# LPS: A Rule-based, Schema-oriented Knowledge Representation System

YUICHIRO ANZAI,* YUICHI MITSUYA,*
SHINYA NAKAJIMA* and SHOJI URA*

A new knowledge representation system called LPS is presented. The global control structure of LPS is rule-based, but the local representational structure is schema-oriented. The present version of LPS was designed to increase the understandability of representation while keeping time efficiency reasonable. Pattern matching through slot-networks and meta-actions from among the implemented facilities of LPS, are especially described in detail.

## 1. Introduction

Efficiency, understandability and flexibility of an artificial intelligence system depend largely on the design of knowledge representation. It is just the reason why many different kinds of representation schemes have been presented so far: frames [1], production rules [2], and semantic networks [3] are well-known examples. Each of them is oriented towards some specific aspect of knowledge pieces and their organizations. For example, a production system is appropriate for knowledge that can be decomposed into piecewise rules, and a frame-based system properly represents interrelated, structured knowledge.

That a system is competent for one aspect of knowledge does not mean its adequateness for all aspects. Squeezing all aspects into one representation often leads to forfeiture of simple control structure, understandability, or flexibility of representation, each of which is relevant to any general-purpose knowledge representation system. This paper describes LPS, a knowledge representation system designed based on this lesson: programs and control of our system should be comrehensible and flexible enough from the user's viewpoint, while maintaining reasonable efficacy.

On one hand, LPS's global control is carried out by the rule-based recognize-act cycle. In this regard, the global control structure is related to production system architecture [4], especially OPS [5]. To avoid redundancy, we do not repeat here what LPS and OPS commonly include, and suppose that OPS and general production system architecture are familiar to the reader. On the other hand, LPS's basic representation inherits general feature of frames. We also assume that the reader is familiar with general frame systems [6]. Consequently, what was originally in LPS is how the characteristics of production systems and frame systems can be integrated into another comprehensible and flexible representation system. LPS has been applied by our group to knowledge-based problem solving [7], natural language understanding [8], and simulation of human cognitive processes. But this paper is not concerned with these applied topics, and concentrates on description of the original ideas embedded in the LPS system.

## 2. A Glimpse of LPS

Each piece of knowledge in LPS is represented usually as a frame, or a *schema*, but avoids complexity in slots by forbidding writing a procedure in a slot of a schema. Instead, we notice that a procedure can be represented as a production rule, i.e., a pair of the condition and action parts, and each condition or action can be a knowledge unit in a schema as exemplified below.

Fig. 1 shows simple examples of LPS schemata. The first schema in Fig. 1 is one for the concept "marry." Its first element, MARRY, represents the *schema-name*, whereas the atoms, ISA, AKINDOF, #OBJ1, #OBJ2, #C1, #C2 and ACT, are called *slotnames*. To each slotname corresponds a list called the *slotvalue* except

```
1-a.
      (MARRY ISA (MOVEMENT)
             AKINDOF (HUMAN.BEHAVIOR)
             #OBJ1 (!ISA MAN $M)
             #OBJ2 (!ISA WOMAN $W)
             #C1 (LOVE $M $W)
             #C2 (LOVE $W $M)
             ACT (MARRY $M $W))

1-b.
      (TAROW ISA (MAN)
             AGE (22))

1-c.
      (HANAKO ISA (WOMAN)
              AGE (20))
```

Fig. 1 Example of LPS schemata (I) "MARRY", "TAROW" and "HANAKO".

*Department of Administration Engineering, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223, Japan.

for ACT. ACT is allowed to have an arbitrary number of slotvalues. The schema MARRY is interpreted as: "marry" is a movement, is a kind of human behavior, and if $M is a man, $M is a woman, $M loves $W, and $W loves $M, then $M marries $W, where $M and $W are variables. Thus, the schema includes the declarative knowledge for categorization of "marry" by ISA and AKINDOF, and also one piece of procedural knowledge that if a man and a woman love each other, then one marries the other. #OBJ1, #OBJ2, #C1 and #C2 in the schema, i.e., the slotnames with "#" at their heads, refer to conditions, and ACT is for actions, of the procedure.

LPS allows at most one production rule in a schema as illustrated in Fig. 1-a. The reader might understand that a schema is basically a production rule possibly with declarative attachment. So control structure for production systems still works for LPS as explained below. With this structure of representation, LPS is able to represent declarative and procedural fragments of knowledge homogeneously in one schema. As shown in the following sections, this compactness of representation is not merely for grafting two different species of knowledge. It yields modifiability of schemata, which is one of the essential features of LPS.

Another restriction in LPS is put on its control structure. To keep transparency of the computation process, we wanted to limit the channel of pattern matching for schemata to only one through *working memory*. Under this restriction, LPS is incorporated with some pattern matching facilities which work on the schema representation of LPS.

