BRAND X: LISP SUPPORT FOR SEMANTIC NETWORKS

Peter Szolovits and William A. Martin

MIT 1 aboratory for Computer Science and
MIT Artificial Intelligence Laboratory

ABSTRACT

BRAND X is a simple representation language implemented as a pure extension of LISP. BRAND X provides the following additional facilities over LISP: *Unique* and *canonical* structures, *property lists* for all objects, *labels* for all objects, and a syntax to express each of these, supported by a reader and printer. BRAND X is intended as an "assembly language" for representation languages, attempting to provide facilities generally found useful in the simplest manner, without any strong commitment to specific representational conventions. The fundamental ideas of the language are described, an overview of its features and notation is given, and some inherent problems of implementation and semantics are presented.

INTRODUCTION

The past decade has seen the introduction of numerous programming languages and representational languages specifically intended for use by the AI community. The early seventies saw the introduction of languages (e.g., PLANNER. CONNIVER. Q^4) which incorporated higher level data structures, novel invocation and control structures, context mechanisms, and rule-like representations of knowledge [7. 13. 12). Later research, focusing on problems of the meaning of representations, has led to a new group of languages (e.g., RRI . FRI . RI /ONE) emphasizing the structure of representations, multiple descriptions and viewpoints, "frame-like" systems, and procedural attachment |3, 4. 18] Despite the promise and even popularity of these languages, most AI programs continue to be written in that robust standby which precedes the above by at least a decade, LISP.

The attraction of LISP continues to be its great simplicity and mutability, allowing any user to build features of more power and incorporate them within the language The contrasting weakness of each of the above-mentioned languages is that they make representational and control commitments which, although appropriate to some applications which guided the development of the language, later appear arbitrary or wrong for other potential uses. To successive generations of AI researchers, it continues to seem more attractive to implement their own language extensions on top of LISP than to accept a complete package of conventions provided (or imposed) by the more recent language designers However, some generality docs exist among the facilities that have been repeatedly invented-roughly corresponding to the notion of *semantic networks,* if only as a medium of implementing *frame systems* or indexing structures for logic or production-rule based inference programs. It is obviously desirable to provide as powerful a set of support facilities as possible, without overstepping the bounds of commitment which may make the resulting language unacceptable.

Our interest in extending LISP arises from our recent experience in building large AI programs which represent knowledge in the form of semantic networks In addition to straightforward implementations of such data structures in terms of IISP symbols and property lists, we have also explored implementations built on discrimination networks of expressions [14]. a uniform indexable data type similar to LEAPtriples, and several generations of representation languages embedded within LISP (6. 15).

A useful base language for implementing semantic networks must have facilities for creating, retrieving and manipulating nodes and links among them. We have found that a consistent set of extensions to lisp which support unique expressions and the ability to attach properties to any object serves us well without overcommitment to representational conventions which arc still, the proper subject of various research efforts.

BRAND X has been designed and implemented with the above observations as its guiding principle: it attempts to provide a commonly useful set of facilities but to avoid decisions about how any specific features of semantic networks should be represented. Thus, it is meant to be an "assembly language" of representation languages, providing a bottom layer above LISP in terms of which other conventions will be defined and implemented

The recent lack of success of a second thread of development has also contributed to our desire to make BRAND X a simple extension to LISP rather than a self-contained programming language Although many proposed AI languages suggest that procedures as well as data should be encoded within the language itself, often that course has not been adequately supported Either the issues of how to represent procedures (and how to interpret them) have been put off, leaving the language dependent on its implementation environment (LISP) for procedural support, or a very complex interpreter has been created which has limited (if any) appeal to users other than the ones who developed the language As in the case of data structures mentioned above, the simplicity and power of LISP seem elusive to language builders.

BRAND X'S immediate predecessor is the XI MS language of Hawkinson [6]. which has been used by the authors and their colleagues both as a data base and as the implementation medium of owi [9. 15] BRAND \ was inspired by the observation that many of the facilities provided by XIMS appeared merely to duplicate features already adequately provided by MSP. suggesting that its function could be gteatly simplified by implementing instead a limited, general set of extensions to USP.

In summary, then, we lake the following position: Because LISP is so successful as a base language for AI research, try to build directly on it when creating higher-level representation languages In creating BRAND X. WC have chosen to add new data types and to extend the standard MSP interpreter to exhibit some reasonable behavior for these types In all this, however, wc have tried to assure that all of LISP remains intact Therefore. with virtually no exception, any valid LISP expression is a valid BRAND X expression, and its interpretation remains unchanged.

BRAND X IS implemented as an extension of LISP (MACLISP [11]), and supports the following additional features:

a. *unique and canonical list structures*— providing a data base facility in which expressions may be identical when composed of the same subexpressions,
b *universal property* list.t—permitting the attachment of property/value pairs to any object in the language,
c *labels*—providing an abbreviation for convenient reference to complex expressions,
d. *an extended LISP notation*—allowing a convenient mechanism for reading and printing any LISP and BRAND X structures,
e *triples—a* special, compact data type having three components (especially for the support of owi). which is optionally present in the language.

IMPLEMENTING SEMANTIC NETWORKS

The authors are sometimes asked *if* they can explain in a simple way the advantage of implementing a semantic network in BRAND X over implementing it directly in LISP 'Typically, a semantic network is a difficult data structure to read and print in LISP. It has backpointers which create circular structures.

and it is so strongly interconnected that given any piece of it to print,LISP punt functions often punt the whole network, in LISP the traditional way to meet these difficulties IN lo make every node of the network an atomic symbol, with the network links on its property list. Given an atom, LISP will print just us name, not tracing down the properly list, and thus not Imping through the whole network. There are at least four difficulties with this solution.

