LORD: LISP-ORIENTED RESOLVER AND DATA-BASE

V.M.Briabrin, V.A.Serebriakov, V.M.Yufa
Computing Center, Academy of Sciences, Moscow, U.S.S.R.

Abstract

The paper presents formalism and implementation aspects of the programming system for Artificial Intelligence (AI) applications. The system contains Resolver which is based on the ideas of LISP - like languages combined with new AI formalisms. Semantic Memory is another part of the system which is implemented by means of hierarchical Data-Base. Interaction with a user is governed by Metaprocessor which generates a syntactic analyser driven by the grammar for the specific version of the input language.

Introduction

The AI problems require incorporation of new features into the traditional programming systems. These features include: the availability of powerful vehicles for creation and amendment of structured data elements - lists, sets, texts; comfortable and efficient formalism for representation of knowledge in different problem domains; new techniques for "non-algorithmic" programming style, such as pattern - driven procedure invocation and automatic backtracking; the availability of standard procedures for the associative search in Data Base; automatic logical inference, etc.

Some of these features could be found in LISP, REFUL, SNOBOL, and also in "big" language systems such as PL/1, APL, ALGOL-68. At the same time there exists a growing tendency for the design of new languages combining the best techniques and most suited for AI applications [1].

The system described herein - Lisp-Oriented Resolver and Data-Base - is designed at the Computing Center of the USSR Academy of Sciences with the purpose to provide an efficient instrument for different AI applications [2] . The system is designed on the basis of some essential ideas being derived at the different scientific groups and materialised in the implementation of some experimental systems, viz., PLAINER, CONNIVER, LISP-70, SAIL, QLISP, POPLER 1.5.

General System Organisation and Data-Base

The LORD system consists of three relatively independent parts: Resolver, Semantic Memory and Metaprocessor. Each of these parts uses similar techniques for memory allocation, function storage and call mechanisms, lexical analysis procedures etc The Data-Base (DB) is the common ground for keeping and retrieval of both program and data elements.

DB has a fixed number of hierarchical levels. The largest section of DB is an area. Each area is identified by its name and may contain an arbitrary number of sets. A set consists of records each of which contains a full characteristic of an object. Usually, such an object is generated and used within the boundaries of a specific user task. The access to the objects is performed by means of hash function (standard of individual for each set).

Objects may have several different properties, and each property has its own value (sometimes a list of values). List of references to the properties is stored in the body of the record corresponding to the given object. Each property in its turn is stored as a separate object with property indicator used as an object name*

Thus in the hierarchical DB we have the following levels associated with system notions:

Data-Base divisions	LORD notions
area	problem domain
set	context
record	object
field	property
contents of the field	value of the property

The LORD access language is provided with functions for establishing the name of the problem domain and the name of the working context. Special references are generated during the creation of new objects and addition of new properties to the existing objects. According to these references the objects and the values of their properties may be derived from DB later.

In many AI applications it is necessary to create the hierarchy of contexts (and even the hierarchy of problem domains). This is provided by automatic bookkeeping of special "set of references" containing the objects which values are pointers at the different DB sections. The hierarchy of contexts or problem domains is represented by interconnections of the objects in this reference set.

Semantic Memory

One of the most important problems in AI systems is accumulation of information representing "knowledge" about some problem domain. There are two distinct approaches to the techniques of knowledge representation. One way is to use a set of expressions describing separate facts or hypotheses. Single expression may represent the relation between two objects or between an object and its properties, or between two or more different facts. Another approach is based on the utilisation of a set of procedure*. Procedure execution checks the existence of definite relations binding specific objects, properties or facts.

These two approaches, "relational" and "procedural", are often intermixed in actual implementations. Semantic Memory in the LORD system is intended to serve the same purpose. Accumulation of knowledge about different problem domains, structuring of accumulated information according to predetermined heirarchy, and retrieval of relevant notions and facts - these are the main functions provided by Semantic Memory.