For example, suppose that working memory contains two elements, (TAROW WITH A DIAMOND RING) and (HANAKO WITH FLOWERS), and we have three schemata as shown in Fig. 1. The slotvalues, (!ISA MAN $M) and (!ISA WOMAN $W), in schema MARRY in Fig. 1-a imply that $M matches a working memory element which ISA MAN, and $W matches one which ISA WOMAN. Here, $M and $W can match (TAROW WITH A DIAMOND RING) and (HANAKO WITH FLOWERS) respectively because the schemata TAROW and HANAKO, whose names are the first elements of the two lists, in Figs. 1-b and 1-c are visible from MARRY through those first elements of the lists, and those two schemata indicate in them that TAROW ISA MAN, and HANAKO ISA WOMAN. The use of !ISA is explained in more detail in a later section.

Since LPS works through pattern matching via working memory as above, the problem is now how working memory is updated. It is done in LPS by the recognize-act cycle similar to OPS [5]. When the slotvalues with the symbol # at the heads of their slotnames matched working memory, the slotvalues of ACT are executed one after another. If the currently executed slotvalue is not a function, it is just deposited into working memory. If it is, it is evaluated, and may modify working memory.

A special feature of LPS is that those functions include *meta-actions* that modify a schema. Part of a schema is modifiable by meta-actions such as ADD-SLOT-VALUE, DELETE-SLOT and so on. It contributes substantially to flexibility and learnability of the LPS system. The details of meta-actions are explicated later.

Thus in short, LPS is a schema-oriented, but rule-based knowledge representaion system. It is schema-oriented because each chunk of knowledge is represented as a schema, and rule-based since schemata are driven similarly to productions in a production system by executing actions embedded in the schemata.

As a programming system, LPS can be used interactively via the LPS Monitor. The I/O routine, ANALYZE routine for constructing internal representation, EXECUTE routine for matching and recognize-act cycling, and the LPS Editor comprise the lower-level modules of the LPS system, and are all called from the Monitor. Fig. 2 illustrates the relations among these modules and the Monitor. The system is all written in Interlisp, and the first version is presently running on DEC-20 under the TOPS-20 operating system.

## 3. LPS Representation

As exemplified in Fig. 1, an LPS schema is a list of the schema-name and the associated body, that is, a set of slots: a slot is defined as a pair of a slotname and a slotvalue (or slotvalues), and represents a knowledge fragment in a chunk of knowledge. For convenience, a slot is usually referred to by its slotname.

Part of the BNF description of LPS programs is shown in Table 1. As defined in Table 1, a slot is either a condition slot (*c-slot*), attribute slot (*att-slot*), or action slot (*act-slot*). For example, in Fig. 1, the slots ISA, AKINDOF and AGE are att-slots, #OBJ1, #OBJ2, #C1 and #C2 are c-slots, and ACT is the act-slot. ACT is a reserved slotname used only for the act-slot. As mentioned before, an LPS schema has at most one act-slot whose slotname is ACT. A c-slot is identified by the symbol # at the head of its slotname. Any slotname that is not ACT, and not headed by "#", corresponds to an att-slot.

If for a schema, all the c-slotvalues were *true*, then the act-slotvalues may be *executed* from left to right. To define the terms, true and executed, we need to explain ⟨c-slotvalue⟩ and ⟨act-slotvalue⟩. First, a c-slotvalue is

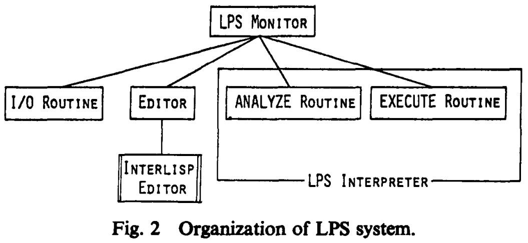

Fig. 2 Organization of LPS system.

Table 1 Partial description of LPS program syntax.

```
<LPS program>::=<working memory><schema>†
<schema>::=(<schema-name><body>)
<schema-name>::=any literal atom
<body>::=<c-slot>|<att-slot>|<act-slot>|<body>*
<c-slot>::=<c-slotname><c-slotvalue>
<c-slotname>::=any literal atom headed by #
<c-slotvalue>::=<simple pattern>|<function>|<ABS-pattern>|
              <IFEXIST-pattern>
<att-slot>::=<att-slotname><att-slotvalue>
<att-slotname>::=any literal atom not headed by #, and except
              ACT
<att-soltvalue>::=any list
<act-slot>::=<act-slotname><act-slotvalue>*
<act-slotname>::=ACT
<act-slotvalue>::=<act-pattern>|<function>
```

either a *simple pattern*, a *function*, an *ABS-pattern*, or an *IFEXIST-pattern*. A simple pattern is a list, and is true as a c-slotvalue if it matched an element in working memory. An ABS-pattern, defined as a list (*ABS <simple pattern>), is true as a c-slotvalue if there was no element in working memory that matched the argument <simple pattern>. A function is true as a c-slotvalue if it was evaluated not to be nil. That is, a function, if written in a c-slot, is regarded as a *predicate* evaluated to be nil or not nil. It should be noted that the name of any function in an LPS program must always put "*" at its head to be distinguished from other expressions. An IFEXIST-pattern, written in the form (*IFEXIST <simple pattern>), is *always* true as a c-slotvalue. If the <simple pattern> matched working memory, variables in the pattern are bound and those bindings are carried to the same variables in the schema. But if the pattern did not match, variables in it are bound to newly generated default values, and those bindings by defaults are transmitted to the same variables in the schema, so derived the name IFEXIST. The system has some functions for handling default values generated by processing IFEXIST's. An example of such functions is shown in Section 5.

