traverse(Obj, List1, List0) :-
    traverse(Obj!left, List1, List2),
    List2 = [{Obj!key, Obj!data} | List3],
    traverse(Obj!right, List3, List0);

local

  insert(Key0, Key, Data, Obj) :-
    Key0 > Key, !,
    insert_element(left, Key, Data, Obj);
  insert(Key0, Key, Data, Obj) :-
    Key0 >= Key, !,
    Obj!data :- Data;
  insert( __, Key, Data, Obj) :-
    insert_element(right, Key, Data, Obj);

  insert_element(Position, Key, Data, Obj) :-
    P = Obj!Position,
    ( atom(P), !,
      :new(#binary__tree3, Element),
      Obj!Position := Element,
      Element!key := Key,
      Element!data := Data;
      :add(P, Key, Data) );

  traverse([], List, List) :- !;
  traverse(Obj, List, List0) :-
    :traverse(Obj, List, List0);

end.

```

### 3.2 Inheritance

The definition part of inheritance specifies the classes to be inherited and their order.

```
[Example]  nature    class1,    class2;
           nature    class1, *,  class2;
```

Write keyword "nature" followed by the class name atoms of the classed to be inherited or an asterisk (indicating that the self-class is to be inherited), each delimited by a comma, and write a semicolon at the end. The order of inheritance is significant and affects the inheritance of a method or a slot. (See section 3.3 to 3.5.)

The order of inheritance is determined as follows:

- (1) If no inheritance definition is specified, the self- class is the only one class to be inherited.
- (2) If an inheritance definition is specified, the self- class and the classes specified in the definition are inherited. The order of inheritance is determined by the order in which the classes are written. The self-class is designated by an asterisk (\*). If the asterisk is omitted, it is assumed that the self-class is at the beginning.
- (3) If the same class name is specified more than once, the class name appearing first is valid.

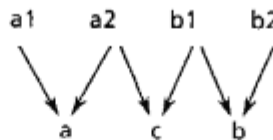
The following shows an inheritance example.

Suppose that class "a" is inheriting classes "a1" and "a2", and class "b" is inheriting classes "b1" and "b2", and class "c" is inheriting classes "a2" and "b1".

```
class a has
  nature a1,a2;
end.
```

```
class b has
  nature b1,b2;
end.
```

```
class c has
  nature a2,b1;
end.
```



- ① class ex1 has  
nature a,b;  
end.

In this case, the order of inheritance is as follows:

$$\text{ex1} \rightarrow \frac{a \rightarrow a1 \rightarrow a2}{\text{inheritance of a}} \quad \frac{b \rightarrow b1 \rightarrow b2}{\text{inheritance of b}}$$

- ② class ex2 has  
nature a,c ;  
end.

In this case, the order of inheritance is as follows:

$$\text{ex2} \rightarrow \frac{a \rightarrow a1 \rightarrow a2}{\text{inheritance of a}} \quad \frac{c \rightarrow b1}{\text{inheritance of c}}$$

- ③ class ex3 has  
nature a\*,c ;  
end.

In this case, the order of inheritance is as follows:

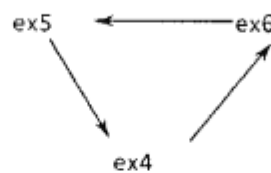
$$\frac{a \rightarrow a1 \rightarrow a2}{\text{inheritance of a}} \quad \text{ex3} \rightarrow \frac{c \rightarrow b1}{\text{inheritance of c}}$$

As above, the order of inheritance is on the depth-first basis. The inheritance loop as described below is not allowed.

class ex4 has  
nature ex5;  
end.

class ex5 has  
nature ex6;  
end.

class ex6 has  
nature ex4;  
end.



All classes implicitly inherit the meta-class “class”, in which some basic methods have already been defined, including method “new” that generates new instance objects. For details of class “class”, see section 3.6.

### 3.3 Method Definition

A method is inherited to the child class. Those methods having the same name and the same number of arguments, among the inheritance classes, become the same predicates.

```
class ex1 has
  :p( __, 1 );
end.
class ex2 has
  nature ex1 ;
  : p( __, 2) ;
end.
```

In this case, the class method “p” of the class “ex2” has the alternative and has the same meaning as when written as below.

```
class ex2 has
  :p( __, 2);
  :p( __, 1);
end.
```

There are the following three kinds of methods:

- (1) Before-demon predicate  
This is defined as follows:

```
before:method-name(arguments) [:-body ] ;
```

- (2) Principal predicate  
This is defined as follows:

```
:method-name(arguments) [ :-body ] ;
```

- (3) After-demon predicate  
This is defined as follows:

```
after:method-name(arguments) [ :-body ] ;
```

A demon indicates preprocessing or postprocessing for an ordinary principal predicate. (From a viewpoint of a logic programming language, a demon limits the application of axioms.)

```

class lock has
instance
  component lock := unlocked ;
  :unlocked(Obj) :- Obj!lock == unlocked ;
  :lock(obj) :- Obj!lock := locked ;
  :unlock(Obj) :- Obj!lock := unlocked ;
end.

class with__a__lock has
instance
  component lock is lock ;*
  before :open(Obj) :- :unlocked(Obj!lock) ; ..... ①
end.

class door__with__a__lock has
  nature with__a__lock ;
instance
  :open(Obj) :- ..... ; ..... ②
end.

```

The instance object of class “door\_\_with\_\_a\_\_lock” indicates a door with a lock. The door can be opened only when it is not locked. That is, when method “:open(Obj)” is issued to the instance object of class “door\_\_with\_\_a\_\_lock”, the before-demon predicate shown as ① is executed before the principal predicate shown as ②, and if it is ascertained that the door is unlocked, then the door can be opened. If the before-demon predicate ① fails, the principal predicate ② is not executed and method “:open(Obj)” results in failure.

---

\* This indicates to generate an instance object of the class “lock” and store it in the slot “lock” in advance. (See section 3.5.)

A method is formed from an AND combination of the following three predicates:

(1) AND combination of before-demon predicates

This is what is made by performing an AND operation to the before-demon predicates of all inherited classes in the order of the inheritance.

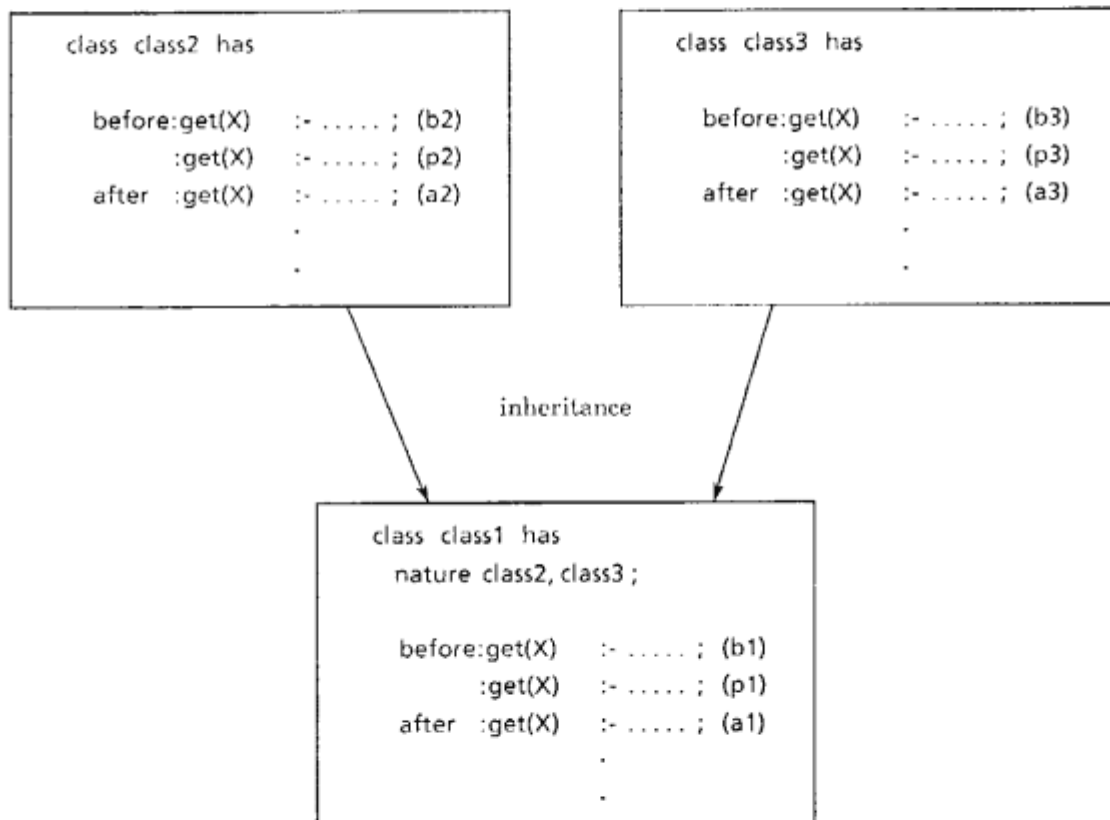
(2) OR combination of principal predicates

This is what is made by performing an OR operation to the principal predicates of all inherited classes in the order of the inheritance.

(3) AND combination of after-demon predicates

This is what is made by performing an AND operation to the after-demon predicates of all inherited classes in the order of the inheritance.

The before-demon means preprocessing, and what is first inherited is executed first. After-demon means post-processing, and what is first inherited is executed last. That is, they grow towards the outside of the principal predicate.



(a1)(p1)... mean the predicates in a class (i.e., a group of clauses having the same predicate name and the same number of arguments).

With the above example, method “:get(X)” of class “class1” is composed as follows:

:get(X):-

<u>b1, b2, b3,</u>	<u>(p1;p2;p3;fail),</u>	<u>a3, a2, a1;</u>
AND combination of before-demon predicates	OR combination of before-demon predicates	AND combination of before-demon predicates

Since they are combined as above, the following points need to be noted:

- (1) Since before-demon predicates are combined by AND, if any of them fails, the principal predicates are not executed. If a middle predicate has an entity that is to be unified with an argument, the result is passed to a later predicate call. Therefore, it is possible to determine the value of an argument in a before-demon predicate and pass that information to the principal predicate.

```

class ex1 has
  before :p(__, Y) :- Y = 1 ; ..... ①
end.

class ex2 has
  nature ex1 ;
  :p( __, X) :- body ; ..... ②
end.

```

If you call class method “p” with :p(#ex2, A), ① causes unification of 1 with A and then ② is called. Therefore, it is the same as the case when ② is called with :p(#ex2, 1).

- (2) If a principal predicate has alternatives (i.e., multiple solutions), they may be limited to one of them by an after-demon predicate.

```

class ex3 has

    :q( __, 1) ;
    :q( __, 2) ;
    :q( __, 3) ;

end.

class ex4 has
    nature ex3 ;

    after :q( __, __) :- relative __cut(1) ;*

end.

```

A call by `:q(#ex3, X)` results in alternatives, while a call by `:q(#ex4, X)` does not.

---

\* Multi-level cut. `!` and `relative __cut(0)` perform the same operation. In the case of the parsed run code, however, the operation is not guaranteed.



### 3.4 Method Call

There are the following three kinds of method calls:

#### (1) Normal method call

```
[ Example ]      : append (Obj, [ 1, 2 ] , [ 3, 4 ] , X)
```

	method	class object or instance object
	name	

#### (2) Class method call with a class name specified

(The class name is not preceded by a colon.)

```
[ Example ]      ex3 : create (#ex4, X)
```

	class	method	class
	name	name	object

#### (3) Instance method call with a class name specified

(The class name is preceded by a colon.)

```
[ Example ]      : ex3 : initialize (Obj, X)
```

	class	method	instance object
	name	name	

With a normal method call, the method to be actually called is dynamically determined by the object specified at the first argument. The argument only needs to be an object at the call at run time and thus it may be a variable that will be unified with an object just before the actual call.

With a class method call with a class name specified, the class method of an explicitly specified class (ie., a combination of the method defined in the class and the method defined in the parent class) is called. The first argument of the method call must be a class object, and the specified class must be the parent class of that class object. The class object only needs to be determined at run time, but the class name must be explicitly specified at class definition.

```

class ex5 has
  :p( __, 1) ;
  :p( __, 2) ;
  :p( __, 3) ;
end.

class ex6 has
  nature ex5 ;
  after :p( __, __) :- relative __cut(1) ;
end.

class ex7 has
  nature ex5, ex6 ;
end.

```

If class method “p” is called with `ex5:p(#ex7,X)`, the class method of class “ex5” which has alternatives is called. If class method “p” is called with `ex6:p(#ex7,X)`, the class method of class “ex6” which has no alternatives is called.

An instance method call with a class name specified is also accomplished in the same way, except that the class name must be preceded by a colon.

For a method call, as described above, the object at the first argument only needs to have been determined by the run time. In the usual notational convention, however, the method name must be specified with an atom at class definition. ESP supports a means by which the method name and the number of arguments may be dynamically determined. This is achieved using method “`refute`” of class “`class`”\* which is automatically inherited to any class.

```

:refute  (object, method-name, argument-vector)
         class object or instance object

```

---

\* For details of class “class”, see section 3.6.

This calls the method (specified by "method-name") of the object (specified by "object") using the elements of "argument-vector" as the arguments. The argument-vector is a vector whose elements are related to the arguments sequentially from the beginning. The method-name and the argument-vector may be specified by a logical variable, which only needs to have been defined by the run time of :refute.

```
class ex0 has
  :p( __, Obj, Pos, ArgVect) :-
    method__data(Vector),
    vector__element(Vector, Pos, Method__name),
    :refute(Obj, Method__name, ArgVect) ;      ..... ①
local
  method__data( {test0, test1, test2}) ;
end.
```

If class method "p" is called with :p(#ex0,Obj,0,{X,Y}), then ① calls :test0(Obj,X,Y). If class method "p" is called with :p(#ex0,Obj,2,{ }), then ① calls :test2(Obj).

### 3.5 Slot

ESP is a programming language developed by introducing the feature of an object-oriented language to Prolog, and has two kinds of variables:

- Logical variable of Prolog
- Slot, which stores the nature and internal state of an object

The logical variable features (1) it is defined by unification and (2) it is undefined by backtracking. In other words, it is a variable that does not have side effects. On the other hand, the slot has the same nature as a variable used in a procedural language. That is, the slot is a physical variable that can be redefined arbitrarily. Also, it is not undefined by backtracking. In other words, the slot is a variable that has side effects.

The logical variable and the slot are the same in that they do not need to have their data type determined in advance, that is, they can have a value of any data type, such as an integer number, a floating-point number and a string. (However, the slot cannot have a stack vector or a variable that does not have a value.\*) That slot is one of those functions that support the ESP's powerful programming capability.

To reference a slot, you need to specify the object and the slot name as a pair. Generally, more than one instance object is generated for a class and these instances have the same slot name. Because of this, you need to specify a particular slot of a particular object.

There are two kinds of slots: attribute and component. A slot name must be an atom. A slot is defined, as shown below, starting with the keyword ("attribute" or "component"), followed by a slot definition (delimited by a comma if more than one) and ending with a semicolon.

```
attribute slot1, slot2;  
component slot3, slot4;
```

You can specify the initial value of a slot at slot definition.

#### (1) Slot name (atom) only

Integer 0 is stored as the initial value.

#### (2) slot-name := initial-value

Give the initial value by a known value. The initial value must be an atomic, string or class object (#class-name). (You cannot specify such a term that cannot originally be substituted to a slot, such as a vector, list or variable.)

---

\* you may use the pool function (see chapter six) to store the structural data (e.g., list or vector) as the slot value.

(3) slot-name is class-name

Generate an instance object for the class specified by "class- name" and store it as the slot value.

(4) slot-name := initial-value :- body

This is what is called a slot initialization predicate, which stores the execution result of "body" as the slot value.

Storage of a slot initial value (i.e., execution of a slot initialization predicate), if it is a class slot, is performed when the class is generated. Therefore, since the class is not generated if the initialization predicate fails, you should not use an initialization predicate for a class slot if possible. Note that this applies not only to the case of (4) above, but also to the case of (3) and the case of the storage a class object of (2). For an instance slot, the storage of the initial value is performed when an instance object is generated by method "new".

```
class ex0 has
  attribute    a ;
  component    b := 12, c := "string", d := #ex1 ;
  attribute    ( d := W :- :create(#essential__window, W),
                :set __size(W, 100, 100)),
  e is lock ;
```

As shown above, you may write the keyword ("attribute" or "component") as many times as you want. (However, you cannot write it between method definitions.) If the body part of a slot initialization predicate has more than one goal, enclose them in parentheses as follows:

```
attribute
  (a := X :- body1, body2),
  (b := Y :- body3, body4);
```

If you write "a := (X :- body1, body2)" instead, it is interpreted as an instruction to store a compound term having ":" as the functor into the slot, and it results in an error at initialization.

---

\* Because the priority order of the operators are:

:- greater than , greater than :=

A slot can be initialized as described above. However, you should pay attention when using a slot initialization predicate for the following reasons:

- If the initialization predicate fails, generation of the class object or method “new” fails too.
- The execution order of initialization predicates is uncertain. Pay attention especially when writing a predicate that has side effects.
- When initializing an instance slot, you cannot use the object of the instance slot itself.

For initialization of a class slot, it would be safer to use an initialization class method instead of the above initialization specification function. For initialization of an instance slot, you can issue an instance method of its own using an after-demon for method “new”.

Since the second argument of the after-demon of method “new” has already been unified with the instance object just generated, you can issue an instance method of its own class and access to the instance slot. From this viewpoint, it is more advantageous than when using an initialization predicate. On the other hand, initialization by a demon has a difficulty in alteration by overriding using an inheritance. You may use either technique as required.

```
class ex1 has
  attribute (window := W := :create(#essential__window, W),
            :set __size(W, 100, 100));

  instance
  attribute list := L :- :create(#list, L);
end.
```

```
class ex1 has
  attribute window ;
  :set __window(Class) :-
    :create(#essential__window, W),
    :set __size(W, 100, 100),
    Class!window := W ;

  after :new( __, Obj) :-
    :create(#list, List),
    Obj!list := List ;

  instance
  attribute list ;
end.
```

There are four kinds of slots:

- Class attribute slot
- Class component slot
- Instance attribute slot
- Instance component slot

Only the following two pairs among the four slots can have the same slot name in the same class:

Class attribute slot and instance component slot

Class component slot and instance attribute slot

The other pairs cannot have the same slot name. For example, the following definitions are not permitted:

```
class ex1 has
  attribute slot1 ;
  component slot1 ;
end.
class ex2 has
  component slot2 ;
  instance
    component slot2 ;
end.
```

An attribute slot and a component slot are different in the following points. An attribute slot corresponds to a global variable of a procedural language and can be referenced by specifying a pair of an object and a slot name atom from outside the class definition. A component slot corresponds to a local variable and cannot be referenced from outside the class definition. It can be referenced within the same class definition, similar to an attribute slot, by specifying a pair of an object and a slot name.

Attribute slot	:	Global variable
Component slot	:	Local variable

An attribute slot name is associated with a special identifier for each class, and the slot is actually associated with the identifier only for slot access within the class. Therefore, if a slot name is specified statically at definition, you cannot access a slot of the same name in a different class. Contrarily, if a slot name is specified dynamically at definition, it is not converted to the identifier specific to a class, you cannot access the component slot even in the same class definition.

```

class ex0 has
  component slot0 := 0 ;
  :get0(Class, X) :- X = Class!slot0 ;
  :get01(Class, Obj, X) :- X = Obj!slot0 ;
  :get02(Class, Obj, X) :- p(Slot), X = Obj!Slot ;
  local
    p(slot0) ;
end.

class ex1 has
  nature ex0 ;
  :get1(Class, X) :- X = Class!slot0 ;
end.

class ex2 has
  nature ex0 ;
  component slot0 := 2 ;
  :get2(Class, X) :- X = Class!slot0 ;
end.

class ex3 has
  attribute slot0 := 3 ;
end.

```

The particular results are as follows:

```

:get0(#ex0, X) ... X = 0
:get1(#ex1, X) ... Error
:get0(#ex0, X) ... X = 0
:get2(#ex2, X) ... X = 2
:get0(#ex2, X) ... X = 0
:get01(#ex0, #ex0, X) ... X = 0
:get01(#ex0, #ex3, X) ... Error
:get02(#ex0, #ex0, X) ... Error
:get02(#ex0, #ex3, X) ... X = 3

```



An attribute slot and a component slot are processed differently at class inheritance. If the same variable name exists in both inheriting and inherited class definitions, the slot is processed as follows:

Attribute slots :	degenerated to one variable.
Component slots :	remain as they are, and are not degenerated.

```
class c1 has
  attribute a, b ;
  component d, e ;
end.
```

```
class c2 has
  nature c1 ;
  attribute a, c ;
  component d, f ;
end.
```

In this case, the class object of class "c2" has the area to store the following slot values:

attribute

a
b
c

component

d of c1
e of c1
d of c2
f of c2

```

class with__a__front__door has
  instance
  component door := closed ;
  :open__front(Obj) :- Obj!door := opened ;
  :close__front(Obj) :- Obj!door := closed ;
end.

```

```

class with__a__back__door has
  instance
  component door := closed ;
  :open__back(Obj) :- Obj!door := opened ;
  :close__back(Obj) :- Obj!door := closed ;
end.

```

```

class room has
  nature
  with__a__front__door ;
  with__a__back__door ;
end.

```

Therefore, you should take the above difference into consideration when you determine whether to define a slot as an attribute slot or a component slot.

In the above example, if you define the slots “door” of “with\_\_a\_\_front\_\_door” and of “with\_\_a\_\_back\_\_door” as an attribute slot instead of a component slot, then both slots are degenerated and thus when you open the front\_\_door, the back\_\_door reacts as if it is opened too. If attribute slots are degenerated, the initial value first inherited has priority over the others.

```

class a1 has
  attribute slot0 := 1 ;
end.

```

```

class a2 has
  attribute slot0 := 2 ;
end.

```

```

class a3 has
  nature a1, a2 ;
end.

```

```

class a4 has
  nature a2, a1 ;
end.

```

The initial value of #a3!slot0 is 1, and that of #a4!slot0 is 2. If an attribute slot and a component slot coexist as a result of the inheritance, they become as follows:

```

class a1 has
  attribute slot0 := 1 ; ..... ①
end.

```

```

class a2 has
  nature a1 ;
  component slot0 := 2 ; ..... ②
  :get2(Obj, X) :- X := Obj!slot0 ;
end.

```

```

class a3 has
  nature a2 ;
  :get3(Obj, X) :- X = Obj!slot0 ;
end.

```

With class "a2", component slot "slot0" has priority over attribute slot "slot0". This priority order holds regardless of the order of inheritance. Since component slot "slot0" cannot be accessed from class "a3", the attribute slot is accessed instead. Therefore, variable X of `:get2(#a3,X)` is unified with 2, and variable Y of `:get3(#a3,Y)` is unified with 1. In other words, you cannot use format "object!slot-name" to access the attribute slot ① from within class "a2", and similarly you cannot use the same format to access the component slot ② from within class "a3".

As described above, you cannot store such a data item as an undefined variable, stack, vector or list in a slot. To store structural data in a slot, you may use the pool function (which is a structure having side effects, actually an instance object.) For details of the pool function, see section six "BASIC CLASSES".

The following program example has a function to convert a general term to an instance object (strictly speaking, an instance object with the converted data store in the instance slot). The program has the following three slots:

```
frozen      ..... Maintains information about the stored undefined variables and vectors.

variables   ..... Maintains the number of the stored undefined variables.

externals   ..... Maintains items that can originally be stored (e.g., integer numbers and strings).
```

frozen consists of a 32-bit string, whose upper 2 bits are used to indicate the type of stored data.

```
10 ..... Indicates a variable. The remaining 30 bits are used to indicate the variable number.

11 ..... Indicates a stack vector. The remaining 30 bits are used to indicate the length.

00 ..... Indicates normal data. The remaining 30 bits are used to indicate the position of the data
        stored in slot "externals".
```

Since any of them does not need as much as 30 bits in most cases, data compression using an 8-bit string instead of the 32-bit string would improve efficiency.

Two methods are provided: class method "freeze" which converts a term to an instance object, and instance method "melt" which reconverts it.

```
:freeze (#frozen__term, [1, 2, X, Y, X], Obj)
```

This example converts the list [1,2,X,Y,X] to a format suitable for storage in the slot and stores it in the slot of instance object "Obj".

Then, if you execute

```
:melt(Obj,Out)
```

[1,2,A,B,A] is returned to Out. The undefined logical variable names are just for convenience and do not have a special meaning. When “freeze” is performed, if the undefined variables are identical to each other (since they have been unified with each other, for example), they remain identical even after “melt”. The same function as this program example has been provided by SIMPOS with the same class name (“frozen\_\_term”), which uses the built-in predicates “stack\_\_to\_\_heap\_\_vector” and “heap\_\_to\_\_stack\_\_vector”. However, you may learn a data storage technique from this program example.

