MIT/LCS/TM-186

BRAND X MANUAL

Peter Szolovits
William A. Martin

November 1980

Brand X Manual

Peter Szolovits
and
William A. Martin

Laboratory for Computer Science
and
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

November, 1980

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), and monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

Table of Contents

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.

1. 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, QA4) which incorporated higher level data structures, novel invocation and control structures, context mechanisms, and rule-like representations of knowledge [8, 14, 13]. Later research, focusing on problems of the meaning of representations, has led to a new group of languages (e.g., KRL, FRL, KL/ONE) emphasizing the structure of representations, multiple descriptions and viewpoints, "frame-like" systems, and procedural attachment [2, 12, 16]. 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 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 does exist among the facilities that have been repeatedly invented. It is obviously desirable to provide as powerful a set of such facilities as possible, without overstepping the bounds of commitment which make the resulting language unacceptable.

BRAND X has been designed and implemented with these observations as the guiding principle. Its immediate predecessor is the XLMS language of Hawkinson [7], which has been used by the authors and their colleagues both as a data base and as the implementation medium of OWL [9, 15]. BRAND X was inspired by the observation that many of the facilities provided by XLMS appeared merely to duplicate features already adequately provided by LISP, suggesting that its function could be greatly simplified by implementing instead a limited, general set of extensions to LISP.

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 strucutures, and it is so strongly interconnected that given any piece of it to print, LISP print functions often print the whole network. In LISP, the traditional way to meet these difficulties is to make every node of the network an atomic symbol, with the network links on its property list. Given an atom, LISP will print just its name, not tracing down the property list, and thus not looping through the whole network. There are at least four difficulties with this solution. First, these node atoms take 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 to a node by typing in its atom name. Second, 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 of the semantic network. Third, because the standard LISP printer will print only the name of a node, the programmer must generally write a set of special purpose print functions to print just those links which are desired. Such code must be carefully written to avoid printing semantic network loops. Finally, 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 for 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. 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 [10].

2. Overview

BRAND X is implemented as an extension of LISP (MACLISP [11]), and thus completely includes its facilities, 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 lists*—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 OWL), which is optionally present in the language.

The rest of this section gives a brief introduction to the concepts underlying the above features, and the rest of this report lists the available functions which support the features.

2.1 Equality, Identity, Uniqueness and Composed Objects

The traditional definition of equality holds that two objects are 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 computation, however, the above definition of equality is not the one usually favored. This is because computer models of the real world often permit tests of distinguishability which are artifacts of the implementation. Typically, computer implementations can distinguish objects based on their 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 to think of the two numbers 999 and 999 as equal, some LISP implementations find them distinguishable under the EQ predicate, which tests equality of address.

One standard solution to the undesirable nature of strict equality is to distinguish between *identity*—true indistinguishability—and *equality*—now taken to mean indistinguishable in the real world, even though distinguishable in the implementation. LISP's EQ and EQUAL predicates capture these notions of identity and equality, respectively.* 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-effects 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 EQ—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 atomic symbols. This solution is motivated by the desire to use EQ as the standard equality test, for the reasons cited in the last paragraph.

*Both are defined more technically, of course. EQ is defined as identity of address, and EQUAL is defined as a recursive test which checks for the identity of primitive objects and the equality of constituents of compound objects. EQUAL is further modified so that equality of numbers is tested by identity of type and numerical equality ($x - y = 0$) of the values.

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) [6].

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 atomic symbols to 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 are atomic symbols which are spelled identically. The LISP 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.* This assures that (ordinarily) all instances read of the same-spelled symbol are EQ—thus, atomic symbols are unique.

To understand how we might define uniqueness for composite objects, consider in detail the ideal nature of the INTERN function. Formally, INTERN is 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 atomic symbols is partitioned by the function SAMEPNAMEP. If LISP interned all atomic symbols, we would say that

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

We take 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 criterial, 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 characteristic of these types is that:

- ULISTs are unique (with respect to EQ) under the condition that their criterial components (CAR and CDR) are identical (EQ), and
- CLISTs are unique (with respect to EQ) under the condition that their criterial components (CAR and CDR) are equal (EQUAL).