First, the names of the network nodes are often not meaningful. Especially if user programs create new network nodes, the nodes are typically given names such as G0001. G0002. etc These names do nothing to improve the intelligibility ol the semantic network Nodes created by the user can of course be given readable (if long) hyphenated names such as LEFT-TUSK Of-ELEPHANT-CLYDE-1. If such names are. however, uninterpreted by the system (as G0001 must be), then the user has no assurance that the given node name actually represents what its name implies Such assurance can be gained by explicitly checking that the relationships the node enters into are consistent with its name, but in that case the common information is redundantly stored in the network—once in the names and once in the links One possible solution occasionally proposed is to permit the system to extract that information from the node name when needed. Because names encoded as LISP atomic symlx>ls have no innate structute. this requires settling on a grammar of names and a parser for them- not an appealing strategy for the fundamental building blocks of a system In our view, the cncoding of some attributes ol a node in its name is quite appropriate; we depart from the solution criticized here chiefly in using innately structured objects for names.

Second, because the standard LISP printer will print only the name of a node, the programmer must generally write a set of special purpose print lunctions lo print just those links which are desired Such code must be carefully written to avoid printing semantic network loops. Although special primers for displaying interesting portions of any large collection of interconnected objects will often prove necessary, we have found that the standard printing action for BRAND is often adequate and is certainly useful for browsing.

Third, there is no declarative notation which the programmer can use to make the task of inputting a semantic network easy The programmer may again write special functions lot this purpose typically. the input notation as defined by these functions will differ from what is generated by the special-purpose print functions Thus, the application may lack LISP'S useful ability to read back whatever can be printed

finally, the atoms which represent nodes lake up quite a bit of memory space This space can be reduced somewhat if the atoms are not made unique in memory, but then one cannot refer lo a node by typing in its atom name.

In BRAND X. we provide the user with an alternative to using atoms for nodes and we deal with the above difficulties in a general manner The essence of the solution is to make it possible for data structures other than atoms to be unique and to have properties These then become an alternative to atomic symbols for representing nodes. A systematic approach to reading and printing all data structures is then provided. Other features and conventions for building semantic networks may also be useful, but the authors do not want to be committed to them at the implementation language (BRAND X) level [10J.

EQUALITY, IDENTITY, UNIQUENESS, COMPOSED OBJECTS

If semantic network nodes are to be named by decomposable objects, we must have the ability to write the same name more than once—thus, the ability to compose objccls in unique ways. This desire leads to an analysis of the meaning of "same" and points lo a reasonable solution and some troubling puzzles.

The traditional definition of equality holds that two objects arc equal if they are indistinguishable by any known test In that case, of course, one may as well speak of just one object, although various paradoxes based on that interpretation have been suggested ' In compulation, however, the above definition of equality is not the one usually favored This is because computer models of the real would often permit tests of distinguishabilily which are artifacts of the implementation, Typically, computer implementations can distinguish objccls based on then address in the memory of the computer; thus, two otherwise-equal objects may be distinguishable by being at different addresses For example, although we would like lo think of the two numbers 999 and 999 as equal, some LISP'S implementations lind them distinguishable under the EQ predicate, which tests equably of address.

One standard solution lo the undesirable nature of strict equality is to distinguish between *identity*—true indistinguishabilily—and *equality—now* taken to mean indistinguishable in the real world, even though distinguishable in the implementation, list's EQ and EQUAL predicates capture these notions of identity and equality, respectively.[2] *The* distinction between identity and equality is important not only for very significant efficiency considerations. but also because the ability of programs to cause side-elfects permits them to distinguish among EQUAL but non-identical (non-EQ) objects.

A second standard solution is to adopt a convention and mechanism for uniqueness, in which objects intended to be equal are indeed made FQ-i.e.. objects are made *unique* according to the equality criteria, so the system permits the existence of only the single unique representative of an (equivalence) class of EQUAL objects. LISP'S *interning* mechanism performs essentially this function for alomic symbols. This solution is motivated by the desire lo use EQ as the standard equality test, for the reasons cited in the last paragraph The additional effort needed on input to create or find the unique object implied by the one actually input is more than rewarded by the possibility of efficient algorithms based on the assumption that some class of EQUAl objects is indeed unique (EQ) [5]

The case of LISP'S handling of atomic symbols deserves investigation in its own right, as a guide to how interning is to be viewed in general. Suppose we intend LISP'S alomic symbols lo be the same whenever they are spelled the same Then the desired equality test on lisp atomic symbols is SAMEPNAMEP, which is true just if its two arguments arc atomic symbols which arc spelled identically. The LISP'S reader, using the system-provided INTERN function, chooses a unique instance of all symbols with the same spelling and assures that that same instance is read each time a symbol of the same spelling is input.[1] This assures that (ordinarily) all instances read of the same-spelled symbol are EQ—thus, alomic symbols are unique.

lo understand how we might define uniqueness for composite objects, consider in detail the ideal nature of the INI E RN function, formally. INTERN ts a function which maps atomic symbols to atomic symbols such that its result is the representative of the equivalence class into which its argument falls when the set of alomic symbols is petitioned by the function SAMEPNAMEP. If LISP'S interned all atomic symbols, we would say that

- atomic symbols are unique (with respect lo EQ) under the predicate SAMEPNAMEP.

We lake all objects to have certain characteristics known as *criterial. These* are the characteristics used by the partitioning predicate of an interning scheme for example, the spelling of an atomic symbol (the sequence of characters in its written form) is criteria!, but other characteristics such as its value, properties, and address are not. in the above interning scheme.

BRAND x introduces two new list data types with different criteria for uniqueness: ULISTs *(Unique* LISTs) and CLISTs *(Canonical* LISTs). The fundamental interesting characteristics of these types are that:

- ULISTs arc unique (with respect to EQ) under the condition that their criteria! components (CAR and CDR) are identical (EQ). and
- CLIST's are unique (with respect to EQ) under the condition that their criterial components (CAR and CDR) are equal (EQUAL).