Second, an act-slotvalue is either an *act-pattern* or a *function*. LPS allows any LISP function to be written in the act-slot. An act-slotvalue must be a list, but its first element can either be or not be the name of a function. If the first element is a function name, it is called a function, and if not, it is called an act-pattern. Execution of an act-pattern as an act-slotvalue results in deposition of it into working memory after functions included in it are evaluated. A function is said to be executed as act-slotvalue when it is evaluated. Whether its returned value is deposited into working memory or not depends on the user's definition: the user can predetermine which functions should deposit returned values.

To an element in working memory, which is restricted to be a list in the present version, is attached a non-negative integer called the *time index*. It is attached to the element when the element is deposited into working memory, and indicates when the element is deposited. It increases by one unit every time an element is deposited.

Let us provide an example in Fig. 3. Suppose that the current working memory looks like a list ((2 HUMAN TAROW) (1 CANNIBAL PUFFIE)), where 1 and 2 denote time indices and are neglected in pattern matching. Note that working memory is a list of lists kept in the decreasing order of the time index, and that the most recently deposited element is headed by the highest time index.

The schema CANNIBAL in Fig. 3 includes eight slots. Among them, four (#PAT1, #PAT2, #ABSENT, #PRED) are c-slots, one (ACT) is the act-slot, and the remaining three (ISA, REPRESENT, PROPERTY) are att-slots.

As can be inferred from Fig. 3, att-slots are usually used to represent knowledge pieces declaratively: CANNIBAL ISA race, represents CANNIBALISM, and so on.

On the other hand, the act-slot and c-slots are concerned with a procedure that if c-slots were all true, then the act-slotvalues are executed. As working memory contains (HUMAN TAROW), the c-slotvalue (HUMAN $HUM), which is a simple pattern, of the slot #PAT1 matches it, and hence $HUM can be bound to TAROW. Similarly, the c-slotvalues (CANNIBAL $CAN) matches (CANNIBAL PUFFIE) in working memory, and $CAN may be bound to PUFFIE. The c-slotvalue of #ABSENT is an ABS-pattern. Its value is true here because there is no element in working memory that matches (EAT-ENOUGH $CAN) when $CAN was bound to PUFFIE. The slotvalue of #PRED is a predicate, where *NOT and *EQUAL correspond to LISP functions, NOT and EQUAL, respectively. This predicate evaluates to true unless the binding of $CAN is equal to the binding of $HUM, in which case one may keep away from eating his or her own flesh. Now as the case bindings of $CAN and $HUM are PUFFIE and TAROW, the c-slotvalue (*NOT (*EQUAL $CAN $HUM)) returns true.

Thus we saw that all the c-slots of CANNIBAL were true when $CAN and $HUM were bound to PUFFIE and TAROW. The act-slotvalues can be executed after these variable bindings are carried over to the act-slot. After this, the act-slot looks like:

ACT (EAT PUFFIE TAROW)

```
(CANNIBAL ISA (RACE)
          REPRESENT (CANNIBALISM)
          PROPERTY (EAT HUMAN FLESH)
          #PAT1 (CANNIBAL $CAN)
          #PAT2 (HUMAN $HUM)
          #ABSENT (*ABS (EAT-ENOUGH $CAN))
          #PRED (*NOT (*EQUAL $HUM $CAN))
          ACT (EAT $CAN $HUM)
              (*DELETE (HUMAN $HUM)))
```

Fig. 3 Example of LPS schemata (II) "CANNIBAL".

(*DELETE (HUMAN TAROW)).

First, (EAT PUFFIE TAROW) is executed. As it is an act-pattern, it is deposited into working memory after the time index, 3, is attached to it. Next, (*DELETE (HUMAN TAROW)) is executed. DELETE is a system-defined function which deletes its argument from working memory. Hence, when this act-slotvalue was executed, working memory must be:

((3 EAT PUFFIE TAROW) (1 CANNIBAL PUFFIE))

Before leaving this section, note that usually more than one schema matches working memory. To determine a schema to be executed, LPS includes a set of *conflict resolution rules* in the EXECUTE routine. Presently seven rules are incorporated. As they are similar to the rules implemented in OPS [5], we only list the LPS rules in the order of priority: (1) refractory inhibition for c-slots, (2) recency by time indices for c-slots, (3) the number of simple patterns in c-slots, (4) the total number of ABS-patterns, IFEXIST-patterns and fuctions in c-slots, (5) the number of constants in simple patterns in c-slots, (6) recency by *ages* for conflicting schemata, and (7) arbitrary tie-breaking. Each schema has an index called age which refers to the time when the schema was created.