BRAND X 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.†

Other MACLISP data types (arrays and hunks) are considered innately unique and canonical.‡

3. Constructor and Selector Functions

3.1 List Structure

Corresponding to LISP's CONS, BRAND X additionally provides UCONS and CCONS to create new ULISTs and CLISTs. As a convenience, other functions which form unique and canonical list structure are also provided.

---

*Even this has exceptions, as LISP'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, LISP's definition of EQUAL unfortunately yields false when two symbols are SAMEPNAMEP but not EQ.

†BRAND X exists in versions which either include or do not include support for triples. The following documentation is for the version including triples, but it notes where differences arise in the non-triple version.

‡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.

UCONS *x y*

Creates (or finds) the unique ULIST whose CAR is (EQ) *x* and CDR is (EQ) *y*. If both arguments are canonical, the corresponding CLIST is returned instead. The arguments are not altered, except under the control of the flags CCONS, UATOM, and UNUMBER, described below. Except for these circumstances, the standard form LISP conditions, that (EQ X (CAR (UCONS X Y))) and (EQ Y (CDR (UCONS X Y))), are satisfied.

CCONS *x y*

Creates (or finds) the canonical CLIST whose CAR EQUALs *x* and whose CDR EQUALs *y*. Because the criterial components of a canonical structure must be canonical, canonical copies of any non-canonical arguments must be used to form the new structure (see CANONICAL, below). Therefore, it is not in general true that (EQ X (CAR (CCONS X Y))) and (EQ Y (CDR (CCONS X Y))).

ULIST $x_1, \ldots, x_n$

Makes a ULIST of its arguments by repeated application of UCONS.

ULIST* $x_1, \ldots, x_n$

Makes a ULIST just as ULIST, except that the list ends with the final CDR being x_n rather than NIL. ULIST* is to ULIST as LISP's LIST* is to LIST. Note that (ULIST* *x y*) is the same as (UCONS *x y*), and that (ULIST* *x*) is the same as *x*, except for possible conversions due to the flags CCONS, UATOM, or UNUMBER.

*ULIST *arglist*

Applies the function ULIST to its single argument, which becomes the list of arguments to ULIST.

ULIST *arglist*

Applies the function ULIST* to its single argument, which becomes the list of arguments to ULIST*.

CLIST $x_1, \ldots, x_n$

Makes a CLIST of its arguments by repeated application of CCONS.

CLIST* $x_1, \ldots, x_n$

Makes a CLIST of its arguments, whose final CDR is x_n. See ULIST* for more details.

*CLIST *arglist*

Explicit list of arguments version of CLIST. See *ULIST.

CLIST *arglist*

Explicit list of arguments version of CLIST*. See *ULIST*.

The selector functions for unique and canonical list structure are simply LISP's CAR and CDR.

3.2 Triples

The data type TRIPLE, provided in some versions of BRAND X, is a three-component structure, whose components are called ILK, TIE, and CUE. The following functions compose and decompose such objects:

`TRIPLE` *ilk tie cue*

Creates a non-unique, non-canonical triple whose three components are *ilk*, *tie*, and *cue*.

`UTRIPLE` *ilk tie cue*

Creates (or finds) the unique `TRIPLE` with components each `EQ` to the given arguments. The flag `CCONS`, if non-`NIL`, causes `CTRIPLE` to be used instead. The flags `UATOM` and `UNUMBER` have the same effect as for `UCONS`.

`CTRIPLE` *ilk tie cue*

Creates (or finds) the canonical `TRIPLE` with components each `EQUAL` to the given arguments. Because the components of canonical structures must be canonical, the `CANONICAL` versions of the three arguments are used.

`ILK` *triple*

This function returns the `ILK` component of a triple. The function `+ILK` has the same effect but performs no error checking.*

`TIE` *triple*

This function retrieves the `TIE` component of its argument. `+TIE` may be used more efficiently if the argument is known to be a triple.

`CUE` *triple*

This function retrieves the `CUE` component of its argument. `+CUE` exists for the same effect without error checking.