```

class frozen__term has

:freeze(Class, Term, Frozen) :-
    :new(Class, Frozen),
    ( freeze(Term,
        0, Var,
        0, Frz, Frozen__list, []),
      0, Ex, Ext__list, []),
      new__string(Str, Frz, 32), fill__string(Frozen__list, 0, Str),
      new__heap__vector(Vct, Ext), fill__vector(Ext__list, 0, Vct),
      Frozen!variables := Var,
      Frozen!externals := Vct,
      Frozen!frozen    := Str,
      fail;
      true );

instance
attribute
    frozen, variables, externals;

:melt(Frozen, Melted) :-
    new__stack__vector(Vars, Frozen!variables),
    string(Frozen!frozen, Length, 32),
    melt(0, __, Frozen!frozen, Vars, Frozen!externals, Melted);

```

local

```
freeze(X, V0, V, F0, F, FLO, FL, E0, E, ELO, EL) :-
    stack_vector(X, 2), first(X, Y), unbound(Y), !,
    freeze_vector(2, X, V0, V, F0, F, FLO, FL, E0, E, ELO, EL);
freeze('$$VAR'(V), V, V+1, F, F+1,
    [V+16#"80000000"[FL], FL, E, E, EL, EL) :- !;
freeze('$$VAR'(N), V, V, F, F+1,
    [N+16#"80000000"[FL], FL, E, E, EL, EL) :- !;
freeze(X, V0, V, F0, F, FLO, FL, E0, E, ELO, EL) :-
    stack_vector(X, N), !,
    freeze_vector(N, X, V0, V, F0, F, FLO, FL, E0, E, ELO, EL);
freeze(X, V, V, F0, F0+1, [E|FL], FL, E, E+1, [X|EL], EL);

freeze_vector(S, X, V0, V, F0, F,
    [S+16#"C0000000"[FLO], FL, E0, E, ELO, EL) :-
    freeze_args(0, S, X, V0, V, F0+1, F, FLO, FL, E0, E, ELO, EL);

freeze_args(S, S, __, V, V, F, F, FL, FL, E, E, EL, EL) :- !;
freeze_args(K, S, X, V0, V, F0, F, FLO, FL, E0, E, ELO, EL) :-
    vector_element(X, K, XK),
    freeze(XK, V0, V1, F0, F1, FLO, FL1, E0, E1, ELO, EL1),
    freeze_args(K+1, S, X, V1, V, F1, F, FL1, FL, E1, E, EL1, EL);

fill_string([], __, __) :- !;
fill_string([HIT], K, S) :-
    set_string_element(S, K, H), fill_string(T, K+1, S);

fill_vector([], __, __) :- !;
fill_vector([HIT], K, S) :-
    set_vector_element(S, K, H), fill_vector(T, K+1, S);

melt(K0, K, S, V, E, M) :-
    string_element(S, K0, SK),
    melt_one(SK>>30, SK^16#"3FFFFFFF", K0+1, K, S, V, E, M);

melt_one(0, K, N, N, S, V, E, EK) :- !,          % normal
    vector_element(E, K, EK);
melt_one(1, K, N, N, S, V, E, K) :- !;
melt_one(2, K, N, N, S, V, E, VK) :- !,          % variable
    vector_element(V, K, VK);
melt_one(3, Size, N0, N, S, V, E, X) :-          % vector
    melt_vector(Size, N0, N, S, V, E, X);

melt_vector(Size, N0, N, S, V, E, X) :-
    new_stack_vector(X, Size),
    melt_args(0, Size, N0, N, S, V, E, X);

melt_args(Size, Size, N, N, __, __, __, __) :- !;
melt_args(K, Size, N0, N, S, V, E, X) :-
    melt(N0, N1, S, V, E, XK),
    vector_element(X, K, XK),
    melt_args(K+1, Size, N1, N, S, V, E, X);
end.
```

### 3.6 Class "class"

Class "class" is implicitly inherited to any ESP classes and provides the basic methods that can be used in any classes.

#### (1) Methods common to class/instance

`:get__slot(Object, Slot-name, ^Value)`

Unifies the value of the slot specified by the "Slot-name" atom of "Object" with the value passed to the argument, value.

`:set__slot(Object, Slot-name, Value)`

Updates the value of the slot specified by "Slot-name" atom of "Object" with the value passed to the argument, value.

`:method(Object, ^Method-name, ^Number-of-arguments)`

Succeeds if "Object" has the method of "Method-name" and "Number of arguments", otherwise fails. If this method is called with "Method-name" or "Number-of-arguments" remaining as a variable, it returns the information of all methods of the object as the alternatives one after another.

`:slot(Object, ^Slot-name)`

Succeed if "Object" has the slot of "Slot-name", otherwise fails. If the method is called with "Slot-name" remaining as a variable, it returns the names of all slots of the object as the alternatives one after another.

`:is__class(Object)`

Succeeds if "Object" is a class object and fails if it is an instance object.

`:refute(Object, Method-name, Argument-vector)`

Calls the method specified by "Method-name" with the "Object" and the elements of "Argument-vector" used as the arguments. To specify "Argument-vector", use a stack vector.

`:class__object(Object, ^Class-object)`

For a normal method call, it returns the class object of the class to which the specified object belongs. For a method call with a class name specified, it returns the class object of the specified class.

`:undefined__method(Object, Method-name, Argument-vector, Address)`

This method is called when an undefined method attempts to be called, and generates an exception together with the given information.

"Address" is the address (integer) of the method call that has performed the call for the undefined method. Usually the user does not use this.

The user can handle an undefined method call by overriding this method in the class definition. Since this method has been implemented in a special way, it will not be called as an alternative even if the overridden method fails.

**:undefined\_\_slot(Object, Type, Slot-name, Value, Address)**

This method is called when access of an undefined slot is attempted, and generates an exception together with the given information.

"Type" indicates the type of built-in predicate that has accessed an undefined slot (the type is either "get\_\_slot" or "set\_\_slot"). "Value" is the third argument of "get\_\_slot" or "set\_\_slot". "Address" is the address (integer) at which the undefined slot access has been made. Usually the user does not use this address.

The user can handle an undefined slot access call by overriding this method in the class definition. Since this method has been implemented in a special way, it will not be called as an alternative even if the overridden method fails.

## (2) Class method

**:new(Class, ~ Instance-object)**

Generates one new instance object and returns it in the second argument. The slot of the generated instance is initialized according to the instructions written in the program.

**:package\_\_name(Class, ^Package-name)**

Unifies the package name atom of the package belonging to "class" with the package name passed to the argument, value.

**:class\_\_name(Class, ^Class-name)**

Unifies the class name atom of "Class" with the class name passed to the argument, value.

**:class\_\_id(Class, ^Class-identifier)**

Unifies the class identifier atom of "Class" with the class identifier passed to the argument.

**:super(Class, ^Parent-class-object)**

Succeeds if the second argument is the specified class object of the parent class of the class to which this method belongs, otherwise fails. If the method is called with the second argument remaining as an undefined variable, it unifies all class objects of the parent class with the second argument as the alternatives one after another.



### (3) Precautions

For improvement of the execution speed, methods “get\_\_slot” and “set\_\_slot” have been implemented in a special way. Because of this, you cannot define a principal predicate (a normal method) and a demon for these methods.

Also, the following methods cannot define a principal predicate (they can define demons.)

new, super, and class\_\_object

To the other methods, you can define a principal predicate and demons freely.

## 3.7 Macro

### 3.7.1 General

The ESP language supports various macro expansion functions and this is one of the main features of ESP. A macro expansion function automatically works to a specific pattern appearing in an ESP program and expands it to a prescribed pattern. Besides the system-defined macros, you can define your own macros.

The example programs described in this manual use many macros. Generally, macros may be classified into three kinds:

- (i) Macros to express a constant value

Example: `16#A000' => 40960`  
`double __bytes#{123} => a 16-bit string whose length is 1 and whose 0th element is 123`

- (ii) Arithmetic operations

Example: `Z = X + Y => add(X,Y,Z)`  
An arithmetic operation expression using general operators is converted to a built-in predicate.

- (iii) Macros related the ESP execution mechanism

Example: `#class-name => a class object`  
`X: = Obj!slot => set_slot(Obj,slot,X)`

The system-defined macros are expanded as follows:

`<object pattern> => <generating predicate>,  
<expanded pattern>,  
<testing predicate>*`

For example,

`p(X):-q(X,Y),r(X + Y);`  
`_____`  
`<object pattern>`

is expanded to

`p(X):-q(X,Y),add(X,Y,Z),r(Z);`  
`_____`  
`<generating predicate> <expanded pattern>`

---

\* No system-defined macros require a testing predicate. It is related to a user-defined macro. (See section 3.8.)

The way a macro is expanded differs a little depending on the position where the object pattern appears.

- ① When the object pattern appears as one of the goal in the body part

Example 1:

```
p(X):-.....,one__goal(X),.....;
           <object pattern>
```

is expanded to

```
p(X):-....., <generating predicate>,
           <expanded goal>,
           <testing predicate>,.....;
```

Example 2:

```
p(Obj,X):-X = Obj!element;
```

is expanded to

```
p(Obj,X):-slot(Obj,element,X);
```

- ② When the object pattern appears as a term which is an element of a body argument or of a structural argument

Example 1:

```
p(X):-.....,one__goal(term),.....;
           <object pattern>
```

is expanded to

```
p(X):-....., <generating predicate>,
           one__goal(term'),
           <expanded pattern>
           <testing predicate>,.....;
```

Example 2:

$p(X,Y,Z):-q(\underline{\underline{X+Y}} \mid Z);$

<object pattern>

is expanded to

$p(X,Y,Z):-\underline{\underline{add(X,Y,A)}},q(\underline{\underline{A}} \mid Z);$

<expanded pattern>      <generating predicate>

- ⊕ When the object pattern appears as a term which is an element of a head argument or of a structural argument in the head part

Example 1:

$p(\underline{\underline{term}}):-\dots\dots,\text{last\_goal}(\text{Args});$

<object  
pattern>

is expanded to

$p(\underline{\underline{term'}}):-<\text{testing predicate}>,\dots\dots,\text{last\_goal}(\text{Args}),<\text{generating predicate}>;$

<expanded pattern>

Example 2:

$p(A,\underline{\underline{A+B}}):-q(A,B);$

<object pattern>

is expanded to

$p(A,X):-q(A,B),\underline{\underline{add(A,B,X)}};$

<expanded pattern>      <generating predicate>

Pay attention to the following case:

$p(\text{Obj},\text{Obj!slot\_name}):!;$

At a glance, slot access seems to be performed before the cut, but in fact it is expanded as follows:

```
p(Obj,S):-!,slot(Obj,slot__name,S);
```

If you want to make it be a program that branches based on the slot value, you must explicitly specify slot access in front of the cut of the body part.\*

```
p(Obj,S):-S = Obj!slot__name,!;
```

### 3.7.2 System-Defined Macros

#### (1) Macros to express a constant value

##### **Base#'Character-string'**

The "Character-string" is expanded to an integer value of the radix specified by the "Base". The base may be between 2 and 36. The character string may contain ten digits from 0 to 9 and letters from a to z (or A to Z). The uppercase and lowercase letters are not discriminated. Up to what digit or letter can be used depends on the radix used.

Example: 16#`'A000'` -> 40960

##### **#'Character"**

The "Character" is expanded to the corresponding character code (integer).\*\*

Example: #`'A'` -> 9025

##### **control#'Character'**

##### **meta#'Character'**

##### **control\_\_meta#'Character'**

The "Character" is expanded to the same code (integer) as when you press that character key on the keyboard while holding down the control key, the meta key, or both, respectively.

##### **key#Name, meta#Name**

The "Name" is expanded to the code (integer) corresponding to the special key designated by that name on the keyboard. The following special key names are supported:

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

---

\* This point is also related to TRO (Tail Recursion Optimization) described later. That is, even if the last goal is written as indicating itself, a generating predicate is inserted after it when the macro is expanded, and TRO may not work.

\*\* The standard character code system of PSI conforms to the JIS 16-bit code system.

### **pf#Number**

The "Number" is expanded to the code corresponding to the function key of that number on the keyboard. The number may be an integer value between 0 and 18.

### **keypad#Character**

The "Character" is expanded to the code corresponding to the character on the key pad provided on the right side of the keyboard. The character may be a digit (0 to 9) or a symbol (comma, period, or hyphen).

### **mouse#Click**

The "Click" is expanded to an atom corresponding to a mouse click input. The click may be 1, 11, m, mm, r, or rr (which corresponds to one left click, two left clicks, one middle click, two middle clicks, one right click, and two right clicks, respectively.) The expanded atom may be expressed in the form of 'mouse#1', for example.

### **ascii#Character-string**

The "Character-string" is expanded to the corresponding ASCII code string (a string of 8-bit codes).

### **string#Character-string, jis#Character-string**

The "Character-string" is expanded to the same string of JIS 16-bit codes as when you simply write "Character-string". This macro has been provided on PSI so as to maintain compatibility with the developing cross-system. Usually you do not need to use this macro on PSI.

<b>words#{Vector}</b>	... 32-bit string
<b>double__bytes#{Vector}</b>	... 16-bit string
<b>bytes#{Vector}</b>	... 8-bit string
<b>bits#{Vector}</b>	... 1-bit string

Each element of the string is given by an integer value (code), not a character. You may give an integer value for each element that can be expressed by as many bits as allowed for that string.

Example: `bits#{1,0,0,1,1}`  
`double__bytes#{16#`FF00`,16#`12AB`}`

At expansion of these macros, the object pattern directly corresponds to the expanded pattern without any generating predicate added.

## **(2) Arithmetic operations**

The arithmetic operations are grouped mainly into two kinds: those related to the unification and the comparison and the other related to the operators.

## ① Unification, comparison

<code>X is Y</code>	.....	<code>unify(X, Y)</code>
<code>X = Y</code>	.....	<code>unify(X, Y)</code>
<code>X == Y</code>	.....	<code>equal(X, Y)</code>
<code>X \= Y</code>	.....	<code>not_equal(X, Y)</code>
<code>X =; Y</code>	.....	<code>identical(X, Y)</code>
<code>X \=; Y</code>	.....	<code>not_identical(X, Y)</code>
<code>X &lt; Y</code>	.....	<code>less_than(X, Y)</code>
<code>X =&lt; Y</code>	.....	<code>not_less_than(Y, X)</code>
<code>X &gt; Y</code>	.....	<code>less_than(Y, X)</code>
<code>X &gt;= Y</code>	.....	<code>not_less_than(X, Y)</code>

For details of comparisons using built-in predicates for each data type, see the built-in predicate manual. The following briefly describes only those comparisons that are often used:

### (i) Comparison of atoms

Atoms are compared with respect to their atom numbers (integer), regardless of the corresponding names.

### (ii) Comparison of strings

Strings are compared with respect to their character codes on the lexical order basis.

```
'aa' < 'bb'
'a' < 'ab'
```

A comparison using operator `==` or `:=` succeeds only if both entities are identical as well as their stored locations.

```
'abc' = 'abc' ... fails
'abc' == 'abc' ... fails
'abc' := 'abc' ... fails
equal_string('abc', 'abc') ... succeeds*
```

....., V = {"abc", 0, 1},	The string of the 0th element of V is unified with
first(V, X), . . . .	X. The string of the 0th element of V is unified
first(V, Y),	with Y.
X == Y,	Succeeds.
X := Y, .....	Succeeds.

---

\* This is the built-in predicate that compares the strings, only with respect to their elements.

(iii) Comparison of stack vectors

The length and elements of vectors are compared. First their length are compared. If their lengths are identical, then their elements are compared, sequentially starting from the 0th element.

$\{1,1\} > \{2\}$   
 $\{1,2\} > \{1,1\}$

"=" or "is" causes unification.

$\{X,1\} = \{2,Y\}$  ... X and 2, and 1 and Y are unified.

A comparison using "=" succeeds if the number of elements is identical and if every corresponding element holds the "=" relationship. A comparison using "==" succeeds only if their stored locations are identical as well.

$\{1, 2, a\} == \{1, 2, a\}$  Succeeds  
 $\{1, 2, a\} =: \{1, 2, a\}$  Fails

....., V = { { 1, 2, a}, 3},  
first(V, X),  
first(V, Y),  
X == Y, ..... Succeeds

(iv) Comparison of different data types

The following shows the comparison between some of different data types. (For details, see the built-in predicate manual.) This comparison is not made on any logical basis but just for convenience. Therefore, you should avoid comparison between different data types if possible.

(less) undefined variable - atom - integer number -  
floating-point number - vector - string (greater)



## ② Arithmetic operations

The following macro operators have been provided for two- argument and one-argument arithmetic operations. For details of each built-in predicate, see chapter four.

Object pattern	Generating predicate*	Expanded pattern	
$X + Y$	<code>add(X, Y, Z)</code>	$Z$	Addition
$X - Y$	<code>subtract(X, Y, Z)</code>	$Z$	Subtraction
$X * Y$	<code>multiply(X, Y, Z)</code>	$Z$	Multiplication
$X / Y$	<code>divide(X, Y, Z)</code>	$Z$	Division
$X \text{ div } Y$	<code>divide__with__remainder(X, Y, Z, __)</code>	$Z$	Division (integer)
$X \bmod Y$	<code>divide__with__remainder(X, Y, __, Z)</code>	$Z$	Remainder
$-X$	<code>minus(X, Y)</code>	$Y$	Reverse sign
$X \setminus Y$	<code>and(X, Y, Z)</code>	$Z$	Logical multiply
$X \setminus / Y$	<code>or(X, Y, Z)</code>	$Z$	Logical add
$X \text{ xor } Y$	<code>xor(X, Y, Z)</code>	$Z$	Exclusive OR
$X >> Y$	<code>shift__right(X, Y, z)</code>	$Z$	Right shift
$X << Y$	<code>shift__left(X, Y, Z)</code>	$Z$	Left shift
$\setminus(X) **$	<code>complement(X, Y)</code>	$Y$	Reverse bit

## (3) Macros related to ESP execution mechanism

The following macros have been provided which are directly related to ESP program execution.

### **#Class-name**

Use this to denote a class object of the class specified by "Class-name".

### **:Method-name(Arguments)**

Is expanded to an ESP method call.

### **Class-name:Method-name(Arguments)**

Is expanded to a call for the class method of the class specified by "Class-name". This may be used to call an instance method of the parent class which has been overridden by the child class.

### **:Class-name:Method-name(Arguments)**

Is expanded to a call for the instance method of the class specified by "Class-name". This may be used to call an instance method of the parent class which has been overridden by the child class.

---

\* Usually, the macros are expanded on PSI to these built-in predicates.

\*\* Because a backslash ( \ ) is not declared as an operator by default, you need to enclose the argument in parentheses.

```
X = Obj!Slot__name
X = = Obj!Slot__name
Obj!Slot__name = X
Obj!Slot__name = = X
```

Gets the value of the slot (specified by "slot\_\_name") of the object (specified by "Obj!") and unifies it with X. "Obj!" and "Slot\_\_name" may remain as a variable and only need to have a value (object and slot name atom respectively) by run time.

**Obj!Slot\_\_name:= X**

Substitutes X for the slot (specified by "Slot-name") of the object (specifies by "Obj").

**Obj!Slot\_\_name**

This is a pattern that appears as an element of an argument or a structural argument. The predicate that reads out the value of the slot (specified by "Slot\_\_name") of the object (specified by "Obj") becomes the generating predicate, and the result becomes the expanded pattern.

#### (4) Other macros

**unique\_\_atom(undefined-variable)**

Generates a unique atom and replaces with it. All those undefined variables that are identical in the class are replaced with the generated atom. For example, describing unique\_\_atom(A) causes not only that description to be replaced with the generated unique atom but also every variable A in the class is replaced with the generated atom.

**standard#input**

Is expanded to the standard input port provided in the processing being currently executed (usually an instance object for the standard input is set up.) For the debugger and the shell, the initial value will be the window.

**standard#output**

Is expanded to the standard output port provided in the process being currently executed (usually an instance object for the standard output is set up.) For the debugger and the shell, the initial value will be the window.

**standard#message**

Is expanded to the message output port provided in the process being currently executed (usually an instance object for the standard output is set up.) For the debugger and the shell, the initial value will be the window.

## (5) Suppression of macro expansion

In principle, macros are automatically expanded. However, you may suppress the macro expansion by using a back-quote symbol (```). The back-quote key is at the top right on the PSI keyboard.

```(term)`

Suppresses expansion of macros in the term and treats the term as is.

`^(term)`

Suppresses expansion of only top-level macros in the term. The deeper-level macros are expanded.

`p(X, Y) :- q( (X-2)-(Y-3) );`

↓

`p(X, Y) :- subtract(X, 2, A),  
          subtract(Y, 3, B),  
          subtract(A, B, C),  
          q(C);`

`p(X, Y) :- q( ``((X-2)-(Y-3)) );`

↓

`p(X, Y) :- q( (X-2)-(Y-3) );`

Vectors combined by the operator “-”

`P(X, Y) :- q( ` ( (X-2)-(Y-3) ) )`

↓

`p(X, Y) :- subtract(X, 2, A),  
          subtract(Y, 3, B),  
          q(A-B);`

Macro expansion does not recognize the difference between a macro that appears in a goal of the body (which is directly executed) and a macro that appears in an argument (which is treated as data).

`p(X, Y) :- q(X = Y);`

↓

`p(X, Y) :- q(unify(X, Y));`

`p(X, Y) :- q(X : Y);`

This attempts to be expanded to a method call and results in an error.

With this example, if you want to write a compound term whose functor is = or :, you must use a single back-quote (') or a double back-quote (``) to suppress the macro expansion.

```
p(X, Y) :- q(' (X = Y) );
p(X, Y) :- q('' (X : Y) );
```

A single or a double back-quote (' or ``) is an atom. For example, ``(X) is a compound term whose functor is ``.

```
p(X, Y) :- q(' (X-2, Y-3) ); ..... A two-element compound term whose functor is ``.*
      ↓
P(X, Y) :- subtract(X-2, A),
            subtract(Y-3, B),
            q(' (A, B) );

p(X, Y) :- q('' ((X-2, Y-3) ));
      ↓
P(X, Y) :- q( (X-2, Y-3) );
```

If you want to write a one-element compound term whose functor is ' or '', write as follows:

```
' ( { ' , A } )      ' ( { '' , B } )
```

### 3.7.3 User-Defined Macros

Besides the standard system-defined macros described in the previous section, you may add your own macros and modify them. Different from a class definition, definition of a user-defined macro is accomplished by a macro bank. The macro banks that you have defined are stored in the system and you can use them at any time.

The ESP macro expansion function does not only replace a certain pattern with a predefined pattern but also inserts necessary goals before and after the goal containing that pattern and/or adds a newly generated predicate. An ESP program consists of a class definition part (which defines inheritance classes, class slots and class methods), an instance definition part (which defines instance slots and instance methods) and a local definition part (which defines local predicates). Each part is composed of a series of Prolog clauses terminated by a semicolon. Macro expansion is carried out in two levels: the first level macro expansion is performed when a given term is disassembled to clauses (class- definition-level

---

\* As explained in section 2.4 "Precautions for Operator- Applied Terms and Compound Terms", p(X,Y) and p((X,Y)) have different meanings.

expansion). The second level macro expansion is performed when each clause is disassembled to components (clause-level expansion).

A macro bank is defined as follows:

```
macro_bank macro-bank-name has
    (atom)
[ nature macro-bank-names; ]

    slot definition

    macro definition

local

    local predicate definition

end.
```

Slot definition, similar to a slot of ESP, corresponds to an instance slot. Since macro definition is performed by the instance object of this macro bank, you may use slot definition to maintain the intermediate state of macro expansion.

Macro definition is a series of macro definition statements terminated by a semicolon. There are three kinds of macro definition statements:

- i) macro definition statement for clause-level expansion
- ii) macro definition statement for class-definition-level expansion
- iii) insertion statement for a clause

#### (1) Clause-level macro definition

A clause-level macro is for an argument of the head part of a Prolog clause and for a goal and its argument of the body part. A clause-level macro is defined as follows:

```
<object pattern> => <expanded pattern>
    [when <a series of generating goals> ]
    [where <a series of testing goals> ]
    [with <a series of added predicates> ]
    [by <instance object>
    [:- <expansion condition>
```

The underline parts indicate a keyword. The object pattern and the expanded pattern are a term and may include a variable. You may omit those entities that are enclosed in square brackets and . If you omit a series of added predicates, a term (a unifiable term) that matches the object pattern is generally previous section.

```

object pattern  =>  generating predicate
                  expanded pattern
                  testing predicate

```

Similar to a system-defined macro, the generating predicate, the expanded pattern and the testing predicate are placed in different places depending on where the object pattern has appeared.

If you define

```

ve(X,N) => E when vector__element(X,N,E)

```

the macro is expanded as follows:

```

p(X, Y) :- q(ve(X, Y));
      ↓
p(X, Y) :- vector__element(X, Y, Z),
          q(Z);

p(X, Y, ve(X, Y)) :- q(X, Y);
      ↓
p(X, Y, Z) :- q(X, Y),
             vector__element(X, Y, Z);

```

With a macro definition statement, similar to the body part of an usual ESP program, you may specify, in the part following ":-", execution of a local predicate call, a method call or a built-in predicate. A macro bank defined as described above always inherits class `esp__macro__expander` (the class for standard macro expansion). That is, the definition is made by altering a standard macro. Therefore, if you want to suppress all standard macros except for user-defined macros, add the following at the end of the macro definition:

```

X => __ :-!,fail;

```

Generally, if you write a cut and "fail" in the body part like below:

```

object pattern => __ :-!,fail;

```

Also, specification of "nature" allows inheritance of other macro banks. By default, whether "nature" is specified or not, the standard macro `esp__macro__expander` is inherited.