The examples of simple expressions processed by the Semantic Memory:

```
(PRODUCT =IS-SBT= (CARS, OIL,CRAIN))(A1)
(PRODUCT =HA5-PR0PERTY= SHORTAGE)
(PRODUCT =BECOME-AN-OBJECT-OF:s
   BLACK-MARKET-OPERATION)             (A2)
```

After processing of these expressions special DB sections are filled. These sections are named NOTION - LIST and FACT - LIST.

1) New objects: PRODUCT, CARS,... appear in the NOTION-LIST if they were not put there earlier.

2) Expressions A1 and A2 are placed into the PACT-LIST, and identifiers A1 and A2 become the names of the corresponding objects, text of the expression becomes the value of the main object property.

3) Each object in the NOTION-LIST is accompanied by the property list. One of the properties has the value showing the object type, another property is evaluated into a list of references to all facts containing the given object. Examples of objects being stored in the NOTION-LIST:

PRODUCT → type SET,facts (A1 A2)
OIL → type SET-ELEMENT, facts (A1)
SHORTAGE → type PROPERTY, facts (A2)

Besides the above mentioned sections

there Is also RELATION-LIST in DB. In a sense this list is a subset of NOTION-LIST, - all system and user defined relations are kept in this sections, e.g., IS-SBT, HAS-PROPERTY etc.

Semantic Memory performs processing of the input expression and all the necessary changes in the DB. Furthermore, the Semantic Memory processor provides answering simple questions of the type:
(OIL =IS-SET-ELEMENT-OF= ?X) **(?)**

The answer is based on binding the pattern variable ?X with the object PRODUCT, which results from the analysis of the property list of the object OIL and "backward" processing of the fact A1.

Besides filling DB with the simple facts, expressions describing complex structures may also be input to the Semantic Memory. One form of comlex expression could be the definition of the function which governs the maintainance of Semantic Memory. An example of such a function is:
(PROPERTY-TRANSFE

This function acts as an analogous IF-ADDED function in CONNIVER, i.e., evaluates operators in the function body when the fact matching the given pattern is added to the Semantic Memory. Here **?Y, ?P, ?X, ?R, ?Q** are pattern-variables and **$Y, $P,...** are the corresponding values, obtained by pattern-variables after successful matching.

Therefore the apperance of the fact (OIL =RAS-PROPERTY= SHORTAGE) in the presence of A1, A2 and A3 implies automatic addition of the fact:
(OIL =BECOMB-AN-OBJECT-OF=
BLACK-MARKET-OPBRATION) (A4)

It is worth noting that A3 has the same format as expressions defining the usual relations, e.g., A1 and A2. That is why A3 may be amended in the same way as any other fact in the Semantic Memory. For example, the following question may be asked:
(PROPERTY-TRANSFER **=IS-FUNCTION-OF= ?W, =TYPE= ?X, =PATTERN= ?Y, =BODY= ?Z)**
(?)

In order to derive an answer pattern-variables **?W, ?X, ?Y, ?Z** get as values the corresponding texts ploked up from A3. This feature gives a convenient ability to investigate and modify the system by its own means.

Standard object types are established at the time of system creation, e.g., constant, object, relation name, function name, etc. A user may also introduce his own types using existing relations, their logical compositions and modalities.

Any fact and notion have a limited scope - DB context. We call it "static" context as opposed to "dynamic" context being dealt with in control structures discussed in the next paragraph. The Semantic Memory access language contains operations for manipulating static contexts - rearrangement of the context tree, removing contexts, uniting them with each other, generating new contexts. As it was mentioned, these operations are performed by processing the special reference set.

Resolver

While the Semantic Memory is used mainly for storing and retrieval of facts, functions, objects and their properties, the Resolver serves for evaluating the procedures carrying out different kinds of logical inference, searching AND/OR trees, reduction to disjunctive normal form and other general or specific functions.

The Resolver may be considered as an extension of LISP containing new facilities both in the input language and in processor implementation. A brief list of these facilities follows.

Notation

One of the doubtless requirements for Resolver implementation is the ability to process programs written in standard LISP [3]. At the same time special preprocessor can accept and translate into standard LISP-notation the expressions written in ALGOL-like input language, e.g.,,

```
   BEGIN NEW X,Y; X:= '(1 3 ) : =
         X CONS CDR(X); RETURN(Y) END
```

The given notation is similar to that of MLISP2 [5] and is characterized by the absence of superfluous parentheses, infix notation of most operators and usual mathematical notation for function calls.