3.3 Control Flags

The following flags control whether unique lists (and triples) are formed at all or always made canonical, and whether atomic symbols and numbers must be interned before being used as components of unique structures.

`CCONS`

If this flag is non-`NIL`, `UCONS` always acts like `CCONS`—this eliminates `ULISTs` from BRAND X.

`UATOM`

If this flag is non-`NIL`, then `INTERNed` versions of all atomic symbol arguments are used by `UCONS`—this eliminates the use of non-interned atomic symbols from unique list structure.

`UNUMBER`

If this flag is non-`NIL`, then `NINTERNed` (see below) versions of all numbers are used by `UCONS`—this assures that all numbers appearing in unique structure are also unique.

Each call to `UCONS`, `CCONS`, `UTRIPLE`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 canonicity, and is also often useful for other indexing and searching applications. The function `CAR-1` retrieves the list of objects so indexed under its argument.

`CAR-1` *object*

Retrieves the list of objects whose `CAR` (or `ILK`) is known to be *object*. These will typically be only unique or canonical objects, because others are not cross-indexed by the creation functions.

*In the current implementation, the `ILK` is in fact the `CAR` of the MACLISP structure used to represent triples. Thus, (`CAR` *triple*) is identical with (`ILK` *triple*), and this fact may be useful for writing efficient algorithms. It is better programming practice, however, to use `ILK` or `+ILK` when a triple is involved, because of the desire for data abstraction.

4. Unique and Canonical

The functions UNIQUE and CANONICAL are 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 criterial 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.

4.1 UNIQUE

atomic symbol — print name — SAMEPNAMEP

Atomic symbols are made unique by the LISP primitive INTERN. The representative element of the equivalence class under SAMEPNAMEP 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 NINTERNed.

LIST or ULIST — CAR and CDR — EQ-CAR-CDR

Yields the unique ULIST (or CLIST if the components are canonical or the flag CCONS is nonNIL) formed of the CAR and CDR of the input. 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 BRAND X implements LISTs and ULISTs as different LISP data types.

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

Yields the unique UTRIPLE (or CTRIPLE if the components are canonical or if the flag 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 LISP 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 BRAND X.

4.2 CANONICAL

atomic symbol — print name — SAMEPNAMEP

Atomic symbols are made canonical by the LISP 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 — EQUAL

The canonical representative of any form of list is a CLIST. 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 BRAND X.

4.3 Interning of Numbers

The function NINTERN is used to intern numbers in BRAND X. NINTERN guarantees that any two NINTERNed numbers of the same LISP type (i.e., FIXNUM, FLONUM, or BIGNUM) will be EQ. Under control of the UNUMBER flag, numbers may be automatically interned when used as part of any structure in the BRAND X notation.

5. Predicates

BRAND X provides a number of predicates to test objects for their types. The most general of these is BRAND-X-OBJECTP, which returns NIL except for BRAND X objects, for which it returns the type of the object. The first group of predicates test their argument for being of a particular type or types. The next are general tests for the uniqueness or canonicity of their argument. Finally, the predicate KNOWN is used to determine if a structure of a given form is already present in the BRAND X database.

5.1 General Type-Testing Predicates

BRAND-X-OBJECTP *object*

Yields NIL for any argument unless it is one of the BRAND X types ULIST, CLIST, TRIPLE, UTRIPLE, or CTRIPLE (the latter three only in BRAND X with triples). In this case, the atomic symbols listed above are returned, as an indication of the value. A generalization of LISP's TYPEP function, which always returns the type of its argument, would be (OR (BRAND-X-OBJECTP *x*) (TYPEP *x*)).

UCONSP *object*

This predicate tests whether its argument is a unique cons cell (ULIST).

CCONSP *object*

This predicate tests whether its argument is a canonical cons cell (CLIST).

CUCONSP *object*

This predicate tests whether its argument is either a unique or a canonical cons cell (ULIST or CLIST). If true, it returns the atoms ULIST or CLIST, respectively. It could be defined as (OR (AND (UCONSP *x*) 'ULIST) (AND (CCONSP *x*) 'CLIST)), but is implemented more efficiently. In BRAND X without triples, this function is identical to BRAND-X-OBJECTP.