“with <a series of added predicates>” adds the specified clauses to the local definition. This may be usual where a condition is an interactive pattern, as shown in the following example:

```
some(X in List, Condition, Tail) =>
    dummy(List, X, Tail)
with(dummy([X | T], X, T) :- Condition, !;
    dummy([_ | R], X, T) :- dummy(R, X, T));
```

“in” can be used as an operator through the operator declaration (described later). If you make a definition as above, expansion is performed as follows:

```
P(List) :-
    some(X in List, (integer(X), X>10), Tail),
    q(X, Tail);
    ↓
P(List) :- dummy(List, X, Tail), q(X, Tail);
dummy([X | T], X, T) :- integer(X), X>10, !;
dummy([_ | R], X, T) :- dummy(R, X, T);
```

In this example, because you have named the condition testing predicate “dummy” in advance, it may be inconvenient if you want to use another condition since the predicate names may contend with each other. In this case, if you use standard macro `unique__atom(logical variable)`, you can cause the predicate name to be determined dynamically. You may alter the above macro definition statement as follows:

```
some(X in List, Condition, Tail) =>
    {unique__atom(Dummy), List, X, Tail}
with({Dummy, [X | T], X, T) :- Condition, !;
    {Dummy, [_ | R], X, T) :- {DummyR, X, T}};
```

An undefined variable specified by keyword “by” is unified with the instance object of the macro bank in current macro expansion. This makes it possible to access the slot defined in the slot definition part.

```
macro __bank m1 has
attribute unique__number := 0;
unique__number => Num by Macro
    :- Num = Macro!unique__number,
       Macro!unique__number := Num + 1;
end.
```

With this macro definition, the macros

```
p(unique__number); p(unique__number);  
r(unique__number);
```

are expanded to

```
p(0) ; p(1) ;  
r(2) ;
```

## (2) Class-definition-level macro expansion

An ESP program itself consists of one large term. Class- definition-level macro expansion disassembles the whole term of a given definition into ESP clauses in the top-down approach and expands each pattern. Therefore, because the whole term is regarded as an expansion object at the first stage, you may use a macro function to write an application program for preprocessing that converts to an ESP program. A class- definition-level macro is defined as follows:

```
<object pattern> ==> <expanded pattern>  
[ with <a series of added predicates> ]  
[ by <instance object> ]  
[ :- <expansion condition> ]
```

The functions of these keywords are the same as those of a clause-level macro. Different from a clause-level macro, <expanded pattern> may be omitted. If omitted, no expanded pattern appears. (Addition of the other patterns and execution of the body part are performed.)

```
(Functor <- List) ==> Expanded  
:- convert(List, Functor, Expanded);  
convert([One], F, {F, One}) :- !;  
convert([One | Rest], F, ({F, One}; Tail)):-  
convert(Rest, F, Tail);
```

"<-" can be used as an operator through operator declaration expansion performed as follows:

```
author <- [kondoh, hagio, ishibasi, chikayama]  
      ↓  
author(kondoh); author(hagio);  
author(ishibashi); author(chikayama)
```



### (3) Insertion of a clause

You can define a clause insertion statement as well as a macro definition statement in the macro definition part. A clause insertion statement unconditionally inserts a specified clause into a specified part in the class definition (class, instance or local). A clause insertion statement adds an entity at a specific place, while the inheritance shares entities such as methods with the parent class. Another difference from the inheritance is that a clause insertion statement can be applied to a local predicate. A clause insertion statement is defined as follows:

```
:- inserta(Type, Clauses) ;  
:- insertz(Type, Clauses) ;
```

The statement "inserta" adds clauses at the beginning, while the statement "insertz" adds clauses at the end. As described later, a macro bank has an inheritance function, and an insertion statement is applied according to the inheritance rules of ESP. "Type" designates the type of clauses. The type may be specified by a letter "c", "i" or "l", which corresponds to the class method, instance method or local predicate respectively. "Clause" is one or more clauses (each delimited by a semicolon if more than one).

```
macro __bank m1 has  
:- inserta(l, (p(0); p(1)));  
:- insertz(l, (p(4)));  
end.
```

If you make a definition as above, expansion is performed as:

```
class c1 with __macro m1 has  
  
:p( __, X) :- p(X);  
local  
p(2);  
p(3);  
end.  
  
class c1 has  
:p( __, X) :- p(X);  
local  
P(0);  
p(1);  
p(2);  
P(3);  
p(4);  
end.
```

If the name of a macro bank as defined above is specified in front of “has” in a class definition, macro expansion in that class is performed using the specified macro bank.

```
class class-name with__macro macro-bank-name has
```

class definition

```
end.
```

```
macro __bank macro__example has
```

```
ave(X, Y) => Z when Z = (X + Y)/2 ;
```

```
end.
```

```
class ex1 with__macro macro__example has
```

```
  :p( __, X, Y, Z) :- p(ave(X, Y), Z) ; ..... ①
```

```
  :q( __, X, Y, ave(X, Y)) :- q(X, Y) ; ..... ②
```

```
  local
```

```
    p(X, Y) :- ...
```

```
    q(X, Y) :- ...
```

```
end.
```

ave(X,Y) at ① and ② in the example above is macro- expanded as follows:

```
:p( __, X, Y, Z) :- ..... ①
```

```
  add(X, Y, A),
```

```
  divide(A, 2, B),
```

```
  p(B, Z) ;
```

```
:q( __, X, Y, Z) :- ..... ②
```

```
  q(X, Y),
```

```
  add(X, Y, A),
```

```
  divide(A, 2, Z) ;
```

A macro bank as specified above is effective in the class where it has been declared. You can specify only one macro bank after `with__macro`. However, use of "natur" allows specification of multiple macro banks.

```
macro__bank macro__set has
  nature macro1, macro2, macro3 ;
end.
```

```
class example with__macro macro__set has
```

class definition

```
end.
```

Also, upon definition of a macro bank, you may specify another macro bank and use it in that macro bank. That is, a function that has been specified by the inheritance is inherited, and used when it is applied to an actual ESP program. On the other hand, a function specified after `with__macro` is used for macro expansion when the macro bank itself is stored in the system.

```
macro__bank macro-bank-name with__macro
                                expansion-macro-bank-name  has
```

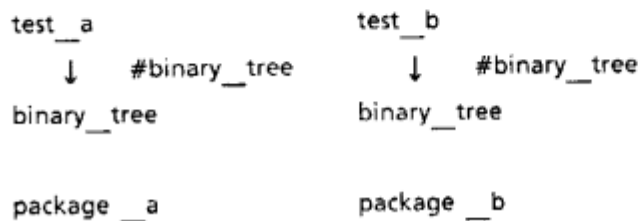
macro definition

```
end.
```

### 3.8 Multi-Class Name Space (Package)

A class name is defined by an atom. Therefore, this class name is usually used for inheritance or reference of a different class. However, when more than one programmer attempts to construct a system, it is likely that the programmers may use the same class names unintentionally. To avoid contention of class names, ESP has a mechanism called the package, which multiplies the class name space.

For example, suppose that programmers A and B are intending to make two independent classes with the same class name "binary\_\_tree". In this case, if there is only one class name space, specification of class name "binary\_\_tree" to take out the class object is processed regardless of the difference between the two classes, consequently, only either class can exist. This kind of class name contention cannot be avoided as long as several programmers use the same processing system. In ESP, as shown below, programmers A and B may generate their own packages "package\_\_a" and "package\_\_b" and generate the desired classes within the separate packages.



In this case, if "#binary\_\_tree" appears in the class "test\_\_a" generated in the package "package\_\_a", this is interpreted as the class object of class "binary\_\_tree" that was generated by programmer A. Similarly, if "nature binary\_\_tree;" appears in the class "test\_\_b" generated in package "package\_\_b", class "binary\_\_tree" that was generated by programmer B is inherited. Use of a package allows you to discriminate those classes that are stored with the same class name in the system.

#### 3.8.1 Class Specification

A package name is specified by an atom. No duplication of package names is allowed. In which package a class is to be generated is determined when the class is stored (in SIMPOS, it is determined usually when the class is catalogued.) No duplication of class names is allowed within a package.

Reference or inheritance of a class is specified in the following two ways:

- (1) class-name-atom
- (2) package-name-atom##class-name-atom

Generally, a class within the self package can be referenced or inherited in format (1), and any class in the system can be referenced or inherited in format (2). A class name may be specified in the following four places in an ESP program:

(1) Class object

```
#class-name #package-name##class-name
```

(2) Inheritance

```
nature class-name, package-name##class-name:
```

(3) Method call with a class name specified

```
Package-name##class-name:method-name(class-object,argument, ....)  
:class- name:method-name(instance-object,argument,...)
```

(4) Macro bank specification

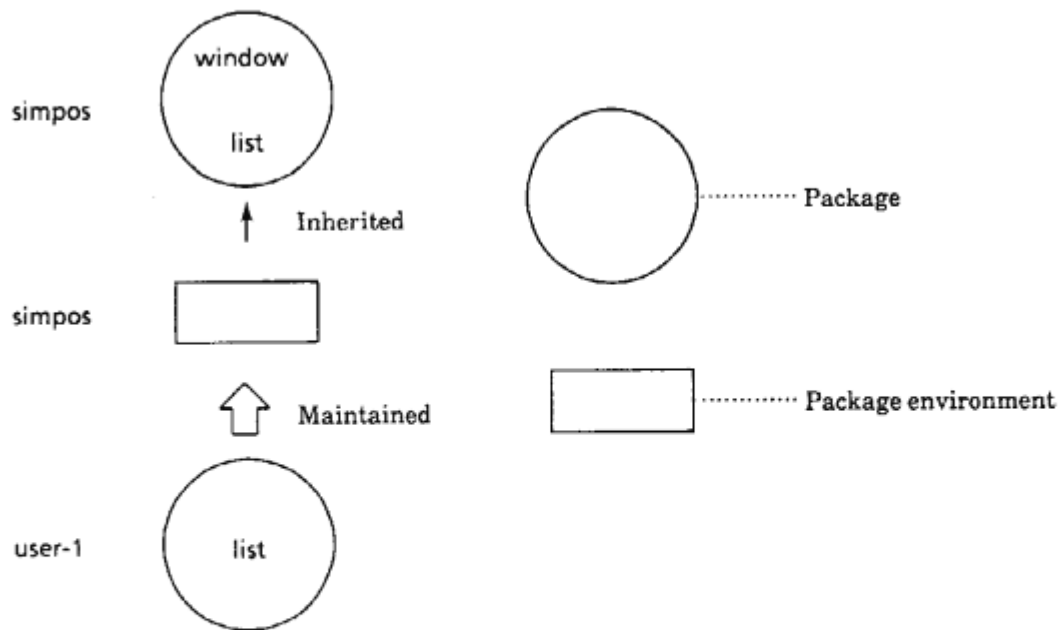
```
class class-name with__macro macro-bank-name has  
    . . . . .  
  
end.  
class class-name with__macro package-name##macro- bank-name has  
    . . . . .  
  
end.
```

Actual linkage between these specifications and the classes is performed when the classes are stored (strictly speaking, when the macros are expanded.)

### 3.8.2 Package Environment and Externally Declared Class

As described in the previous section, a class in the self package can be referenced or inherited by specifying the class name only. If you use a package environment and external declaration of a class, you can also reference or inherit a class in a different package by specifying the class name only.

A package environment specifies a package to which those classes that can be referenced or inherited by only the class name belong. Each package can have one package environment at most. A package environment name is specified by an atom. A package environment can multiple- inherit a package and a different package environment. The most general use of a package and a package environment is to use the package environment "with\_\_simpos" that inherits system package "simpos" as as shown below.



As for the classes in "user\_\_1", the classes in package "user\_\_1" and those classes that have been declared as external (public declaration) in package "simpos" can be referenced or inherited by only the class name. Now, let's think about reference and inheritance when the following program is made and registered in package "user\_\_1". Suppose that classes "window" and "list" exist in package "simpos" and a homonymous class "list" exists in package "user\_\_1".

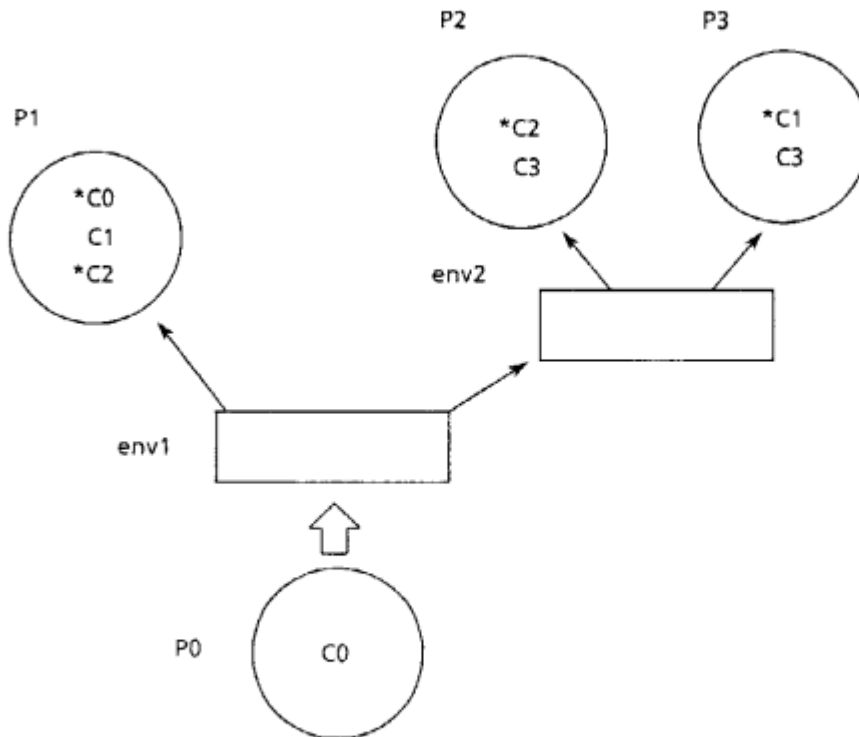
```

class ex1 has
nature list; ..... ①
:p(Class) :- :create(#window, W), ..... ②
              :create(#simpos##list, List), ..... ③
              . . . . . ;
end.

```

With "list" in ① since class "list" is defined in the self package, class "list" of "user\_\_1" is inherited. With "window" in ② since class "window" is not defined in the self package, class "window" of package "simpos" is referenced. With "list" in ③ since the package name is specified, class "list" of specified package "simpos" is referenced.

As above, the classes that can be referenced or inherited only by the class without the package name specified are: (1) every class in the self package, and (2) among the classes belonging to the packages inherited (either directly or indirectly) by the package environment of the self package, only those classes that have been declared as external. If an inherited package exists, there may be homonymous classes. In this case, a class that belongs to a package of a higher inheritance order has priority over that of a lower inheritance order. The inheritance order conforms to the inheritance rules of ESP.



As shown above, package "p0" has a package environment. The left package has a higher inheritance order than the right one. The class name preceded by an asterisk (\*) is a class that has been declared as external. In this case, if only a class name is specified in package "p0", this is interpreted as follows:

```
c0 ... class "c0" of package "p0"
c1 ... class "c1" of package "p3"
c2 ... class "c2" of package "p1"
c3 ... error since there is no applicable class
```

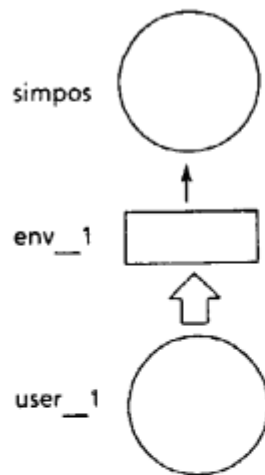
To reference a class other than the above, you must specify the package name like "p2##c2". If you specify a package name, you can also reference or inherit such a class that is not declared as external.

### 3.8.3 Use of Package in SIMPOS

Generation of a package or a package environment and external declaration of a class can be accomplished by operating the SIMPOS librarian and performing a library method call. For details, see "SIMPOS OPERATION MANUAL" and "SIMPOS PROGRAMMING MANUAL". This section describes general use of a package in SIMPOS.

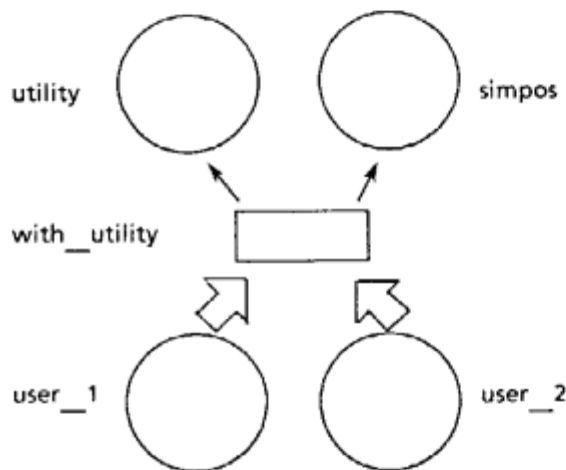
#### (1) When using a personal package

If you want to make your own package to avoid class name contention with other users and you want to reference a class of "simpos" only by the class name, make the following structure:



In this case, you can generate a class having the same name as one of “simpos” in package “user\_1”.

(2) When sharing a utility class



Suppose that packages “user\_1” and “user\_2” want to specify a common utility by the class name only. Use the following procedure:

- (1) Generate the utility class in package “utility” and declare the interface class as external.
- (2) Generate the package environment “with\_utility” that inherits packages “utility” and “simpos”.
- (3) Generate packages “user\_1” and “user\_2” and set up the package environment “with\_utility”.

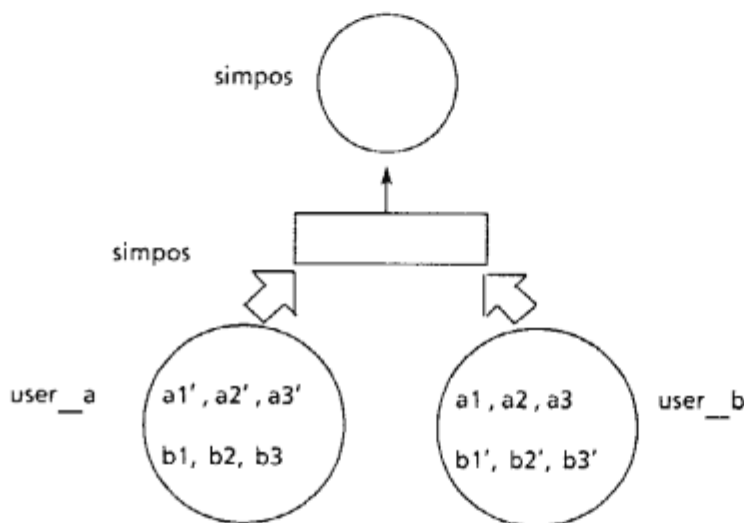


When you inherit a package as above, a change in a class in the upper package greatly affects the lower package. Therefore, the upper package should be limited to a set of classes that has been completed to some extent like a utility program. If you need to change a package quite often, it is preferable to specify a class name together with the package name (in the form of "package-name##class-name") than to inherit the package and specify a class by only the class name.

(3) When more than one user is constructing one system

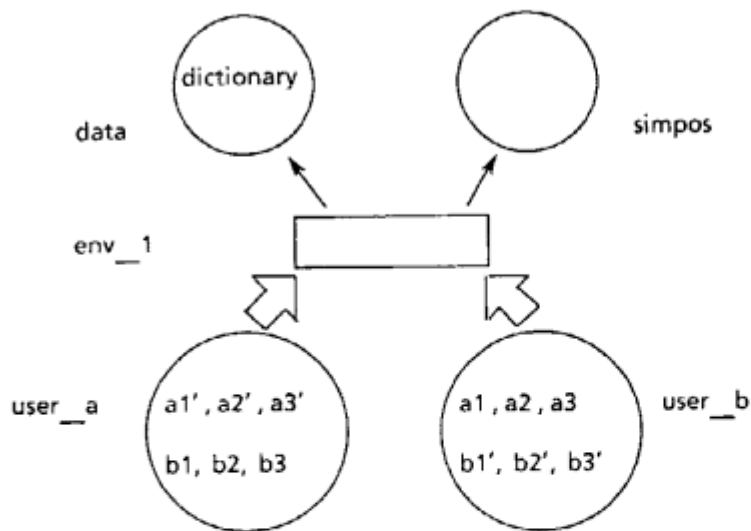
When more than one user is constructing one system, it may happen that one user releases his classes to the other users for testing, then he wants to change the classes for improvement, consequently, the other users cannot use the system for a while. To avoid this, you may generate your own package and store multiple versions in the system.

Suppose that user A is in charge of classes a1, a2 and a3, and user B is in charge of classes b1, b2 and b3.



Users A and B may temporarily release a1, a2, a3, b1, b2 and b3, then change their own classes in their own package. When user A wants to release the improved class a1 to user B, he can restore (i.e., recatalogue) it in package "user\_\_b".

As shown below, you may also have a structure where common routines are stored in an inheritance package. This may be used when sharing data, dictionary data for example.



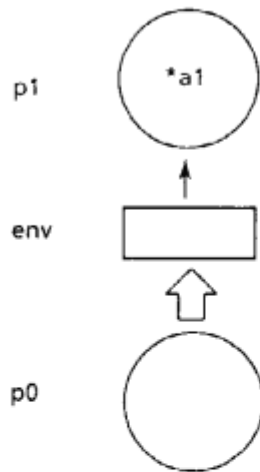
With this structure, however, a class in the upper package cannot reference a class in the lower package by specifying only the class name. Therefore, the classes you store in the upper package should be of closed reference and inheritance relationship and should be completed ones that hardly need to be changed.

### 3.8.4 Restrictions

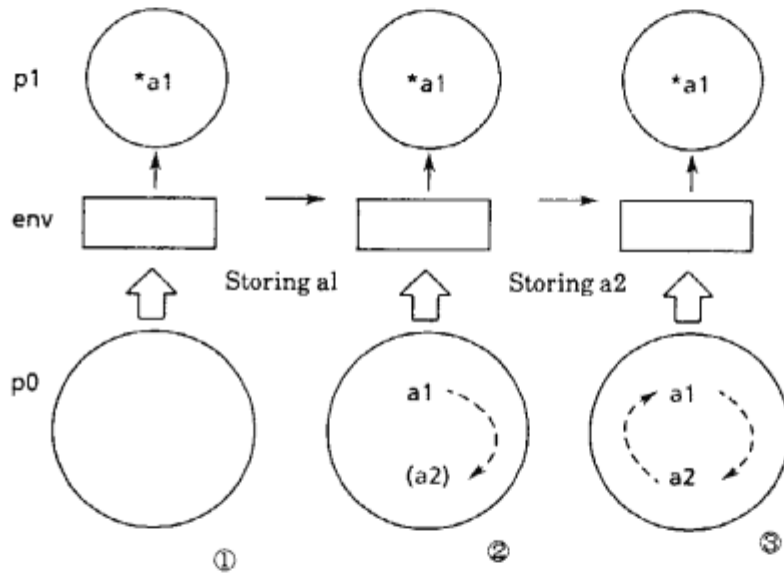
As described above, a class name written in a source program is linked to the actual class statically upon macro expansion. Therefore, the reference and inheritance relationship differs depending on the situation when macro expansion begins. Think of the following two classes.

class a1 has	class a2 has
:p(#a2) :-	:q(#a1) :-
. . . .	. . . .
end.	end.

Suppose that the package is as follows when you start storing the class:

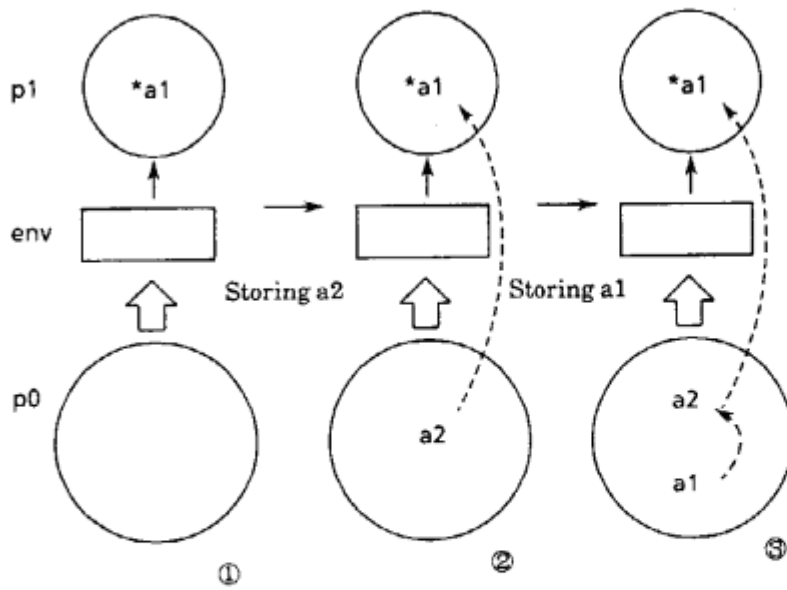


Suppose that external-declared class a1 already exists in package p1. In this case, if the above classes a1 and a2 are stored in package p0 in that order, they reference classes a2 and a1 of package p0 respectively.

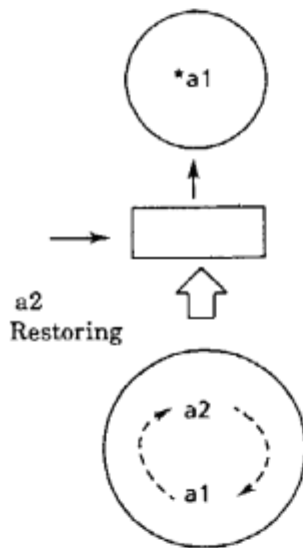


The state ② indicates that only the name of class a2 is automatically stored in package p0 supposing that class a2 will be generated later.