The operator and function set is provided with the priority system which facilitates writing and reading complex expressions.

A-points and backtracking

Resolver has the ability to backtrack programs which implies restoring of program and data state in some previously passed point and choosing and initiating an alternative path of solution as in [5]. A-point (alternation point) is set up by the call for one of special functions: REP (repetition), ALT (alternation) and OPT (option). Generation of failure and return to the last A-point is produced by the function FAIL which is called directly or indirectly, e.g., on unsuccessful pattern matching. Backtracking mechanism compels talking about "dynamic" context defined by access link, binding link, control link and process state in the sense of Bobrow [1].

The numbers of generated and eliminated alternative branches (dynamic contexts) are fixed as values of special system variable APOINT. There is a possibility to transfer new values to the "higher" dynamic contexts. This is accomplished by the following generalized assignement operator:

```
     (SETO u (n1 n2 ... np) v) ,
```

where n1, n2, ..., np stand for A-point numbers (possibly expressions evaluated to numbers) indicating the dynamic contexts, where variable u has to accept the new value v. The particular form of this operator:

```
          (SETO u v)
```

changes the value of u in the current dynamic context. Another particular case:

```
          (SETQ u (GLOBAL) v)
```

changes the value of u in the whole program.

Indirect function calls and debugging aids

One of the most important trends in modern "non-procedural" programming is the use of indirect function calls. In LORD this is achieved by means of pattern - directed function call, suspension of function evaluation and "by-passing" of functions.

The idea and implementation of pattern-directed function call are analogous to the corresponding facilities of PLANNER and CONNIVER (V). The main point is the inclusion of function call pattern into the function definition expression. Resolver performs only one type of pattern call, namely: a function with a pattern is called when some variable is assigned a value which represents an object matching the function pattern. This WHEN-ASSIGNED type of call is used for initiation of relevant procedures when specific information appears in the current dynamic context.

Two other types of call, WHEN-ADDED and WHEN-REMOVED, relate to the Semantic Memory processor, i.e., to addition and removing the structures matching the function pattern.

The suspension of function evaluation is performed by the operator (STOP f m) which interrupts evaluation of the function f containing this operator. The control is transfered to the dynamic context, embracing the call of function f. The argument m is a message (a value of interrupted function f) being sent to the embracing program. If later the operator (CONT f) is met then evaluation of f will be resumed from the point of interruption. This tool permits the synchronisation of computational processes; moreover, combined with conditional expressions it may be used for organisation of alternative branches.

"By-passing" of function calls facilitates debugging operations. It is implemented by substitution of a debugging function g everywhere instead of "suspected" function f. The substitution is performed after evaluation of the expression (BYPASS f p g) . Predicate p is evaluated each time when function f is called. By-passing is taking place only if p is true.

Other debugging aids are: setting-up the maximum number of calls for the specified function, establishing the "alarm clock" for the current dynamic context, etc.

Interaction with Semantic Memory

The Resolver interacts with Semantic Memory processor by means of functions ADD, REMOVE, FIND, CHECK and others, which perform storing and removing expressions from the DB, associative search by a given pattern in the static context, check for the presence/absence of definite object properties etc. While the Resolver is dealing with atoms, lists and texts, Semantic Memory may contain objects of different types, such as set, tree, arbitrary structure. The Semantic Memory processor is capable of performing such actions as union and intersection of sets, check for membership, etc.

Special types of predicates in Resolver - existential and universal quantifiers - are also implemented with the use of FIND and CHECK operations in the Semantic Memory. The corresponding expressions in Resolver:

(EXIST (x1 x2 ...) e c) and
(FORALL (x1 x2 ...) e c) ,

where x1, x2, ... - quantifier variables, e - expression, evaluating the conditions of quantifier application, c - static context.

There is also an aggregate operator:

(FOREACH x p e),

implying the execution of expression e for each object x in Semantic Memory satisfying predicate p.