Though the above description gave the basic characteristics of the LPS representation, it is far from complete. LPS is provided with some special pattern matching mechanisms, and actions that modify a program itself. We explain these features in the following sections.

## 4. Slot-net Patterns and Slot-net Variables

The global structure of LPS, i.e., the rule-based recognize-act cycle, is appropriate for knowledge that can be decomposed into piecewise schemata. However, its critical deficit is its difficulty in representing structured knowledge often clearly describable as schemata. For instance, consider an LPS schema as in Fig. 4.

Suppose that, as we have discussed with Fig. 1, working memory contains lists (TAROW WITH A DIAMOND RING) and (HANAKO WITH FLOWERS). These lists do not suffice to fire the schema MARRY in Fig. 4 if lists like (MAN TAROW) and (WOMAN HANAKO) are lacking from working memory.

To keep somewhat "time-invariant" information such as (MAN TAROW) and (WOMAN HANAKO) constantly in working memory is not only awkward for recognize-act cycle computation, but also cumbersome for understanding the program structure: it is more readable if the schema TAROW itself includes the declarative information that Tarow is a man, also for the schema HANAKO. Slot-net patterns and slot-net variables are introduced to avoid such difficulty.

A slot-network is a network of slotvalues connected by a particular slotname, and used for representing structured knowledge in the LPS database. For example, a point is a part of a line segment, a line segment is a part of a triangle, and a triangle is a part of a tetrahedron. Tarow is a handsome boy, a handsome boy is a boy, a boy is a man, and a man is a human. These hierarchical relations can be represented in schemata by APARTOF- and ISA-networks, respectively. For instance, if we have five schemata in Fig. 5, then the ISA-network embedded in them looks as shown in Fig. 6.

A merit of the LPS representation is that the user can embed slot-networks explicitly in a program, while keeping the global recognize-act cycle structure. But this merit should be buttressed by some procedure to use the embedded networks. It is explicitly performed in LPS by what we call slot-net patterns and slot-net variables that restrict ranges of pattern matching

```
(MARRY ISA (MOVEMENT)
       AKINDOF (HUMAN.BEHAVIOR)
       #C1 (LOVE $M $W)
       #C2 (LOVE $W $M)
       #C3 (MAN $M)
       #C4 (WOMAN $W)
       ACT (MARRY $M $W))
```

Fig. 4 Example of LPS schema without slot-net patterns or variables.

```
(CANNIBAL ISA (RACE)
          REPRESENT (CANNIBALISM)
          #PAT1 ($AKINDOF-CANNIBAL-CAN)
          #PAT2 ($ISA-HUMAN-HUM)
          #PRED (*NOT (*EQUAL $CAN $HUM))
          ACT (EAT $CAN $HUM)
              (*DELETE ($HUM))

(PUFFIE ISA (MAN)
        AKINDOF (CANNIBAL))

(TAROW ISA (MAN)
       AGE (22))

(MAN ISA (HUMAN))

(JOHN ISA (DOG)
      AGE (7))
```

Fig. 5 Example of LPS schemata (III).

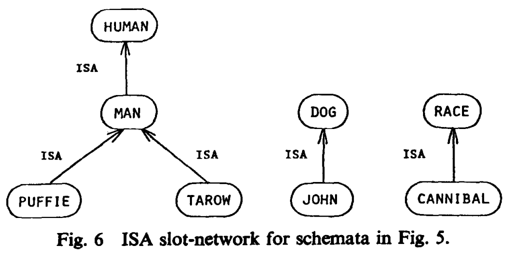

Fig. 6 ISA slot-network for schemata in Fig. 5.

through slot-networks embedded.

A slot-net pattern is a kind of simple pattern, and denoted basically as (!⟨slotname⟩⟨slotvalue⟩⟨variable⟩). ⟨Slotname⟩ is a slotname which defines the type of a slot-network; one such as ISA in Fig. 6. When a slot-net pattern is used, any list in working memory matches the slot-net pattern if the node named ⟨slotvalue⟩ is accessible in the related slot-network from the first element of the list. For instance, the list (HANAKO WITH FLOWERS) matches (!ISA WOMAN $W) because the schema HANAKO has the slot ISA and the value (WOMAN). Also (TAROW WITH A DIAMOND RING) matches (!ISA HUMAN $X) if the ISA-network in Fig. 6 was used: it is equivalent to accessibility of the node HUMAN from TAROW in the ISA-network. ⟨Variable⟩ in a slot-net pattern is the variable to which a matched working memory element is to be bound. The matched information, i.e., (TAROW WITH A DIAMOND RING) in the last example, is substituted into the ⟨variable⟩ $X, and used anywhere else in the schema. But ⟨variable⟩ can be omitted if no substitution is necessary.

