ICOT Technical Report: TR-044

TR-044

ESP Reference Manual

by
Takashi Chikayama

February, 1984

ICOT

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

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

ESP is a logic programming language based on **PROLOG** and is intended to be used on sequential inference machines currently being developed at ICOT research center. ESP is designed for ease of writing systems software including the operating system of the sequential inference machine itself.

ESP is compiled into **KL0**, the machine language of the sequential inference machine. KL0 is a PROLOG-like language with several extensions. Various features of KL0 are almost directly available in ESP. They include:

- **Unification**, as the basic parameter passing mechanism,
- **Backtracking**, as the basic control structure, and
- **Various Built-in Predicates.**

Main features of the ESP language, aside from those provided by KL0, are:

- **Object Oriented Calling Mechanism**,
- **Class and Inheritance Mechanism**, and
- **Macro Expansion Mechanism.**

This manual describes language features of ESP except the features of KL0.

CHAPTER I

INTRODUCTION

I.1 Language Summary

ESP is a logic programming language based on **PROLOG** and is intended to be used on sequential inference machines currently being developed at ICOT research center [Uchida 83]. ESP is designed for ease of writing systems software including the operating system of the sequential inference machine itself [Hattori 83].

ESP is compiled into **KLO**, the machine language of the sequential inference machine [Chikayama 84]. KLO is a PROLOG-like language with several extensions. Various features of KLO are almost directly available in ESP. They include:

- **Unification**, as the basic parameter passing mechanism,
- **Backtracking**, as the basic control structure, and
- **Various Built-in Predicates.**

Main features of the ESP language, aside from those provided by KLO, are:

- **Object Oriented Calling Mechanism**,
- **Class and Inheritance Mechanism**, and
- **Macro Expansion Mechanism.**

This manual describes language features of ESP except the features of KLO. For about KLO, a summary of the KLO built-in predicates is given in an appendix. For more details, see [Chikayama 84].

I.1.1 Object

An **object** in ESP represents an axiom set, which is basically the same concept as "worlds" in some PROLOG systems [Caneghem 82]. The same predicate call may have different semantics when applied in different axiom sets. The axiom set to be used in a certain call can be specified by giving an object as the first argument of a call and preceding the call with a colon (:) to notify the set that the semantics of the call should depend on the first argument.

An object may have **slots** each of which has its name and its value. Such values can be examined by certain predicate using their names, i.e., the slot values defines a part of the axiom set. The slot values can also be changed by certain predicate calls. This corresponds to altering the axiom set represented by the object. This mechanism is similar to "assert" and "retract" of DEC-10 PROLOG. The difference is that only the slot values can be changed in ESP, while any axioms can be altered in DEC-10 PROLOG.

I.1.2 Class and Inheritance

An **object class**, or simply a **class**, defines the characteristics common in a group of similar objects, i.e., objects which differ only by their slot values (only values; slot names are common in the objects of the same class). An object belonging to a class is said to be an **instance** of that class. A class itself is an object which represents a certain axiom set.

A **class definition** consists of a **nature definition**, **slot definitions** and **clause definitions**. There are two kinds of slots and clauses. Slots and clauses for the class itself and those for the instances of the class. The former ones are called **class slots** and **class clauses**, while latter ones are called **instance slots** and **instance clauses**, respectively.

The **nature definition** defines the **inheritance** relationship concerning the class. Multiple inheritance mechanism similar to that of the **Flavor** system [Weinreb 81], rather than the single inheritance seen in **Smalltalk-80** [Goldberg 83], is provided in ESP.

The most remarkable feature of the inheritance mechanism of ESP is methods are consisting of a certain AND-OR combination of predicates defined in inherited classes. By this inheritance mechanism, classes form a network of "IS-A" and, with the aid of **slots**, "PART-OF" hierarchy can also be formed.

Clause definitions are used for defining a PROLOG-like clause. In a declarative point of view, a clause expresses an axiom in a form of a Horn clause. In a procedural point of view, a clause specifies procedural steps to be taken when a predicate is called. Like slots, there are **class clauses** and **instance clauses**. There also are **local clauses** which define non-object-oriented local predicates.

Clauses for methods are further classified into three types, namely, **before demon clauses, primary clauses** and **after demon clauses.** Before demon clauses with the head having the same functor and same arity form a **before demon predicate**; primary clauses and local clauses form a **primary predicate,** and after demon clauses form a **after demon predicate,** respectively.

Predicates are not called directly, except local predicates. Predicates are called through **methods.** A method is a certain AND-OR combination of these predicates defined in a class or its super classes.

I.1.3 Macro Expansion

Macros are for writing meta-programs which specify that programs with so and so structure should be translated into such and such programs. Macros can be defined in a form of an ESP program, fully utilizing the pattern matching and logical inference capability of logic programming languages.

A macro invocation in ESP is not only expanded to the specified pattern at the place where it is given, but may also affect the program around it, by inserting certain goals specified in the macro definition. This gives a very flexible ability to the macro expansion mechanism of ESP.

I.2 Syntax Description

A somewhat extended BNF is employed in this manual for description of the language syntax. Extensions are:

- **"X"** indicates a **terminal symbol** X. Within X, consecutive two double-quotes mean one double-quote, i.e., """" means a terminal symbol consisting of a single double-quote;
- **{ X }** indicates arbitrarily many (including zero) repeated appearances of X;
- **[X]** indicates X or void, i.e., X is optional.

CHAPTER II

LEXICAL STRUCTURE

This chapter describes the lexical structure of ESP programs.

II.1 Characters

Characters used in ESP programs are categorized as follows:

SYNTAX

```
<lowercase letter> ::=
    <kanji 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"

<uppercase letter> ::=
    "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> ::=
      " " | <line separator> | <tabulation code>

<delimiting character> ::=
    "(" | ")" | "{" | "}" | "[" | "]"
  | "," | ";" | """ | "'" | "|"
```

II.2 Lexical Element

ESP programs consist of **lexical elements**. They are names, numbers, character strings, variables, and delimiters.

SYNTAX

```
<lexical element> ::=
      <name>
    | <number>
    | <character string>
    | <variable>
    | <delimiter>
```

Examples of lexical elements:

```
icot      123      "abc"      X      }
```

Each adjacent lexical elements must be separated by formatting characters or comments when otherwise ambiguous. An arbitrary number of formatting characters or comments can be inserted between two lexical elements without changing the meaning of the program, unless explicitly stated otherwise in this manual.

A **comment** starts with a percent character ("%") and is terminated by a line separator. Any characters except a line separator can appear in a comment (including percent characters).

Example of a comment:

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

No formatting characters are allowed inside a lexical element except as a name string character or a string character, appearing in quoted character strings (see below). Comments also are not allowed inside a lexical element.

II.2.1 Name

Some entities in ESP programs are accessed using their **names.**

SYNTAX

```
<name> ::=
      <lowercase letter> { <letter> | <digit> | "_" }
    | <special character> { <special character> }
    | <quoted name>

<quoted name> ::=
      "'" { <name string character> } "'"
```

```
<name string character> ::=
      <any character except apostrophes> | "''"
```

Examples of names:

```
icot        name_with_underscore        'A Quoted Name'
```

Names are associated with a character string called name string, which consists of characters used in the name. For quoted names, apostrophes in both sides are omitted and any duplicated apostrophes inside are contracted into one.