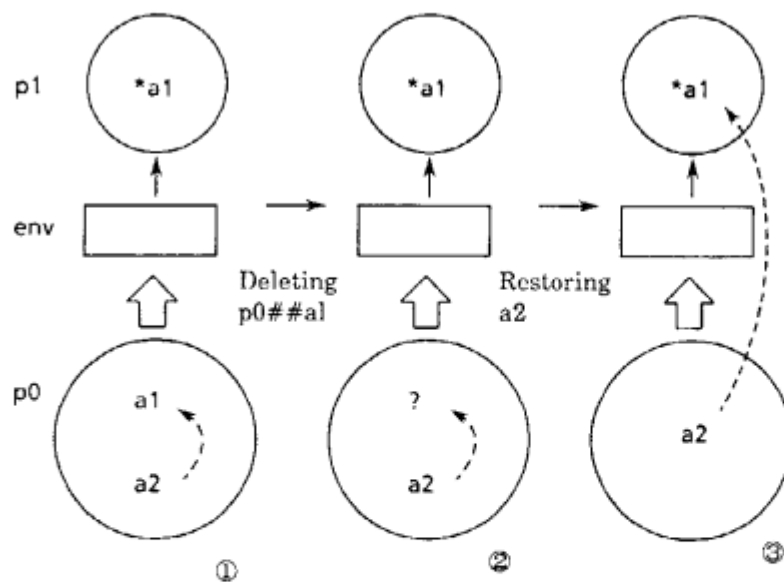
If classes a2 and a1 are stored in package p0 in that order different from the previous case, class a2 references class a1 of package p1, and class a1 references class a2 of package p0.



When class `a2` is restored later, classes `a1` and `a2` of package `p0` reference each other.



The same problem occurs when a class is deleted.



If you delete **p0##a1** in state ②, class **a2** has no referencing class, and execution of class **a2** may result in an error. When you restore class **a2** later, class **a2** references **p1##a1**.

As above, if you define in the lower package a class of the same name as one in the upper package, various kinds of problems may occur upon change, storage and deletion of a class to keep the correctness. Therefore, in principle, you had better not define a class of the same name as one in the upper package.

### 3.9 ESP Program in PSI

SIMPOS has a function to add or delete the operations when parsing a source program. You can perform user-defined operator declaration by specifying it in a source program.

To do this, SIMPOS supports the following four statements, and you can write any of them in an ESP source file. Each statement must be terminated by a period.

- Class definition
- Macro bank definition
- Operator declaration
- Macro declaration
- Package declaration
- "include" statement

#### (1) Class definition

This is a usual class definition that starts with "class" and ends with "end."

#### (2) Macro bank definition

This is a macro bank definition that starts with "macro\_\_bank" and ends with "end.". Since macro expansion is needed at compilation, a macro bank definition needs to be stored before the class to be used. (If you use the same file, you must write a macro bank definition in front.) Since a macro bank definition is treated internally as a class, it can also be saved by the librarian. Since it can be automatically loaded, once you store it in the library, you can use it without a special procedure. Because a class generated by a macro bank definition (that starts with "macro\_\_bank") inherits SIMPOS class "esp\_\_macro\_\_expander", you can debug a macro bank or dynamically use it by using the following method:

```
:expand__clause (Obj, Head0, Body0, ^Head, ^Body)
    Expands the clause (Head0:-Body0) and unifies the expanded result with (Head:-Body).

: expand__goals (Obj, Goals0, ^Goals)
    Expands the goals and unifies the expanded result with "Goals".
```

When you execute

```
:expand__clause(Obj,p(Obj,X + l6#`F0`),q(X),Head,Body)
```

you have the following results:

```
Head = (p(Obj,Y))
Body = (q(X),add(X,240,Y))
```

That is, clause `p(Obj,X + l6#`F0`):-q(X);` is converted to

```
p(Obj,Y):-q(X),add(X,240,Y);
```

However, if you write it as is directly in a program, the macro expansion function is evoked when the program is compiled. Therefore, you need to suppress macro expansion at compilation by using a double back-quote (` `). For goals that have no head part, use `expand__goals`.

### (3) Operator declaration

This is a user-defined operator declaration. An ESP source program is parsed according to the operator precedence grammar. You can modify the standard operators. The following shows the operator types:

Argument	<b>Op</b>	:	An operator (atom) or a list whose elements are operators
	<b>Type</b>	:	Operator type (fx, fy, xfx, yfx, xfy, xf or yf)
	<b>Pre</b>	:	Precedence of operator (specified by an integer number between 0 and 1200)

`add__operator(Op,Type,Pre).`

Adds an operator (or operators) specified by “Op” to the registered operator group. If the specified operator has already been registered with a different type, these operators must have the same precedence. If you specify 0 for the precedence, `remove__operator(Op,Type)` is applied.

`remove__operator(Op,Type).`

`remove__operator(Op).`

Deletes an operator (or operators) specified by “Op” from the registered operator group. If you do not specify “Type”, the operators of all types are deleted.

The standard operators of ESP are listed in appendix B. An operator declaration is effective only after the place where it is declared within that file.

### (4) Macro declaration

Usually an ESP source program is expanded by class “`esp__macro__expander`” which is a standard macro. However, it can be expanded by a user-defined macro bank. As described in section 3.7.3 “User-Defined Macros”, you can specify a macro bank for expansion of the class by using keyword “`with__macro`”. The macro declaration described below is effective only after the place where the declaration is specified within the file.

`use__macro([package-name##]class-name).`

If a package name is omitted, the default package of the process being currently executed is assumed.

### (5) Package declaration

You can specify a package to use. The specified package must be a defined package and is effective only within the source file.

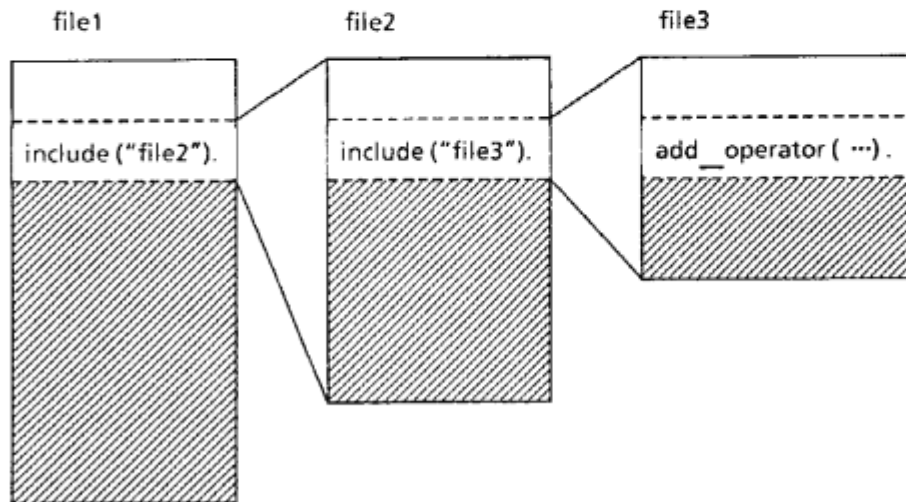
`use__package(package-name).`

(6) "include" statement

An "include" statement makes the specified file behave as if it exists there.

```
include("file-name").
```

This is useful when more than one file uses the same operator declaration. In the following example, **add\_\_operator** of file3 is applied to the shaded parts. The file to be included can contain a class definition, a macro bank definition, an operator declaration, and another "include" statement.



The following shows an example where a new operator is added.

```
add__operator(ave, yfx, 500) .
```

```
macro__bank macro1 has
```

```
  ` (Z = X ave Y) => true
    when C
      :- ex(X, Y, Z, C) ;
  X ave Y => Z
    when C
      :- ex(X, Y, Z, C) ;
```

```
local
```

```
  ex(X, Y, Z, C) :-
    C = `` (Z = (X + Y) / 2) ;
```

```
end.
```



```

class ex0 with __macro macro1 has
  :p( __, X, Y, Z) :- P(X, Y), Z = X ave Y ; ..... ①
  :q( __, X, Y, Z) :- P(X ave Y ave Z, Z) ; ..... ②
  :r( __, X, Y, X ave Y) :- p(X, Y) ; ..... ③

  local
    :P(X, Y) :-
  end.

```

In this case, ①, ② and ③ are expanded as follows:

① :p( \_\_, X, Y, Z) :- p(X, Y),  
                   add(X, Y, A),  
                   divide(A, 2, Z) ;

② Since “ave” is of the yfx type, (X ave Y ave Z) is interpreted as ((X ave Y) ave Z). Therefore, ② is expanded as follows:

```

:q( __, X, Y, Z) :- add(X, Y, A),
                  divide(A, 2, B),
                  add(B, Z, C),
                  divide(C, 2, D),
                  p(D, Z) ;

```

③ :r( \_\_, X, Y, A) :- p(X, Y),  
                   add(X, Y, B),  
                   divide(B, 2, A) ;

## CHAPTER FOUR BUILT-IN PREDICATES

This chapter describes the functions of the built-in predicates provided in ESP. The built-in predicates are explained by functions. They are categorized into the following eight functional groups:

- (1) Data manipulation
- (2) Arithmetic operation
- (3) Logical operation
- (4) Comparison operation
- (5) Data type conversion
- (6) Execution order control
- (7) System control
  - Process/processor control
  - Memory management
  - I/O operation
- (8) Built-in predicates for language processing system

The built-in predicates of groups (1) to (6) may be used by an ESP programmer. Group (7) is used only inside the operating system. Group (8) is used by the language processing system program such as the compiler and the interpreter when generating an executable object of an ESP program. This chapter describes only the main built-in predicates among groups (1) to (6) that a general ESP programmer may use. For details of the particular built-in predicates, see "KLO-BUILT-IN PREDICATE MANUAL". First the input/output mode of arguments is explained, then the built-in predicates are explained. Originally the built-in predicates have been introduced to the system for optimization and they are prepared in the system to perform a specific processing at high speed. Therefore, they do not necessarily need flexibility in the input/output mode for arguments, while flexibility may be required for a general user-defined predicate. The input/output mode of every argument of the ESP built-in predicates has been fixed: if an argument attempts to be passed in a different mode, it causes an exception at execution and results in an error. For example, if you specify atoms and integer numbers for the arguments of the built-in predicate for addition "add(X,Y,Z)", it causes an "Illegal Input" error. In the following explanations of the built-in predicates, the input mode is assumed for every argument unless otherwise specified. That is, at call time, an argument assumes to have been bound with a value. There are several modes of arguments. For example, the arguments of such a predicate that performs data generation or arithmetic operation have the output mode, the reference mode and the unifiable mode as well as the input mode. These argument modes are explained below and listed in Table 4-1.

- Input mode

An argument that must have been bound with a value before execution of the built-in predicate. In the manual, arguments of this kind are designated with no particular mark added in front.

Example: `integer(X)`

- Output mode (`~`)

An argument that the value is determined by the execution of the built-in predicate. It must be in the undefined state (not bound with a value) before execution. In the manual, arguments of this kind are designated with “`~`” mark added in front. Do not write that mark in an actual program.

Example: `new__atom(~X)`

- Reference mode (`?`)

An argument that can be in the undefined state before execution of the built-in predicate but will not be bound with any value by execution. In the manual, arguments of this kind are designated with a question mark (`?`) added in front. Do not write that mark in an actual program.

Example: `equal(?X,?Y)`

- Unifiable mode (`^`)

An argument that either can be in the undefined state or can be bound with a value before execution of the built-in predicate and that is bound with a value (or is undefined) after execution. The unification rule for an undefined variable is explained in section 4.4. In this manual, arguments of this kind are designated with “`^`” mark added in front. Do not write that mark in an actual program.

Example: `add(X,Y,^Z)`

Table 4.1 Argument Modes

Notational symbol	Mode	Condition at input	State at output
<code>X</code>	Input mode	Defined	Defined (Same as at input)
<code>~ X</code>	Output mode	Undefined	Defined
<code>? X</code>	Reference mode	Don't care	Don't care (Same as at input)
<code>^ X</code>	Unifiable mode	Don't care	Conform to the unification rule

## 4.1 Data Manipulation

The built-in predicates for data manipulation perform basic manipulation operations on various kinds of data handled in an ESP program. Data manipulations are categorized into the following four kinds:

- (1) Check the data attribute of the value of a variable.
- (2) Generate new data.
- (3) Read or write elements of structure data.
- (4) Read or write a substructure of structure data.

### (1) Data attribute checking

The built-in predicates for data attribute checking are used to check the data type of the value bound with a variable. Therefore, several built-in predicates have been provided to cope with every data type. Each built-in predicate succeeds if the type of data given to the argument is the expected one, otherwise it fails. For structural data, not only the data type but also the size and length of the structure can be checked, and also the attribute information can be read out through unification. Besides these built-in predicates for checking the type of each data, there are some built-in predicates to classify data according to whether the variable is defined or not and whether the data is of the structure type or not.

This chapter does not describe those built-in predicates that are not usually used by a programmer. See the "KLO BUILT-IN PREDICATE MANUAL" for these predicates.

#### **atom(?X)**

If X is an atom, the predicate succeeds. Otherwise it fails.

#### **integer(?X)**

If X is an integer number, the predicate succeeds. Otherwise it fails.

#### **floating\_\_point(?X)**

If X is a floating-point number, the predicate succeeds. Otherwise it fails.

#### **atomic(?X)**

If X is an atom, an integer number or a floating-point number, the predicate succeeds. Otherwise it fails.

#### **number(?X)**

If X is an integer number or a floating-point number, the predicate succeeds. Otherwise it fails.

#### **heap\_\_vector(?X, ^Length)**

If X is a heap vector, the predicate unifies the vector length with "Length". Otherwise it fails.

**stack\_\_vector(?X, ^Length)**

If X is a stack vector, the predicate unifies the vector length with "Length". Otherwise it fails.

**string(?X, ^Length, ^Type)**

If X is a string, the predicate unifies the length with "Length", and the string type (an integer 1, 8, 16 or 32) with "Type". Otherwise it fails.

**structure(?X)**

If X is structural data (heap vector, stack vector, string or location), the predicate succeeds. Otherwise it fails.

**bound(?X)**

If X is already bound with a value, the predicate succeeds. If X is still undefined, it fails.

**unbound(?X)**

If X is not bound with a value yet (that is, if X is still undefined), the predicate succeeds. If X is already bound with a value, it fails.

## (2) Data generation

The built-in predicates for data generation are used to generate a new atom or structural data.

**new\_\_atom(~Atom)**

Generates a new atom and unifies it with "Atom".

**new\_\_heap\_\_vector(~H\_\_V, Length)**

Generates a heap vector of the length specified by "Length" and unifies it with "H\_\_V". The area of the heap vector generated in the main memory is not released by backtracking. However, the value of argument H\_\_V is released.

**new\_\_stack\_\_vector(~S\_\_V, Length)**

Generates a stack vector of the length specified by "Length" and unifies it with "S\_\_V". The area of the stack vector generated in the main memory is released by backtracking. The value of argument S\_\_V is also released.

**new\_\_string(~S, Length, Size)**

Generates a string having the length specified by "Length" and the bit length of each element specified by "Size" (integer 1, 8, 16 or 32), and unifies it with "S". The area of the string generated in the main memory is not released by backtracking. However, the value of argument S is released.

### (3) Access to structure data element

The built-in predicates for access to a structure data element are used to read or write an element or structure data. The built-in predicates for the reading read out the contents of a structure element through unification. That is, if you give the output argument with a variable, you can read the element. If you give it with a constant, you can compare the content of the element with the constant.

The built-in predicates for writing can be performed on structure data except for a stack vector. Writing is carried out by overriding a value over the element. You can perform overriding to any data other than an unbound variable and a stack vector.

**vector\_\_element(V,Position,^Element)**

Unifies the element at the position specified by "Position" of vector "V" (stack vector or **heap\_\_vector**) with "Element". For the vector position, specify the first element as the 0th element and so on.

**first(V,^Element)**

Unifies the first element (i.e., the 0th element) of vector "V" (stack vector or heap vector) with "Element".

**second(V,^Element)**

Unifies the second element (i.e., the 1st element) of vector "V" (stack vector or heap vector) with "Element".

**string\_\_element(S,Position,^Element)**

Unifies the element at the position specified by "Position" of string "S" with "Element". For the string position, specify the first element as the 0th element and so on.

**set\_\_vector\_\_element(H\_\_V,Position,Element)**

Overrides the data specified by "Element" to the position specified by "Position" of heap vector "H\_\_V". The overridden result is not retracted to the original value by backtracking.

**set\_\_first(H\_\_V,Element)**

Overrides the data specified by "Element" to the first element (i.e., the 0th element) of heap vector "H\_\_V". The overridden result is not retracted to the original value by backtracking.

**set\_\_second(H\_\_V,Element)**

Overrides the data specified by "Element" to the second element (i.e., the 1st element) of heap vector "H\_\_V". The overridden result is not retracted to the original value by backtracking.

**set\_\_string\_\_element(S,Position,Element)**

Overrides the data specified by "Element" to the position specified by "Position" of string "S". The overridden result is not retracted to the original value by backtracking.

#### (4) Access to substructure

The built-in predicates for access to a substructure are used to manipulate a part of structure data. A substructure means a structure that consists of part of the original structure which can be specified by the starting element position and the length (number of elements). Figure 4-1 shows the relationship between a structure (heap vector) and a substructure (partial heap vector).

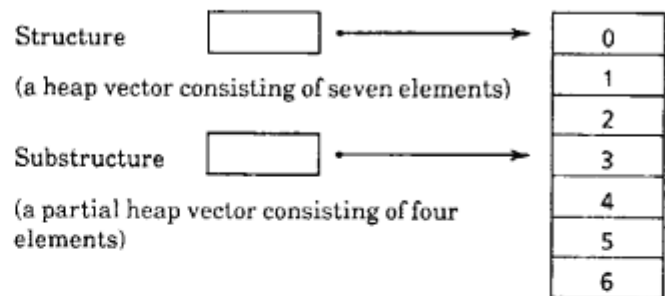


Fig. 4-1 Structure and Substructure

**subvector(V,Position,Length,^SubV)**

Unifies the partial vector that starts from the position specified by "Position" of vector "V" (heap vector or stack vector) and has the number of elements specified by "Length", with "SubV".

**vector\_\_tail(V,Position,^SubV)**

Unifies the partial vector that starts from the position specified by "Position" of vector "V" (heap vector or stack vector) and ends at the end of that vector, with "SubV".

**substring(S,Position,Length,^SubS)**

Unifies the partial string that starts from the position specified by "Position" of string "S" and has the number of elements specified by "Length", with "SubS".

**string\_\_tail(S,Position,^SubS)**

Unifies the partial string that starts from the position specified by "Position" of string "S" and ends at the end of that string, with "SubS".

**set\_\_subvector(H\_\_V,Position,Length,SubV)**

Overrides the vector specified by "SubV" to the partial vector that starts from the position specified by "Position" of heap vector "H\_\_V" and has the number of elements specified by "Length".

**set\_\_substring (S, Position, Length, SubS)**

Overrides the string specified by "SubS" to the partial string that starts from the position specified by "Position" of string "S" and has the number of elements specified by "Length".

**move\_\_string\_\_elements (S, Position, Length, Shiftcount)**

Shifts the partial string that starts from the position specified by "Position" of string "S" and has the number of elements specified by "Length", to the left or right by the number of elements specified by "Shiftcount". It is shifted to the right if the "Shiftcount" value is positive, and to the left if the value is negative.





<code>divide(N1,N2,^R)</code>	Macro expression:	<code>^R is N1/N2</code>
-------------------------------	-------------------	--------------------------

Divides number "N1" by number "N2", and unifies the result with "R". The operation differs depending on the data type of "N1" and "N2".

If "N1" and "N2" are both integers, fixed-point division is performed. Division is performed in the manner that a quotient is given with the remainder of the same sign as the dividend. The remainder is ignored. The quotient is unified with "R" as an integer number.

If "N1" and "N2" are both floating-point, floating-point division is performed. The quotient is unified with "R" as a floating-point number.

`divide__with__remainder(I1,I2,^Q,^Rem)`

Divides integer "I1" by integer "I2", and unifies the quotient with "Q", and the remainder with "Rem".

<code>increment(I,^R)</code>	Macro expression:	<code>^R is R + I</code>
------------------------------	-------------------	--------------------------

Adds I to integer "I", and unifies the result with "R".

<code>decrement(I,^R)</code>	Macro expression:	<code>^R is R-I</code>
------------------------------	-------------------	------------------------

Subtracts I from integer "I", and unifies the result with "R".

<code>minus(N,^R)</code>	Macro expression:	<code>^R is -N</code>
--------------------------	-------------------	-----------------------

Reverses the sign of the value of N, and unifies the result with "R".

## (2) Double-length word integer operations

The built-in predicates for the double-length word integer operations are used to perform basic arithmetic operations on double-length word integer data. In ESP, a double-length word integer number is expressed in a double length form of 64 bits by using two words of 32-bit unsigned integer. Therefore, to perform a double-length word integer operation, you must operate an integer expressing the upper 32 bits and an integer expressing the lower 32 bits, as sign judgment is made on your own responsibility.

`add__extended(X,Y,^R__up,^R__low)`

Adds integers "X" and "Y" regarding them as 32-bit unsigned integers, and unifies the lower 32 bits of the result with "`R__low`", and the carry from the 32nd bit with "`R__up`". Note that an integer number dealt with by this built-in predicate is not a binary complement.

**subtract\_\_extended(X,Y,~S,~R)**

Subtracts integer “Y” from integer “X” regarding them as 32-bit unsigned integers, and unifies the absolute value of the result with “R”. Also, the sign of the result is unified with “S”: integer 0 is unified if  $X \geq Y$ ; integer 1 is unified if  $X < Y$ . Note that an integer number dealt with by this built-in predicate is not a binary complement.

**multiply\_\_extended(X,Y,~R\_\_up,~R\_\_low)**

Multiplies integers “X” and “Y” regarding them as 32-bit unsigned integers, and unifies the lower 32 bits of the product with “R\_\_low”, and the upper 32 bits with “R\_\_up”. Note that an integer number dealt with by this built-in predicate is not a binary complement.

**divide\_\_extended(X\_\_up,X\_\_low,Y,~Q,~R)**

Divides an unsigned 64-bit integer number (whose upper 32 bits are specified by X\_\_up and lower 32 bits are specified by X\_\_low) by 32-bit unsigned integer Y, and unifies the quotient with Q, and the remainder with R both as a 32-bit unsigned integer. Note that an integer number dealt with by this built-in predicate is not a binary complement.

### 4.3 Logical Operations

The built-in predicates for logical operations are used to perform a logical operation on integer numbers or strings. These operations are grouped into the following two kinds:

- (1) Logical operations for integer data
- (2) Logical operations for strings

#### (1) Logical operations for integer data

For logical operations for integer data, six built-in predicates have been provided: logical multiply (AND), logical add (OR), exclusive OR (XOR), one's complement (COMPLEMENT), left shift (SHIFT-LEFT), and right shift (SHIFT-RIGHT). These built-in predicates may be expressed by using a macro expression in an ESP program.

**and(I1,I2,^R)**

Macro expression:

**^R is I1 /\ I2**

Performs a logical add (OR) operation on integers I1 and I2 on a bit basis, and unifies the result with R.

**xor(I1,I2,^R)**

Performs an exclusive OR (XOR) operation on integers I1 and I2 on a bit basis, and unifies the result with R.

**complement(I,^R)**

Macro expression:

**^R is \ (I)**

Obtains a one's complement of the value of integer I on a bit basis, and unifies the result with R.

**shift\_\_left(I,Count,^R)**

Macro expression:

**^R is I << Count**

Shifts the value of integer I to the left on a bit basis by the number of bits specified by Count, and unifies the result with R. The bit positions at the right side where the original bits have been shifted are padded by zeros.

**shift\_\_right(I,Count,^R)**

Macro expression:

**^R is I >> Count**

Shifts the value of integer I to the right on a bit basis by the number of bits specified by Count, and unifies the result with R. The bit positions at the left side where the original bits have been shifted are padded by zeros.

## (2) Logical operations for strings

For logical operations for strings, four built-in predicates have been provided: logical multiply (AND-STRING), logical add (OR-string), exclusive OR (XOR-STRING), and one's complement (COMPLEMENT-STRING). These logical operations are performed to the part of a string whose starting position and length are specified by arguments, with a masking string specified by an argument. The operation result is overridden at the position of the partial string in the original string. If the masking string is shorter than the partial string, the operation is performed with as many zeros as necessary added at the end of the masking string. If the masking string is longer than the partial string, the operation is performed with the surplus part of the masking string cut off.

### **and\_\_string(S,Position,Length,MaskS)**

Performs an AND operation between the partial string that starts from the position specified by **Position** (on the zero origin basis) in string **S** and has the number of elements specified by **length**, and masking string **MaskS**. The result is overridden to the partial string in string **S**. The overridden value is not retracted to the original value by the backtracking. If **MaskS** is shorter than **Length**, the operation is performed with as many zeros as necessary added at the end of **MaskS**. If **MaskS** is longer than **Length**, the operation is performed with the surplus part cut off.