The following predicates are defined only for BRAND X with triples:

TRIPLEP *object*

This function returns NIL for any object which is not a triple, and it returns one of the atoms TRIPLE, UTRIPLE, or CTRIPLE if its argument is a non-unique, unique, or canonical triple, respectively.

NUTRIPLEP *object*

This predicate tests whether its argument is a non-unique triple.

UTRIPLEP *object*

This predicate tests whether its argument is a unique triple.

CTRIPLEP *object*

This predicate tests whether its argument is a canonical triple.

5.2 Tests for Unique and Canonical Structure

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.

UNIQUEP *object*

A predicate which determines whether its argument is unique. It is the case that (UNIQUEP (UNIQUE *x*)) is T. Also, for any object which passes this predicate, it is also the case that (EQ *x* (UNIQUE *x*)). From this follows that atomic symbols are UNIQUEP just when they are the representative of their equivalence class chosen by INTERN, numbers when chosen by NINTERN, LISP lists and non-unique triples are never unique, and all other objects are unique. Note that all canonical objects are also unique, but not the other way.

CANONICALP *object*

A predicate which determines whether its argument is unique. It is the case that (CANONICALP (CANONICAL *x*)) is T. Also, for any object which passes this predicate, it is also the case that (EQ *x* (CANONICAL *x*)). Atomic symbols and numbers are canonical if they are unique, lists, unique lists (UCONSP), non-unique and unique triples (NUTRIPLEP and UTRIPLEP) are not canonical, and everything else is.

INTERNP *atomic-symbol*

Determines whether its argument is the canonical representative of its equivalence class chosen by INTERN. This function depends on the internal method used by MACLISP's INTERN function.

NINTERNP *number*

Determines if its argument is the canonical representative of its equivalence class chosen by NINTERN.

5.3 KNOWN

BRAND X provides a special form, KNOWN, for the convenient operation of testing to see whether a unique or canonical BRAND X object of a certain form has already been created or not. This form is also useful for testing the presence of properties and labels, which are described below.

KNOWN *expression*

The expression is evaluated in an environment wherein none of the BRAND X functions which normally form new structures, add property values, or assign labels (see below) is allowed to do so. Instead, each checks whether the requested formation, property, or assignment is already present. If yes, the specified object is returned; otherwise, NIL. For example, (KNOWN (CCONS 'A 'B)) will return T if and only if the canonical cons of 'A and 'B has previously been made. LISP's *backquote* facility, as extended to BRAND X objects, is convenient for specifying the expression argument to KNOWN. Thus, the typical use of KNOWN is to check for the presence of some form, label, or properties in the same syntax in which they would be input: (KNOWN `[BALL ,NUM &COLOR RED]).* Note that this macro serves to check only BRAND X facilities supported in the BRAND X syntax; specifically, those functions which ultimately use UCONS, CCONS, UTRIPLE, CTRIPLE and ASSIGN-LABEL, and internal functions which cause properties to be added in response to input expressions. It does *not* control other BRAND X functions nor basic LISP ones such as CONS.†

6. Properties

BRAND X extends the LISP notion of properties to all objects of the language, including all MACLISP data types and other BRAND X objects. Thus, *any* object may have a property list [1]. The functions GETP, GETPL, PUTP, REMP, PROPLIST, and SETPROPLIST are extensions of and subsume the corresponding MACLISP functions GET, GETL, PUTPROP, REMPROP, PLIST, and SETPLIST. GETPL1 is provided as an efficient alternative to GETPL. In addition, BRAND X supports two functions, ADDP and DELP, which assume the convention that a property has a *list* of values rather than a single value. In this case new values are added at the front of the list; if an existing value is again added, it causes that value to come to the front of the list, but is not duplicated. Attempts to use ADDP or DELP on properties which do not have list values is in error.

GETP *item prop*

Retrieves the *prop* property of *item*, or NIL if it has none. This corresponds to LISP's GET. Note that although *prop* may be any object, searching of the property list is by EQ; therefore, typically only canonical objects should be used as property indicators.