II.2.2 Number

There are two types of numbers in ESP, namely, **integer numbers** and **floating-point numbers.**

SYNTAX

```
<number> ::= <integer number> | <floating-point number>

<integer number> ::= <digit sequence>

<floating-point number> ::=
      <digit sequence> "." <digit sequence>
      [ <exponent part> ]

<digit sequence> ::= <digit> { <digit> }

<exponent part> ::=
      "^" [ "+" | "-" ] <digit sequence>
```

Examples of numbers:

```
1984        3.14159265        51.2^8
```

Note that only positive numbers are provided as a lexical element. Negative numbers can be represented by complex structures consisting of a prefix operator "-" and a positive number.

II.2.3 Character String

There are three kinds of strings in ESP. They have 1-bit, 8-bit and 16-bit elements and are called bit strings, byte strings and double-byte strings respectively. Only double-byte strings have a **character string** format in lexical element level to denote their values. Strings of other types can be denoted using a vector-like compound syntax (see below).

SYNTAX

```
<character string> ::=
      """" { <string character> } """"

<string character> ::=
      <any character except double-quotes>
    | """" """"
```

Examples of character strings:

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

Double-quote characters must be repeated twice in a character string. They are interpreted as a single double-quote character.

II.2.4 Variable

SYNTAX

```
<variable> ::=
      <variable beginning character>
      { <variable trailing character> }

<variable beginning character> ::=
      <uppercase letter> | "_"

<variable trailing character> ::=
      <lowercase letter>
    | <uppercase letter>
    | <digit>
    | "_"
```

Examples of variables:

```
X       P3C       _a_variable
```

All the occurrences of variables denoted by the same character string appearing within a macro definition, a clause definition or a slot definition item designates the same logical variable. The only exception is that occurrences of variables denoted by only one underline character are considered to be designating independent logical variables. Such variables are called anonymous variables.

II.2.5 Delimiter

Delimiters consist of a single delimiting character and are used for various purposes described below.

SYNTAX

<delimiter> ::= <delimiting character>

CHAPTER III

SYNTACTICAL STRUCTURE

A program of ESP consists of one or more definitions, each being represented as a **term**. A term is constructed using one or more lexical elements according to the syntax rules given in this chapter.

III.1 Term

Basically, the syntax rules of ESP are those of an operator precedence grammar augmented with special notations for vectors, strings, compound terms and linear linked list structures.

SYNTAX

```
<term> ::=
      "(" <term> ")"
    | <variable>
    | <atomic literal>
    | <compound literal>
    | <class object>

<compound literal> ::=
      <vector>
    | <string>
    | <compound term>
    | <operator application>
    | <list>
```

Examples of terms:

a N f(X, Y) {a, b, c} X+Y (3*X+Y)

III.2 Atomic Literal

An **atomic literal** is either a name representing a symbol or a number.

SYNTAX

```
<atomic literal> ::= <symbol> | <number>

<symbol> ::= <name>
```

Examples of atomic literals:

```
icot      1984      3.14159
```

An atomic literal itself is sometimes called the **principal functor** of itself. The **arity** of an atomic literal is 0.

III.3 Vector

The **vector notation** is one of the most basic notations for compound literals. Another basic notation is that for strings. Others are merely shorthand notations of this vector notation.

SYNTAX

```
<vector> ::= <null vector> | <non-null vector>

<null vector> ::= "{}"

<non-null vector> ::= "{" <term list> "}"

<term list> ::= <term> { "," <term> }
```

Examples of vectors:

```
{}      {1, 2, 3}
```

Elements of a non-null vector written in this syntax are the terms which appear in the term list. Their indices begin with zero and go up to the number of the elements minus one, from left to right.

The first (index 0) element of a non-null vector is called the **principal element** of the vector. When the principal element of a vector is a symbol, it is sometimes called the **principal functor** of the vector. When the principal element is a symbol, a vector can also be denoted in a form of a compound term. In such a case, the number of elements of the vector minus 1 is called the **arity** of the vector. Note that an atomic literal is said to be the principal functor of itself, in which case the arity is 0.

III.4 String

There are three kinds of strings in ESP. They have 1-bit, 8-bit and 16-bit integer elements and are called **bit strings**, **byte strings** and **double-byte strings** respectively. Double-byte strings have a character string format to denote their values. Strings of all these three types can be denoted by specifying their components one by one as integer values.

SYNTAX

```
<string> ::=
      <character string>
    | <string type specifier> ":" <string elements>

<string type specifier> ::=
      "bits" | "bytes" | "double_bytes"

<string elements> ::=
      "{}" | "{" <integer list> "}"

<integer list> ::=
      <integer number> { "," <integer number> }
```

Examples of strings:

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

Values allowed as integer numbers appearing in string elements are:

1. 0 or 1, for bit strings,
2. between 0 and 255, for byte strings, and
3. between 0 and 65535, for double-byte strings.

These integer numbers are the elements of the string with their indices zero through the number of elements minus one, from left to right.

III.5 Compound Term

SYNTAX

```
<compound term> ::= <functor> "(" <argument list> ")"

<functor> ::= <symbol>

<argument list> ::= <argument> { "," <argument> }

<argument> ::= <term>
```

Examples of compound terms:

```
f(X)      g(M, N+3)
```

No formatting characters nor comments are allowed between the functor and the left parenthesis of a compound term. This exception to the general rule is for discriminating a prefix operator application from a compound term. A term "**+ (X, Y)**" is not a compound term with two arguments "**X**" and "**Y**", rather, it is an application of a prefix operator "**+**" to a term "**X,Y**".

Compound terms are a shorthand notation for a vector whose principal element is a symbol. The length of the vector is the number of arguments in the argument list of the compound term plus one. Its principal (index 0) element is the functor symbol and following elements (index 1 and up) are the arguments of the compound term. The index 1 element is the first argument, index 2, the second, and so on. For example, a compound term of the form "**a(b,c)**" is interpreted as a vector "**{a, b, c}**", whose principal element is "**a**" and arity is 2.

III.6 Operator Application

SYNTAX

```
<operator application> ::=
      <prefix operator application>
    | <postfix operator application>
    | <infix operator application>

<prefix operator application> ::=
      <prefix operator> <term>

<postfix operator application> ::=
      <term> <postfix operator>
```

```
<infix operator application> ::=
      <term> <infix operator> <term>

<prefix operator> ::= <simple name>

<postfix operator> ::= <name>

<infix operator> ::= <name>
```

Examples of operator applications:

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

Operator applications also are shorthand forms of vector notations. A prefix or a postfix operator application represents a vector with two arguments with its principal (index 0) element being the operator itself, and its second (index 1) element being the operand. An infix operator application represents a vector with three elements with its principal element being the operator itself, the left hand side operand being its second (index 1) element and the right hand side operand being its third (index 2) element.

Operators can arbitrarily be defined by **operator definition banks.** Such operators must be used without any ambiguity. There are two ways to solve such an ambiguity:

(1) Parentheses

"(X + Y) + Z" is interpreted as "+(+(X,Y),Z)" and "X + (Y + Z)" is interpreted as "+(X,+(Y,Z))".

(2) Precedence of operators

When there are two candidate operators to be applied, one which precedes the other is applied first. This precedence relation is affected by whether one operator appears left or right to the other operator. For example, the infix operator "+" precedes "-" when "-" appears to the right of the appearance of "+", but is preceded by "-" in a reverse situation. This makes "X + Y - Z" to be interpreted as "(X + Y) - Z" and "X - Y + Z" as "(X - Y) + Z". Precedence relations can be declared in operator definitions.