### **or\_\_string(S,Position,Length,MaskS)**

Performs an OR operation between the partial string that starts from the position specified by **Position** (on the zero origin basis) in string **S** and has the number of elements specified by **Length**, and masking string **MaskS**. The result is overridden to the partial string in string **S**. The overridden value is not retracted to the original value by backtracking. If **MaskS** is shorter than **Length**, the operation is performed with as many zeros as necessary added at the end of **MaskS**. If **MaskS** is longer than **Length**, the operation is performed with the surplus part cut off.

### **xor\_\_string(S,Position,Length,MaskS)**

Performs an XOR operation between the partial string that starts from the position specified by **Position** (on the zero origin basis) in string **S** and has the number of elements specified by **Length**, and masking string **MaskS**. The result is overridden to the partial string in string **S**. The overridden value is not retracted to the original value by backtracking. If **MaskS** is shorter than **Length**, the operation is performed with as many zeros as necessary added at the end of **MaskS**. If **MaskS** is longer than **Length**, the operation is performed with the surplus part cut off.

### **complement\_\_string(S,Position,Length)**

Obtains the complement of the partial string that starts from the position specified by **Position** in string **S** and has the number of elements specified by **Length**. The result is overridden to the partial string. The overridden value is not retracted to the original value by backtracking.

#### 4.4 Comparison operations

The built-in predicates for comparison operations are used to compare data in terms of equality or the less-greater relation or to search for a string element. Comparison operations are grouped into the following four kinds:

- (1) Unification
- (2) Equality and identity comparison
- (3) Less-greater comparison
- (4) String element search

##### (1) Unification

Unification is usually performed at a predicate call between an argument of the calling side and an argument of the called side. Built-in predicate "unify(X,Y)" performs completely the same function as ordinary unification between two arguments X and Y. The same unification rules as those used at a user-defined predicate call apply as shown in Table 4-2. You can use this built-in predicate to check the value of a variable or to substitute a value with a variable.

`unify(`X,`Y)`

Macro expression:

``X = `Y`

Unifies X with Y. The following unification rules apply:

- (i) If both arguments are undefined  
Unification always succeeds. Both arguments become logically identical.
- (ii) If one argument is undefined and the other is defined  
Unification always succeeds. The undefined argument (the undefined variable) has the value of the other argument and becomes the defined variable.
- (iii) If both arguments are defined  
The data type and the value of both arguments are compared. If the arguments are identical, unification succeeds. In other words,
  - a. Unification fails if they are of different data types.
  - b. Unification succeeds if they are of the same data type and are equal to each other. (See Table 4-3 for the equality conditions by data types.)

Table 4-2 Unification Rules

<div> Data Type </div> <div> Data Type </div>	Undefined	Atomic Atom Integer number Floating-point number	Structure with side-effect Heap vector String	Stack vector
Undefined	○	○	○	○
Atomic Atom Integer number Floating-point number	○	Equal : ○ Not equal : ×	×	×
Structure with side-effect Heap vector String	○	×	Equal : ○ Not equal : ×	×
Stack vector	○	×	×	Equal : ○ Not equal : ×

○ : Unification succeeds.

× : Unification fails.

Equal : Both data type and value are identical.

## (2) Equality and identity comparison

Six built-in predicates have been provided to check the equality or the identity between two data given as the arguments. Equality means that the two data represent the same logical value. Identity means that the two data are identical to each other in terms of memory image. Some built-in predicates are provided specially for strings to check whether or not two unequal strings are logically identical to each other.

Table 4-3 shows the conditions by the data types in which equality or identity holds.

Table 4 – 3 Conditions for Equality and Identity

Data Type		Equality	Identity
Atomic · Atom · Integer number · Floating-point number		Data type and value are the same.	
Structure with side-effect	Heap vector	· The logical address of the first element and the number of elements are the same.	
	String	· The logical address of the first element and its position in the word, the element length and the number of elements are the same.	
Stack vector		· The elements are the same and each corresponding element is equal.	· The logical address of the first element and the number of elements are the same.
Undefined		The logical address of the undefined variable cell is the same.	

`equal(?X,?Y)`

Macro expression:

`?X == ?Y`

Checks whether **X** is logically equal to **Y**. No unification is performed when this predicate is executed: the values of the arguments remain unchanged.

`not_equal(?X,?Y)`

Macro expression:

`?X \= ?Y`

checks whether **X** is logically unequal to **Y**. No unification is performed when this predicate is executed: the values of the arguments remain unchanged.

`equal__string(S1,S2)`

Checks whether the contents of strings **S1** and **S2** are the same. Equality holds if the type and length of the two strings are the same and the corresponding elements of the strings are the same (including the case the lengths of the two strings are both zero).

`not_equal__string(S1,S2)`

Check whether the contents of strings **S1** and **S2** are the same. Equality holds if the type and length of the two strings are the same and the corresponding elements of the strings are the same (including the case the lengths of the two strings are both zero).

`identical(?X,?Y)`

Macro expression:

`?X == ?Y`

Checks whether **X** is identical to **Y**. No unification is performed when this predicate is executed: the values of the arguments remain unchanged.

`not__identical(?X,?Y)`

Macro expression:

`?X == ?Y`

Checks whether **X** is identical to **Y**. No unification is performed when this predicate is executed: the values of the arguments remain unchanged.

### (3) Less-greater comparison

Two built-in predicates have been provided to perform a less-greater comparison between two data. This comparison can be applied not only to numerical values but to all data types including undefined variables. For the less- greater comparison methods, see Table 4-4. A less-greater comparison between different data types is performed with respect to the tag that indicates a data type. Since the less-greater relationship between different data types does not have any general meaning, you should try not to use this comparison.

`less__than(?X,?Y)`

Macro expression:

`?X < ?Y`

`?Y > ?X`

If **X** is less than **Y**, the predicate succeeds. Execution of the predicate causes no change in the values of the arguments.

`not__less__than(?X,?Y)`

Macro expression:

`?X >= ?Y`

`?Y == < ?X`

If **X** is not less than **Y**, the predicate succeeds. Execution of the predicate causes no change in the values of the arguments.



Table 4-4 Less-Greater Comparison Methods by Data Types

Data Type	Less-greater comparison method
Atom	Atom numbers are compared regarding them as a 32-bit positive integer
Integer number	The numerical values are compared.
Floating-point number	The numerical values are compared.
Heap vector	Comparison is made using the following procedure: (1) Comparison is performed with respect to the number of elements (positive integer). (2) If the number of elements is the same, then comparison is performed with respect to the logical address of the first element.
String	Comparison is made using the following procedure: (1) Comparison is performed with respect to the element size 1-bit < 8-bit < 16-bit < 32-bit (2) If the element sizes are the same, then the contents of both strings are compared starting from the first element (treated as a positive integer). (3) During the comparison of (2), if the comparison has reached the end of one string, the string having more elements is judged as being greater than the other string.
Stack vector	Comparison is made using the following procedure: (1) A comparison is performed with respect to the number of elements. (2) If the number of elements is the same, then both stack vectors are compared starting from the first element. The definition described in this section is applied to the comparison of the elements.
Undefined variable	Comparison is performed with respect to the logical address (32-bit integer) of the variable cell.

#### (4) String element search

Four built-in predicates have been provided to search for a string element. All these built-in predicates except for "search\_string\_difference" are only for 16-bit element strings.

**search\_\_character(S,Start,End,Charactercode,^Position)**

Searches for the same character as specified by "Charactercode" within the area whose starting and ending positions are specified respectively by "Start" and "End" in string "S". If the same character is found, the element position of the first occurrence is unified with "Position". The searching direction is determined by the relationship between "Start" and "End": forward direction if "Start" ≤ "End"; backward direction if "Start" > "End".

`search__character__backward(S,Start,Table1,Table2,^Position)`

First (1) read out one element (16-bit element) at the position specified by "Start" in string "S". Then (2) regarding the value (0 to 2 16-1) of that element as an element number, read out the element corresponding to that element number from the string in "Table1" (8-bit element). Again (3) regarding the value (0 to 2 8-1) of that 8-bit element as an element number, read out the element corresponding to that element number from the string in "Table2" (1-bit elements). If that 1-bit element has value "1", then unify the position of the element currently concerned in string "S" with "Position". If not "1", repeat above operations (1) to (3) for the next younger element of string "S". This set of operations may be repeated up to the 0th element of string "S".

`search__character__forward(S,Start,Table1,Table2,^Position)`

First (1) read out one element (16-bit element) at the position specified by "Start" in string "S". Then (2) regarding the value (0 to 2 16-1) of that element as an element number, read out the element corresponding to that element number from the string in "Table1" (8-bit element). Again (3) regarding the value (0 to 2 8-1) of that 8-bit element as an element number, read out the element corresponding to that element number from the string in "Table2" (1-bit elements). If that 1-bit element has value "1", then unify the position of the element currently concerned in string "S" with "Position". If not "1", repeat above operations (1) to (3) for the next younger element of string "S". This set of operations may be repeated down to the last element of string "S".

`search__string__difference(S1,S2,^Position)`

Strings "S1" and "S2" are compared starting from their first elements. If the corresponding elements are different, the element number of the different element at the first occurrence is unified with "Position".

#### 4.5 Data Type Conversion

The built-in predicates for data type conversion are used to change the data type from the integer type to the floating-point type, or vice versa. The built-in predicates for arithmetic operations can be applied to either the integer type or the floating-point type, but not to both of them mixed. Therefore, you must convert the data types to either of them before the operation.

`integer__to__floating__point(I,^F)`

Converts integer number I to a floating-point number, and unifies it with F. If integer number I is a hexadecimal number of more than 6 digits, the upper 6 digits become the mantissa of the floating-point number, and the lower two digits are rounded off. The converted floating-point number is normalized.

`floating__point__to__integer(F,^I)`

Converts floating-point number F to an integer number, and unifies it with I.

## 4.6 Execution Order Control

The built-in predicates for execution order control are used to control program execution by forcing a clause to succeed or fail or by cutting the alternatives of a clause.

### **true**

This predicate always succeeds.

### **fail**

This predicate always fails. That is, when "fail" is executed, backtracking occurs and the most recent alternative is executed.

### **!**

When this predicate is executed, the alternative of the clause to which ! belongs is cut. If ! is used in an OR clause, the alternative of the OR clause is cut. Note that this predicate handles the alternative of an OR clause differently from ! of Prolog.

#### Example 1

```
P: - Q, !, R; ..... ①  
    P: - S; ..... ②
```

When ! in clause ① is executed, the alternative of P, namely clause ②, is cut.

#### Example 2

```
P: - (Q, !, R; S); ..... ③  
    └──┬──┘  
      ①  ②  
  
P: - T; ..... ④
```

When ! in goal ① is executed, the alternative of this OR clause, namely goal ②, is cut. The alternative of clause ③, namely clause ④, is not cut. In Prolog, ! cuts also clause ④.

### **cut\_and\_fail**

The predicates fails after ! is executed.

### **relative\_cut(Level)**

When this predicate is executed, all alternatives below the level specified by argument "Level" (the depth of levels backward from the current level) are cut. You may use this predicate to realize the cut in OR in a clause of Prolog. Note that this predicate is effective only for a compiled program; if the interpretive code is executed, it functions the same as !.

### Example 1

P: - (Q, relative\_\_cut(1), R; S); ..... ①  
P: - T; ..... ②

When relative\_\_cut(1) in clause ① is executed, the alternative of the clause in this OR, namely goal S, is cut and also the alternative ② of P is cut.

### Example 2

P: - (Q, !, (R, relative\_\_cut(2), S; T)); ..... ①  
P: - U; ..... ②

When relative\_\_cut(2) in clause ① is executed, the alternative of the clause in this OR, namely goal T, is cut and also the alternative ② of P is cut.

### relative\_\_cut\_\_and\_\_fail(Level)

This predicate fails after executing relative\_\_cut(Level). This predicate is effective only for a compiled program.

### absolute\_\_cut(Level)

All alternatives below the level specified by "Level" (including this Level) are cut. This predicate is effective only for a compiled program.

### absolute\_\_cut\_\_and\_\_fail(Level)

This predicate fails after executing absolute\_\_cut(Level). This predicate is effective only for a compiled program.

### level(Level)

Unifies the depth of the current predicate call with Level as integer. This predicate is effective only for a compiled program.

### succeed(Level)

Assumes that all those clauses that have the level value specified by "Level" have succeeded and cuts all of their alternatives. This predicate is effective only for a compiled program.

## CHAPTER FIVE PROGRAMMING TECHNIQUE

This chapter describes some of the programming techniques that help you write an ESP program. Stress is put upon how to write in ESP a procedural program that may otherwise be written in a conventional language, because you often have to write procedural expressions when you write a large program in ESP.

Techniques for efficient memory use and for high speed execution are also explained.

The explanations are made with many program examples. However, they are mainly written in local predicates for ease of understanding. In some examples, goals of ESP are designated by uppercase letters P and Q, simplifying the pair of predicate name and arguments. For instance, [Example 1] may be described as [Example 2].

```
[Example 1]  reverse(X, Y) :- rev(X, [], Y) ;
              rev([], L, L) :- ! ;
              rev([_:_], L, R) :- rev(T, [HIL], R) ;
```

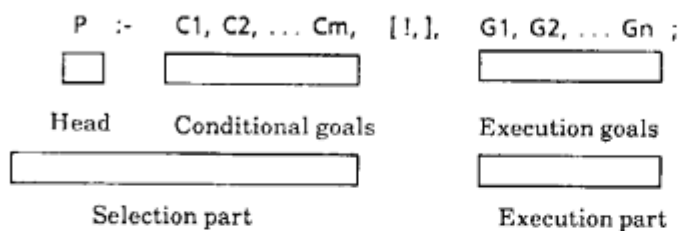
```
[Example 2]  P :- Q ;
              Q1 :-! ;
              Q2 :-Q ;
```

### 5.1 Writing a clause

Most of an ESP program is clause descriptions. This section explains the standard clause form at and some techniques to write an efficient program.

#### (1) Standard clause format

A clause has generally the following standard format:



A clause consists of a selection part and an execution part, and the selection part consists of a head and conditional goals (a conditional goal may be null.) In the selection part, you write conditions for judging whether or not to execute the execution part. If the selection part succeeds, that is, if the unification of the head succeeds or if each goal of the conditional goals succeeds, then control goes to the execution of the goals in the execution part. If the selection can be deterministic, that is, if predicate P has the alternative and if the alternative can be cut, insert ! (cut) after the selection part.

You may write a clause in a format other than the standard format. However, the standard format is, so to speak, a formula. Writing clauses in the standard format makes the program efficient and readable.

A clause that contains two or more cuts can be easily realized by a combination of standard clauses as follows:

```
P :- C1, C2, !, D1, D2, !, E1, E2 ;
      ↓
P :- C1, C2, !, Q ;
Q :- D1, D2, !, E1, E2 ;
```

If you want to use OR (:) in a clause, you may have to use more than one cut in one clause. This format may be regarded as an abbreviated form of a combination of standard formats and may rather often result in a readable program.

```
P :- Q, C3, !, G4 ;
Q :- C1, !, G1 ;
Q :- C2, !, G2 ;
Q :- G3 ;
      ↓
P :- (C1, !, G1 ; C2, !, G2 ; G3), C3, !, G4 ;
```

However, when you want to use OR in a clause, take the following points into consideration:

- If you need to nest several ORs, use of OR in a clause makes the program very unreadable. In this case, write it in separate clauses.

Example:

```
P :- (C1, !, (D1, !, (E1, ! ; E2) ; D2) ; C2), C3 ;
      ↓
P :- (C1, !, Q ; C2), C3 ;
Q :- (D1, !, R ; D2) ;
R :- (E1, ! ; E2) ;
```

- If you want to select one type of processing from among several types of processing according to the value of the variable passed through unification of the head, use head unification for condition judgment instead of using OR in a clause. This improves the processing speed because a compiler technique called clause indexing (explained later) is applied.

Example: Make a predicate p(X) that executes processing a, b or c if the value passed to the argument is 0, 1 or 2 respectively.

[ When OR is used in a clause ]

```
p(X) :- (X = 0, !, a;
        X = 1, !, b;
        x = 2, !, c) ;
```

[ When head unification is used. ]

```
p(0) :- !, a;
p(1) :- !, b;
p(2) :- c;
```

Solution 2 is better in both program readability and execution speed.

## (2) Determinate termination of predicate

The following explains a technique to determinately terminate a predicate, which is very important on writing an efficient program.

If a predicate is terminated without the alternative remaining, the predicate is said to be determinately terminated. One of the ESP features is the nondeterminacy function by backtracking. In fact, many parts of a large program do not need to use this function. It is not better in program execution speed and the efficient memory use that a predicate terminates with the alternative remaining if the predicate could be determinately terminated. To write an efficient program, you should take into consideration the states when predicates terminate.

In other words, determinate termination of a predicate means when the execution of a clause of a predicate has completed, no alternatives remain for every goal in the body part of the clause and for the predicate. It occurs when either of the following two conditions i) and ii) holds:

- when the last clause of a predicate has been executed.

```
P1 :- . . . ;
P2 :- . . . ;
P3 :- . . . ;
P4 :- . . . ; ← When this clause has been executed.
Q  :- . . . ;
```

At this time, the goals called in the body part of clause P4 must not have any alternatives.

- ii) When a clause other than the last clause has been executed, if the subsequent clauses do not remain as the alternatives due to the cut.

```

P1 :- . . . ;
P2 :- . . . , !, . . . ; ← When this clause has been executed.
P3 :- . . . ;
P4 :- . . . ;
Q  :- . . . ;

```

At this time, the goals after ! in P2 must not have any alternatives.

Determinate termination of a predicate has the following advantages:

- Execution speed is improved.

If unnecessary alternatives remain, it takes unnecessary time to search for the alternatives, reducing the execution speed.

- Memory capacity can be saved.

If unnecessary alternatives remain, the system needs to maintain the information about them in memory, reducing greatly memory use efficiency.

- Debugging can be easily done.

If unnecessary alternatives remain, an unexpected alternative might be executed during program debugging, making debugging difficult.

As described above, it is important to write predicates in the determinate manner for the program part that does not need nondeterminate processing. The following shows an example of a determinate predicate expression.

Example: Make a predicate loop(N,X) that repeats do(X) specified N number of times.

[ Bad example ]

```

loop(0, _) ; ..... (a)
loop(N, X) :- do(X), ..... (b)
               loop(N-1, X) ;

```

In this program, it is not specified that clause (b) is not executed if the value of N is 0. Therefore, clause (b) remains as the alternative. Consequently, even when do(X) has been executed N times, the program does not stop, falling in an infinit loop.



[ Good example ]

```
loop(0, _) :- ! ;          ..... (c)
loop(N, X) :- do(X), ..... (d)
                loop(N-1, X) ;
```

With this program, when (c) is executed, (d) is not the alternative any longer due to the cut (!), and the program stops after executing do(X) N times.

In ESP, writing a predicate in the nondeterminate manner allows that predicate to be used bidirectionally. However, this has a disadvantage as regards efficiency. Because of this, it would be better if you prepare both a determinately described predicate and nondeterminately described predicate and use either of them case by case.

Example: Make a predicate "member(X,List)" that judges whether X is an element of the list List and also unify the elements of the list with X one by one by backtracking.

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

This program is adequate for reading the list elements one by one by backtracking. For use for judgment, since unnecessary alternatives remain, the program is not easy to use and is inefficient. Therefore, it would be better to prepare two separate programs: for element reading and for element judgment.

[For element reading]

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

[For element judgment]

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

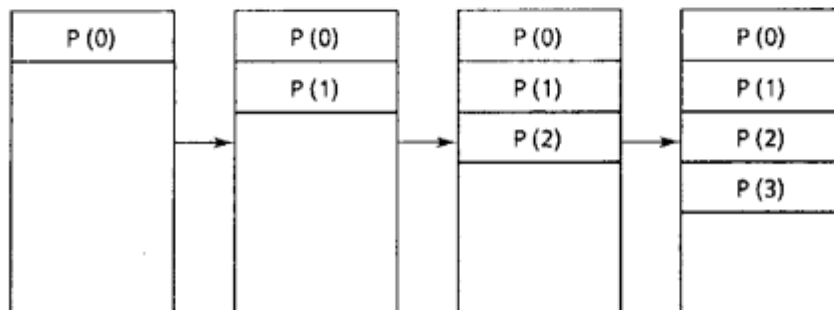
### (3) Recursive call

In ESP, a predicate can call itself from the body part of a clause in that predicate. This is called a recursive call. If the goal that calls its own predicate is the last goal of the body part, it is specially called a tail recursion. Upon tail recursion, only if the predicate is determinately terminated, memory use optimization is accomplished by firmware. This optimization method is called TRO (Tail Recursion Optimization), which is very efficient in terms of memory capacity and speed.

Example 1: When TRO is not applied.

P ; ..... (a)  
P :- R, S, P ; ..... (b)

In clause (b), P is called tail recursion. In this case, if R or S has the alternative, TRO is not applied. The following simplified figures show the states of the stack in memory when P is executed.

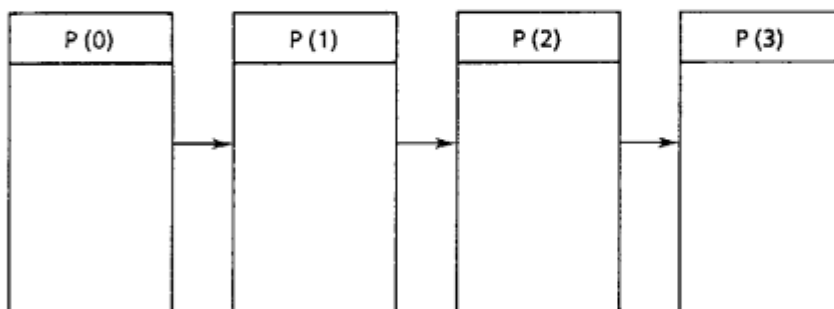


P(N) designates that predicate P is called recursively at the N'th time.

Example 2: When TRO is applied.

P ; ..... (a)  
P :- R, S, !, P ; ..... (b)

In clause (b), P is called as tail recursion. In this case, because the alternatives are cut, TRO is applied and the stack does not grow. The following simplified figures show the states of the stack in memory when P is executed.



## 5.2 Controlling Program Execution

ESP does not have execution control commands such as **GO TO**, **CALL**, **IF THEN ELSE**, **DO**, and **WHILE**. However, these execution control functions can be realized by combining some ESP functions such as backtrack, cut and recursive call as follows:

Execution control function	Realization in ESP
Unconditional branch (GO TO)	none
Subroutine call (CALL)	Equivalent to goal call.
Conditional branch (IF)	Can be realized by using backtrack and cut.
Multi-way branch (CASE)	Can be realized by using backtrack, cut and recursive call.

In ESP, you cannot write a statement equivalent to a **GO TO** statement. Since the structure programming regards a **GO TO** statement as an undesirable function, no function corresponding to a **GO TO** statement has been provided. In fact, you can make a program without a **GO TO** function. Because no **GO TO** statement is available in ESP, you can naturally write a program matching the structured programming concept.

A subroutine call (**CALL** statement) can be realized by a goal call of the body part of a clause. **P:-Q,R** means **P:- call(Q),call(R)**, that is, a call function is implicitly used. However, the goal call is different from a **CALL** statement of a procedural language in the following points: the body part of the called clause is executed only if unification of the head part succeeds, and if it fails backtracking is applied.

The conditional branch, the multi-way branch and the iteration functions can be realized as follows:

- (1) Conditional branch function can be realized by using the cut function and the backtracking function as shown in the following example. Note that in ESP, you have to define a separate predicate that accomplishes a conditional branch and call it from the body part desiring it.

Example:

Procedural language	ESP
:	R: - ..., Q ...
(1) if P then S1	→
else S2	Q: - P, !, S1 ;
	Q: - S2 ;
(2) if P1 then S1	
else if P2 then S2	→ Q: - P1, !, S1 ;
else if P3 then S3	Q: - P2, !, S2 ;
else S4	Q: - P3, !, S3 ;
	Q: - S4 ;
(3) if P1 then	
if P11 then S11	→ Q: - P1, !, Q1 ;
else S12	Q: - P2, !, Q2 ;
else if P2 then	Q: - S3
if P21 then S21	Q1: - P11, S11 ;
else S22	Q1: - S12 ;
else S3	
	Q2: - P21, S21 ;
	Q2: - S22 ;

## (2) Multi-way branch (CASE statement)

The multi-way branch can be realized by using a combination of conditional branches (IF). However, it can be realized more concisely by using the unification function of the head part.

Example:	Procedural language	E S P
	case X is	Q :- ..., P (X, ...) , ...
	when C1       → S1	
	when C2       → S2	P (C 1, ...) :- ! , S1 ;
	when C3       → S3	P (C 2, ...) :- ! , S2 ;
	when others   → S4	P (C 3, ...) :- ! , S3 ;
		P ( __, ...) :- S4 ;

\* where C1, C2 and C3 are constants.

If you write a multi-way branch in this form of ESP, it can be executed at high speed through the optimization function called the clause indexing (explained later).

## (3) Iteration

There are two typical cases of iteration: one is equivalent to a DO loop of FORTRAN that repeats an execution for the specified number of times, and the other repeats an execution until a certain end condition is detected like a record read/write operation during a file handling operation.