Of these types. CLISTs are the easier to comprehend because they have the properly that "EQUAL imp 1 i es EQ." Any canonical combination of canonical objects yields another canonical object and EQ tests are sufficient to determine equality. The other type. ULIST. is one of a possibly large number of other useful types which partition the universe of expressions by a predicate stronger than EQ bul weaker than EQUAL. ULISTs are particularly useful for enabling the user to re-create the same (FQ) structure on a second unique construction of the same pair of non-canonical objects. Such an ability may not be very important, though it has been proposed for example as a way of associating values with previously evaluated expressions [Global]. BRANDX automatically creates canonical rather than unique structures when all components of the structure are canonical, and under the control of a flag may always construct canonical structures.

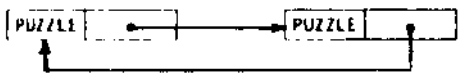
BRAND \ also provides a new data type TRIPLE, and notions of uniqueness and canonicity for it A triple is a compound object with three

criterial parts, its ILK, TIE, and CUE. Just as ULISTs and CLISTs are defined for list structure, BRAND X defines UTRIPLEs and CTRIPLEs.[4]

Other MACLISP data types (arrays and hunks) are considered innately unique and canonical.[5]

A Puzzle

The essence of BRAND X canonical structures is that "EQUAL implies EQ." Yet is this a consistently tenable formulation? Consider the list structure P.



which could he created by evaluating the LISP'S expression

```
((LAMBDA (X) (RPLACD (CDR X) X))
      (LIST 'PUZZLE 'PUZZLE)).
```

BRAND X provides the function CANONICAL which, given any object as argument, returns its canonical representative. (It is the canonical version of INIERN) What are we to expect if we hand P to CANONICAL[7] The result (let us call it *PP)* should clearly be a canonical list structure whose CAR and CADR arc both PUZZLE. whose CDDR is EQ to itself, and whose CDR is not. However. (CDR PP) should obviously be EQUAL to PP itself, yet the two structures cannot be identical without doing violence to our intuition that the canonical form of a two-cycle should not be a one-cycle.[6]

The simplest solution to this puzzle is to claim that EQUAL cannoi be applied to structures such as P or PP because LQUAL is defined as a recursive tree-comparison algorithm and is therefore inapplicable to circular structures. Although this is certainly true in use (e.g.. (EQUAL P (CDR P)) will lead to looping). we feel that this begs the question—it reflects EQUALS failure to capture our motion of equality rather than a true solution.

A more attractive solution would be to follow the definition of CANONICAL as the parallel of INTERN LISP'S INTERN does not guarantee that only a single atom with the same name is ever created; it merely chooses one as the representative of the equivalence class and maps all others onto that We could act similarly in this case and sav that PP and (CDR PP) both fall into the same equivalence class and CANONICAL chooses, say. PP as the representative Then

```
(CANONICAL (CDR PP)) -> PP
```

This proposal has a grave problem, though The point of having canonical structures is to enable us to make use of the "EQUAL implies EQ" rule. Yet now PP and (CDR PP) *cannoi* be EQ. and mapping the latter onto the former by CANONICAl is little or no help In the case of INTERN, the mapping onto the representative *was* useful because it was done consistently on input (by READ) and because LISP does not ordinarily create non-interned atomic symbols. By contrast, simply taking the CDR of PP retrieves the non-canonical instance of PP The canonicalization could be performed at "run time" by requiring CAR and CDR to yield the canonical versions of components of canonical structures, but at a very high cost.

Currently Brand x \ therefore adopts no solution—such circular structures cannot in general be made canonical[7] The puzzle presented here, however, is interesting not only for its difficulties for Brand x. We observe that no formal scheme can permit the specification of cyclic structures and also enforce * EQUAL implies EQ." .

BRAND X LANGUAGE OVERVIEW

For a complete description of BRAND X. we refer the reader to the BRAND x Manual [16]. In this section we merely summarize the facilities provided for support of unique and canonical objects, for generalized properly lists, and for reading and printing.

Structure Building, Decomposition and Testing

The fundamental construction operators of Brand x are unique and canonical versions of LISP'S CONS UTONS and CCONS For support of triples, the functions TRIPLE. UTRIPLE and CTRIPLE serve the analogous purpose. CAR and CDR decompose lists. ultsts and clists. and ILK. TIE and CUE decompose all kinds of inples Higher level constructor functions such as LISP'S LIST and LIST* are also provided in unique and canonical versions. Predicates to test an object for being of a particular type are also given; the general predicate BRAND-X-OBJECTP yields the type (ULIST. CLIST. TRIPLE. UTRIPLE. or CTRIPLE) of any of the HKAND x lypcs and NIL for anything else.

An existing unique or canonical object may of course be found by simply re-creating it: iLs uniqueness assures that no second copy is made. This technique cannot be used merely to discover if the object already exists, because it has the effect of creating it if it does not exist[8] HKAND X therefore provides a syntactic form. KNOWN, which causes its argument to be retrieved if already present but NIL to be returned if the argument is not present. LISP'S *buckquote* syntax [9] is especially useful in this application, allowing us to write (KNOWN [.A . B]) to find out whether any unique list structure whose CAR is the value of A and whose CDR is B has been constructed.

Properties

BRAND X extends the lISP notion of properties to all objects of the language, including all MACI ISI* data types and other HKAND X objects. Thus. *any* object may ha\e a property list (2) The functions GIT P. GETPL. PUTP. REMP. PROPLIST. and SETPROPLIST arc extensions of and subsume the corresponding MACLISP'S functions GET. GETL. PUTPROP. REMPROP. PLIST. and SET PLIST'[0]

In addition to the above, we have also found it useful to support explicitly the manipulation of pmpcnics which have lists as their values. The functions ADDP and DELP add and remove one value to or from a list of values on a ptoperty of any object New values arc added at the front of the list, and duplicates are deleted Equality checking for duplicates and for deletion is bv *LQ.* The motivation lot these functions is that we wanted to provide a syntax (see below) which allowed the expression of multiple values for properties such as DAUGHTERS or CHARACTERIZATION.