III.7 List Structure

SYNTAX

```
<list> ::= <null list> | <non-null list>

<null list> ::= "[]"

<non-null list> ::=
      "[" <term list> [ "|" <term> ] "]"
```

Examples of lists:

```
[]      [a, f(X), 3]      [Head | Tail]
```

The **list notation** also is a shorthand notation of the vector notation. A list notation of the form **"[X | Y]"** is a shorthand of **"{Y, X}"** (note that elements are reversed). A list notation of the form **"[X, ...]"** is a shorthand of **"[X | [...]]"**. A null list denotes a symbol whose print name is **"[]"**. Thus, a list **"[a, b, c | d]"** is a shorthand for **"{ { {d, c}, b}, a}"**.

CHAPTER IV

CLASSES

A class defines characteristics common in a group of similar objects. Here, two objects are called similar when they differ only by their **slot values**. When the characteristics of an object is described in a class, the object is said to be an **instance** of the class.

IV.1 Class Object

A class itself is also an object. Such an object is sometimes called a **class object** to distinguish it from usual instance objects.

Class objects can be written down as a constant using the class names. Only class objects are static objects which have the explicit literal notation and directly accessible. Instance objects are always created dynamically and should be bound to a certain variable or put into a slot of another accessible object, if they ever are of any later use.

SYNTAX

```
<class object> ::= "#" <class name>

<class name> ::= <name>
```

Examples of class objects:

```
#basic_window        #class_with_a_long_name
```

IV.2 Class Definitions

A **class definition** defines slots and predicates of the class itself and those of the instances of the class. **Class definition** is the form of defining executable programs of ESP.

SYNTAX

```
<class definition> ::=
    "class" <class name>
        [ <macro bank declaration> ]
    "has"
        [ <nature definition> ";" ]
        { <class slot definition> ";" }
        { <class clause definition> ";" }
  [ "instance"
        { <instance slot definition> ";" }
        { <instance clause definition> ";" } ]
  [ "local"
        { <local clause definition> ";" } ]
    "end" "."
```

The macro bank declaration declares which macros are to be expanded in the class definition. The nature definition defines what other classes should be inherited by the class. Slot definitions and clause definitions define the **template** of the class: the template of a class defines slots and predicates to be inherited to the classes which inherits the class. It is called a **template** because the class template itself is not enough for making the class object. Especially, methods are only defined by gathering predicates concerned from all the inherited classes. For details, see the following sections.

Example class definitions are given in programming examples in an appendix.

IV.3 Macro Bank Declaration

Macros defined in a macro bank whose name appears in the macro bank declaration can be used in the class definition.

SYNTAX

```
<macro bank declaration> ::=
    "with_macro"
        <macro bank name> { "," <macro bank name> }

<macro bank name> ::= <name>
```

IV.4 Nature Definition

A **nature definition** defines the set and the order of the **inherited classes** of a class.

SYNTAX

```
<nature definition> ::=
    "nature"
        <super class name> { "," <super class name> }

<super class name> ::= <class name> | "*"
```

The **inherited classes** and the **order of the inheritance** are defined by the following rules:

(1) If no nature definition is given in a class definition, the currently defined class itself is the only inherited class.

(2) Otherwise, all the inherited classes of the classes whose names are given in the nature definition are inherited. The order of the inheritance is such that the inherited classes of the class whose name appears first are inherited first. Exceptions to this general rule are:

- o When "*" is given as the super class name, the currently being defined class itself is the inherited class.
- o When a class would be inherited twice by the above rule, then only the first inheritance is valid and the rest are ignored.

Note that, by the rule (2), when a class **a**, say, is inherited by another class **b**, and yet another class **c** has the name of **b** in its nature definition, the class **a** is also inherited by the class **c**, because it is an inherited class of **b**.

The definitions of the inherited classes (except the defined class itself) should be given before the definition of the class inheriting it. This inhibits looped inheritance: when **a** inherits **b**, **b** can neither directly nor indirectly inherit **a**.

IV.5 Clauses, Predicates and Methods

Clauses are defined by clause definitions given in a class definition. A set of clauses with the same clause type and heads with the same functor and arity defines a **predicate.** Predicates may define a **local predicate,** or, with predicates defined in inherited classes, define a **method.**

IV.5.1 Clauses

SYNTAX

```
<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> { "," <goal> }

<goal> ::= <method call> | <predicate call>
```

Heads and goals must be a term with a symbol as its principal functor. Thus, terms such as "p" or "q(a,X)" can be a head or a goal while neither "1", "[a, b, c]" nor "{{a}}" can.

Three kinds of clauses can be defined in a class definition: class clauses, instance clauses, and local clauses. Class clauses define class method which describe the characteristics of the class itself. Instance clauses define instance method which describe the characteristics of the instances of the class. Local clauses define local predicates.

There are three types of clauses for methods: **Before demon clauses** preceded by the clause type identifier **"before", after demon clauses** preceded by **"after"** and **primary clauses** which have no clause type identifiers.

IV.5.2 Predicates

SYNTAX

```
<predicate call> ::= <term>
```

Examples of predicate calls:

```
reconc(X, [], Y)      member(X, L)
```

Predicates are the form of executable programs of ESP. A predicate is defined by a set of clause definitions given in a class definition having the same clause type and having heads with the same principal functor and the same arity.

One class definition, thus, can have up to three different predicates with the same functor and the same arity: a **before demon predicate** consisting of before demon clauses, a **primary predicate** consisting of primary clauses, and an **after demon predicate** consisting of after demon clauses.

Local clauses form a **local predicate** which can only be accessed within the clauses given in the same class definition. Local predicates are the only predicates which can be directly called.

Predicates consisting of class clauses are called **class predicates,** while predicates consisting of instance clauses are called **instance predicates.**

Only calls for local predicates can be directly specified in the program text; other predicates, i.e., class predicates and instance predicates will be parts of the class template and can only be called via **method calls** after constructing a **method** by gathering such predicates from templates of inherited classes. Once called, execution of a **predicate** in ESP is that of PROLOG except for **built-in control forms** specially provided by KLO (See [Chikayama 84]).

A primary class predicate with one argument and the functor name "**new**" is implicitly defined for each class. The program may not explicitly define a class predicate with one argument and the functor name "new". However, demon predicates for "new" may be defined.

IV.5.3 Methods

IV.5.3.1 Demon Combination

A method is identified by the class it is belonging to, its name, arity, and whether it is a class method or an instance method. Methods of a class which is defined by class and instance predicates of all the inherited classes of the class (including the class itself).

A **method** actually consists of an AND combination of the following three:

(1) AND combination of calls of all the **before demon** predicates defined in templates of the inherited classes, in the order of the inheritance.

(2) OR combination of calls of all the **principal** predicates defined in templates of the inherited classes, in the order of the inheritance.

(3) AND combination of calls of all the **after demon** predicates defined in templates of the inherited classes, in the reverse order of the inheritance.

The order of combination is opposite for before and after demons. By this ordering, demon predicates defined in a class will be properly nested.

This combination is sometimes called a **demon combination.** As the same arguments are passed to all of the predicates in a demon combination, various sorts of communication is possible among demons. For example, a before demon can pass some information to the primary predicate by instantiating certain parts of the arguments. It is also possible that the primary predicates should provide several alternative successes, only one of which should be accepted by the after demons.

IV.5.3.2 Method Calls

SYNTAX

```
<method call> ::=
    <normal method call>
  | <class method call>
  | <instance method call>

<normal method call> ::= ":" <term>

<class method call> ::= <class name> ":" <term>

<instance method call> ::= ":" <class name> ":" <term>
```

Examples of method calls:

```
% Normal method calls
     :refresh(Window)
     :output(Window, Character)

% Class method calls
     basic_window:create(Window_class, New_window)
     list_handler:append(Some_package, X, Y, Z)

% Instance method calls
     :basic_window:refresh(Window)
     :as_output_window:output(Window, Character)
```

In normal method calls, the method actually called is determined dynamically depending on the first argument of the call. Thus, methods must have at least one argument.