The following shows some examples of iteration in ESP:

### (1) Using a tail recursion to realize a DO loop

Procedural language	E S P
DO I= 1, N	..., do (1,N + 1), ...
CALL p(I)	
END	do(N,N) :- ! ;
	do(I,N) :- p(I),
	do(I + 1,N) ;

With this technique, if P(I) does not have any alternative, TRO is applied improving the efficiency greatly. If it has alternatives, however, the stack grows, therefore this technique is not adequate for a large loop.

(2) Using alternatives and failing to realize a DO loop

Procedural language	E S P
DO I = 1 , N	..., do(1, N), ...
CALL p(I)	do(I1, N) :- for(I1, I, N),
END	p(I), fail ;
	do( __, __ ) ;
	for(N, N, N) :-!;
	for(I, I, N) ;
	for(I0, I, N) :-
	for(I0 + 1, I, N) ;

Compared with the technique (1) above, this technique has the advantage that the stack does not grow although it has the disadvantage that the speed is reduced. Also, p(X) can use this technique only for a processing that has side effects. Take these advantages and disadvantages into consideration when deciding which technique (1) or (2) to use. In principle, use the technique (2) for a large loop to perform a processing with side effects, and the technique (1) for other loop.

The following shows two examples of an ESP program to realize a loop that terminates only when an end condition is detected.

(3) Using tail recursion to realize a loop that terminates only when an end condition is detected

Procedural language	E S P
while not eof(file) do	loop(end_of_file) :-! ;
read(ch)	loop( __ ) :-
end	read(Ch) ,
	loop(Ch) ;

(4) Using repeat to realize a loop that terminates only when an end condition is detected

Procedural language	E S P
while not eof(file) do	loop:-repeat,
read(h)	read(Ch) ,
end	Ch == end_of_file, ! ;
	or
	loop:-repeat,
	read(end_of_file), ! ;

### 5.3 Data Manipulation

In ESP, the data input/output operations to the outside of the system, like **READ** and **WRITE** statements of a conventional language or the Prolog built-in predicates **get(X)** and **put(X)**, are all accomplished with methods provided by a class of SIMPOS.

For example, reading a line of characters entered the keyboard through a window is accomplished by issuing the following method call to the object of the window.

```
:getl (Window, String)
```

A similar method call can also be used to read a line of characters from a file or other I/O device.

```
:getl (File, String)
```

For details of a method call for input/output with the outside of the system, see the explanations of window and file in chapter six. This section explains data used in an ESP program.

“Data” generally means information that can be referenced or manipulated by a program. Data is classified into the following types according to the meaning and the operation unit:

- Atom, integer, floating-point

These types of data cannot be divided into smaller entities any further.

- Structure

This is structure data that has several data items as the elements. A vector, string, list, and compound term are of this type of data. To handle more complex structure data, SIMPOS has a group of classes called the pool subsystem. An object of each class provided by the pool subsystem can handle data further extended from a structure data.

The following shows program examples in which data of each data type are used.

## (1) Atom data

Within a program, atom data can be handled as internal data that has a printable name.

Example: Make a window, and make a predicate "query" that displays in the window an English month name corresponding to a number (JIS 16-bit code) entered from the keyboard.

```
query :-   : create (#essential__window, [size(300, 100), position(10, 10) ], W),
           : activate(W),
           loop(W) ;

loop(W) :- repeat,
           :getc(W, Month),
           month(Month, Atom),
           :putt(W, Atom), :new__line(W),
           fail ;

month("#1", january) :- ! ;      month("#7", july) :- ! ;
month("#2", february) :- ! ;    month("#8", august) :- ! ;
month("#3", march) :- ! ;       month("#9", september) :- ! ;
month("#4", april) :- ! ;       month("#A", october) :- ! ;
month("#5", may) :- ! ;         month("#B", november) :- ! ;
month("#6", june) :- ! ;        month("#C", december) :- ! ;
                                month(__, '%%-Error-%%') ;
```

## (2) Integer data

Integer data is used to give a constant value used within a program. In a program, integer data can be used as an integer number for four basic arithmetic operations (addition, subtraction, multiplication and division). Single-precision integer numbers range from -2147483648 to 2147483647 (one sign bit + 31 data bits).

Example: Make a predicate "square\_\_root(N,Result)" that calculates the square root of a positive integer number specified by variable N rounding off the fractions at the first place of decimals, and unifies the approximated value with Result. The predicate must fail if N is negative.

Let X be an approximated integer value of  $\sqrt{N}$ . First, give a value to initial value X0, then calculate  $X_{n+1} = (X_n + N/X_n)/2$  for each X incremented until  $X_n \leq X_{n+1}$  holds. Then, choose a result nearer to the real number of  $\sqrt{N}$  as X.



```

square__root(N,R):-N<0,!fail; % If negative, cut the alternative and fail.
square__root(0,0):-!; % If zero, return "0".
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,!; % Jugement of iteration end
square__root(N,Xo,_,V):- (Xo+N/Xo)-2=Xn, % Approximate calculation of square root
    square__root(N,Xn,Xo,V); % Termination recursive call
initial__value(N,N):-N<100,!; % predicate to set the initial value
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/100000);
round(X,Y,V,V):-X<Y,!;
round(_,_,V,V+1);

```

### (3) Floating-point data

Floating-point data is used to give a constant value used within a program. In a program, floating-point data can be used as a floating-point number for four basic arithmetic operations (addition, subtraction, multiplication and division).

A floating-point number is a normalized hexadecimal number consisting of one sign bit, 7 exponent bits and 24 mantissa bits. A significant figure is a 6-digit decimal number. The effective range of real numbers is from  $10^{-78}$  to  $10^{75}$ .

Example: Make a predicate "exponent(X,Result)" that calculates an exponential value (ex) of a real number specified by variable X by using the following equation and unifies it with Result.

$$ex = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \dots = \frac{x^n}{n!} + \dots$$

```

exponent(0,0,1,0):-!;
exponent(1,0,2.71818):-!;
exponent(X,R):-exp(X,2.0,X,1.0+X,R); % Initial value setting
exp(_,_,T,R,R):-T<1.0^-5,!; % Jugment of iteration end
exp(X,N,T,Ro,R):-T*X/N=Tn, % Approximate calculation by recurrence formula
    exp(X,N+1.0,Tn,Ro+Tn,R); % Tail recursion

```

The following recurrence formula is used:

$$x^n/n! = (x^{n-1}/(n-1)!) * (x/n)$$

#### (4) Structure data

ESP supports a vector and a string as structure data. Since a list and a compound term can be expressed by using vectors, you can virtually use the following five kinds of structure data in ESP: a) stack vector, b) heap vector, c) compound term, d) list, and e) string. Also, use of a class provided by the SIMPOS pool subsystem allows you to use structure data such as array and keyed list array, as an object.

##### a) Stack vector

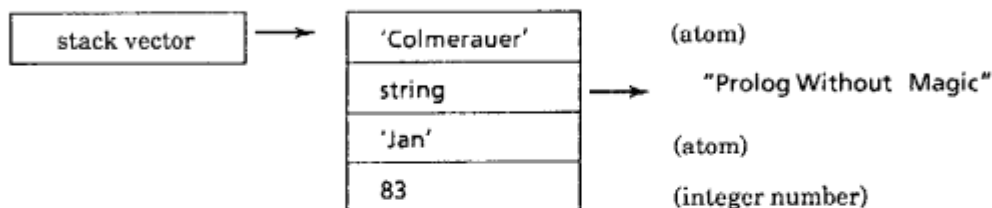
A stack vector is structure data of one-dimensional array, which is generated in the memory area (global stack) temporarily upon program execution. The memory area is released when backtracking is performed. A list and a compound term, later explained, are kinds of stack vectors. A stack vector can be either written in a source program or generated by a built-in predicate at run time. Note that a stack vector is made in a temporary memory area and cannot be saved in a slot. Consequently, a compound term and a list cannot be saved in a slot as well. Also, unification is performed for every element of a vector. Use examples of a stack vector are described below.

Example 1: Writing a stack vector in a program

Make a predicate "refer( Author, Title, Month, Year )" to search for a reference document of an ESP manual.

```
refer( { 'Colmerauer', "Prolog Without Magic", 'Jan', 83 } );  
refer( { 'Clockin', "Programming in Prolog", 'Jun', 81 } );  
refer( { 'Kowalski', "Logic as a Computer Language", 'Feb', 80 } );  
refer( { 'Pereira', "Pure LISP in Pure Prolog", 'Jul', 82 } );  
...
```

As shown in this example, you can write a stack vector in a source program by enclosing it in braces and . In this case, the stack vector has the following internal structure:



A stack vector can be unified with another stack vector. For example, you can use the predicate "refer" with an author name specified to find the document name as follows:

```
?- refer( { 'Kowalski', Title, __, __ } ).
```

Title = "Logic as a Computer Language"

A stack vector is also used internally when a list or a compound term is written in a program. This will be discussed later.

**Example 2: Using a built-in predicate to generate a stack vector**

Make a predicate that is equivalent to the Prolog built-in predicate "functor (Term, Name, Arity)". For simplicity, let "Term" be always a variable and suppose that "Name" and "Arity" have already been unified. In ESP, a compound term must be expressed by a stack vector, whose first element is a functor.

```
functor(Term, Name, Arity) :-  
    new_stack_vector(Term, Arity + 1),  
    first(Term, Name);
```

In this program, `new_stack_vector(Term, Arity)` is a built-in predicate that generates a stack vector having the number of elements specified by Arity. Also, `first(Term, Name)` is a built-in predicate that unifies data specified by Name with the 0th element of the stack vector specified by Term.

You can use this predicate "functor" to generate a new compound term.

```
?- functor(Term, foo, 3).
```

```
Term = foo(X, Y, Z)
```

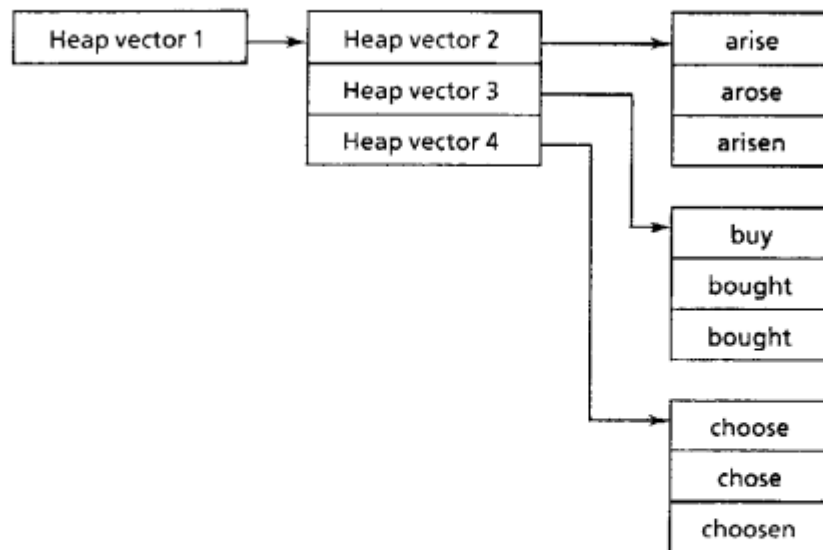
Where, X, Y and Z are logical variables.

**b) Heap vector**

A heap vector, similar to a stack vector, is structure data of a one-dimensional array, however, which is generated in the heap area at run time, and the area is not released by backtracking. You cannot write a heap vector in a source program: you need to use a built-in predicate to generate it. A generated heap vector can be saved in a slot. Note that unification is not performed for every element of a heap vector, that is, whether or not the vectors are completely identical to each other is only judged. Use examples of a heap vector are described below.

**Example:** Make predicate "create (Dictionary)" that makes a table of irregular verb forms of English, using a heap vector. Also make predicate "get (Dictionary, Present, Past, Past\_participle)" that finds the past form and the past participle form from the present form of a given verb through the irregular verb forms table. For simplicity, let the table have the following structure:

### Structure of dictionary



### Program example of predicate "create (Dictionary)"

```

create (H __ V) :-
    new __heap__vector (H __ V, 3),
    set __element (H __ V, 0, {arise, arose, arisen}),
    set __element (H __ V, 1, {buy, bought, bought}),
    set __element (H __ V, 2, {choose, chose, chosen});

set __element (H __ V, Position, {Present, Past, Past __Participle}) :-
    new __heap__vector (Element, 3),
    set __vector__element (H __ V, Position, Element)
    set __vector__element (Element, 0, Present),
    set __vector__element (Element, 1, Past),
    set __vector__element (Element, 2, Past __participle);
  
```

### Use example

?- create (Dictionary).

Dictionary = heap \_\_vector : { . . . }

Program example of predicate "get(Dictionary,Present,Past,Past\_\_participle)"

```
get (H __V, Present, Past, Past __participle) :-  
    heap__vector (H __V, Length),  
    search (H __V, 0, Length, Present, Past, Past__participle) ;  
  
search (H __V, Last, Last, __, __, __) :- !, fail ;  
search (H __V, Position, Length, Present, Past, Past __participle) :-  
    vector__element (H __V, Position, Element) ,  
    (vector__element (Element, 0, Present) , !,  
     vector__element (Element, 1, Past) ,  
     vector__element (Element, 2, Past__participle) ;  
     search (H __V, Position + 1, Length, Present, Past, Past __participle) ) ;
```

Use example

?- get (Dictionary, buy, Past, Past\_\_participle) .

Past = bought,  
Past\_\_participle = bought

c) Compound term

In ESP, a compound term is expressed internally by a stack vector whose 0th element represents the functor.

For example, a compound term  $f(X,Y,Z)$  can be unified with a stack vector  $\{f,X,Y,Z\}$ .

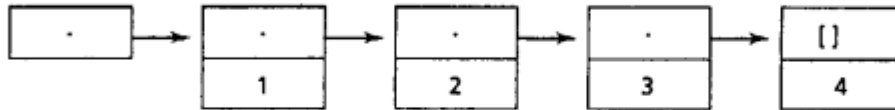
$f(X,Y,Z) = \{f,X,Y,Z\}$   
 $f(X) = \{P,Q\} = [Q|P] *$

\*This is explained in the following section "List".

d) List

In PSI, a list is expressed internally by a series of 2-element stack vectors. For example, a list [1,2,3,4] can be unified with the following stack vectors:

$$[1, 2, 3, 4] = \{\{\{\{\[], 4\}, 3\}, 2\}, 1\}$$



Similarly, lists [a,b|X] and [X|Y] can also be unified with the following stack vectors respectively:

$$[a, b|X] = \{\{X, b\}, a\}$$

$$[X|Y] = \{Y, X\}$$

Pay attention to the last example. Since [X|Y] is unified with f(Z), you must not use [X|Y] to judge whether a data item is a list.

$$[X|Y] = \{Y, X\} = f(Z)$$

For comparison, let's see the internal expression of a list in Prolog. In Prolog, usually a list is expressed internally by a compound term whose functor is designated by '.'. Therefore, a list of Prolog can be unified with the following compound term:

$$[1, 2, 3, 4] = '.' (1, '.' (2, '.' (3, '.' (4, []))))$$

$$[X|Y] = '.' (X, Y)$$

As described above, a list in ESP and a list in Prolog are different in their internal expression, and you must pay attention to this difference when handling a list.

#### e) String

A string is a structure data to represent a string of characters or bits. There are four kinds of strings by the bit width of an element:

- 1-bit string
- 8-bit string
- 16-bit string
- 32-bit string

Example: Make a predicate “getl(Window,String)” that stores characters (JIS 16-bit codes) entered from the keyboard through the window “Window”, to a 16-bit string. In this case, judge the end of entry by a carriage return (key#cr).

```
getl(Window, String) :- read __line(Window, [], Reverse __list),
                        reverse(Reverse __list, [], List),
                        list__to__string(List, String);
read __line(Window, Old __list, List) :- :getc(Window, Char),
    ( Char == key#cr, !, List = Old __list;
      read __line(Window, [Char | Old __list], List) );
reverse([], L, L) :- !;
reverse([H | T], L, R) :- reverse(T, [H | L], R);

list__to__string(List, String) :- length(List, 0, Length),
    new __string(String, Length), 16,
    set __element(List, 0, String);
length([], Length, Length) :- !;
length([H | T], Counter, Length) :-
    length(T, Counter + 1, Length);
set __element([], __, __) :- !;
set __element([H | T], Position, String) :-
    set __string__element(String, Position, H),
    set __element(T, Position + 1, String);
```

#### f) Pool subsystem

The pool subsystem consists of a group of classes that provide various kinds of structure data manipulation functions. The pool subsystem makes up for the description function of ESP and may be virtually regarded as an extended function of the ESP language.

The pool subsystem provides structure data such as array, list, stack and set, as an object. In a conventional procedural language, structure data provides a data area only. An object of the pool, however, provides not only the function to read or write data at a specified location but also the functions to read specified data, to inquire the information of stored data, and to write or read data at high speed by a hashing algorithm. These high-speed structure data manipulation functions greatly improve the productivity of programs. Furthermore, use of the inheritance function of ESP allows you to modify or add a new function to those objects provided as standard by the pool subsystem to make optimized structure data objects. The pool subsystem is realized internally by using all data types provided in ESP. Therefore, compared with when using ESP built-in predicates as are, use of the pool subsystem generally reduces the execution speed if applied to data of a simple data structure. Take this point into consideration when using the pool subsystem.

For details and usage of the pool subsystem, see section 6.1 "Pool".



## 5.4 Slot

There are two kinds of slots: attribute and component. For definition of slots to a class, it is recommended that a slot used for inheritance should be an attribute slot and a slot used only within the class should be a component slot.

For a component slot, you can read the value of a slot using the following format even if the slot belongs to a different class.

Obj! slot\_\_name

However, this usage is not recommendable since it is against the information hiding principle of object. To read the value of a slot, you should prepare a slot value reading method in the class to be referenced and always call this method.

Bad example:

- Definition of class a

```
class a has
  attribute value;
end.
```

- Reading the slot value

?- X is #a!value.

Good example:

- Definition of class a

```
class a has
  attribute value;
  :get__value (Class, Class!value) ;
end.
```

- Reading the slot value

?- :get\_\_value (#a, Value).

The following discusses data that can be substituted for a slot. Different from a logical variable of ESP, a slot is a variable that is not undefined by backtracking and is realized by using side effects. Therefore, note that you cannot substitute a variable or a stack vector (as well as a compound term or a stack vector) for a slot. The following lists data that can be substituted for a slot:

Data that can be substituted for a slot

atom, integer number, floating-point number, string, heap vector, object
-----------------------------------------------------------------------------

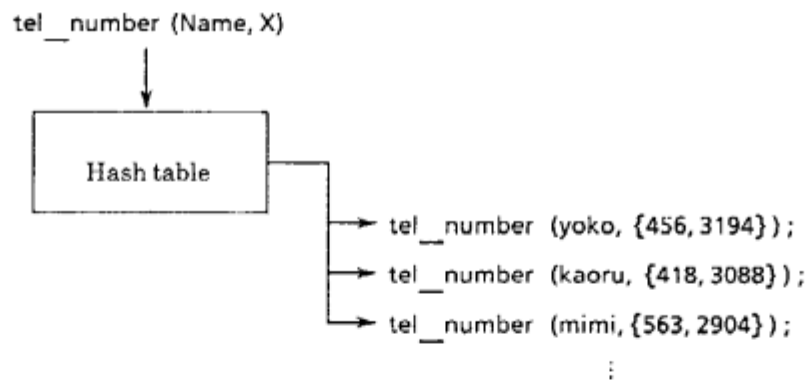
## 5.5 Clause Indexing

Clause indexing is a technique to improve the execution speed which is often used in a Prolog processing system. The clause indexing algorithm is as follows: looking at the arguments of a predicate, a hash table is made for the contents of each argument. Use of this hash table allows high-speed head unification at a predicate call.

Let's take the following example. If you want to write a program for a data base containing pairs of personal names and telephone numbers using predicate "tel\_\_number(Name,{Area\_\_code,Number})", you may have the following program:

```
tel__number (yoko, {456, 3194});  
tel__number (kaoru, {418, 3088});  
tel__number (mimi, {563, 2904});  
:  
:
```

With this program, when calling goal tel\_\_number(hanako,X), if every clause is to be checked serially from the beginning and there are n number of clauses, these clauses need to be checked on average  $n/2$  times. Therefore, the more clauses that exist, the more execution time it takes. The clause indexing technique is used to improve the clause search speed, in which the ESP compiler internally creates a hash table as looking at the argument values in the program. This allows retrieval of the desired clause within an almost constant time regardless of the number of clauses existing.



In ESP, optimization by clause indexing is performed if the following conditions hold:

- For local predicates, the first argument is to be indexed, and for method calls, the second argument is to be indexed.
- There are some consecutive clauses which have a constant value at the argument position specified above, and the argument may have two or more different values.
- If the argument to be indexed is a stack vector\*, its first element is to be indexed.

Upon programming, you need to place the argument that you want to index at the first argument for local predicates, and at the second argument for method calls. Note that if a clause that has a variable as the argument is encountered, clause indexing is terminated at that position since unification for a variable always succeeds.

The high speed effect by clause indexing is great if many clauses exist. Therefore, you should use it in cases where it is applicable.

- \* Since a compound term is internally a stack vector, it can be indexed. Therefore, with the following program, the data item such as yoko, yuko, hiromi and linda can be indexed.

```
tel__number (yoko (22) , {456, 3194} )  
tel__number (yuko (18) , {483, 8173} )  
tel__number (hiromi (25) , {456, 3196} )  
tel__number (linda (13) , {456, 1999} )
```

## APPENDIX A GRAMMAR OF ESP

### ©Notational Rules

- “X” designates terminating symbol X. To specify a double quote as is, give two successive double quotes (“”).
- {X} indicates that you can repeat X zero or more times.
- [X] indicates that one X exists or nothing exists being omitted. That is, it indicates an arbitrary selection.