Each call to UCOK'S. CCONS. UTRIPLF and CTRIPLE causes the created cell to be added under the CAR-1 (inverse CAR) property of its CAR (ILK. in the case of triples) This structure is necessary to maintain uniqueness and canomcity. and is also often useful for other indexing and searching applications. The function CAR-1 retrieves the list of objects so indexed under its argumenL

L abels

Any R R W D x object may be labeled. In a system in which names are typically constructcd by composition of other names, a facility for abbreviating names is essential to avoid printing an expression of size at least 2^n for objects built by n compositions. By convention, labels are most often atomic symbols, although any object may label any other The assignment of labels is intended to be permanent, and numerous implementation problems result if labels are reassigned The relationship between a label and the object it labels is not to be viewed as the relationship between vanables and values. Both the label and the object are read as the identical entity.

A label may be used in HKAND X before the expression it labels has been defined, and an expression may also be used and later have a label assigned to it However, problems again arise if both the label and the expression are used before they are associated.

Unique and Canonical

The functions UNIQUE and CANONICAL arc provided to return the unique and canonical versions of their inputs. Listed here are

1. the BRAND x types to which each function applies.
2. the components of those types considered criteria! by the function.
3. the equality predicate used on objects of that type, and
4. a description of the resulting value returned by the function.

UNIQUE

atomic symbol print name SAME P NAME P

Atomic symbols are made unique by the list' primitive INTERN. The representative element of the equivalence class under SAME P NAME P is the first symbol seen with that name.

number — type and value — EQUAL

Numbers are interned by NINTERN, which uses a hashing scheme defined by BRAND x and also used by the property list mechanism. The representative is the first number of that type and value which is NINTER Ned.

LIST or ULIST — CAR and COR — EQ-PARTS

Yields the unique ULIST (or CLIST if the components are canonical or the flag CCONS is non-NIL) formed of the CAR and CDR of the inpuL If the input is a ULIST, it is unchanged. If a LIST, its CAR and CDR are UCONSed. The representative element cannot be EQ the first LIST made unique because BRANDx implements LISTs and ULISTs as different LISP data types.

TRIPLE or UTRIPLE — ILK, TIE and CUE EQ-PARTS

Yields the unique UTRIPLE (or CTRIPLE if the components are canonical or if the (lag CCONS is non-NIL) formed of the ILK, TIE and CUE of the input. If the input is a UTRIPLE, it is itself returned If it is a TRIPLE, its components are combined by UTRIPLE.

CLIST or CTRIPLE — self — EQ

CLISTs and CTRIPLEs are innately unique because there is no mechanism for forming other copies of them —EQUAL implies EQ.

other — self — EQ

Hunks, arrays, and other USP data types are considered to be unique, forming singleton classes under the chosen equivalence relation. A relatively straightforward extension of the ideas used for the creation of unique list structure could also be applied to hunks and arrays, but this has not been done in nRAND x.

CANONICAL

atomic symbol — print name — SAMEPNAMEP

Atomic symbols are made canonical by the USP primitive INTERN. The representative element of the equivalence class under SAMEPNAMEP is the first symbol seen with that name. This is the same as under UNIQUE.

number — type and value — EQUAL

Numbers are interned by NINTERN. which uses a hashing scheme defined by BRAND x and also used by the property list mechanism. The representative is the first number of that type and value which is NINTERNed. This is the same as under UNIQUE.

LIST. ULIST, or CLIST — CAR and CDR — TQUAL

The canonical representative of any form of list is a Cl. 1ST. The canonical form of a CLIST is itself. That of a LIST or ULIST is the *CCONS* of the CANONICAL versions of its CAR and CDR. Note that this may require the complete copying of LIST or ULIST structure into CLIST structure. CLISTs are easily recognized in the implementation, so that EQUAL tests can be performed by EQ and canonicalization is trivial.

TRIPLE, UTRIPLE or CTRIPLE — ILK. TIE and CUE EQUAL

The canonical representative of any form of triple is a CTRIPLE. The canonical form of a CTRIPLE is itself. That of a TRIPLE or UTRIPLE is the CTRIPLE of the CANONICAL versions of its ILK. TIE and CUE. Note that, as for list structure, this may require the extensive copying of TRIPLE and UTRIPLE structures. EQUAL tests on CLISTs may also be performed by EQ.

other — self — EQ

Hunks, arrays, and other LISP data types are considered to be canonical as well as unique, forming singleton classes under the chosen equivalence relations. A relatively straightforward extension of the ideas used for the creation of canonical list structure could also be applied to hunks and arrays, but this has not been done in RRAND X.

The predicates UNIQUEP and CANONICALP test whether their arguments are unique and canonical respectively. The meaning of these predicates is that any object which passes one would itself be returned by the functions UNIQUE and CANONICAL. INTERNP and NINTERNP are special cases of these for atomic symbols and numbers.

NOTATION

One of the powerful simplicities of USP is that, on the whole, any object may be printed out in such a way that it can later be reconstituted by reading in that printed representation.[11] This notion is preserved and extended to BRAND x objects.

Lists, Ulists, and Clists

Our goal has been to preserve USP syntax as much as possible. Therefore, LISTs and CONSes may be formed as in LISP:

reading (A . B) is equivalent to evaluating (CONS 'A 'B), and
reading (A B C) is equivalent to evaluating (LIST 'A 'B 'C).

ULISTs and CLISTs are written in a manner similar to LISTs, but with square brackets instead of parentheses. Thus,

reading [A . B] is equivalent to evaluating (UCONS 'A 'B), and
reading [A B C] is equivalent to evaluating (ULIST "A 'B 'C).

ULISTs. when they are composed of non-unique components, are shown in the appropriate mixture of parentheses and square brackets. Thus, for example,

reading [A (B)] is equivalent to evaluating (UCONS 'A '(B)).