The first argument of a method call must be an object, i.e., a class object or an instance object of a certain class. It must be an object at the time of the call: it may be a variable which would be instantiated to an object at run time. If the first argument of a normal method call is a class object, a class method of the class is called; if it is an instance of a certain class, an instance method of the class is called. In both cases, the called method is one with the same name and arity as the call.

In class method calls in which a class name is explicitly given in the program, a class method of the specified class is called. In this case, the first argument of the method call must be a class object (either literally or dynamically instantiated to it) and that class must inherit, either directly or indirectly, the class explicitly specified in the call.

In instance method calls, an instance method of the specified class is called. In this case, the first argument of the method call must be an instance object. As an instance object cannot be denoted as a literal constant, the first argument should be a variable in the program text which will be instantiated to an instance object. The class of the instance object must inherit, either directly or indirectly, the class specified in the call.

IV.6 Slots

IV.6.1 Slot Definition

Slot definitions are special forms of method definitions. They give definitions of implicitly defined methods **"get_slot"** and **"set_slot"**.

SYNTAX

```
<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 names> [ <slot initiation> ]
  | "(" <slot names> [ <slot initiation> ]
        ":-" <slot initiation code> ")"

<slot names> ::=
    <slot name>
  | "(" <slot name> { "," <slot name> } ")"

<slot initiation> ::=
    ":=" <term> | "is" <class name>

<slot initiation code> ::= <goal list>
```

There are two types of slots, namely **attribute** slots and **component** slots. Attribute slots can be accessed from wherever in the program, while component slots can only be accessed inside the same class definition, because implicitly defined methods for component slots cannot be accessed outside the class definition.

A class has the following slots:

(1) All the component slots defined in templates of the inherited classes.

(2) Attribute slots defined in templates of the inherited classes. If there exist more than one attribute slots with the same name, only the first one when searched in the inherited order is defined. Other attribute slots with the same name are eliminated.

A **class slot** belongs to the class itself, while an **instance slot** belongs to the instances of the class: each instance has its own instance slots.

A component slot name cannot also be used for an attribute name. For example, when a class component "**x**" is defined, instance attributes of that class cannot have the name "**x**".

Each slot has a value, which is initiated by the value specified in the slot initiation. If the initiation code is specified, it is executed before the slot is initiated. This initiation process is effected as an after demon for the class predicate **new**.

The initiation code for class slot may contain built-in predicates only. This restriction is not applied to initiation of instance slots.

When the slot initiation has the form ":= <term>", the term will be the initial value. Otherwise, i.e., when the slot initiation has the form "**is** <class name>", a new instance of the specified class is created by calling the class predicate "**new**" of the class, and the slot is initiated by that newly created object. This initiation style using "**is**" is only for instance slots and is not available for class slots.

IV.6.2 Slot Access Methods

There are implicitly defined methods named "**get_slot**" and "**set_slot**" both as class methods and instance methods. They are for accessing slots of the class object and instances of the class, respectively.

"Get_slot" examines the value of the slot, while "set_slot" alters the value of the slot. The first argument of these methods must be an object just as in explicitly defined methods; the second argument is the name symbol of the slot; the third argument is the value of the slot: "Get_slot" unifies the third argument with the slot value while "set_slot" sets the third argument to the slot. For a component slot, these method can only be used inside the same class definition. Not even in a class inheriting the class which defines the component.

The method "set_slot" virtually alters the definition of the value getting method, like "**assert**" and "**retract**" in various PROLOG systems, though in a much more restricted manner.

Example of slot access methods:

Assume that the class **a** has a class attributes **c** and an instance attribute **i**. Then the class **a** implicitly defines the following four method clauses:

Class method clauses:

```
get_slot(Class, c, Value),
set_slot(Class, c, Value)
```

Instance method clauses:

```
get_slot(Instance, i, Value),
set_slot(Instance, i, Value)
```

IV.7 Implementing Semantic Networks

A **semantic network** can be easily implemented using the multiple inheritance, object slot, and demon combination mechanisms of ESP. This section briefly describes a standard way of implementing such a network using ESP.

It should be obvious enough that an "IS-A" hierarchy can be implemented using the inheritance mechanism and the demon combination. For example, to represent the fact that sparrows are a bird and birds are an animal, the class **sparrow** should inherit the class **bird**, which, in turn, should inherit the class **animal**. By this way, for a certain method of **sparrow**, all the primary predicates defined in the class **animal**, **bird** and **sparrow** are OR'ed together. In other words, all the axioms applicable to animals and birds are naturally also applicable to sparrows.

A "PART-OF" hierarchy is implemented using this "IS-A" hierarchy and object slots. Assume that we want to make two instances of class **wing** to be parts of an instance of class **bird**. First, the definition of the class **wing** should be given. Then, a class **with_wings** should be defined so that instances of the class **with_wings** holds two instances of **wing** in its slots. Here, a method named **fly** may be defined. Finally, the class **bird** is defined to be a subclass of **with_wings** (and, probably, also a subclass of **animal**); in other words, the class **sparrow** IS-A class of objects **with_wings**.

Note that the multiple inheritance capability of ESP provides a very flexible way of implementing such networks. As the class **with_wings** can be defined separately from the class **bird**, any chimera class, **pegasus** for example, can be quite naturally implemented by inheriting two classes, **horse** and **with_wings**.

CHAPTER V

OPERATORS

Operators are used to denote compound literals in the form of an **operator application.** Operator definitions, which give meta-information for the program are given separately in a form of **operator definition banks.**

V.1 Operator Bank

An **operator bank** defines a set of operators to be used in programs.

SYNTAX

```
<operator bank> ::=
    "operator_bank" <operator bank name> "has"
        { <operator definition> ";" }
    "end" "."
```

V.2 Operator Definition

Three types of operators, namely **prefix**, **postfix** and **infix** operators can be defined (along with their precedence) using operator definitions given in operator banks.

SYNTAX

```
<operator definition> ::=
      <operator kind> <operators>
        { <precedence definition> }

<operator kind> ::=
      "prefix" | "infix" | "postfix"

<precedence definition> ::=
      <precedence relation>
        [ <direction> ] <operators>
```

```
<precedence relation> ::=
      "precedes" | "is_preceded_by"

<direction> ::=
      "left" | "right"

<operators> ::=
      <operator>
    | "(" <operator list> ")"

<operator list> ::= <operator> { "," <operator> }

<operator> ::= <name> | "(" <name> ")"
```

Examples of operator definitions:

```
infix ','
    precedes ';'
    precedes left ','

infix (*, /)
    precedes (+, -)
    precedes right (*, /)
    is_preceded_by ^
```

Note: By the definitions above, commas will be right associative and plus and minus signs will be left associative operators.

When more than one operator are defined in a operator definition, it has the same effect as when the same operator definition for all the operators are given separately.

When more than one operator are defined in a precedence definition, it has the same effect as when the precedence definitions for all the operators are given, including cross terms. Thus, a precedence definition of the form "(Op1, Op2) **precedes right** (Op3, Op4)" is equivalent with four precedence definitions "Op1 precedes right Op3", "Op1 precedes right Op4", "Op2 precedes right Op3" and "Op1 precedes right Op4".