GETPL *item list-of-props*

Retrieves part of *item*'s property list starting with the first property found that is among *list-of-props*, or NIL if none is found. This corresponds to LISP's GETL.

*This syntax will be explained below.

†Internally, KNOWN establishes a CATCH tag, NO-ULIST-CREATION-TAG, and binds the variable *DO-NOT-CREATE-ULIST to T, then evaluates its argument in that environment. When one of the above-mentioned functions detects a failure of the form to match existing structure, it throws NIL.

GETL1 *item* $prop_1, \ldots, prop_n$

A variable-argument version of GETPL, which avoids the need to create the *list-of-props* list. As implemented, this is most efficient when n is 1, which is a common case.

PUTP *item val prop*

Puts (or replaces) *val* as the *prop* property of *item*, and returns *val*. This corresponds to LISP's PUTPROP.

REMP *item prop*

Removes the *prop* property of *item*. The returned value is that part of the former property list whose CAR is the removed property, or NIL if the property was not present.

PROPLIST *item*

Returns the whole property list of *item*, or NIL if it has no properties. This corresponds to LISP's PLIST.

SETPROPLIST *item proplist*

Sets (or replaces) the property list of *item* with *proplist*. This corresponds to LISP's SETPLIST. Note that this operation is generally poor programming practice, as it may destroy information on the property list of an item needed by some package independent of the one doing the SETPROPLIST.

As described above, ADDP and DELP manage properties with lists as their values. These lists are maintained as non-duplicating, in reverse order of addition. Duplication is checked for by the predicate EQ, but under control of the UNUMBER and UATOM flags, numeric and atomic symbol arguments may be made canonical before they are added or deleted. For example, if UNUMBER is non-NIL, then (ADDP 'X 999 'P) done twice will leave only the single NINTERNed 999 among X's P properties. If UNUMBER is NIL, however, the same double addition would leave two (in MACLISP, where two instances of 999 are not EQ).

ADDP *item val prop*

Adds the value *val* to the *prop* property of *item*. That property must either already be a list or must not exist before the ADDP, or else this is an error. If *val* is a number or atomic symbol, its canonical instance may be used under control of the UNUMBER and UATOM flags. The new value is added at the front of the value list, but if that value is already present (by EQ), it is simply pulled to the front of the list. The function returns the new list of values.

DELP *item val prop*

Deletes the value *val* from the *prop* property of *item*. That property must be a list or must not exist. If *val* is not among the values, no action is taken. Comparison of *val* to the existing values is by EQ, except that if *val* is a number or atomic symbol, its canonical instance may be used under control of the UNUMBER and UATOM flags. If the last value in a list is deleted, the corresponding property is removed. The function returns the new list of values (possibly NIL).

A note on implementation: BRAND X properties for atomic symbols are implemented using LISP's property list mechanism. Property lists for UCONS, CCONS, TRIPLE, UTRIPLE, and CTRIPLE objects are included as one component of the internal representation of the object. Property lists for numbers are stored in a hash array which is also used by NINTERN; hashing is by SXHASH and comparison by EQUAL. Property lists for all other LISP objects, including ordinary CONS cells, are stored in a separate hash array, hashed by MAKNUM and compared by EQ.

7. Labels

Any BRAND X object may be labeled. When printed, an object's label may (under user control) be printed instead of the object itself.

`ASSIGN-LABEL` *label item*

Assigns the label *label* to the object *item*. *Label* must not already have been assigned to another object.

`REASSIGN-LABEL` *label new-object*

Assigns the label *label* to the object *new-object* and removes *label* as the label of whatever object it had been assigned to previously. In addition, an attempt is made to *alter* all accessible past uses of *label* to now use *new-object* instead of what *label* previously labeled. Note that this operation may have very undesirable side-effects and should be used only as a temporary measure, e.g., to correct the state of a data base. It is dangerous to use, because not all previous uses of *label* can be found and therefore those not found (e.g., in non-unique, non-canonical structure, in the `CDR` position of any structure) will continue to use the old object.

`LABEL-1` *label*

Retrieves the object labeled by *label*, or `NIL` if the argument is not in fact the label of anything.