<lowercase character> ::=

<kanji> | <hiragana> | <katakana> | <Russian> | <Greek>

| "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"

<uppercase character> ::=

"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"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<special character> ::=

"@" | "#" | "\$" | "%" | "&" | "\*" | "+" | "-" | "="  
| "~" | "`" | "?" | "/" | "\" | "." | "<" | ">" | ":",

<formatting character> ::= " " | <new-line> | <tab code>

<delimiting character> ::= " ( " ) " . " | " | " | " { " | " } " | " ! " | " | " | " , " | " ; " |

<lexical element> ::= <atom> | <number> | <character string> |

<variable> | <delimiter>

<atom> ::=

<lowercase character> {<subsequent character>}  
| <special character> {<special character>}  
| <quoted atom>

<subsequent character> ::= <lowercase character> | <uppercase character> | <digit> | "\_"

<quoted atom> ::= "" {<character for atom>} ""

<character for atom> ::= <any character except for single quote> | "" ""

<number> ::= <integer number> | <floating-point number>

<integer number> ::= ["-"] <digit string>

<floating-point number> ::= ["-"] <digit string> "." <digit string> [<exponent part>]

<digit string> ::= <number> {<number>}

$\langle \text{exponent part} \rangle ::= \text{"^"} [ "+" | "-" ] \langle \text{digit string} \rangle$   
 $\langle \text{character string} \rangle ::= \text{"\""} \{ \langle \text{character for string} \rangle \} \text{"\""}$   
 $\langle \text{character for string} \rangle ::= \langle \text{any character except for double quote} \rangle | \text{"\""}$   
 $\langle \text{variable} \rangle ::= \langle \text{first character of variable} \rangle \{ \langle \text{subsequent character} \rangle \}$   
 $\langle \text{first character of variable} \rangle ::= \langle \text{uppercase character} \rangle | \text{"\_"}$   
 $\langle \text{delimiter} \rangle ::= \langle \text{delimiting character} \rangle$   
 $\langle \text{term} \rangle ::= \langle \text{variable} \rangle | \langle \text{atomic literal} \rangle | \langle \text{compound literal} \rangle$   
 $\langle \text{compound literal} \rangle ::= \langle \text{vector} \rangle | \langle \text{string} \rangle | \langle \text{compound term} \rangle |$   
 $\langle \text{operator-applied term} \rangle | \langle \text{list} \rangle$   
 $\langle \text{atomic literal} \rangle ::= \langle \text{symbol} \rangle | \langle \text{number} \rangle$   
 $\langle \text{symbol} \rangle ::= \langle \text{atom} \rangle$   
 $\langle \text{vector} \rangle ::= \langle \text{null vector} \rangle | \langle \text{non-null vector} \rangle$   
 $\langle \text{null vector} \rangle ::= \text{"\{\}"}$   
 $\langle \text{non-null vector} \rangle ::= \text{"\{"} \langle \text{term list} \rangle \text{"\}"}$   
 $\langle \text{term list} \rangle ::= \langle \text{term} \rangle \{ \text{","} \langle \text{term} \rangle \}$   
 $\langle \text{string} \rangle ::= \langle \text{character string} \rangle$   
 $\langle \text{compound term} \rangle ::= \langle \text{functor} \rangle \text{"("} \langle \text{argument list} \rangle \text{"}"}$   
 $\langle \text{functor} \rangle ::= \langle \text{symbol} \rangle$   
 $\langle \text{argument list} \rangle ::= \langle \text{argument} \rangle \{ \text{","} \langle \text{argument} \rangle \}$   
 $\langle \text{argument} \rangle ::= \langle \text{term} \rangle$   
 $\langle \text{operator-applied term} \rangle ::=$   
 $\quad \langle \text{prefix operator-applied term} \rangle |$   
 $\quad \langle \text{postfix operator-applied term} \rangle |$   
 $\quad \langle \text{infix operator-applied term} \rangle$   
 $\langle \text{prefix operator-applied term} \rangle ::= \langle \text{prefix operator} \rangle \langle \text{term} \rangle$   
 $\langle \text{postfix operator-applied term} \rangle ::= \langle \text{term} \rangle \langle \text{postfix operator} \rangle$   
 $\langle \text{infix operator-applied term} \rangle ::= \langle \text{term} \rangle \langle \text{infix operator} \rangle \langle \text{term} \rangle$   
 $\langle \text{list} \rangle ::= \langle \text{null list} \rangle | \langle \text{non-null list} \rangle$

```

<null list> ::= "[ ]"

<non-null list> ::= "[" <term list> "]" <term> "]"

<class definition> ::=
    "class" <class name>
    [ <macro bank declaration> ]
    "has"
    { <inheritance declaration> "," }
    { <class slot definition> ";" }
    { <class clause definition> ";" }
    ["instance"
    { <instance slot definition> ";" }
    { <instance clause definition> ";" }]
    ["local"
    { <local clause definition> ";" }]
    "end" " "

<class name> ::= <atom>

<package name> ::= <atom>

<class identifier> ::= <class name> | <package name> "##" <class name>

<macro bank identifier> ::= <macro bank name> | <package name> "##"
    <macro bank name>

<macro bank declaration> ::= "with__macro" <macro bank identifier>

<macro bank name> ::= <atom>

<inheritance declaration> ::=
    "nature"
    <inheritance class name> { " " <inheritance class name> }

<inheritance class name> ::= <class identifier> | "*"

<class clause definition> ::= <method clause definition>

<instance clause definition> ::= <method clause definition>

<local clause definition> ::= <clause definition>

<method clause definition> ::= [ <demon type> ":" <clause definition> ]

<clause definition> ::= <head> [ ":" <body> ]

<demon type> ::= "before" | "after"

```

```

<head> :: = <term>

<body> :: = <goal list>

<goal list> :: = <goal list> {"," <goal list>}
                | "(" <goal list> {"," <goal list>} ")"
                | <goal>

<goal> :: = <method call> | <predicate call>

<predicate call> :: = <term>

<method call> :: =
    <usual method call>
    | <class method call>
    | <instance method call>

<usual method call> :: = ":" <term>

<class method call> :: = <class identifier> ":" <term>

<instance method call> :: = ":" <class identifier> ":" <term>

<class slot definition> :: = <slot definition>

<instance slot definition> :: = <slot definition>

<slot definition> :: =
    <slot type> <slot definition item>
    {"," <slot definition item>}

<slot type> :: = "attribute" | "component"

<slot definition item> :: =
    <slot name> {<slot initialization>}
    | "(" <slot name> ":" <term>
      ":" - <slot initialization code> ")"

<slot name> :: = <atom>

<slot initialization> :: =
    ":" <term> | "is" <class name>

<slot initialization code> :: = <goal list>

```



```

<macro bank definition> ::
    = "macro__bank" <macro bank name>
      [ <macro bank declaration> ]
      "has"
      [ <inheritance declaration> ";"]
      { <slot definition> ";"}
      { <macro definition> ";"}
      ["local"
        { <local clause definition> ";"} ]
      "end" "."

<macro definition> :: =
    <object pattern> "= >"
    <expanded pattern> [ <clause expansion execution condition> ]
    ":-" <expansion condition> |
    <object pattern> "= >"
    [ <expanded pattern> ] [ <execution condition> ]
    ":-" <expansion condition> |
    <clause insert statement>

<object pattern> :: = <term>

<expanded pattern> :: = <term>

<clause expansion execution condition> :: =
    ["when <generating goal list>]
    ["where" <testing goal list>]
    [ <execution condition> ]

<execution condition> :: =
    ["with" { <local clause definition> ";"}]
    ["by" <variable> ]

<generating goal list> :: = <goal list>

<testing goal list> :: = <goal list>

<expansion condition> :: = <body>

<clause insert statement> :: = <insert type> "(" <predicate type. ">,"
{ <clause definition> ";"} ")"

<insert type> :: = "inserta" | "insertz"

<predicate type> :: = "c" | "i" | "l"

```

## APPENDIX B TABLE OF STANDARD OPERATORS

class	fx	1180	is	xfx	700
has	xfx	1160	= . .	xfx	700
instance	fx, xfx	1155	= =	xfx	700
local	fx, xfx	1154	\ = =	xfx	700
;	xfx, xf	1150	@ <	xfx	700
	xfx	1150	@ >	xfx	700
with__macro	xfx	1140	@ = <	xfx	700
before	fx	1140	@ > =	xfx	700
after	fx	1140	= : =	xfx	700
nature	fx	1120	= \ =	xfx	700
component	fx	1120	<	xfx	700
attribute	fx	1120	>	xfx	700
: -	fx, xfx	1100	= <	xfx	700
? -	fx	1100	> =	xfx	700
- - >	xfx	1100	: =	xfx	700
= >	xfx	1099	+	fx, yfx	500
= = >	xfx, xf	1099	-	fx, yfx	500
when	yfx	1098	/\	yfx	500
where	yfx	1098	\ /	yfx	500
with	fx, yfx	1098	xor	yfx	500
by	fx, yfx	1098	*	yfx	400
mode	fx	1080	/	yfx	400
public	fx	1080	< <	yfx	400
- >	xfy	1040	> >	yfx	400
,	xfy	1000	div	yfx	400
'	fx	999	mod	xfx	400
spy	fy	900	^	xfy	200
cspy	fx	900	!	yfx	200
nospy	fy	900	:	fy, xfx	100
\ +	fy	900	#	fy, xfx	90
=	xfx	700	# #	xfx	50

## APPENDIX C LIST OF STANDARD MACROS

### (1) Notation of constant value

radix#"character string"	Radix-of-2 to -36 notation
#"character"	Character code
control#"character"	Control-keyed character code
meta#"character"	Meta-keyed character code
control__meta#"character"	Control-meta-keyed character code
key#name	Special key
meta#name	Meta-keyed special key
pf#number	Function key
keypad#"character"	Keypad key
mouse#click	Mouse entry
ascii#"character string"	ASCII code string
string#"character string"	JIS 16-bit code string
jis#"character string"	JIS 16-bit code string
words#"character string"	32-bit string
double__bytes#{vector}	16-bit string
bytes#{vector}	8-bit string
bits#{vector}	1-bit string

### (2) Arithmetic operations

X is Y	Unify
X = Y	Unify
X = = Y	Equal
X \ = = Y	Not equal
X = : = Y	Identical
X = \ = Y	Not identical
X < Y	Comparison
X = < Y	Comparison
X > Y	Comparison
X > = Y	Comparison
X + Y	Addition
X - Y	Subtraction
X * Y	Multiplication
X / Y	Division
X div Y	Division (for integer only)
X mod Y	Modulo residue
- X	Reverse sign

<code>X /\ Y</code>	Bitwise logical product (AND)
<code>X \/ Y</code>	Bitwise logical sum (OR)
<code>X xor Y</code>	Bitwise exclusive OR
<code>X &gt;&gt; Y</code>	Right shift
<code>X &lt;&lt; Y</code>	Left shift
<code>\(X)</code>	Bit complement

### (3) Execution of ESP

<code>#class-name</code>	Class object
<code>:method-name(arguments)</code>	Method call
<code>class-name:method-name(arguments)</code>	Class method call with a class specified
<code>: class-name:method-name(arguments)</code>	Instance method call with a class specified
<code>X = Obj ! Slot __name</code>	Slot access
<code>X = = Obj ! Slot __name</code>	Slot access
<code>Obj ! Slot __name = X</code>	Slot access
<code>Obj ! Slot __name = = X</code>	Slot access
<code>Obj ! Slot __name</code>	Slot access
<code>Obj ! Slot __name : = X</code>	Slot assignment

### (4) Others

<code>unique__atom(logical-variable)</code>	Unique atom
<code>standard#input</code>	Standard input port
<code>standard#output</code>	Standard output port
<code>standard#message</code>	Standard message output port

### (5) Macro expansion suppression

<code>`` (term)</code>	All suppressed
<code>` (term)</code>	One level suppressed

## APPENDIX D LIST OF BUILT-IN PREDICATES

`absolute__cut (Level)`  
`absolute__cut __and __fail (Level)`  
`add (N1, N2, ^ R)`  
`add __extended (I1, I2, ^ R __up, ^ R __low)`  
`and (I1, I2, ^ R)`  
`and __string (S, Position, Length, MaskS)`  
`apply (Predicate, S __V __Arg)`  
`atom (?X)`  
`atomic (?X)`  
`bind __hook (?X, Goals)`  
`bound (?X)`  
`code (?X, ^ Length)`  
`complement (I, ^ R)`  
`complement __string (S, Position, Length)`  
`!`  
`cut __and __fail`  
`decrement (I, ^ R)`  
`divide (N1, N2, ^ R)`  
`divide __extended (I1 __up, I1 __low, I2, ^ Q, ^ Rem)`  
`divide __with __remainder (I1, I2, ^ Q, ^ Rem)`  
`equal (?X, ?Y)`  
`equal __string (S1, S2)`  
`exception __hook (Exception, Predicate)`  
`fail`  
`first (V, ^ Element)`  
`first __location (H __V, ^ Loc)`  
`floating__point (?X)`  
`floating__point __to__integer (F, ^ I)`  
`hash (?X, ^ Hash)`  
`heap__vector (?X, ^ Length)`  
`identical (?X, ?Y)`  
`increment (I, ^ R)`  
`integer (?X)`  
`integer __to__floating__point (I, ^ F)`

```

less__than(?X, XY)
level ( ^ Level)
location(?X)
location__element(Loc, ^ Element)
method__call(Object, Method, ^ Arg1, ..., ^ ArgN)
minus(N, ^ R)
move__string__element(S, Position, Length, Shiftcount)
multiply(N1, N2, ^ R)
multiply__extended(I1, I2, ^ R __up, ..., ^ R __low)
new__atom( ~Atom)
new__heap__vector( ~H __V, Length)
new__protected__object( ~P, Key, Value)
new__stack__vector( ~S __V, Length)
new__string( ~S, Length, S __size)
not__equal(?X, ?Y)
not__equal__string(S1, S2)
not__identical(?X, ?Y)
not__less__than(?X, ?Y)
number(?X)
on__backtrack(Goals)
or(I1, I2, ^ R)
or__string(S, Position, Length, Masks)
overridden__method__call(Table, Method, ^ Arg1, ..., ^ ArgN)
predicate__call(Predicate, ^ Arg1, ... ^ ArgN)
protected__type(?X)
protected__value(P, Key, ^ Value)
raise(Exception, S __V __8)
relative__cut(Level)
relative__cut__and__fail(Level)
search__character(S, Start, End, Charactercode, ^ Position)
search__characters__backward(S, Start, Table1, Table2, ^ Position)
search__characters__forward(S, Start, Table1, Table2, ^ Position)
search__string__difference(S1, S2, ^ Position)
second(V, ^ Element)
second__location(H __V, ^ Loc)
set__first(H __V, Element)
set__location__element(Loc, Element)
set__protected__value(P, Key, Value)

```

```

set __second (H __V, Element)
set __slot (Object, Slot, Value)
set __string__element (S, Position, Element)
set __substring (S, Position, Length, SubS)
set __subvector (H __V, Position, Length, SubV)
set __vector__element (H __V, Position, Element)
shift __left (l, Count, ^ R)
shift __right (l, Count, ^ R)
slot (Object, Slot, ^ Value)
stack __vector (?X, ^ Length)
string (?X, ^ Length, ^ S type)
string__element (S, Position, ^ Element)
string__tail (S, Position, ^ Element)
structure (?X)
substring (S, Position, Length, ^ SubS)
subtract__extended (l1, l2, ^ S, ^ R)
subvector (V, Position, Length, ^ SubV)
succeed (Level)
true
type (?X, ^ Tag)
unbound (?X)
unify (^ X, ^ Y)
value (?X, ^ Value)
vector__element (V, Position, ^ Element)
vector__element __location (H __V, Position, ^ Loc)
vector__tail (V, Position, ^ SubV)
xor (l1, l2, ^ R)
xor __string (S, Position, length, Masks)

```

## APPENDIX E KEYBOARD CODE ENTRY TABLE

control						control					
	JIS kanji	ASCII	control	meta	meta		JIS kanji	ASCII	control	meta	meta
a	2361	61	1	E1	81	A	2341	41	1	C1	81
b	2362	62	2	E2	82	B	2342	42	2	C2	82
c	2363	63	3	E3	83	C	2343	43	3	C3	83
d	2364	64	4	E4	84	D	2344	44	4	C4	84
e	2365	65	5	E5	85	E	2345	45	5	C5	85
f	2366	66	6	E6	86	F	2346	46	6	C6	86
g	2367	67	7	E7	87	G	2347	47	7	C7	87
h	2368	68	8	E8	88	H	2348	48	8	C8	88
i	2369	69	9	E9	89	I	2349	49	9	C9	89
j	236A	6A	A	EA	8A	J	234A	4A	A	CA	8A
k	236B	6B	B	EB	8B	K	234B	4B	B	CB	8B
l	236C	6C	C	EC	8C	L	234C	4C	C	CC	8C
m	236D	6D	D	ED	8D	M	234D	4D	D	CD	8D
n	236E	6E	E	EE	8E	N	234E	4E	E	CE	8E
o	236F	6F	F	EF	8F	O	234F	4F	F	CF	8F
p	2370	70	10	F0	90	P	2350	50	10	D0	90
q	2371	71	11	F1	91	Q	2351	51	11	D1	91
r	2372	72	12	F2	92	R	2352	52	12	D2	92
s	2373	73	13	F3	93	S	2353	53	13	D3	93
t	2374	74	14	F4	94	T	2354	54	14	D4	94
u	2375	75	15	F5	95	U	2355	55	15	D5	95
v	2376	76	16	F6	96	V	2356	56	16	D6	96
w	2377	77	17	F7	97	W	2357	57	17	D7	97
x	2378	78	18	F8	98	X	2358	58	18	D8	98
y	2379	79	19	F9	99	Y	2359	59	19	D9	99
z	237A	7A	1A	FA	9A	Z	235A	5A	1A	DA	9A
@	2177	40	0	C0	80						
[	214E	5B	1B	DB	9B						
]	214F	5D	1D	DD	9D						
\	2140	5C	1C	DC	9C						
^	2130	5E	1E	DE	9E						
_	2132	5F	1F	DF	9F						



	JIS kanji	ASCII	meta		JIS kanji	ASCII	meta
0	2330	30	B0	!	212A	21	A1
1	2331	31	B1	"	2149	22	A2
2	2332	32	B2	#	2174	23	A3
3	2333	33	B3	\$	2170	24	A4
4	2334	34	B4	%	2173	25	A5
5	2335	35	B5	&	2175	26	A6
6	2336	36	B6	'	2147	27	A7
7	2337	37	B7	(	214A	28	A8
8	2338	38	B8	)	214B	29	A9
9	2339	39	B9	*	2176	2A	AA
				+	215C	2B	AB
				,	2124	2C	AC
				-	215D	2D	AD
				.	2125	2E	AE
				/	213F	2F	AF
				:	2127	3A	BA
				;	2128	3B	BB
				<	2163	3C	BC
				=	2161	3D	BD
				>	2164	3E	BE
				?	2129	3F	BF
				`	2146	60	E0
				{	2150	7B	FB
					2143	7C	FC
				}	2151	7D	FD
				~	2141	7E	FE

						meta
pf#0	80	keypad#"0"	B0	key#bell	7	87
pf#1	81	keypad#"1"	B1	key#bs	8	88
pf#2	82	keypad#"2"	B2	key#tab	9	89
pf#3	83	keypad#"3"	B3	key#lf	A	8A
pf#4	84	keypad#"4"	B4	key#cr	D	8D
pf#5	85	keypad#"5"	B5	key#esc	1B	9B
pf#6	86	keypad#"6"	B6	key#del	7F	FF
pf#7	87	keypad#"7"	B7	key#help	88	88
pf#8	88	keypad#"8"	B8	key#abort	89	89
pf#9	89	keypad#"9"	B9	key#up	8F	8F
pf#10	8A	keypad#","	AC	key#down	90	90
pf#11	8B	keypad#"-"	AD	key#left	91	91
pf#12	8C	keypad#"."	AE	key#right	92	92
pf#13	8D			key#enter	FF	FF
pf#14	8E					
pf#15	8F					
pf#16	90					
pf#17	91					
pf#18	92					

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

< 付録 F > J I S コード表

APPENDIX F

	1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35	36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50
1D	1E	1F
20	21	22
23	24	25
26	27	28
29	2A	2B
2C	2D	2E
2F	30	31
32	33	34
35	36	37
38	39	3A
3B	3C	3D
3E	3F	40
41	42	43
44	45	46
47	48	49
4A	4B	4C
4D	4E	4F
50	51	52
53	54	55
56	57	58

[illegible]



51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	84	85	86
57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	84	85	86						
57	58	59	5A	5B	5C	5D	5E	5F	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F	80	81	82	83	84	85	86						

## APPENDIX G PRECAUTIONS FOR CONVERSION FROM PROLOG TO ESP

Appendix G discusses the differences in language specifications and implementation method between Prolog and ESP, which should be taken into consideration when converting a Prolog program to an ESP program. The following discussions concern the most common various Prolog processors since much of the language specifications and implementation method depend on a particular language processor. Therefore, all of the following discussions may not apply to the Prolog language processor you are particularly thinking of. The discussions include the following items:

- |                               |                       |
|-------------------------------|-----------------------|
| 1. Data type                  | 5. Meta logic         |
| 2. Arithmetic operations      | 6. Program operation  |
| 3. Data comparison            | 7. Internal data base |
| 4. Execution sequence control |                       |

### 1. Data type

Prolog does not have the string and stack vector data types of ESP. Also, a list and a compound term are different in internal data form at between Prolog and ESP. This section describes the precautions related to the data type difference between Prolog and ESP that you need to know when reforming a Prolog program to an ESP program.

#### (1) Stack vector and compound term

In ESP, a compound term is expressed internally by a stack vector. A compound term of Prolog can be regarded as a special form of ESP stack vector. However, there is one big difference: the functor of a stack vector can be a variable. A stack vector has the following standard form at:

$$\{X0, X1, X2, \dots, Xn\}$$

With this form at, if X0 is an atom, f for example, this stack vector is equal to the data of the following form at, which can be written in a program.

$$f(X1, X2, \dots, Xn)$$

Therefore, you can use a Prolog compound term as is in ESP without paying any attention except that f(X) can be unified with {A,B}.

## (2) List

In Prolog, a list is expressed internally as a two-element compound term that has an atom '.' as the operator. Therefore, the following goal calls always succeed:

```
[1,2,3] = '.' (1, '.' (2, '.' (3, [])))
[X|Y] = '.' (X,Y)
functor([X|Y], '.', 2)
arg(1, [X|Y], X)
arg(2, [X|Y], Y)
```

In ESP, a list is expressed internally as a two-element stack vector. Therefore, the following goal calls always succeed:

```
[1,2,3] = [{[ ], 3}, 2], 1}
[X|Y] = (Y, X)
stack_vector([X|Y], 2)
vector_element([X|Y], 0, Y)
vector_element([X|Y], 1, X)
```

The difference in the internal expression between the Prolog and the ESP lists usually gives no problem upon manipulation of lists in an ordinary application program. However, note that a list whose CDR part (the rest of the list excluding the first element) is a variable is unified with a one-element compound term. For example, ---

[X|Y] = f(Z)

it succeeds, and X and Y are unified with Z and f respectively (i.e., X=Z and Y=f).

Therefore, predicate "is\_\_list(X)" that judges whether or not given data X is a list may be written as follows in Prolog and ESP:

- is\_\_list(X) written in Prolog

```
is __list(X) :- var(X), !, fail.
is __list([_ | _]).
is __list([ ]).
```

Note that in this case, is\_\_list('.(X,Y)) succeeds.



- `is__list(X)` written in ESP

```
is__list(X) :- stack__vector(X, 2), !,
              first(X, Cdr),
              (unbound(Cdr), !;
              is__list(Cdr));
is__list([]);
```

Note that in this case, `is__list({X,Y})` succeeds.

### (3) String

Prolog does not have the data type of string. However, writing "ABC" is internally expressed as an ASCII code list [65,66,67]. ESP has the data type of string, which is completely different from an ASCII code list in Prolog.

The usage of a character string enclosed in double quotes (referred to as a string hereinafter for both Prolog and ESP) differs between Prolog and ESP when strings are unified. In Prolog, since a string is a list of ASCII codes, unification between strings is performed according to the rules for unification between lists, and unification is performed for every element. For example, the following unifications succeed:

```
"ABC" = "ABC"
"ABC" = [X, Y, Z]
```

The following unifications fail:

```
"ABC" = "ACB"
"ABC" = [65, 67, 66]
```

In ESP, since a string is data that has side effects, unification between strings is performed according to the rules shown in table 4-2, and unification is not performed for every element. Therefore, the following unifications all fail:

```
"ABC" = "ABC"
"ABC" = [X, Y, Z]
"ABC" = "ACB"
"ABC" = [65, 67, 66]
```

If you want to check the equality of all elements between strings in ESP, you can use the following built-in predicate:

```
equal__string("ABC", "ABC")
```

#### (4) Atom

In Prolog, an atom is deeply related to its print name: the print name of an atom can be obtained by a built-in predicate. For example, to obtain the print name of an atom called "atom", you use the built-in predicate:

```
name(atom,Name)
```

This predicate unifies the ASCII code list [97,116,11,109] of the atom "atom" with Name.

In ESP, atoms are associated with their print names by using the function of SIMPOS basic class "symbolizer". For example, to obtain the print name of an atom, you use the following:

```
:get__atom__token(#symbolizer,atom,Name)
```

This unifies the JIS 16-bit code string of atom "atom" with Name. Note that comparison between atoms is not accomplished by their literal names. (See table 4-4).

## 2. Arithmetic operations

### (1) Execution by macro expansion and interpreter

One of the main differences in arithmetic operations between Prolog and ESP is that an arithmetic operation is executed after being macro-expanded or is executed directly by the interpreter. With a Prolog interpreter, an arithmetic expression is executed directly by the interpreter, while in ESP, an arithmetic expression is macro-expanded to a sequence of built-in predicates, then executed.

Note that even in Prolog, if an arithmetic expression is compiled, it is expanded similar to ESP. However, in Prolog, the execution by the interpreter is not compatible with the execution by the compiler as shown in the following program example:

[Program]                       $p(X,Y):-Y \text{ is } X + 1$

[Execution example 1]         $?-p(3,Z).$   
In both ESP and Prolog, Z is unified with 4.

[Execution example 2]         $?-p(3 + 2,Z).$   
In ESP and in Prolog interpreter, Z is unified with 6. However, the execution process differs greatly between them as described below.

In ESP, when the goal  $p(3 + 2,Z)$  is entered, it is macroexpanded to the following, then executed:

```
add(3,2,X), p(X,Z)
```

Therefore, when goal *p* is called, the first argument of *p* is already 5. In the Prolog interpreter, however, the first argument of *p* is unified with  $3 + 2$ , namely, with a two-element compound term with operator “+”, and operation  $3 + 2 + 1$  is executed in built-in predicate “is”. In Prolog, a call such as *p*( $3 + 2, X$ ), if it is issued after the program has been compiled, results in an error.

## (2) Word length

In ESP, an integer number is expressed as a 32-bit signed or unsigned integer number, and an overflow is notified to the user by exception handling. In Prolog, handling an integer number depends on the machine and the language processor. Therefore, you cannot make a general discussion. The following is an example of a language processor.

In Prolog, an integer number is expressed in 18 bits, and no overflow check is performed. In the course of an operation, however, a 36-bit integer is allowed. to compensate for this disadvantage, Prolog has two-argument operator *xwd*, allowing 36-bit operations. For example, compound term *xwd*(1,5) can be used in an arithmetic operation as integer 262,149 (1 is shifted to the left by 18 bits and the result is added with 5)..

## (3) Arithmetic operators

Some arithmetic operators are supported in both ESP and Prolog, and some are supported in only one of them.

- Arithmetic operators supported in ESP and Prolog

$X + Y$	Add X and Y.
$X - Y$	Subtract Y from X.
$X * Y$	Multiply X and Y.
$X / Y$	Divide X by Y.
$X \bmod Y$	Obtain the remainder of the division of X by Y.
$-X$	Reverse the sign of X.
$X \setminus Y$	Obtain the logical product of X and Y.
$X \setminus / Y$	Obtain the logical sum of X and Y.
$\setminus X$	Obtain the bit complement of X.
$X << Y$	Shift X to the left by Y bits.
$X >> Y$	Shift X to the right by Y bits.

- Arithmetic operators supported only in ESP

$X \text{ div } Y$	Divide X by Y and convert the result to integer.
--------------------	--------------------------------------------------

**Note:** With a Prolog language processor that can handle only integers,  $X \text{ div } Y$  is virtually equal to  $X/Y$ .

- Arithmetic operators supported only in Prolog (of a particular processor)

<code>!(X)</code>	Remainder of the division of X by $2^{18}$ if $0 \leq X \leq 2^{18} - 1$ holds
<code>\$(X)</code>	Remainder of the division of X by $2^{18}$ if $-2^{17} \leq X \leq 2^{18} - 1$ holds
<code>xwd(X,Y)</code>	Shift X to the left by 18 bits, and add Y to it.
<code>[X]</code>	Integer value of X

#### (4) Floating-point number

ESP can handle floating-point numbers while many Prolog language processors cannot.