When the direction is omitted in a precedence definition, it has the same effect as when precedence definitions for both directions are given.

V.3 Precedence Relation

Suppose two operators **X** and **Y** are applicable at a certain context if no precedence is taken for account. Precedence relations have effect in such cases: If **X** precedes **Y**, then **X** is applied first; otherwise, if **Y** precedes **X**, then **Y** is applied first; if both are not the case, the operator application is ambiguous and thus erroneous.

The precedence relation is affected by the textual order of the operators concerned. Now, suppose **X** and **Y** appears textually in this order, that is, **X** appears to the left of **Y** and **Y** appears to the right of **X**. In this case, what matters is whether **X** precedes right **Y**, or **Y** precedes left **X**, or neither.

In such a case the precedence of operators **X** and **Y** is determined according to the following rules:

1. **Precedence Definitions**
   If there is a precedence definition for **X** stating that "**X** precedes right **Y**", then **X** precedes right **Y**. Otherwise, if there is a precedence definition for **Y** stating "**Y** is preceded by left **X**", then **X** precedes right **Y**.

2. **Transitivity**
   If **X** precedes right **Z** and **Z** precedes right **Y**, then **X** precedes right **Y**.

Example of the effect of precedence relations

> In "1 + 2 - 3", without precedence relations, it is not sure whether "+" should be applied first or "-" should be applied first. However, according to the standard precedence relations, "+" precedes right "-". Thus, "+" is applied first, and the original term is interpreted as "(1 + 2) - 3".
>
> Note that "1 - 2 + 3" is interpreted as "(1 - 2) + 3", as "-" precedes right "+" in turn. This is an example of the case when the textual order of the operators are important.

To avoid cases where two operators precede each other, a precedence definition resulting in a circular precedence is not permitted: Defining an operator **X** to be preceding another operator **Y** which, according to already given precedence definitions and the transitivity rule, precedes **X**, is an error. The language processor should check out such an error in the operator definitions given in the operator banks whose names are listed in the operator bank declaration in module definitions.

CHAPTER VI

MACROS

Macros are used for defining **meta-programs** which translate one program to another.

VI.1 Macro Bank Definition

A **macro bank definition** defines a set of macro definitions. Macros defined in a macro bank are effective in the class definitions with the macro bank declaration having the name of the macro bank.

SYNTAX

```
<macro bank definition> ::=
    "macro_bank <macro bank name> "has"
        { <macro definition> ";" }
    "end" "."
```

VI.2 Macro Definition

SYNTAX

```
<macro definition> ::=
      <invocation pattern> "=>"
        <expanded pattern> { <runtime condition> }
        ":-" <expansion condition>

<invocation pattern> ::= <term>

<expanded pattern> ::= <term>

<runtime condition> ::=
        "when" <generator>
      | "where" <checker>
```

<generator> ::= <goal list>

<checker> ::= <goal list>

<expansion condition> ::= <body>

The **invocation pattern** and the **expanded pattern** may include logical variables. When one appears in both, they refer to the same entity. A sub-term in an invocation of this macro, which appears at the position corresponding to that variable in the invocation pattern, will be included in the expanded result at the position where that same variable appears in the expanded pattern.

The **runtime conditions** can also be included in the expanded result so as to be executed in runtime, i.e., when the expanded result is executed. The way they are expanded will be described in the following sections.

The **expansion condition** is also a condition associated with the macro expansion. It is executed when a macro invocation is expanded, rather than included in the expansion and executed in runtime. The execution process of the expansion condition is the same as that of the body of a clause, except that when one successful execution is found, no alternatives will be tried later.

For examples of macro definitions, see the pseudo definitions of standard macros given in an appendix.

VI.3 Macro Invocation and Expansion

A **macro invocation** is any term which can be matched with the invocation pattern of a certain macro definition and the execution of its expansion condition terminates successfully. Matching of an invocation pattern and the (possibly) expanded term differs from unification in that, as for matching, the variables in the expanded term is treated as constants.

Invocations appearing in a goal and those in a clause head are expanded in different manners.

VI.3.1 Macro Expansion in a Goal

Macro invocations appearing in a goal in a body, including cases where goals themselves are a macro invocation, are expanded as follows:

(1) The invocation is matched with the invocation pattern of the macro definition.

(2) The **expansion condition** of the macro definition, if any, are executed in the same way as the body of a clause.

(3) The goal including the invocation is replaced by a logical conjunction of the following three parts:

   1) **the generator**,

   2) **the original predicate call with the invocation concerned substituted** by the expansion,

   3) **the checker.**

Example of macro expansion in a goal:

With a macro definition:
```
    X - Y => Z when subtract(X,Y,Z)
```
the clause:
```
    repeat(N) :- repeat(N-1)
```
is expanded into:
```
    repeat(N) :- subtract(N,1,M), repeat(M)
```
where "M" is an arbitrary chosen unique variable name.

VI.3.2 Macro Expansion in a Head

Macro invocations appearing in the **head** of a clause are expanded in a slightly different manner:

(1) The invocation is matched with the invocation pattern of the macro definition (the same as when invoked within a goal).

(2) The **expansion condition**, if any, are executed in the same way as the execution of the body of a clause (same).

(3) The invocation in the head is replaced by the expanded pattern. The body of the clause is replaced by a logical conjunction of the following three in this order:

   1) **the checker**,

   2) **the original body**, and

   3) **the generator.**

Example of macro expansion in a head:

```
With a macro definition:
    X - Y => Z when subtract(X,Y,Z)
the clause:
    sub1(N, N-1)
is expanded into:
    sub1(N, M) :- subtract(N,1,M)
where "M" is an arbitrary chosen unique variable name.
```

VI.3.3 Repeated Macro Expansion

Expanded result of a macro may include another macro invocation as its sub-term which, in turn, will be expanded, including the cases where the expanded pattern itself is another macro invocation.

However, macro expansions are tried only in a top-down manner. Once a term is examined and is not recognized as a macro invocation, it will never be treated as a macro invocation even if a later macro expansion of its sub-terms has made the parent term unifiable with a certain macro invocation pattern.

Macros are expanded when the term containing the invocations is compiled. Even if a pattern looking like a macro invocation is generated during program execution, macro expansion does not take place.

VI.3.4 Controlling Macro Expansion

When a pattern unifiable with a macro invocation pattern should be treated as it is rather than being expanded, it should be quoted using a prefix operator "`". The term prefixed by "`" is never expanded even if it is a macro invocation pattern. The quoting operator "`" is eliminated: "`**X**" means "**X**". The pattern "`**X**" itself can be written as "``**X**".

This quoting is effective only for the top-level of the pattern. Thus, if the quoted term has macro invocations as its sub-terms, they will be expanded as usual.

Example of controlled macro expansion:

```
With a macro definition:
    X - Y => Z when subtract(X,Y,Z)
the clause:
    p(M,N) :- q(`((M-3)-(N-2)))
is expanded into:
    p(M,N) :-
        subtract(M,3,X),
        subtract(M,2,Y),
        q(X-Y)
```

rather than into:

```
p(M,N) :-
    subtract(M,3,X),
    subtract(M,2,Y),
    subtract(X,Y,Z),
    q(Z)
```

where "X" and "Y" are arbitrary chosen unique variable names.

VI.4 Standard Macro Definitions

A set of macros are defined as standard and can be used without any explicit declarations.

VI.4.1 Accessing Object Slots

A notation of the form "**Object!Slotname**" is expanded into a form which refers the specified slot of the object.