`GET-LABEL` *label*

Retrieves the object labeled by *label*, or creates a dummy object whose label is *label* if *label* labels no object.

`*DUMMY-LABEL-CREATOR`

If this variable is `NIL`, BRAND X's standard method is used for creating dummy label objects. In BRAND X with triples, a dummy label object is formed by `(UTRIPLE` *label* `nil 'DUMMY-LABEL)`; in BRAND X without triples, by `(UCONS` *label* `'DUMMY-LABEL)`. In either of these cases, a value of `T` is added as the `DUMMY-LABEL` property of the object. If the variable is non-`NIL`, it is a function of one argument (the label), which is called to create dummy label objects. Default is `NIL`.

`UGL`

Returns all Undefined Global Labels. A utility function which finds all those atomic symbols in the current `OBARRAY` which label a dummy label object.

Included here is a short discussion of a number of problems which can arise in the use of labels. These problems are not easily solved, and are solved not at all or only badly by the current implementation of BRAND X. This section, in small font, may be skipped by all but the cogniscenti and those suspecting labels as the source of their unexplainable troubles.

The intent of a label is to be simply an abbreviation for the object it labels. If labels were used only after 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 to refer to something in an interactive environment before having completely defined it, labels do get used before they are assigned. The implementor 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 to ban all use of such objects, but this fails the criteria outlined above and is also difficult to 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 the algorithm will also succeed or fail appropriately for these. Therefore, BRAND X permits the examination of the representation of unassigned labels, which have the structure described above under the definition of `*DUMMY-LABEL-CREATOR*`.

The use of labels before they are assigned creates other very serious problems:

1. 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 to assure that all previous uses of the two will indeed refer to the identical object. Thus, formerly-made references to 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 to 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 BRAND X implementation) if the label is unused before its assignment; then, the dummy object is never created. 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`.
2. Canonicalization of data structure can fail because some identity that depends on label assignment may not be known when it is computed. For example, `(CLIST 'A 'B 'C)` and `(CCONS 'A (GET-LABEL 'FOO))` may appear to have little in common; yet, if `(ASSIGN-LABEL 'FOO (CCONS 'B 'C))` is later done, the two expressions are seen to be identical. However, the second canonical structure was created before this was known, and cannot be simply made to be `EQ` to the first. The same problem also appears when labels are used as a mechanism for creating circular structures. For instance, after `(ASSIGN-LABEL 'FOO (CCONS 'BAR (GET-LABEL 'FOO)))`, we have a new structure whose `CAR` is `BAR` and whose `CDR` is itself. Repeating this operation with different labels will create distinct such structures, although of course there should be only one because it is canonical. The solution to this problem is extremely hard, requiring a fast algorithm for identifying isomorphic structures and possibly a complete traversal of the data base any time a circular structure is formed.

8. Notation

One of the powerful simplicities of LISP 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.* This notion is preserved and extended to BRAND X objects.

8.1 Lists, Ulists, and Clists

Our goal has been to preserve LISP syntax as much as possible. Therefore, `LIST`s and `CONS`es 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)`.

`ULIST`s and `CLIST`s are written in a manner similar to `LIST`s, but with square brackets instead of parentheses. Thus,

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

*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.

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].

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

8.3 Labels

Labels are assigned by using the syntax

(<label> = <expression>) or [<label> = <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 . !AN-EXAMPLE]

is entirely equivalent to

[NOW THIS IS AN EXAMPLE],

and will be printed in the shorter form.

8.4 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 criterial 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 &COLOR RED GREEN BLUE],

we have

(GETP '[BALL 1] 'COLOR) => (RED 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 &COLOR &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 BRAND X, however, we interpret that form as equivalent to just `X`.* This provides a syntactic means of attaching properties (and labels as well) to any BRAND X object. For example, we use the following notation to attach `POSSIBLE-VALUES` to `COLOR`:

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

8.5 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, we may need to refer to the subject role of the frame in close proximity to our specification of the expression representing the frame itself. For example,

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

to 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 to 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 &ROLES
                  [SUBJECT : &C PERSON]],
```

which is read identically with the expanded form above. Here, the colon (`:`) acts as an anaphor, referring to the criterial 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 &ROLES
                  [SUBJECT : &C (PERSON ::)]],
```

stands for `(PERSON [RUN INTO TROUBLE])`.