#### (5) Double-word operation

ESP can perform double-word (64 bits) operations while many Prolog language processors cannot.

### 3. Data comparison

#### (1) Equality and identity comparison

Prolog and ESP support the following four predicates for equality and identity comparison:

<code>X = Y</code>	Succeed if X is equal to Y.
<code>X \= Y</code>	Succeed if X is not equal to Y.
<code>X := Y</code>	Succeed if X is identical to Y.
<code>X \= Y</code>	Succeed if X is not identical to Y.

Since identity comparison differs depending on a particular language processor, it is recommended to use it only for comparison of atom and integer data.

Although these four predicates are common between Prolog and ESP, attention needs to be paid to the equality comparison of strings: in Prolog, since the data type of string is not supported, the equality comparison between strings means the equality comparison between lists. In ESP, the data type of string is supported, and "`=`" or "`\=`" does not perform equality comparison of every element. To perform the equality comparison of every element between strings in ESP, use built-in predicate "`equal_string`" or "`not_equal_string`".

#### (2) Less-greater comparison

Prolog has two kinds of built-in predicates for less-greater comparison.

- Less-greater comparison between integer numbers: `<`, `=`, `>`, `=`, `>`
- Less-greater comparison between terms: `@<`, `@=`, `@>`, `@=`, `@>`

In ESP, the built-in predicates for less-greater comparison  $<$ ,  $=$ ,  $>$ ,  $=$ , and  $>$  can be applied to both integer numbers and terms. When using these built-in predicates, note that the less-greater comparison of atoms is completely different between Prolog and ESP. In Prolog, the less-greater comparison of atoms is performed with respect to the alphabetical order of the atom name. In ESP, since atoms are associated with their names in the software level (by the symbolizer of SIMPOS), the atom names are unknown when the built-in predicate is executed. Therefore, the less-greater comparison of atoms is performed with respect to the atom number. The atom number has nothing to do with the atom name: it is a number given to an atom in the order of the appearance of atoms.

#### 4. Execution sequence control

Among the built-in predicates for program execution control, the following are completely the same between Prolog and ESP:

$P, Q$	If the execution of goal $P$ succeeds, execute goal $Q$ .
$P;Q$	If the execution of goal $P$ fails, execute goal $Q$ .
<code>true</code>	Always succeeds.
<code>fail</code>	Always fails.
$X = Y$	Unifies $X$ with $Y$ .
<code>repeat</code>	Has infinite number of alternatives.

For “;”, since the priorities of functors “:-” and “;” are different between ESP and Prolog, these functors must be written differently between ESP and Prolog.

Example: In Prolog

$R :- P;Q.$                       or    $R :- (P;Q).$

In ESP

$R :- (P;Q);$

Built-in predicate “!” is supported both in Prolog and ESP. However, note that when it is used in an OR clause, the range of the cut differs between Prolog and ESP. (For details, see section 4.6.)

Example: In Prolog

$P :- Q, (R, !; S).$        .....    ①  
 $P :- T.$                                .....    ②

! of clause ① also cuts clause ② which is the alternative of  $P$ .

Example: In ESP

```
P :- Q, (R, ! ; S). ..... ①
P :- T. .... ②
```

! of clause ① cuts goal S which is the alternative in the OR clause, but does not cut clause ② which is the alternative of P.

In ESP, if you want to use the same function as the cut in OR clause of Prolog, you may use built-in predicate "relative\_\_cut" in the following way. However, this is effective only after compilation.

Example: To realize the cut in OR clause of Prolog in ESP

```
P :- Q, (R, relative__cut(1) ; S) ;
P :- T ;
```

The built-in predicate specifying negation is different in the name between Prolog and ESP, but the same in the function.

- In ESP ..... not(P)
- In Prolog ..... \+ (P)

## 5. Meta logic

### (1) Data type checking

The following are the built-in predicates to check the data type, which are the same in function and name between Prolog and ESP.

atom(X) Succeeds if X is bound currently with an atom.

integer(X) Succeeds if X is bound currently with an integer number.

atomic(X) Succeeds if X is bound currently with either an atom or an integer number. In ESP, however, it succeeds even if X is bound with a floating-point number.

The following are the built-in predicates which are the same in the function but different in the name between Prolog and ESP.

In ESP ..... Unbound(X), bound(X)

In Prolog ..... var(X), nonvar(X)

### (2) Manipulation of structures

Prolog has three built-in predicates to manipulate structures: "functor", "arg" and "=..". In ESP there are no corresponding built-in predicates. However, the equivalent functions can be easily realized in ESP as follows:

- Realization of functor (T,F,N) in ESP
- When T is unified with a structure

```

functor (T, F, N) :-
    structure (T) , ! ,
    first (T, F) ,
    stack__vector (T, N1) ,
    N = N1-1 ;

```

- When T is unified with an atom

```

functor (T, F, N) :-
    atom (T) , ! ,
    F = T ,
    N = 0 ;

```

- When T is a variable and F is unified with an atom, and N is unified with a positive integer number or zero.

```

functor (T, F, N) :-
    unbound (T) ,
    N1 = N + 1 ,
    {
        N = 0 , ! ,
        T = F
    } ;
    new __stack __vector (T, N1) ,
    first (T, F)
};

```

- Realization of  $\text{arg}(I, T, X)$  in ESP

The following gives almost the same function in ESP:

```
arg(I, T, X) :- integer(I),
               stack__vector(T, A),
               I > 0,
               I < A,
               I1 is I-1
               vector__element(T, I1, X);
```

- Realization of  $X = ..Y$  in ESP

- When  $X$  is unified with an atom

```
'=..' (X, Y) :- bound(X),
               atomic(X), !,
               Y = [X];
```

- When  $X$  is unified with a structure

```
'=..' (X, Y) :- bound(X),
               stack__vector(X, N), !,
               make__list(N, 0, X, Y);
make__list(0, __, __, []) :- !;
make__list(N, K, X, [H | T]) :-
    vector__element(X, K, H),
    make__list(N-1, K+1, X, T);
```

- When  $Y$  is unified with a list

```
'=..' (X, [H | T]) :-
    (
        T = [], atomic(H), !,
        X = H
    );
    make__structure([H | T], 0, X)
);
```



### (3) Meta call

While Prolog has the built-in predicate for a meta call "call(P)", ESP has no corresponding predicate. However, the same function of the meta call can be realized in ESP by using method "refute" (every class implicitly inherits class "class" in which "refute" has been defined.)

The following shows an example of a meta call for goal "p(1,2,X)" or "p(Object,1,2,X)".

Example: In Prolog

```
?- call (p(1, 2, X)).
```

Example: In ESP

```
?- :refute (Object, P, {1, 2, X}).
```

## 6. Program operation

Prolog has two built-in predicates, "assert" and "retract", to store or delete predicates into or from the program data base, but ESP has no corresponding built-in predicates. Prolog can store or delete predicates only in the interpretive codes. Also, how the program is executed when clauses are stored or deleted during program execution depends on a particular Prolog language processor. If you want to have the functions equivalent to "assert" and "retract" in ESP, you need to realize by yourself such program data base managing functions in ESP. In many Prolog language processors, the Prolog interpreter is written in Prolog itself. In ESP, these functions can easily be realized by using the pool functions of SIMPOS and built-in predicates.

## 7. Internal database

Prolog has five built-in predicates to utilize the internal data base: "recorded", "recorda", "recordz", "erae" and "instance", while ESP has no corresponding built-in predicates. However, use of the pool functions of SIMPOS allows you to have more versatile data base managing functions than the Prolog built-in predicates.

## APPENDIX H WHAT IS "OBJECT-ORIENTED"

This appendix describes object-oriented programming, which is a very important concept for understanding ESP language specifications.

Object-oriented programming is a programming technique extended from the conventional structured programming technique, which aims at concise expression and description of problems and solutions. The term "object" means an "entity" or "subject that takes an action". A program can be regarded as a description of a real world including a product of human thought (such as an algorithm) in the form of an image mapped into the computer. With an object-oriented language, you disassemble an image to be mapped into the components, define these components as self-organizing objects, and write a program in the format that these objects communicate with each other by passing messages. In the object-oriented language, data and a procedure are dealt with as one entity (which are separate entities in a conventional procedural language) and a standard external interface (called message passing) is set up.

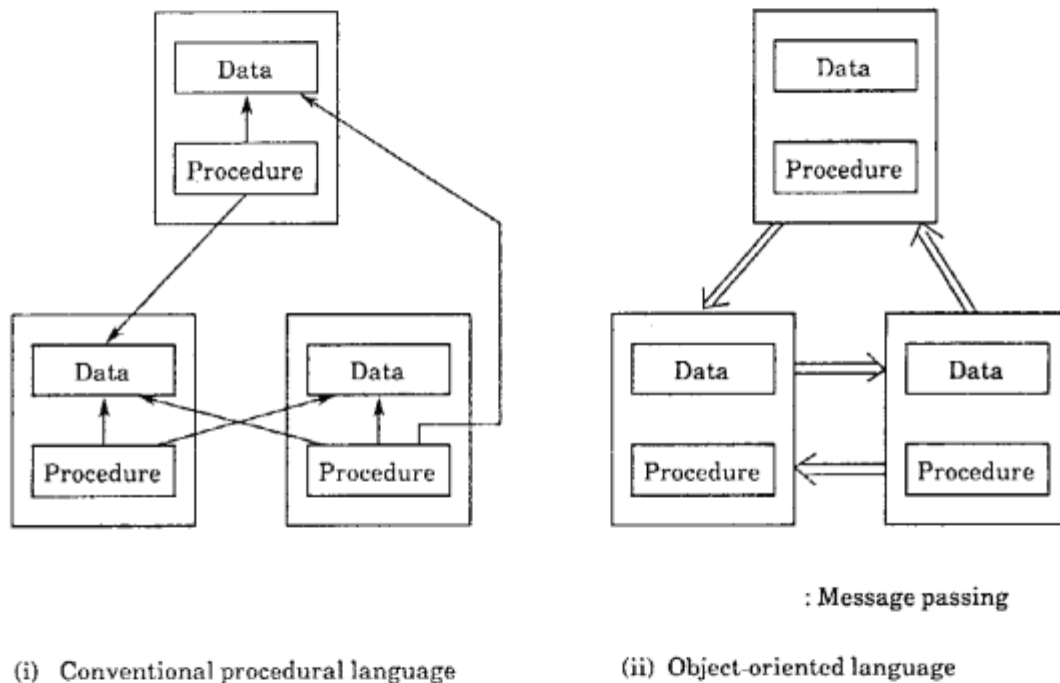


Fig. H-1 Difference in Inter-module Interface between Conventional Language and Object-oriented Language

For example, a window to be displayed on a bit-mapped display is realized as an object. According to instructions from the program or the window manipulator, the window can display characters or graphics and change their position, size or display state. With the conventional programming technique, generally a window is expressed as a set of data, and the window is manipulated by changing this data through an external procedure. On the other hand, in the object-oriented language, a window can be realized as one entity consisting of the data and the procedure for manipulating the window. When you give the window an instruction (namely, a message) to "display a character" or to "move to a specified position", the window executes the requested instruction by evoking a procedure corresponding to that message provided in the window itself.

As described above, an "object" can be regarded as a kind of personification: you describe in a program a lifeless object that originally cannot understand meanings as an active subject that can understand a language and move the body. Simialr to the technique used in animation where you can personify any objects, this technique can be applied to any objects in a program. Let's take an example of a door. A door can be opened, closed or locked. To realize a door as an object, you may build the procedures for these actions into the object called "door" so that the object can respond to a message given from the outside such as "open", "cose" or "lock".

In human society, objects called human beings use spoken or written language as a communication means to exchange thoughts and requests between them, by which an organization such as a government, society, or a social club can be managed in an orderly manner. Object-oriented programming corresponds to the composition of an organization in human society. First, you analyze the tasks needed to realize the desired function, establish an office organization, and associate with each other the individual positions set up according to the office organization to build a unified organization. People are assigned to the respective positions, then the organization begins its service. Suppose the organization is a TV factory. When a request "Produce 1000 TV sets" is given from the outside, the individuals of the organization execute their own roles self-organically such as exchanging messages (instructions) between each other, so as to finally produce 1000 TV sets.

As described above, since object-oriented programming allows exact description of objects, you may have various kinds of advantages from it. These advantages are as follows:

(1) High descriptiveness

A real world can be described directly and definitely in a program: you can easily write a program and the program is readable.

(2) High maintainability and reutilization of a program by information hiding

An object is a package containing the data and procedure of a program and it can be handled like a black box. This allows localization of program modification. Since the access interface to an object is made standard, the modularity (independency) of an object can be made high, allowing the program to be easily used for other applications.

(3) High programming efficiency by the inheritance function

Packaging data and procedures into an object allows use of the efficient programming function called inheritance. Only by modifying or adding new functions to an existing program through the inheritance function, can you make a new program. You can aso combine existing programs in to a new program through the inheritance function.

The following explains the way to write a program in the object-oriented programming.

As described above, since a program is constructed as a set of objects, you may only describe individual objects. However, you may often need several objects of the same attribute. For example, supposing you make a program to simulate the TV factory organization, you usually need multiple production line workers who do the same job at the same section. As for a window on a bit mapped display, you may want to have a theoretically infinite number of windows having the same attribute. In these cases, it is a waste of time to write a similar program many times in every object. To cope with this problem, you can describe all those objects that have the same attribute and behavior as a single entity called a class. A class defines the type (template) of objects that belong to that class, which is treated as a programming unit. In other words, a program is described as a set of classes.

Execution of a program begins with generating the necessary number (one or more) of objects (called instance objects) from each class. (You do not need to prepare all of these objects before program execution. They only have to be generated when they become necessary.) After this, program execution proceeds further as the generated objects are exchanging messages with each other. (See Figure H-2). If you define a class itself as an object, this gives various kinds of advantages. A class regarded as an object is called a class object. Receiving a message asking generation of an instance, a class object generates an instance object.

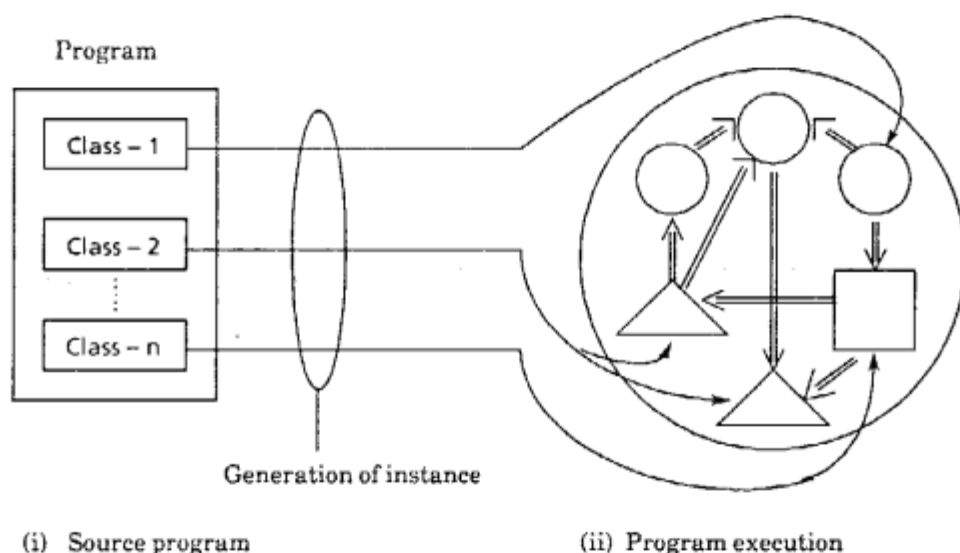


Fig. H-2 Source program and Program Execution in Object-oriented Language

The relationship between a class (class object) and an instance (instance object) may be explained with the following examples: a class is the negative of a film while an instance is one of the positives that can be produced from the negative; or a class is a mold while an instance is a molding produced with the mold; or a class is the mask pattern of an integrated circuit while an instance is a particular IC made with the mask pattern.

When a program is compared to a device consisting of logical circuits, a class may be regarded as the IC mask pattern and an instance is an IC. You may generate the necessary number of ICs from each mask pattern and combine these ICs into one device. Electric signals passing between the ICs may be regarded as messages. This analogy is summarized as follows:

(Object-oriented program)	(Logical circuit device)
Class objects	Mask patterns
Instance objects	ICs
Messages	Electric signals
Program	Device

One of the features that an object-oriented program has is that the program has a dynamic feature: the time a class generates instance objects is when the program is actually executed. This seems intricate apparently. However, this mechanism makes it possible to give the program the powerful descriptiveness.

The following explains the inheritance function. The mechanism to generate multiple objects from one class eliminates the waste of time to write all of those objects that have the same attribute. The inheritance function is a mechanism to more simplify the description of a class definition itself. That is, for definition of different classes, the common part of these different classes is extracted as one independent class. This may be explained by an analogy with the following example of algebra:  $ab+ac$  may be expressed as  $a(b+c)$  by factoring out common factor  $a$ . This seems like a subroutine in a procedural language, but is different in essence. A subroutine is prepared as a physically independent program and called at run time. On the other hand, the inheritance is a function to incorporate a defined class into the self class, thus it becomes one entity even in terms of the physical aspect. (Upon inheritance, the class is not merely copied but is incorporated in such a way that the functions are synthesized or replaced according to the prescribed rules.)

Let's take an example of describing animals for further explanation.

Suppose you make a program where you define a canary, a penguin and a German shepherd as separate classes. One way of describing is to describe their attributes in the respective classes. However, if you factor out the common attribute, you can reduce the describing task. With this example, since a canary and a penguin are both birds, you define the common attributes among birds as a separate class "bird". When you make a class definition for a canary, you can use the definition of class "bird" and add to it only the attributes particular to a canary (such as "yellow" and "sings beautifully"). When expanding this concept to a German shepherd, a German shepherd is a mammal. Since a bird and a mammal are both animals, you define the attributes common to animals as a separate class "animal". Then, for definition of class "bird" and class "mammal", you can use the definition of class "animal" and you only have to add the particular attributes other than the attributes of an animal. You can also modify the attributes already defined. For example, a bird can fly but a penguin cannot. Therefore, when you describe the attributes of a penguin, you may change the attribute "can fly" among those attributes inherited from class "bird" to attribute "cannot fly". (How to represent the knowledge "a bird can fly but a penguin cannot" was one of the important themes in the artificial intelligence field. However, it can be represented by using the inheritance function. Historically, the idea of inheritance was developed as a means for knowledge representation and is realized in an object-oriented language in the format as seen today.) As described above, the function by which you can reutilize the contents of an existing defined class is called the inheritance function.

There are two usages of inheritance:

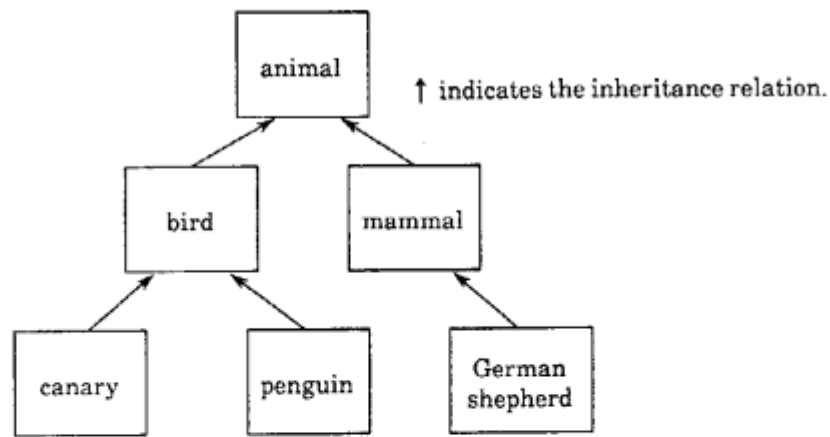


Fig. H-3 Class Definition by Inheritance (Single Inheritance)

- (i) Single inheritance
- (ii) Multiple inheritance

Single inheritance means to inherit only one defined class during a class definition. The above example is an example of single inheritance. As seen in this example, single inheritance is a technique suitable when you define a complex class from a simple (multi-purpose) class. On the other hand, multiple inheritance means to inherit multiple defined classes during a class definition. This technique is used in a case where you define several attribute classes and assemble them into one product.

As described above, single inheritance and multiple inheritance are different beyond the difference in the number of classes to be inherited. You may use either of them case by case. Use of the inheritance function allows concise and simple description of complicated logic. In a procedural language such as Pascal or C, you can use a data inheritance function for definition of a data structure. For example, when you have defined the data type A having a complex structure, you can reuse it as a partial structure of the data type B which has more complex data structure. However, you cannot reuse the procedure that manipulates the data type. In an object-oriented language, data and procedures are packaged into one entity, and it is this mechanism that makes it possible to realize the inheritance function.

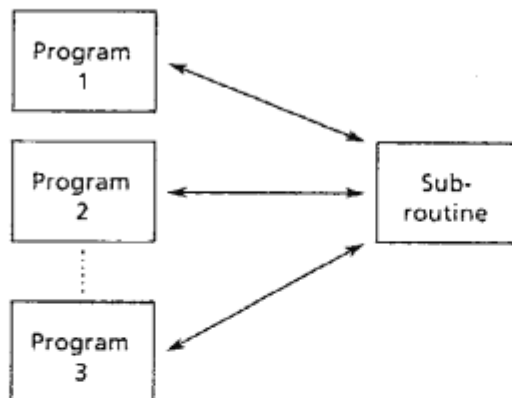
(SUPPLEMENT) Difference between an object and a subroutine in terms of programming

An object is similar to a subroutine of a procedural language. This section describes the difference between them from the viewpoint of programming to help you further understand an object.

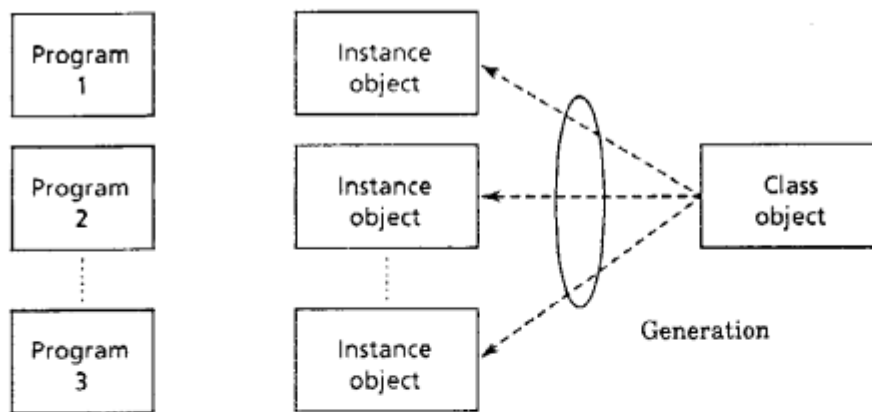
In a procedural language, it is possible to program an object-like subroutine to some extent. For example, when you use function subroutines of FORTRAN, you can receive the results of the calculation for a given input value by writing the subroutine program in a special format. You can modify the internal data structure and the use algorithm of the function subroutine unless you change the external interface.

However, an object and a subroutine have the following definite differences:

- (1) A subroutine is often realized in such a format that it does not maintain the internal state (i.e., the side effects are not left in the subroutine.) For this, calling a subroutine with the same input parameters always causes the same value to be returned. The function subroutines of FORTRAN are typical example. On the other hand, usually an object is used in such a format that the internal state is maintained in it. The internal state is maintained by a slot. (For example, when class "window" generates multiple instance objects, each instance object maintains its window size, position in the screen and window name in the slot provided in the object.) Since the internal state is maintained, sending the same message to the same object usually causes a value different from the previous time to be returned. However, it is possible to make a subroutine that maintains the internal state or an object that does not maintain the internal state, and in fact such a subroutine or an object is often used. That is, there is an essential difference in usage between a subroutine and an object. (See Figure H-4). A subroutine is basically shared as one physical entity, which saves memory capacity. Objects are usually generated as many times as necessary and assigned one by one. This mechanism uses more memory capacity, though, since you do not share an object, you can make it maintain the internal state, greatly improving the flexibility of programming.



(i) Use format of subroutine



(ii) Use format of object

Fig. H-4 Difference in Usage between Subroutine and Object

- (2) A subroutine call in a procedural language is accomplished based on static linkage. that is, the name of a subroutine is explicitly specified in the source program and the linkage to the subroutine is solved by te linkage editor before program execution. On the other hand, an object call is accomplished based on a dynamic linkage. Which object is called is generally determined at run time and thus linkage is also performed dynamically.
- (3) Generally an object has more than one function (method) and which function is used is specified by the argument (method name). On the other hand, which subroutine to be used usually determines which function to be used. (However, it is possible to make an object-like subroutine. But the specifications of the language do not originally provide that function and such a subroutine is made on the programmer's responsibility.)
- (4) Related to (3), though, the format of a method call in an object has been standardized by the language specifications. No other format for information passing exists (you cannot use common variables for example.) Becuase of this, an object can be treated as a black box. On the other hand, a subroutine of a procedural language could be treated as a black box. However, you must make a subroutine in that way on your own responsibility.
- (5) You can dynamically generate as many objects as necessary based on the class definition. With a subroutine, you cannot generally do it.