A notation of the form "**Object!Slotname:Value**" is expanded into a form which alters the specified slot of the object with the specified value.

VI.4.2 Sharp Macros

The sharp symbol ("**#**") is used in various ways for conveniently denoting various entities.

- A notation of the form **#Class_name** denotes the class object with the specified class name. The class name must be a symbol.
- A notation of the form **#"Character"** denotes the character code of the specified character.
- A notation of the form **Radix#"Digit_sequence"** denotes an integer whose notation based on the specified radix is the digit sequence. The radix should be in the range 2 through 36.

VI.4.3 Functional Notation

Arithmetical and bit-wise logical operators can be used in a functional way. For example, a goal "**p(X+3)**" has almost the same semantics as a goal sequence "**add(X,3,Y), p(Y)**".

Available operators are:

- \+ as a prefix no operation operator and as an infix addition operator.
- \- as a prefix arithmetical complement operator and as an infix subtraction operator.
- * as an infix multiplication operator.
- / as an infix division operator.

More detailed description of standard macros is given in an appendix.

- o **mod** as an infix remainder operator.
- o /\ as an infix bit-wise logical and operator.
- o \/ as an infix bit-wise logical or operator.
- o \ as a prefix bit-wise complement operator.
- o << as an infix left shift operator.
- o >> as an infix right shift operator.

APPENDIX A

KLO BUILT-IN PREDICATES

This appendix gives a summary of KLO built-in predicates. Only the mnemonic names of built-in predicates and their arguments with argument category are given here. For details, see [Chikayama 84].

A.1 Categories of Arguments

Arguments to KLO built-in predicates can be classified into four categories depending on the required instantiation status at the time the built-in predicate is executed. They are:

Input Arguments:

Input arguments must already be instantiated at the time of the execution of the built-in predicate; otherwise, an exception is raised. Input arguments are not specially marked.

Unified Arguments:

Unified arguments are unified with a certain result of computation performed by the instruction. Thus, they may be either instantiated or uninstantiated. In the description of the built-in predicates below, unified arguments are prefixed with "^".

Output Arguments:

Output argument must not be instantiated when a built-in predicate is executed; otherwise, an exception is raised. Output arguments are prefixed with "-".

Arbitrary Arguments:

Arbitrary arguments may be instantiated or uninstantiated when the built-in predicate is executed, and are only checked their status and are never unified with anything. Arbitrary arguments are prefixed with "?".

A.2 Data Type and Attributes

```
      atom(?X)
      integer(?X)
      floating_point(?X)
      heap_vector(?Vector, ^Length)
      stack_vector(?Vector, ^Length)
      code(?Code, ^Length)
      protected_type(?X)
    * protection_key(X, ^Key)
      location(?X)
      string(?String, ^Length, ^Size)
      unbound(?X)

      type(?X, ^Type)
      value(?X, ^Value)

      number(?X)
      atomic(?X)
      structure(?X)
```

A.3 Object Creation

```
      new_atom(~X)
      new_heap_vector(~Vector, Length)
      new_stack_vector(~Vector, Length)
    * new_code(~Code, Length)
      new_string(~String, Length, Size)
      new_protected_object(~Object, Key, Value)
```

A.4 Structure

A.4.1 Element Access

```
      vector_element(Vector, Position, ^Element)
      first(Vector, ^Element)
      second(Vector, ^Element)
      string_element(String, Position, ^Element)
    * code_element(Code, Position, ^Element)
      location_element(Location, ^Element)
      protected_value(Protected, Key, ^Value)

      set_vector_element(Vector, Position, Element)
      set_first(Vector, Element)
      set_second(Vector, Element)
```

) Predicates with their names marked with a "" are privileged predicates. Privileged predicates can only be executed if the instruction is stored in a privileged heap area. See [Chikayama 84] for details.

**) Some of the predicates are not designed to be used by the user directly and are intended to be used by a little more high level macro provided by the language processor. Details are not fixed yet.

```
    set_string_element(String, Position, Element)
*   set_code_element(Code, Position, Element)
    set_location_element(Location, Element)
    set_protected_value(Protected, Key, Value)
```

A.4.2 Substructure and Location

```
subvector(Vector, Position, Length, ^Subvector)
vector_tail(Vector, Position, ^Subvector)
substring(String, Position, Length, ^Substring)
string_tail(String, Position, ^Substring)

set_subvector(Vector, Position, Length, Subvector)
set_substring(String, Position, Length, Substring)

vector_element_location(Vector, Position, ^Location)
first_location(Vector, ^Location)
second_location(Vector, ^Location)
```

A.5 Arithmetics

```
add(X, Y, ^R)
subtract(X, Y, ^R)
multiply(X, Y, ^R)
divide(X, Y, ^R)

divide_with_remainder(X, Y, ^Q, ^R)
increment(X, ^R)
decrement(X, ^R)
minus(X, ^R)

add_extended(X, Y, ^R1, ^R2)
subtract_extended(X, Y, ^R1, ^R2)
multiply_extended(X, Y, ^R1, ^R2)
divide_extended(X1, X2, Y, ^R1, ^R2)
```

A.6 Bit-wise Logical Operation

```
and(X, Y, ^R)
or(X, Y, ^R)
xor(X, Y, ^R)
complement(X, ^R)
shift_left(X, N, ^R)
shift_right(X, N, ^R)

and_string(String, Position, Length, Mask_String)
or_string(String, Position, Length, Mask_String)
xor_string(String, Position, Length, Mask_String)
complement_string(String, Position, Length)
```

A.7 Comparison

```
unify(^X, ^Y)

equal(?X, ?Y)
not_equal(?X, ?Y)
equal_string(String1, String2)
not_equal_string(String1, String2)

identical(?X, ?Y)
not_identical(?X, ?Y)

less_than(?X, ?Y)
not_less_than(?X, ?Y)

hash(X, ^Hash_Code)
```

A.8 Type Conversion

```
integer_to_floating_point(Integer, ^Floating)
floating_point_to_integer(Floating, ^Integer)
vector_to_list(Vector, ^List)
list_to_vector(List, ^Vector)
```

A.9 Low-Level Primitives

```
* wait(Wait_Code)
* restart
* halt(Halt_Code)

  address(?X, ^Address)
  physical_address(Address, ^Physical_Address)
* word(Address, ^Tag, ^Value)
* set_word(Address, Tag, Value)
* physical_word(Address, ^Tag, ^Value)
* set_physical_word(Address, Tag, Value)

* area_flags(Area, ^Flag)
* area_top_pointer(Area, ^Address)
* page_map_base(Area, ^Base)
* area_size(Area, ^Size)
* area_size_limit(Area, ^Limit)
* move_page_map(Area, New_Base)
* set_area_flags(Area, Flag)
* set_area_top_pointer(Area, Address)
* set_page_map_base(Area, Base)
* set_area_size(Area, Size)
* set_area_size_limit(Area, Limit)

* collect_garbage

* process_control_block(Process, PCB)
* set_process_control_block(Process, PCB)
* initialize_process(Process, Start_Code, Stack_Area)
* change_process(Process)
```

```
* interrupt_index(Interrupt, ^Process)
* set_interrupt_index(Interrupt, Process)
* interrupt_code(Interrupt, Interrupt_Code)
* trap(Interrupt, Interrupt_Code)

* register(Register, ^Tag, ^Value)
* set_register(Register, Tag, Value)
* system_register(Register, ^Value)
* set_system_register(Register, Value)
* real_time_clock(^T1, ^T2)
* set_real_time_clock(T1, T2)
```