Spaces are normally insignificant in BRAND X except to 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

*The rationale for this is that MACLISP's `LIST*` function, which forms successive conses of its arguments (e.g., `(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.

```
(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, we could write

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

which is not equivalent to the fully written-out version

```
(RUN INTO TROUBLE &ROLES
                  (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.*

8.6 Syntax

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

```
<x-expr> ::= <Lisp-atom> | (<x-expr-body>) | [<x-expr-body>] |
             <label-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>

<prop-spec> ::= &<prop:x-expr> {&<prop:x-expr>} {<val:x-expr>}+

<label-spec> ::= !<x-expr>

<colon-anaphor> ::= {:}+

<quoted-form> ::= '<x-expr>

<backquoted-form> ::= `<x-expr>

<comma-form> ::= ,<x-expr>
```

*Technically, the formation of truly circular expressions presents some difficulties more severe than those encountered in forming structures that are circular through property attachments. For example, in forming the structure [A [B :]] (whose CADADR is EQ to itself), we appear to need the whole structure before we can form its substructure. Although partly-successful tricks such as those involving the creation of circular structure by use of labels can be used here as well, BRAND X does 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 print it with anaphora.

In the above, we have used braces ({ ... }) to indicate optional phrases, * and + as superscripts on optional phrases to indicate zero or more and one or more permitted repetitions, respectively. Metasyntactic variables are in angle brackets (< ... >); such a variable of the form <*name*:type> is an expressive variant of just <type>—thus, <*prop*:x-expr> is merely <x-expr> with a suggestion of its meaning as a property descriptor. Spaces are not significant, except as separators and as noted above between successive colons in colon anaphors. Note that although any x-expr 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 are 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.*

9. Reading and Printing

The LISP reader and printer have been thoroughly "hacked" by BRAND X to produce reasonable behavior. Nevertheless, most normal LISP uses of reading and printing should continue to work.

The most significant change in reading is the assignment of special meaning to numerous characters which are treated as alphabetic in LISP. The characters [,], &, !, =, : and (for BRAND X with triples), * must be typed preceded by a slash (/) if they are to be taken alphabetically.

The function ABSORB is provided to permit the compilation of BRAND X data structures from a file.†

`ABSORB` *any-forms*

ABSORB simply ignores any number of arguments. It is like a variable-number-of-arguments QUOTE, intended to quote "top level" data items.

The BRAND X printing functions use a combined set of functions which have the ability to perform "pretty printing", checking for anaphora, and selective printing of the labels and properties of objects as well as their criterial parts. These printing facilities are controlled by a number of flags.

`BRAND-X-TOPLEVEL-PRIN1` *object (*optional *stream)*

This function prin1's *object* onto *stream* or the standard output if `NIL`, under control of the flags below. This is also the function used by LISP's "top level" read-eval-print loop for BRAND X.

`*PRINT-PROPS`

If non-`NIL`, the top level printer will show the properties of an object being printed. Default is `T`. Only those properties which have lists as values and which are not among `*NOPRINT-PROPS` are printed.

---

*Currently, the use of nested backquoted expressions fails to work correctly. Thus, for example, one cannot write `` `(A B ,C `(,D)) ``, which could be used to represent `(LIST 'A 'B C (LIST D))`.

†This is actually due to an error in the MACLISP compiler, which assumes that quoted forms at "top level" in a file can have no effect; it therefore throws them away. Writing these as arguments to ABSORB merely protects them from this fate.

*NOPRINT-PROPS

A list of property indicators which are not to be used to print properties even if *PRINT-PROPS is non-NIL. Default includes properties used internally by LISP and BRAND X, such as EXPR and LABEL.

*PRINT-STM-PROPS?

If non-NIL, the top level printer will print the properties of non-unique, non-canonical objects as well. Default is NIL.

*PRINT-SYMBOL-PROPS

If non-NIL, the top level printer will print the properties of atomic symbols. Default is T.