Canonical structures must always be composed only of canonical substructures; thus, their printed representation is free of parentheses. In contrast with the above example,

(CCONS *A '(B)) yields [A B].

Triples, Utriples, and Ctriples

In a manner analogous to lists, a notation is defined for triples. The fundamental triple notation looks like a list of three elements, the ILK, TIE and CUE. but with an asterisk between the ILK and TIE. Thus,

reading (A*B C) is equivalent to evaluating (TRIPLE *A 'B 'C).

Similarly,

reading [A*B C] is equivalent to evaluating (UTRIPLE 'A 'B 'C),

and CTRIPLEs are formed when each component of a UTRIPLE is canonical.[12]

Labels

Labels are assigned by preceding the criteria! part of the expression by the label and the symbol ''=". (Note that this is within the brackets delimiting the expression.) For example,

[AN-EXAMPLE = THIS IS AN EXAMPLE]

assigns to the canonical four-list [THIS IS AN EXAMPLE] the label AN-EXAMPLE.

A label is used by prefixing it with an exclamation point in the syntax. Thus, after the last example,

[NOW . IAN-EXAMPLE]

is entirely equivalent to

[NOW THIS IS AN EXAMPLE],

and will be printed in the shorter form.

Properties

LISP does not provide any explicit syntax for the assignment or display of properties. In BRAND X. within the square brackets or parentheses used in writing an expression, the criteria] expression may be followed by any number of property assignment clauses. Each is of the form:

an ampersand (&), followed by the property indicator, followed by any number of values.

The values are added (via ADDP) so that they appear in the order given in the syntax Thus, if [BALL 1] has no COLOR property to begin with, then after

[BALL 1 &.C0L0R REO GREEN BLUE],

we have

(GETP '[BALL 1] 'COLOR) ■> (REO GREEN BLUE).

To support effective optional cross-indexing in the data base, BRAND x permits the specification of both forward and reverse properties at the same time. To specify a reverse property link, follow the initial ampersand and property by a second ampersand and property, before the values. For example, after

[BALL 2 &C0L0R &HAVING-THIS-COLOR RED WHITE],

we have

(GETP '[BALL 2] 'COLOR) -> (RED WHITE), and
(GETP 'RED 'HAVING-THIS-COLOR) => ([BALL 2]).

In LISP, an expression such as (. X) is syntactically invalid, as it appears to CONS nothing onto X. In BKAND X. however, we interpret that form as equivalent to just X." This provides a syntactic means of attaching ptopertics (and labels as well) to any BRAND X object. For example, we use the following notation to attach POSSIBLE-VALUES lo COLOR:

```
[. COLOR &POSSIBLE-VALUES
        RED ORANGE YELLOW GREEN
        BLUE INDIGO VIOLET]
```

Anaphora

We often find it convenient to refer to parts of an expression as that expression is being written. In specifying the representation of a frame in a semantic network, for example, wc may need to refer to the subject role of the frame in close proximity lo our specification of the expression representing the frame itself. For example,

```
[RUN INTO TROUBLE &R0LES
        [SUBJECT [RUN INTO TROUBLE]
            &C PERSON]]
```

lo indicate that *run Into trouble* has a subject role and that whatever satisfies that role must also satisfy the characterization *person.* Note that, after the above,

(GETP '[RUN INTO TROUBLE] 'ROLES)

yields

([SUBJECT [RUN INTO TROUBLE]]).

It is undesirable lo have to repeat the expression [RUN INTO TROUBLE] each time a role of that frame is to be specified or referred to. Instead, BRAND x allows us to write

```
[RUN INTO TROUBLE &R0LES
        [SUBJECT : &C PERSON]],
```

which is read identically with the expanded form above. Here, the colon (:) acts us an anaphor. referring to the criierial expression which is one level of parentheses or brackets out from the appearance of the colon.

BRAND X supports a general facility for anaphora, expressed via successive colons not separated by space. The number of colons specifies the number of levels of parentheses and brackets to move out to find the anaphor being referred to. For example, the (PERSON : :) in

```
[RUN INTO TROUBLE &R0LES
        [SUBJECT : &C (PERSON : : ) ] ] .
```

stands for (PERSON [RUN INTO TROUBLE]).

Spaces are normally insignificant in BRAND X except lo delimit atomic symbols. In the case of colon anaphora, however, spaces may not be placed between the colons. Thus, in the above, if we had written

(PERSON : :)

instead, it would have been read as

(PERSON [SUBJECT [RUN INTO TROUBLE]] [SUBJECT [RUN INTO TROUBLE]]).

When writing non-unique list structure, the colon anaphor is not only a convenience but more essential, because rewriting an expression cannot create uniquely the same one as a previous instance. Thus, if we wished to form a structure like that of the simpler example above, but from non-unique lists, wc could write

```
(RUN INTO TROUBLE &R0LES
        (SUBJECT : &C PERSON)),
```

which is not equivalent lo the fully written-out version

```
(RUN INTO TROUBLE &R0LES
   (SUBJECT (RUN INTO TROUBLE) &C PERSON)),
```

because the two expressions (RUN INTO TROUBLE) are not EQ in the second case.