As matching of a slot-net pattern is concerned with only the first element of a list, but substitution into ⟨variable⟩ is for the whole list, a slot-net pattern is allowed to match a list of any length. On the other hand, a slot-net variable, which is regarded as a kind of variable, is allowed to match only an atom. It is denoted basically as $⟨slotname⟩-⟨slotvalue⟩-⟨variable⟩. Similar to slot-net patterns, ⟨slotname⟩ defines the type of a slot-network on which matching is to be performed. A slot-net variable matches any atom in a working memory element from which the node ⟨slotvalue⟩ is accessible in the slot-network specified by ⟨slotname⟩. ⟨Variable⟩ is the variable to which a matched atom is to be bound, and can be omitted when unnecessary. Let us provide a brief example of slot-net variables below.

Suppose that we have five schemata as shown in Fig. 5. Also assume that the current working memory is a list:

((3 JOHN) (2 TAROW) (1 PUFFIE)),

where 3, 2, and 1 are time indices. The ISA network embedded across those schemata is illustrated in Fig. 6.

Variables $AKINDOF-CANNIBAL-CAN and $ISA-HUMAN-HUM in Fig. 4 are slot-net variables. The variable $ISA-HUMAN-HUM is allowed to match an atom from which the node HUMAN is reachable in the ISA network in Fig. 6. As shown in Fig. 6, the node HUMAN can not be reached from JOHN, but can be from TAROW and PUFFIE. Thus the c-slotvalue ($ISA-HUMAN-HUM) of #PAT1 can match (2 TAROW) of (1 PUFFIE) in working memory. Similarly, $AKINDOF-CANNIBAL-CAN can match (1 PUFFIE) because PUFFIE is AKINDOF cannibal as shown in Fig. 5. Accordingly, as all the c-slots are true in the schema CANNIBAL, its act-slot can be executed after the bindings are passed to $HUM and $CAN in the act-slot. After the execution, working memory looks like:

((4 EAT PUFFIE TAROW) (3 JOHN) (1 PUFFIE)).

As the above example suggests, the slot-net patterns and variables give some semantic restrictions to bindings, using declarative knowledge embedded in LPS schemata.

The above two kinds of semantically limited matching schemes, i.e., slot-net patterns and slot-net variables, are not alternatives, but each can be used in any part of a program where the user considers relevant. As mentioned above, these schemes facilitate computation of the LPS recognize-act cycle, and understanding of program structure.

Notice that both slot-net patterns and variables are effective only in pattern matching through working memory, and they can not be used in direct matching between schemata as in usual frame-based systems. Rather, woking memory elements are often just tags of associated schemata, and slot-net patterns and variables can be regarded as efficient communicating devices for looking into the details of chunks of knowledge through working memory.

## 5. Meta-actions

While slot-networks are essential for representing structured knowledge in the rule-based control structure, it is another fundamental characteristic of the LPS representation that schemata can be generated, deleted and modified easily by actions within the schemata. Thus, LPS retains flexibility and potential learnability of representation. However, though this characteristic was derived from the piecewise schema representation actually inherited from production system architecture, modification of part of a schema is made much easier in LPS by utilizing slotnames. Usual production systems have difficulty in indicating one specific condition to be modified because no name or tag is given to the condition. LPS, on the other hand, is able to add, delete and modify one or more slots in a schema. We call that kind of actions, "meta-actions" since they are executed *on* the representation itself.

Presently, LPS is equipped with around 20 meta-actions mostly developed along our research on natural language understanding [8]. Table 2 shows the list of principal meta-actions often useful.

Meta-actions of LPS can be classified into the following five classes: (a) adding new information to the current representation, (b) deleting information from the current representation, (c) properly modifying the current representation, (d) extracting information from the current representation, and (e) miscellaneous.

For example, among 11 meta-actions in Table 2, BUILD, ADD-SLOT-VALUE and RECOVER are in the class (a), i.e., they add some new information to the current program. BUILD builds a new schema,

Table 2 Typical LPS meta-actions.

1. (*BUILD ⟨schema-name⟩⟨c-list⟩⟨act-list⟩⟨att-list⟩)
2. (*DELETE-SLOT ⟨slotname⟩⟨schema-name⟩)
3. (*ADD-SLOT-VALUE ⟨slotname⟩⟨slotvalue⟩ ⟨schema-name⟩)
4. (*DELETE-SLOT-VALUE ⟨slotname⟩⟨slotvalue⟩ ⟨schema-name⟩)
5. (*CHANGE-SLOT-VALUE ⟨slotname⟩⟨slotvalue⟩ ⟨schema-name⟩)
6. (*SUBSTITUTE ⟨oldvalue⟩⟨newvalue⟩⟨schema-name⟩)
7. (XTRELEMENT ⟨slotname⟩⟨location⟩⟨schema-name⟩)
8. (*XTRSLOT ⟨slotname⟩⟨schema-name⟩)
9. (*FIND-DEF-VALUE ⟨slotname⟩⟨schema-name⟩)
10. (*ERASE ⟨schema-name-list⟩)
11. (*RECOVER ⟨schema-name-list⟩)

where

⟨c-list⟩::=(⟨c-slot⟩*) ! c-slotnames can be omitted.
⟨act-list⟩::=(⟨act-slotvalue⟩*)
⟨att-list⟩::=(⟨att-slot⟩=) ! att-slotnames can be omitted.
⟨slotname⟩::=⟨c-slotname⟩|⟨att-slotname⟩|⟨act-slotname⟩
⟨slotvalue⟩::=⟨c-slotvalue⟩|⟨att-slotvalue⟩|⟨act-slotvalue⟩
⟨oldvalue⟩::=any S-experession
⟨newvalue⟩::=any S-expression
⟨location⟩::=a list of positive integers that indicate the nested position of an S-expression in the designated slot.
⟨schema-name-list⟩::=(⟨schema-name⟩*).