A.10 Debugging Aid

```
ancestor(N, ^Clause)
frame(N, Stack, ^Frame)
read_console(^X)
display_console(?X)
```

A.11 Program Control

```
true
fail

self
apply(Definition, Arguments)

case(Index, Table_Size)
clause_indexing(Position, Table_Size)

level(^N)
success(Level)
absolute_cut(Level, Size)
relative_cut(Level, Size)
absolute_cut_and_fail(Level)
relative_cut_and_fail(Level)
cut(Size)
cut_and_fail

on_backtrack(Code)

bind_hook(?X, Handler)

exception_hook(Exception, Handler)
raise(Exception, Arguments)
* exception_handler_table(Process, Exception_Table)
* set_exception_handler_table(Process, Exception_Table)
```

A.12 Input/Output

* get_byte(Address, ^Data)
* get_double_byte(Address, ^Data)
* get_double_byte_swapped(Address, ^Data)
* get_vector(Address, Vector)
* get_vector_swapped(Address, Vector)
* get_string(Address, String)
* get_string_swapped(Address, String)
* get_with_tag(I/O_Address, Memory_Address, Length)

* put_byte(Address, Data)
* put_double_byte(Address, Data)
* put_double_byte_swapped(Address, Data)
* put_vector(Address, Vector)
* put_vector_swapped(Address, Vector)
* put_string(Address, String)
* put_string_swapped(Address, String)
* put_with_tag(I/O_Address, Memory_Address, Length)

* bus_reset

APPENDIX B

STANDARD MACROS

This appendix gives the pseudo-definitions of standard macros.

Definitions listed here are intended to help understanding the semantics of standard macros. They may not (and, in some cases, cannot) be implemented in the way listed here. Various subroutines required in a practical implementation, both for expansion-time and run-time, including those for error checking, are also omitted.

B.1 Slot Access

```
Obj!Slot => Value when :get_slot(Obj, Slot, Value);

(Obj!Slot:=Value) => :set_slot(Obj, Slot, Value);
```

B.2 Constants

```
Radix#Digits => N

:- string(Digits, , ),

convert_to_integer(Digits, B, 0, N);

#String => Code

:- string(String, 1, ),

string_element(String, 0, Code);

#Class_name => Class_object

:- get_class_object(Class_name, Class_object);
```

B.3 Arithmetics

```
X + Y => Z when add(X, Y, Z);

X - Y => Z when subtract(X, Y, Z);

X * Y => Z when multiply(X, Y, Z);

X / Y => Z when divide(X, Y, Z);

X mod Y    => Z when divide_with_remainder(X, Y, _, Z);

- X => Z when minus(X, Z);
```

B.4 Bit-wise Logical Operations

```
X /\ Y => Z when and(X, Y, Z);

X \/ Y => Z when or(X, Y, Z);

X << Y => Z when shift_left(X, Y, Z);

X >> Y => Z when shift_right(X, Y, Z);

\(X) => Z when complement(X, Z);
```

APPENDIX C

PROGRAMMING EXAMPLES

Example ESP programs are given in this appendix.

C.1 List Handler

The example in this section shows how the class mechanism can be used for encapsulation.

The class "list_handler" defined in the program below provides a list-processing subroutine package. Class methods of the class "list_handler" are the entries of the package. Implementation details are hidden in local predicates.

```
class list_handler has
    :append(_,X,Y,Z) :-          % Entry for append
        append(X,Y,Z);
    :reverse(_,X,Y) :-           % Entry for reverse
        reconc(X,[],Y);
    ...
local
    append([],X,X);              % Definition of append
    append([W|X],Y,[W|Z]) :-
        append(X,Y,Z);
    reconc([],X,X);              % Reverse is defined
    reconc([W|X],Y,Z) :-         %   by a strange routine
        reconc(X,[W|Y],Z);
    ...
end.
```

C.2 Room with two doors

The example given in this section is intended to show the ability of the inheritance mechanism and the demon mechanism of ESP.

The example program defines a class of rooms with two doors. Each door has a lock. A room can be entered through a door only when it is not locked. One can enter the room through either of the two doors.

In the program below, checking the status of lock is effected by the before demon defined in the class "with_a_lock" mix-in class. This check is automatically made **before** opening any object (a door, in this case) "with_a_lock".

Two doors are introduced to a room by inheriting two classes "with_front_door" and "with_back_door". Each class has an instance method named "make_way". Whichever door already open or at least unlocked when the "make_way" method of the room is called, is used for entering and exiting the room.

```
% Lock
class lock has
instance
    component
        state := unlocked;          % Locked or unlocked
    :locked(Lock) :-                % Is locked?
        Lock!state = locked;
    :lock(Lock) :-                  % Locking
        Lock!state := locked;
    :unlock(Lock) :-                % Unlocking
        Lock!state := unlocked;
end.

% With Lock -- MIXIN
class with_a_lock has
instance
    attribute
        lock is lock;
    before:open(Obj) :-             % Must be unlocked
        :unlocked(Obj!lock);        %   before opened.
    :lock(Obj) :-                   % Locking object is
        :lock(Obj!lock);            %   locking the lock.
    :unlock(Obj) :-                 % Unlocking object is
        :unlock(Obj!lock);          %   unlocking the lock.
    :locked(Obj) :-                 % Object is locked when
        :locked(Obj!locked);        %   the lock is locked.
end.
```

```
% Simple Door
class door has
instance
    component
        state := closed;          % Open or closed
    :closed(Door) :-              % Is closed?
        Door!state = closed;
    :open(Door) :-                % Opening
        Door!state := open;
    :close(Door) :-               % Closing
        Door!state := closed;
    :make_way(Door) :-            % If already open,
        Door!state = open, !;     %   do nothing.
    :make_way(Door) :-            % If not,
        :open(Door);              %   then open it.
end.

% Door with Lock(s)
class door_with_a_lock has
    nature
        door,                     % A door
        with_a_lock;              %   with a lock
end.

% Something with Door  --  MIXIN
class with_front_door has
instance
    attribute
        front_door is door_with_a_lock;
    :open_front(Obj) :-           % Opening front door
        :open(Obj!front_door);
    :close_front(Obj) :-          % Closing front door
        :close(Obj!front_door);
    :lock_front(Obj) :-           % Locking front door
        :lock(Obj!front_door);
    :unlock_front(Obj) :-         % Unlocking front door
        :unlock(Obj!front_door);
    :make_way(Obj) :-             % One can make way
        :make_way(
            Obj!front_door);      %   through the front door
end.
```

```
class with_back_door has
instance
    attribute
        back_door is door_with_a_lock;
    :open_back(Obj) :-              % Opening back door
        :open(Obj!back_door);
    :close_back(Obj) :-             % Closing back door
        :close(Obj!back_door);
    :lock_back(Obj) :-              % Locking back door
        :lock(Obj!back_door);
    :unlock_back(Obj) :-            % Unlocking back door
        :unlock(Obj!back_door);
    :make_way(Obj) :-               % One can make way
        :make_way(
            Obj!back_door);         %   through the back door
end.

% Simple Room
class room has
nature ...;
instance
    ...
    :enter(Room) :-
        :make_way(Room), ...;
    :exit(Room) :-
        :make_way(Room), ...;
    ...
end.

class room_with_two_doors has
    nature
        room,                       % A room
        with_front_door,            %   with front door
        with_back_door;             %   and back door
end.
```