*PRINT-LABELS

A variable which controls whether PRINT normally prints the label of an object or the object itself. If non-NIL, the label is printed. Default is T.

*PROPERTIES-OF-INTEREST?

NIL, ALL, or a list of property indicators which are to be printed within nested structures (not only at top level). Default is NIL, and the effect of this flag is overridden by *NOPRINT-PROPS and *PRINT-STM-PROPS.

*PRINT-ALL-ANAPHORA?

If non-NIL, then circularity will be represented by anaphora in print-out at all places. If NIL, printing will be significantly faster (one, rather than two passes), but a structure like (A . (B (:))) will be printed less clearly because the one-pass printer will begin with (A B ...) before realizing that a new list must start with B to provide a reference point for the anaphor. Default is T.

*DELIMIT-TRIPLE-INDICATOR?

If non-NIL, the * delimiting the ILK and TIE of a triple is surrounded by spaces. Otherwise not. Default is NIL.

10. Using Brand X, and Relation to Other Lisp Packages

BRAND X is built using LSB [5], the Layered System Building package, which is not needed by it at run-time. A BRAND X without LSB may be invoked by :BX in ITS, a BRAND X with LSB by :BXLSB, and a BRAND X with LSB and the OWL definitions and support [4] by :BXOWL.

BRAND X supports the use of the LOOP iteration macro facility by providing the following paths:

```
(LOOP FOR X BEING INFERIORS OF Y DO ...)
```

is equivalent to

```
(LOOP FOR X BEING EACH CAR-1 OF Y DO ...),
```

which is in turn equivalent to

```
(DO ((XX (GETP Y 'CAR-1) (CDR XX)) (X))
    ((NULL X))
    (SETQ X (CAR XX))
    ...)
```

The INFERIORS (or INFERIOR) path is special purpose, to help enumerate those objects "under" another. The second form is general, allowing any property indicator to be substituted for CAR-1. For much more functionality and details, refer to the documentation on the LOOP package [3].

Acknowledgment

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.

References

1. Bobrow, D. G., "A Note on Hash Linking," *Comm. ACM* 18, (18) (July 1975), 413–415.
2. Bobrow, D. G., and Winograd, T., *An Overview of KRL, a Knowledge Representation Language,* Technical Report AIM-293, Stanford Artificial Intelligence Lab., Stanford, Ca., (1976).
3. Burke, G. S., and Moon, D., *Loop Iteration Macro,* TM-169, MIT Lab. for Comp. Sci., Cambridge, Mass., (July 1980).
4. Burke, G. S., *Brand X Interpreter Manual,* MIT Lab. for Comp. Sci., Cambridge, Mass., (in preparation).
5. Burke, G. S., *LSB Manual,* MIT Lab. for Comp. Sci., Cambridge, Mass., (in preparation).
6. Goto, E., *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).
7. Hawkinson, L. B., *XLMS: A Linguistic Memory System,* TM-173, MIT Lab. for Comp. Sci., Cambridge, Mass., (1980).
8. Hewitt, C., *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot,* AI-TR-258, MIT Artificial Intelligence Lab., Cambridge, Mass., (1972).
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 Szolovits, 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., Cambridge, Mass., (1974).
13. Rulifson, J. F, Derksen, J. A. and Waldinger, R. J., *QA4: A procedural calculus for intuitive reasoning,* Technical Note 73, SRI International, Menlo Park, Ca., (November 1972).
14. Sussman, G. J., and McDermott, D. V., "From PLANNER to CONNIVER - A Genetic Approach," *Proceedings of the 1976 Fall Joint Computer Conference,,* AFIPS Press, (1976), 1171–1179.
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 Rahmstorf, G., and Ferguson, M., (Eds.), *Proceedings of the Workshop on Natural Language Interaction with Databases,* International Institute for Applied Systems Analysis, Schloss Laxenburg, Austria, Jan. 10, 1977.
16. Woods, W. A., *Research in Natural Language Understanding; Annual Report,* Report No. 4274, Bolt, Beranek, and Newman Inc., Cambridge, Mass., (1979), See especially Chapter 2, "An Introduction to KL/ONE", pp. 13–46.