When a slotname in the ⟨c-list⟩ or ⟨att-list⟩ was omitted, the system automatically generates a new symbol for the slotname.

ADD-SLOT-VALUE adds information to a slot, or creates a slot, in a schema, and RECOVER recovers a once erased schema.

The class (b) includes DELETE-SLOT, DELETE-SLOT-VALUE and ERASE in Table 2. As the names suggest, DELETE-SLOT deletes a specified slot from a schema, DELETE-SLOT-VALUE deletes information from a slot in a schema, and ERASE erases a schema from the body of the program. Actually, ERASE and RECOVER are inverse operations.

CHANGE-SLOT-VALUE and SUBSTITUTE in Table 2 belong to (c). The slotvalue of a slot in a schema can be changed to a new slotvalue by using CHANGE-SLOT-VALUE. SUBSTITUTE substitutes a new expression into a specified old expression in a schema.

The class (d) contains XTRELEMENT, XTRSLOT and FIND-DEF-VALUE among meta-actions in Table 2. XTRELEMENT returns the expression at a specifed location in a schema. Taking the schema CANNIBAL in Fig. 3 as an example, (*XTRELEMENT #ABSENT (2 1) CANNIBAL) returns EAT-ENOUGH, for the atom EAT-ENOUGH is located at the first position in the second element of the slotvalue for #ABSENT, which is (*ABS (EAT-ENOUGH $CAN)). FIND-DEF-VALUE finds all default values in a schema, and returns the list of them. The user can use this kind of meta-actions for dealing with default values to program himself a default handling mechanism. Those meta-actions are able to distinguish default values from other atoms. Notice that XTRELEMENT and XTRSLOT are used not for pattern matching, but only for looking at the inside of a schema without getting through working memory. It is still true that patterns can be matched only through working memory.

We have found from experience that most of what the user wants to do can be written not by using complex user-defined LISP functions, but only by combining system-defined functions and meta-actions. Although the user is allowed to incorporate arbitrary user-defined LISP functions in LPS programs, unlimited use of them sometimes leads to blurred and confused representation. It is safer, and usually enough, to use only functions served by the system. We illustrate examples of meta-actions later in Section 7.

## 6. LPS Monitor

An LPS program is initially compiled into internal representation before execution. It is preferable that the user be able to use the system with no inconvenience even if he knows nothing about the internal structure of LPS. Also one of the difficulties in programming LPS programs is that the user is sometimes hard to foresee which schema will be executed and in what order. For subjugating it, an interactive facility is helpful. The LPS Monitor was designed to satisfy these requirements.

Once the user gets into the Monitor mode, he can access interactively to any of the modules as shown earlier in Fig. 2. It may be sufficient here to briefly describe only the fundamental features of the Monitor:

(1) The user can input schemata and working memory elements for his program by answering questions asked by the Monitor, and does not need to be conscious about the internal representation.
(2) The user can use the LPS Editor interactively in the Monitor. The Editor is essentially based on the Interlisp Editor, but specially tailored to LPS programs. The user can reach a desired original, modified or generated schema, the initial or current working memory, or a user-defined function, by answering questions asked by the Editor.
(3) The user can specify the number of times for recognize-act cycling. For example, the Monitor command "XQT 4" executes cycling four times, and then the run pauses. This facility makes it easier for the user to do interactive monitoring and execution.
(4) The user can suspend and logout while executing an LPS program, and login and execute from that intermediate state.
(5) The user can easily handle the Monitor with schemata generated or modified during the run. There is a separate area for storing these schemata, and it depends on the user whether to save, or delete them.

## 7. How an LPS Program Runs: An Example

This section provides in a casual manner an illustrative example of how a program runs. The cover story itself is only for the reader's ease of comprehension, and not essential for the purpose of this section. Although the example is not complex, it includes most of the features explained in the preceding sections. Fig. 7 shows the set of schemata and the initial working memory that constitute our example LPS program.

The contents of the initial working memory can be paraphrased as follows: (once upon a time,) there were two men, Tarow and Jirow, and a woman, Hanako, at what we call Unknown Island (located in some unknown area), and Tarow and Hanako loved each other. The 11 schemata in Fig. 7 are the only information that describes and controls their world. The execution of the program then simulates what happened to those three people.