Anaphora provide a form of local labeling. They also permit the printing of circular structures, and are capable of extension to permit reading of all such structures as well.[14]

Syntax

To recapitulate the syntax of BRAND X formally, we present an extended BNI description:

```
<x-expr> ::■ <Lisp-atom> | (<x-expr-body>) |
            [<x-expr-body>] | <1abel-spec> |
            <colon-anaphor> | <quoted-form> |
            <backquoted-form> | <comma-form>
<x-expr-body> : = {<label.x-expr>*} <criterial-expr>
                  {<prop-specs>}*
<criterial-expr> ::= {<x-expr>}* {.<x-expr>} |
       <ilk :x-expr>*<tie: x-expr> <cue:x-expr>
<nrop-spec> : : * &<prop:x-expr> {&prop.x-expr>}
                {<va/:x-expr>}+
<label-spec> :: = !<x-expr>
<colon-anaphor> ::■ {:}+
<quoted-form> ::• '<x-expr>
<backquoted-form> ::■ *<x-axpr>
<comma-form> :* ,<x-expr>
```

In the above, we have used braces ({ ... }) lo indicate optional phrases, and + as superscripts on optional phrases to indicate zero or more and one or more permitted repetitions, respectively. Melasyniaclic variables are in angle brackets (< ... >); such a variable of the form <name:type> is an expressive variant of just <typc>—thus. <prop:x-expr> is merely <x-expr> with a suggestion of its meaning as a properly descriptor. Spaces arc not significant, except as separators and as noted above between successive colons in colon anaphors. Note that although any x-cxpr is acceptable as a label, by convention we will use only atomic symbols. Quotation is as in LISP, SO that 'X is a convenient abbreviation of (QUOTE X).

Backquoted forms and comma forms require a little explanation because, although they arc commonly used in MACLISP, they are not innately part of LISP. They provide a facility for abbreviating programs which are to construct structures of a particular form. For example, the form

(A B (C ,D) ,E (F G))

is read as

(LIST 'A 'B (LIST 'C D) E '(F G)).

Variable parts of such a structure are preceded by a comma. Other parts are quoted if they are constants or formed up by the appropriate BRAND X operation.

IMPLEMENTATION

BRAND X is implemented in MACLISP as a number of functions, hooks into the reader, printer and evaluator which are provided by the base language, and objects implemented as MACLISP HUNK structures.

Representation of Objects and Properties

Unique and canonical lists and all triples are represented by MACLISP object of type HUNK 4. For list types, the CAR and CDR of the hunk are the CAR and CDR of the object, and a third component identifies the object as a BRAND x entity and holds its property list. For triples, the hunk contains the ILK, TIE and CUE of the triple as well as the component which identifies its type and holds its properly list. The ILK of a triple is the CAR of its hunk. This is the only storage structure decision allowed to be visible at the language level, because it allows efficient algorithms which use the CAR-1 function to retrieve unique or canonical objects whose CAR or ILK is the given object

All unique and canonical objects are indexed under the CAR-1 property of their CAR or ILK; this is needed to enable BRAND X lo find such an object given its components. This indexing is not symmetrical between the CAR and COR; therefore, finding such an object among many with the same CAR is potentially costly. At the cost of additional overhead in storage, a CDR-1 property could also be maintained and the search could be optimized. There is also other independent motivation for that property, to permit a user-level CDR-1 function, but we have so far not been persuaded to pay the storage cost for its possible benefits Label attachments arc recorded under the LABEL and LABEL-1 properties of the object and label, respectively.

Unfortunately MACLISP does not provide any way lo prevent the components of user-defined types from being examined and altered by the system's primitive functions. The integrity of a BRAND X data base can be seriously dis-

rupted by indiscriminately applying operations such as RPLACA and RPLACD to unique or canonical structures. Similarly, careless changing of the properties used by BRAND X can lead to trouble. Were HRAND X implemented in a strongly-typed language with good modularity and information-hiding practices (e.g., CLU [8]). these problems could be avoided. Our planned implementation on the Lisp Machine will also be able to avoid them by similar means.

Numbers arc made unique by a hashing scheme which uses MACLISP'S SXHASH function. The hash table is also used to store the numbers' property lists. This method assures that EQUAL numbers, which are not generally EQ in MACLISP, can share properties. The primitive type of the numbers matters, and the fixed number 1 is different from the floating number 1.0.

Other non-atomic-symbol MACLISP objects, including normal list structures, are associated with properties by another hashing scheme based on the object's address. Only those objects arc in the table which have properties, and the storage cost here is minor.

Interactions with the Lisp System

MACLISP gives primitive support for user-defined extended data types built as HUNKs The LISP primitive functions EVAL, PRINT. EQUAL, SUBST and SXHASH, when applied to a hunk which is identifiable as an instance of a user-defined extended data type, call a handler function instead of performing their normal action. This mechanism is used in BRAND X to permit a completely integrated use of BRAND X objects in LISP code and data.

The LISP printer has been augmented and put under the control of several flags. Printing in LISP'S READ-EVAL-PRINT loop uses a pretty-printer which displays not only a formatted version of the expression to be printed but also its label, if present, and its properties Anaphora are also used to abbreviate printout and to print circular structures.

The LISP reader has been augmented by the definition of reader-macro functions for characters such as "(" and "[" which direct the parsing of input expressions as specified in the description of the notation, above. The primitive LISP reader continues to be used for reading symbols and numbers. Implementation of the reader, especially because of difficult interactions between the use of anaphora and backquote. was especially difficult. Indeed, although we have been able to identity algorithms lor reading at least non-canonical innately circular structures, we have not yet implemented them because of their difficulty.

In the course of developing BRAND X syntax, we have also designed a reader extension for MACLISP to allow the mapping of different actual characters onto the *virtual character set* of the language. In these terms, HRAND X expressions are read in terms of virtual characters such as BEG IN-UNIQUE - LIST and PROPERTY-INDICATOR, which are mapped in the standard reader environment onto the characters "[" and '&". Such a mechanism, called an *ear* afler a suggestion of Gerald J. Sussman. permits various LISP extension packages to be used together without causing syntactic conflicts; all that is required is to map the total set of logical characters needed in an application onto the available physical character set. This facility is being implemented by Glenn Burke both for MACLISP and the Lisp Machine.

Evaluation

What should it mean to evaluate (in the sense of LISP'S EVAL) various BRAND X objects? By default, we treat evaluation of unique and canonical list structure just as if the lists were normal LISP lists. Thus, evaluating the expression

```
[CAR [QUOTE [A . B]]]
```

is no different from evaluating

```
(CAR (QUOTE (A . B)))
```

Again by default, we say that EVAL cannot be applied to a triple; it yields an error.