Metaprocessor and function compilation

Resolver and Semantic Memory accept the expressions satisfying formal syntax. In the meantime the idea of giving the user the ability to create his own versions of input language becomes more and more popular. For this purpose LORD system contains Metaprocessor which is similar to the one designed for MLISP2 [5].

Metaprocessor manipulates the sequence of grammatical rules which have the following format:

DBF f (x1, x2, ...) =<syntax>
MEAN <semantics> ,

wher f stands for metavariable name or program name; x1,x2, ...- parameters; <syntax> defines the formal structure of an input phrase; <sematics> - sequence of functions to be evaluated by the Resolver or the Semantic Memory processor.

A set of these rules defines a formal context free grammar which describes the specific version of the input language. It is worth noting that this could be a simple "functional" language, where each name f corresponds to the program composing the <semantics> Such a program could be initiated either by functlonal expression **f(x1, x2, ...)** or by some phrase corresponding to the given **<syntax> .**

On the other hand, the nontrivial language with deep phrase structure could be defined by means of metararlables used in the syntax of some rules. Thus a special language system could be Implemented, which finally is interpreted by means of LORD Resolver and Semantic Memory.

The role of <semantlos> could be illustrated by the following example. Suppose we would like to create a LISP-dlalect to be translated Into standard LISP 1.5 program. In this case the sequence of grammatic rules will define the general syntax of the LISP - dialect expression with the topmost rule looking as folloingi

```
DEF f (x1, x2, ... ) =
<general-syntax-of-LISP-dialect-
 expression>
MEAN ( GENER<file>, <output-string>¯
       LISP<file>)
```

where <output-string> contains pieces of LISP 1.5 program icorporating **x1, x2,** ... values provided by syntax analyser.

<semantics> in this example includes two macrocalls:the first one generates LISP 1.5 text in the specified file, the second one calls LISP 1.5 translator with generated file as a source of input.

In the similar way we could define any context free input language with semantic interpretation provided by the programming language which exists in the computer already.

All functions composing the semantics are compiled into macro-assembler language and then into machine code. The modules of compiled oode are stored in the Data-Base. The LORD Monitor calls them from the DB according to Resolver and Semantic Memory functioning.

Besides DEF expressions Metaprocessor accepts also special command operators serving in on - line interaction with the LORD system - this includes editing text files, switching to different operation modes, choosing input/output channels, etc. [6]j.

Conclusion

The design of a new system capable of successful competing with conventional widespread programming languages suoh as LISP or PL/1 is a hard and fascinating task. The ground for optimism, lies, on the one hand, in the fact that many of the ideas comprised by the LORD project are tested to some degree in experimental systems such as CONNIVER, MLISP 2, POPLER 1.5. On the other hand, people connected with this project are involved in system implementation as well as in developing methods of its usage for solving practical AI problems. In particular these problems are connected with system analysis research for business and environment control, construction of information retrieval systems and natural language processing.

The necessity for intelligent systems of this kind is urgent, and even the prototype design will make a valuable contribution to the experience of constructing and usage of AI systems.

References

1. Bobrow D., Raphael B., "New Programming Languages for AI Research", ACM Computing Surveys, 1974,3.
2. Briabrin V.M., Serebriakov V.A., Yufa V.M., "Semantic Memory, Resolver and Metaprocessor in LORD system", Symbolic Information Processing, v.2, 1975, PP. 5-46.
3. Lavrov S.S., Silagadee G.S., "Input Language and Interpreter of the LISP-BESM-6 Programming System", Moscow, Academy of Solences of the USSR, 1969-
4. McDermott D., Suaaman G., "The CONNIVER Reference Manual", AI Memo 259, MIT, 1972.
5. Enea H., Smith D.C., "MLISP-2",

AI Memo 195, Stanford University, 1973.6.

6. Brlabrln V.M. et al., "On-line Translation and Debugging of Programs", Communications on Computational Mathematics, 9, Moscow, Computing Center of the Academy of Sciences of the USSR, 1974.

7. Brlabrln V.M., Pospelov D.A., "DILOS-Dlalog System for Information Retrieval, Computation and Logical Inference", Paper presented at the Workshop on Dialog Systems, IIASA, Vienna, June 1975.