Let us follow the simulation output, which is listed in Fig. 8. As Tarow and Hanako loved each other, first they married at Unknown Island. The execution of the schema MARRY shows this. MARRY includes two meta-actions in the act-slot, and the schemata, TAROW and HANAKO, were modified by them: Tarow became a husband and Hanako a wife. MARRY then deleted all the information concerned with love and the names of the individuals involved in it. This is done by executing four actions such as (*DELETE $C1) at the bottom of the act-slot. As mentioned in Section 3, DELETE

<ins>Initial set of schemata</ins>

```
(MARRY ISA (MOVEMENT)
       AKINDOF
       (HUMAN.BEHAVIOR)
       #C1
       (LOVE $ISA-MAN-M $ISA-WOMAN-W)
       #C2
       (LOVE $W $M)
       #C3
       ($M)
       #C4
       ($W)
       #C5
       (*ABS (MARRY $M $W))
       ACT
       (*PRINT (### MAN $M MARRIES WOMAN $W ###))
       (*CHANGE-SLOT-VALUE ISA $M (HUSBAND))
       (*CHANGE-SLOT-VALUE ISA $W (WIFE))
       (MARRY $M $W)
       (*DELETE $C1)
       (*DELETE $C2)
       (*DELETE $C3)
       (*DELETE $C4))

(MARRIED-COUPLE #C1 (MARRY $MAN $WOMAN)
                #C2
                (*ABS (MARRIED-COUPLE $MAN $WOMAN))
                ACT
                (*ADD-SLOT-VALUE HASWIFE $MAN ($WOMAN))
                (*ADD-SLOT-VALUE HASHUSBAND $WOMAN ($MAN))
                (MARRIED-COUPLE $MAN $WOMAN)
                (*DELETE $C1))

(EAT #C1 (!ISA HUMAN $MAN)
     #C2
     (!AKINDOF CANNIBAL $CAN)
     ACT
     (*PRINT (### $MAN WAS EATEN BY CANNIBAL $CAN ###))
     (EAT $CAN $MAN)
     (*DELETE $C1)
     (*HALT))

(JIROW ISA (MAN))

(TAROW ISA (HANDSOME.BOY))

 (PUFFIE AKINDOF (UNKNOWN.TRIBE)
         #C1
         (AT UNKNOWN.ISLAND)
         #C2
         (*ABS (PUFFIE))
         ACT
         (PUFFIE))

 (UNKNOWN.TRIBE AKINDOF (CANNIBAL))

 (HANDSOME.BOY ISA (MAN))

 (BEAUTIFUL.LADY ISA (WOMAN))

 (MAN ISA (HUMAN)
      #C1
      ($ISA-MAN-M)
      ACT
      (*DELETE ($M))
      (MAN $M))

 (HANAKO ISA (BEAUTIFUL.LADY))
```

<ins>Initial working memory</ins>

```
((6 LOVE TAROW HANAKO) (5 LOVE HANAKO TAROW) (4 AT UNKNOWN.ISLAND)
(3 JIROW) (2 TAROW) (1 HANAKO))
```

Fig. 7 Example LPS program: initial working memory and set of schemata.

```
MNTR>XQT
>>> LET'S START ANALYZING YOUR PROGRAM
>>> READY FOR EXECUTION !

**FIRE :MARRY
(### MAN TAROW MARRIES WOMAN HANAKO ###)

(***** SCHEMA TAROW CHANGED *****)


  (TAROW ISA (HUSBAND))

(***** SCHEMA HANAKO CHANGED *****)


  (HANAKO ISA (WIFE))

((7 MARRY TAROW HANAKO) (4 AT UNKNOWN.ISLAND) (3 JIROW))

**FIRE :MARRIED-COUPLE

(***** SCHEMA TAROW CHANGED *****)


  (TAROW ISA (HUSBAND)
        HASWIFE
        (HANAKO))

(***** SCHEMA HANAKO CHANGED *****)


  (HANAKO ISA (WIFE)
         HASHUSBAND
         (TAROW))

((8 MARRIED-COUPLE TAROW HANAKO) (4 AT UNKNOWN.ISLAND) (3 JIROW))

**FIRE :PUFFIE

((9 PUFFIE) (8 MARRIED-COUPLE TAROW HANAKO) (4 AT UNKNOWN.ISLAND)
(3 JIROW))

**FIRE :EAT
(### (JIROW) WAS EATEN BY CANNIBAL (PUFFIE) ###)

((10 EAT (PUFFIE) (JIROW)) (9 PUFFIE) (8 MARRIED-COUPLE TAROW HANAKO)
(4 AT UNKNOWN.ISLAND))

 ***** LPS HALT ! *****

MNTR>STOP
" LABELLED PRODUCTION SYSTEM VERSION-1  SEE YOU AGAIN !!"
```

Fig. 8 Computational output for program shown in Fig. 7.

deletes its argument from working memory. One simplified notation, $C1 in (*DELETE $C1) for example, was introduced in description of act-slots. That is, if #C is a slotname in a schema, any $C in an act-slotvalue of that schema is substituted by the slotvalue of #C. (This example is for a c-slotname, but the notation is applied to any type of slotnames. # is deleted in the case of c-slotnames.) Thus, (*DELETE $C1), when executed, deletes from working memory (LOVE TAROW HANAKO), the bound slotvalue for the slot #C1.