Our intent is that this hook into the action of LISP'S evaluation mechanism should serve as the basis for a more sophisticated evaluator to work on declarative structures of greater complexity than LISP s-expressions. An experimental version of such an interpreter has been programmed by Glenn Burke.[15] providing pattern-directed invocation of procedures by examining the database of methods. Triples are used to express a hierarchic a-k-o structure in which methods for evaluation are inherited. More specific ideas on how such interpretation ought to be organized will be given elsewhere [10), and other interpreter implementations are planned. One important criterion of these is that they arc to interface consistently with LISP'S normal action of evaluation—evaluating a simple LISP expression should be the same in HRAND X as in LISP.

ANOTHER PUZZLE—WITH LABELS

Included here is a short discussion of a number of problems which can arise in the use of labels. These problems arc not easily solved, and arc solved not at all or only badly by the current implementation of BRAND x.

The intent of a label is to be simply an abbreviation for the object it labels. If labels were used only afler the object they labeled was created, and if labels were never reassigned, then these problems would not arise. However, because of the possible need for mutual recursive reference in data structures, and because of the more frequent need lo refer to something in an interactive environment before having completely defined it. labels do get used before they arc assigned. The implemcntor of a system must choose some representation for an unassigned label and must decide how such an object can be used. One possible choice is lo ban all use of such objects, but this fails the criteria outlined above and is also difficult lo implement in a language like LISP, in which information hiding is impossible. Another choice allows reference to these objects but not an examination of their components. This would permit the use of unassigned labels in constructing other objects, but would prohibit asking for. say, the CAR of such an object This is also impossible to implement in Lisp, and any attempt to enforce such conventions systematically would be very expensive; many algorithms can be significantly speeded up if they need not handle a special case of unassigned labels because the implementor knows that the test already in use in (he algorithm will al..o succeed or fail appropriately for these. Therefore, BRAND X permits the examination of the representation of unassigned labels, which have a standard internal structure.

However, the use of labels before they are assigned creates very serious problems:

If an object and an unassigned label are both used to construct other objects, and then if that label is assigned to that object, then there is in general no good way lo assure that all previous uses of the two will indeed refer lo the identical object. Thus, formerly-made references lo the label will not be EQ to formerly-made references to the object This could be fixed only by an elaborate indexing mechanism which keeps track of the use of all unassigned labels, or by an alternative mechanism which scans the entire data base for uses of the label (references lo its dummy object) and replaces them with the actual object. An alternative, possibly available on different computer architectures, would be to define EQ to follow data indirections ("hidden pointers") before making address comparison tests, but this is not generally feasible. This problem is easily avoided (as it is in the current HRAND x implementation) if the label is unused before its assignment; then, the dummy object is never crealed. If the label has been used, but the object to which it is assigned has not yet been created, the problem is also avoidable if the object can be created "on top of the dummy object representing the label. This is done in the current BRAND X whenever it can be; it fails if the underlying LISP data types of the intended object and the dummy object are distinct—e.g.. if a formerly used dummy label (whose default is created as a UCONS or UTRIPLE. with underlying LISP data type HUNK4), is then assigned to a LIST.

Canonicalizaiion of data structure can fail because some identity that depends on label assignment may not be known when it is computed. For example. [A B C] and (CCONS 'A ' IFOO) may appear to have little in common; yet. if [B C] appears and [F00 = B C] is later done, the two expressions are seen lo be identical However, the second canonical structure was created before this was known, and cannot be simply made lo be EQ lo the first The same problem also appeal's when labels are used as a mechanism for creating circular structures For instance, afler [F00 = BAR . [FOO], we have a new structure whose CAR is BAR and whose CDR is itself. Repealing this operation with different labels will create distinct such structures, although of course there should be only one because it is canonical. This is again the problem we have described in our earlier discussion of the puzzle of the canonicity of circular structures.

CONCLUSION

We have described a limited set of extensions to LISP to support its use for implementing semantic networks. The principal features given in BRAND x are the ability to create unique and canonical list and triple structures, the availability of property lists on all objects, labeling of any object, reading and printing of objects and their properties, and the integration of BRAND X objects into the standard LISP environment so that they appear as first-class data types to the BRAND x user.

This package is available in the MACLISP environment and will shortly be made available on the Lisp Machine. It is currently being used as the representation medium for an English Query System EQS and several smaller projects *of* the authors and our colleagues. So far, it has satisfied its design mandate, to provide us a useful set of extensions to our favorite implementation language, LISP, without overcommitting us to decisions best left to a higher level of language design.

ACKNOWLEDGMENTS

The authors would like to thank Mr. Glenn S. Burke for his invaluable advice and programming help in the construction of BRAND X. especially in integrating it into the MACLISP system. This research was supported (in part) by the National Institutes of Health Grant No. 1 P01 LM 03374-02 from the National Library of Medicine, and by the Defense Advanced Research Projects Agency (DOD) monitored by the Office of Naval Research under Contract No. N00014-75-C-066L

NOTES

1 For example. Black suggests (1) that we consider a universe consisting only of two indistinguishable balls in orbit about each other We have absolutely no features to tell "which ball is which." yet wc would hesitate to claim that the two balls are the same

2 Roth are defined more technically, of course E0 is defined as identity of address, and EQUAL is defined as a recursive test which checks for the identity of primitive onjects and me equality ot constituents of compound objects EQUAL is further modified so that equality of numbers is tested by idenUty of type and numerical equality *{x — y* = 0) of the values.

3 Even this has exceptions, as USP's INTERN actually stores its representative instances in an OBLIST or OBARRAY, of which multiple versions may be maintained In addition, some LISP functions can create atomic symbols without interning them To add to the possible confusion. USP's definition of EQUAL unfortunately yields false when two symbols arc SAME PNAME P but not EQ.