APPENDIX D

COLLECTED SYNTAX

All the syntax rules of ESP are collected in this appendix in an alphabetical order. Use the index to find the page where the syntax rule is given.

```
<argument list> ::= <argument> { "," <argument> }

<argument> ::= <term>

<atomic literal> ::= <symbol> | <number>

<body> ::=
    <goal list>
  | "( <goal list> { ";" <goal list> } ")"

<character string> ::=
      """ { <string character> } """

<checker> ::= <goal list>

<class clause definition> ::= <method clause definition>

<class definition> ::=
    "class" <class name>
        [ <macro bank declaration> ]
    "has"
        [ <nature definition> ";" ]
        { <class slot definition> ";" }
        { <class clause definition> ";" }
  [ "instance"
        { <instance slot definition> ";" }
        { <instance clause definition> ";" } ]
  [ "local"
        { <local clause definition> ";" } ]
    "end" "."

<class method call> ::= <class name> ":" <term>

<class name> ::= <name>

<class object> ::= "#" <class name>

<class slot definition> ::= <slot definition>
```

```
<compound literal> ::=
      <vector>
    | <string>
    | <compound term>
    | <operator application>
    | <list>

<compound term> ::= <functor> "(" <argument list> ")"

<delimiter> ::= <delimiting character>

<delimiting character> ::=
    "(" | ")" | "{" | "}" | "[" | "]"
  | "," | ";" | "*" | "'" | "|"

<demon type> ::= "before" | "after"

<digit sequence> ::= <digit> { <digit> }

<digit> ::=
    "0" | "1" | "2" | "3" | "4"
  | "5" | "6" | "7" | "8" | "9"

<direction> ::=
      "left" | "right"

<expanded pattern> ::= <term>

<expansion condition> ::= <body>

<exponent part> ::=
      "^" [ "+" | "-" ] <digit sequence>

<floating-point number> ::=
      <digit sequence> "." <digit sequence>
      [ <exponent part> ]

<formatting character> ::=
      " " | <line separator> | <tabulation code>

<functor> ::= <symbol>

<generator> ::= <goal list>

<goal list> ::= <goal> { "," <goal> }

<goal> ::= <method call> | <predicate call>

<head> ::= <term>

<infix operator application> ::=
      <term> <infix operator> <term>

<infix operator> ::= <name>

<instance clause definition> ::= <method clause definition>
```

```
<instance method call> ::= ":" <class name> ":" <term>

<instance slot definition> ::= <slot definition>

<integer list> ::=
      <integer number> { "," <integer number> }

<integer number> ::= <digit sequence>

<invocation pattern> ::= <term>

<lexical element> ::=
      <name>
    | <number>
    | <character string>
    | <variable>
    | <delimiter>

<list> ::= <null list> | <non-null list>

<local clause definition> ::= <clause definition>

<lowercase letter> ::=
    <kanji 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"

<macro bank declaration> ::=
    "with_macro"
        <macro bank name> { "," <macro bank name> }

<macro bank definition> ::=
    "macro_bank <macro bank name> "has"
        { <macro definition> ";" }
    "end" "."

<macro bank name> ::= <name>

<macro definition> ::=
      <invocation pattern> "=>"
        <expanded pattern> { <runtime condition> }
        ":-" <expansion condition>

<method call> ::=
    <normal method call>
  | <class method call>
  | <instance method call>

<method clause definition> ::=
    [<demon type>] ":" <clause definition>

<clause definition> ::=  <head> [ ":-" <body> ]

<name string character> ::=
      <any character except apostrophes> | "''"
```

```
<name> ::=
      <lowercase letter> { <letter> | <digit> | "_" }
    | <special character> { <special character> }
    | <quoted name>

<nature definition> ::=
    "nature"
        <super class name> { "," <super class name> }

<non-null list> ::=
      "[" <term list> [ "|" <term> ] "]"

<non-null vector> ::= "{" <term list> "}"

<normal method call> ::= ":" <term>

<null list> ::= "[]"

<null vector> ::= "{}"

<number> ::= <integer number> | <floating-point number>

<operator application> ::=
      <prefix operator application>
    | <postfix operator application>
    | <infix operator application>

<operator bank> ::=
    "operator_bank" <operator bank name> "has"
        { <operator definition> ";" }
    "end" "."

<operator definition> ::=
      <operator kind> <operators>
        { <precedence definition> }

<operator kind> ::=
      "prefix" | "infix" | "postfix"

<operator list> ::= <operator> { "," <operator> }

<operators> ::=
      <operator>
    | "(" <operator list> ")"

<operator> ::= <name> | "(" <name> ")"

<postfix operator application> ::=
      <term> <postfix operator>

<postfix operator> ::= <name>

<precedence definition> ::=
      <precedence relation>
        [ <direction> ] <operators>
```

```
<precedence relation> ::=
      "precedes" | "is_preceded_by"

<predicate call> ::= <term>

<prefix operator application> ::=
      <prefix operator> <term>

<prefix operator> ::= <simple name>

<quoted name> ::=
      "'" { <name string character> } "'"

<runtime condition> ::=
        "when" <generator>
      | "where" <checker>

<slot definition item> ::=
    <slot names> [ <slot initiation> ]
  | "(" <slot names> [ <slot initiation> ]
        ":-" <slot initiation code> ")"

<slot definition> ::=
    <slot type> <slot definition item>
        { "," <slot definition item> }

<slot initiation code> ::= <goal list>

<slot initiation> ::=
    ":=" <term> | "is" <class name>

<slot names> ::=
    <slot name>
  | "(" <slot name> { "," <slot name> } ")"

<slot type> ::= attribute | component

<special character> ::=
    "!" | "@" | "#" | "$" | "^" | "&" | "*" | "+"
  | "-" | "=" | "~" | "`" | "?" | "/" | "\" | "."
  | "<" | ">"

<string character> ::= .
      <any character except double-quotes>
    | """" ""

<string elements> ::=
      "{}" | "{" <integer list> "}"

<string type specifier> ::=
      "bits" | "bytes" | "double_bytes"

<string> ::=
      <character string>
    | <string type specifier> ":" <string elements>

<super class name> ::= <class name> | "*"
```

```
<symbol> ::= <name>

<term list> ::= <term> { "," <term> }

<term> ::=
      "(" <term> ")"
    | <variable>
    | <atomic literal>
    | <compound literal>
    | <class object>

<uppercase letter> ::=
    "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"

<variable beginning character> ::=
      <uppercase letter> | "_"

<variable trailing character> ::=
      <lowercase letter>
    | <uppercase letter>
    | <digit>
    | "_"

<variable> ::=
      <variable beginning character>
      { <variable trailing character> }

<vector> ::= <null vector> | <non-null vector>
```

APPENDIX E

REFERENCES

[Caneghem 82] M.Van Caneghem, PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille (1982).

[Chikayama 84] T.Chikayama: KL0 Reference Manual, to appear as ICOT Technical Report, Institute for New Generation Computer Technology.

[Goldberg 83] A.Goldberg, D.Robson, SMALLTALK-80 -- The Language and its Implementation --, Xerox Palo Alto Research Center (1983).

[Hattori 83] T.Hattori and T.Yokoi: Basic Constructs of the SIM Operating System, New Generation Computing, Vol.1, No.1, Ohmsha and Springer-Verlag (1983).

[Uchida 83] S.Uchida, M.Yokota, A.Yamamoto, K.Taki and H.Nishikawa: Outline of the Personal Sequential Inference Machine: PSI, New Generation Computing, Vol.1, No.1, Ohmsha and Springer-Verlag (1983).

[Weinreb 81] D.Weinreb, D.Moon, Lisp Machine Manual, 4th ed., Symbolics, Inc. (1981).

INDEX