Then, Tarow and Hanako became a married couple, which was simulated by the execution of MARRIED-COUPLE. This schema also includes meta-actions, which modified TAROW and HANAKO further.

At Unknown Island, there lived a cannibal named Puffie. He popped out to the world when the schema PUFFIE was executed. This corresponds to deposition of the list (PUFFIE) in the ACT slot of the schema PUFFIE into working memory. As implied in the structure of the schema, it can be executed if no particular event, e.g., a love affair, was going on at Unknown Island. Lastly, as Puffie and a single man, Jirow, were at Unknown Island at the same time, Jirow was eaten by Puffie. It was simulated by executing the schema EAT. After EAT was executed, Jirow disappeared from the world, and Puffie and the married couple lived together on the island.

## 8. Discussion

Pursuance of efficiency, understandability and flexibility at the same time is not an easy task. There seems to be no resolution at least currently to this problem: the three criteria are conflicting each other, and a design of an artificial intelligence system involves trade-offs among them. Our LPS system pursues reasonable efficiency, understandability and flexibility as a general-purpose knowledge representation system. This section provides a brief discussion on evaluation of LPS from these three dimensions.

*Efficiency.* Time efficiency of LPS is kept reasonable by adopting discrimination nets as internal representation, though computation time depends especially on complexity of functions in c-slots of schemata. One recognize-act cycle averages about 0.5 sec for the current Interlisp version. We have kept time efficiency reasonable by sacrificing a bit of space efficiency. It will be on the agenda for a future version to increase space efficiency while keeping other merits.

*Understandability.* As emphasized in this paper, LPS restricts pattern matching to be only through the narrow channel of working memory visible to the user. Also every detail of knowledge is represented explicitly and homogeneously in a simple form of schemata. The only invisible part is conflict resolution in the recognize-act cycle. No global variable or stack visible from the user is involved except those in working memory. Thus, LPS programs and computation processes are transparent enough even for a naive user.

*Flexibility.* Any knowledge base in a knowledge representation system must be well prepared for frequent automatic or hand modification because any kind of knowledge is exposed to changes of the environment, and to adapt to the change, maintenance of the knowledge base is frequently needed. Confronted with needs for modification, it is far better that knowledge can be altered within the representation system itself. In this sense, LPS is flexible enough as it can modify itself by using meta-actions. A concept close to flexibility is learnability. A system is highly learnable if a new piece of knowledge can readily be assimilated into an existing internal representation. LPS, as well as other rule-based systems, has potential learnability in the sense that it can add a new schema to an existing set of schemata. But LPS's potential in learning is higher than ordinary rule-based systems, since LPS is able to access and modify more detailed part of a schema than merely a whole rule.

## 9. Conclusion

One of the hopes of researchers in artificial intelligence and cognitive science is to have a knowledge representation scheme that is as flexible as the human cognitive system. In this paper we presented the new representation system LPS, motivated basically by this hope. The overall control structure of LPS is rule-based, but the local representational structure is schema-oriented. The design philosophy was to increase understandability and flexibility at the same time, while keeping efficiency reasonable. Though this paper addresses only what LPS is, LPS has been successfully applied to two different areas; knowledge-based problem solving [7] and natural language understanding [8]. Although the high flexibility of the human cognitive system is as yet unattainable for the present, our results advanced the research at least one step further towards symbolic realization of systems as such.

**Acknowledgement**

This work was supported by the Grant-in-Aid of the Ministry of Education No. 446194, and also partly by the Esso Financial Aid for Interdisciplinary Reserch.

**References**

1, Bobrow, D. G. and Winograd, T. An Overview of KRL, a Knowledge Representaion Language. *Cognitive Science*, 1, (1977), 3–46.

2. Davis, R. Buchanan, B. G. and Shortliffe, E. H. Production Rules as a Representation for a Knowledge-based Consultation Program, *Artificial Intelligence*, 8, (1977), 15–45.

3. Woods, W. A. What's In a Link: Foundations for Semantic Networks. In D. G. Bobrow and A. Collins (eds.), Representation and Understanding, New York, N.Y.: Academic Press, (1975), pp. 35–82.

4. Waterman, D. A. and Hayes-Roth, F. (eds.) Pattern-directed Inference Systems. New York, N.Y.: Academic Press. (1978).

5. Forgy, C. and McDermott, J. The OPS4 Reference Manual.

Computer Science Department, Carnegie-Mellon University, (1979).
6. Minsky, M. A Framework for Representing Knowledge. In P. H. Winston (ed.), The Psychology of Computer Vision, New York, N.Y.: McGraw-Hill, (1975), pp. 211–277.
7. Anzai, Y., Ishibashi, N., Mitsuya, Y. and Ura, S. Knowledge-based Problem Solving by a Labelled Production System. *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, (1979), 22–24.
8. Mitsuya, Y. Design and Implementaion of a Japanese Language Understanding System. Unpublished MS Thesis, Administration Engineering Department, Keio University, (1980).

(Received June 18, 1980: revised July 20, 1981)