4 B R A N D X exists in versions which either include or do not include support for triples Such support may be omitted simply to save a small amount of the (virtual) memory space used by the system for large applications which do not require the type Although this was initially an important design consideration for us. experience indicates that all significant applications use triples The remainder of this discussion will assume that they are present

5 There is no reason why notions of uniqueness and canonicity could not be extended to these other structured objects as well, but this was deemed unnecessary and is not currently done The extension would be done in analogy with the definitions for list structure and triples (two- and three-element structures), although more efficient indexing schemes may be needed for larger structures.

6 This issue is equivalent to the previously-cited question of whether two identical objects can even be spoken of as two objects There is almost a formal equivalence between this puzzle and the problem of the universe of two balls

7 Note that this applies only to structures which are circular in that they contain themselves as their own critenal parts Circularity in the more conventional form of self or mutual reference by links is fully supported.

8. This problem is familiar to all LISP users who have tried to tell whether an atomic symbol has already been interned; merely typing the symbol interns it

9. See the later discussion of BRAND X syntax. A formal definition may be found in the Lisp Machine Manual [17).

10 It might perhaps have been better to redefine the original functions and thus to avoid having both GET and GETP We decided, however, to leave the original functions unchanged so that users depending on their error detection capabilities would not be misled

11 This is not completely true, as some data types (e.g. arrays in MACLISP) have no printed representation, and furthermore, circular structures (those which include themselves as a part) cannot normally be printed

12 Note that in a BRAND X without triples, the asterisk has no special significance and. for example. (A*B C) would be read as a list of two atoms. A*B and C.

13 The rationale for this is that MACLISP'S LIST* function, which forms successive conscs of its arguments (eg.. (LIST* 'A B 'C) is equivalent to (CONS 'A (CONS 'B 'C) . which is of course (A B . C)), yields just its single argument if given only one argument

14 Technically, the formation of truly circular expressions presents some difficulties more severe than those encountered in forming structures that arc circular through property attachments lor example, in forming the structure [A [B :]] (whose CADADR is CQ to itself), we appeal to need the whole structure before wc can form its substructure Because of the difficulties noted earlier. BRAND X docs not now support an input syntax for circular structures of the kind in which an expression is its own subexpression If such an expression is formed (e.g. by RPLACA), however, the printer will punt it with anaphora

15 Private communication

REFERENCES

1. Mack, M . "Identity of Indisccrniblcs." *Problems of Analysis: Philosophical Essays,* Greenwood Press. (1954)
2. Bobiow. D G.. A Note on Hash Linking." *Comm. ACM* 18. (7) (July 1975). 413-415
3. Bobrow, D G.. and Winograd. I . *An Overview of KRl.. a Knowledge Representation language,* Technical Report AIM-293. Stanford Artificial Intelligence Lab., Stanford. Ca . (1976)
4. Goldstein. I P. and Roberts. R B. *NUDGE. A Knowledge-fiasco" Scheduling Program,* Al Memo 405. MIT Artificial Intelligence lab, Cambridge. Mass. (1977)
5. Goto, F , *Monocopy and Associative Algorithms in an Extended lisp.* Technical Report 74-03. Information Science laboratories, Faculty of Science. University of Tokyo. Tokyo. Japan, (May 1974)
6. Ilawkinson, L B. *XLMS: A linguistic Memory System,* TMI73, MIT lab for Comp. Sci. Cambridge. Mass. (1980)
7. Hewitt. C . *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot,* AF TR-258, MIT Artificial Intelligence lab. Cambridge. Mass, (1972).
8. Liskov. B H.. Snyder, A. Atkinson. R, and Schaffen. C. "Abstraction Mechanisms in CLU.' *Comm. ACM* 20. (8) (August 1977). 564-576
9. Martin. W A., *Roles, co-descriptors, and the formal representation of quantified English expressions,* TM-139, MIT Lab for Comp Sci.. Cambridge. Mass.. (September 1979)
10. Martin. W A and SzoloviLs. P.. *Semantic Networks in Lisp fundamental concepts and a specific implementation,* MIT lab for Comp Sci. Cambridge. Mass., (in preparation)
11. Moon D A. *MACLISP Reference Manual* MIT lab for Comp Sci. Cam bridge. Mass. (1974)
12. Rulifson, J.E . Derkscn. J A and Waldinger, R J , *QA4: A procedural calculus for intuitive reasoning.* Technical Note 73, SRI International. Menlo Park. Ca. (November 1972)
13. Sussman. G J, and McDermott. D. V . "From PLANNFR to CONNIVFR-A Genetic Approach, *Proceedings of the 1976 fall Joint Computer Conference.* AFIPS Press, (1976). 1171-1179
14. SzoloviLs, P. and Pauker. S G, "Research on a Medical Consultation System for Taking the Present Illness," *Proceedings of the Third Illinois Conference on Medical Information Systems.* University of Illinois at Chicago Circle, (November 1976)
15. Szolovits, P . Hawkinson, L., and Martin, W A , *An Overview of OWL. a Language for Knowledge Representation.* MIT/LCS/TM-86. MIT lab for Comp Sci. Cambridge. Mass. (June 1977), also in Rahmstorl. G . and Ferguson, M.. (Fd$). *Proceedings of the Workshop on Natural language Interaction with Databases.* International Institute for Applied Systems Analysis. Schloss laxenburg. Austria. Jan 10, 1977
16. S/olovits. P, and Martin. W A , *Brand X Manual. TM* 186. MIT lab. for Comp Sci. Cambridge. Mass. (Nov 1980)
17. Weinreb. 1). and Moon, D. *Lisp Machine Manual.* MIT Artificial Intelligence Lab, Cambridge, Mass. (Jan 1979)
18. Woods. W A . *Research in Natural language Understanding; Annual Report.* Report No 4274, Bolt, Beranck. and Newman Inc , Cambridge. Mass., (1979), See especially Chapter 2, An Introduction to KLZONE", pp 13-46