Table of Contents

1. Introduction

This paper describes the preliminary implementation of an Interpreter for concepts and conceptual methods using the Brand X data base. The primary intent is to have automated dynamic procedure selection based on both argument matching and prerequisites. Secondarily, almost as a side effect to attain this, knowledge base conventions and primitives are developed.

Based on the implementation and facilities of the Maclisp evaluator, one can consider there to be three evaluation mechanisms of interest:

(1) evaluation of a form, the **car** of which is a special object

(2) application of a special object to some arguments (via **apply**, **funcall**, or **lexpr-funcall**)

(3) evaluation of a special object itself

This "special object" is a *description* in our knowledge base, known as a **concept**. The primary goal of the system documented here is cases 2 and 3. Accordingly, the handling of case 1 is simplified to just evaluating the arguments and going on to case 2.

Conceptually, what is desired is to be able to automatically select an appropriate procedure based on the arguments given, the relations between the arguments, and the external environment. This is (arbitrarily!) broken down into constraints which can be specified declaratively (by "matching", which will be expounded on later), and those which are must be specified procedurally. Note that the distinction between these is mainly a matter of what can be specified declaratively. This distinction is undoubtedly useful no matter how fluent the declarative language, since it seems unlikely that a declarative language could encompass all possible situations which might be encountered. (For exposition purposes, "procedural" is something for which **eval** or some application counterpart is explicitly used, and "declarative" is where one, for example, compares descriptions in a pre-defined manner.)

The breakdown chosen is to partition the procedures into *methods* or *partial definitions*, each of which is a procedure with some *formal parameters*. Partial definition selection will be based on matching of the arguments to the corresponding formal parameters (the declarative part). Each partial definition can have *prerequisites* (the procedural part) which must be satisfied for that method to be valid. The fine points and mechanism of this are detailed in chapter 7; first, knowledge base conventions are needed.

2. A Version of OWL

This chapter details the knowledge base conventions upon which the evaluator extensions are built. It is obviously subject to drastic redefinition.

2.1 Concept Formation

For the present, a concept is either the Brand X triple **!tao**, defined

```
[tao = tao*tao tao]
```

or a triple whose **ilk** is a concept. The implications of this are

(1) A concept is not circular through the **ilk** path.

(2) One can iterate up the **ilk** of a concept, terminating on a trivial **eq** test.

When an object is created for an undefined Brand X label in OWL, it is made into a concept, for ease of use; this is discussed fully in section 2.8.1, page 10.

Although not strictly tested for, concepts are often assumed to have **ties** which are either atomic symbols or concepts. The creation of a concept with something else as a tie might result in strange errors, such as infinite looping or memory errors.

The **cue** of a concept may be any Lisp object. Certain concept constructs however make assumptions about the cue, which may or may not be checked for validity.

2.1.1 Concept Testing

conceptp *object*
Returns **t** if *object* is a concept, **nil** otherwise.

OWL defines **concept** as a data type, using the predicate **conceptp**. This is used for (automatically generated) argument type checking in many routines defined here.

2.1.2 Concept Creation

make-ltm-concept (concept *ilk*) *tie cue*
make-stm-concept (concept *ilk*) *tie cue*
These two routines open-code into calls to **utriple** and **triple** respectively. It is unclear whether they are necessary, except from the standpoint that in some future implementation concept formation may be more complicated than just triple construction. These do, however, check the *ilk* for **conceptp**, when called interpreted.

2.1.3 Iteration Paths

OWL defines the **superiors** iteration path for the **loop** macro.

```
(loop for var being concept and its superiors
      ...)
(loop for var being the superiors of concept
      ...)
```

step *var* through the superiors of *concept*, up to and including **!tao**. The first starts at *concept* itself, the second at the **ilk** of *concept*.

2.2 Using Concepts in the Hierarchy

When one constructs a concept, one is stating that the new concept is **ako** its **ilk**. The actual implication of this is dependent on the interpretation placed on the makeup of the concept; that is, the interpretation one places on its **tie** and its **cue**, and other description, the properties.

A significant hierarchical relation of two concepts is **underp**. A concept *c1* is said to be **underp** a concept *c2* if one can reach a concept **equal** to *c2* by taking some non-zero number of **ilk**s of *c1*. Note that **equal** is used here; take, for example,

```
(setq a (triple '!tao 't 'a))
  => (!TAO*T A)
(setq b (triple (triple '!tao 't 'a) 't 'b))
  => ((!TAO*T A)*T B)
```

The **ilk** of *b* is **equal**, but *not* **eq**, to *a*. At the same time, the intuitive feeling is that indeed *b* is **underp** *a*. There are other cases where the use of **equal** by the OWL system is less obviously correct; some of these will be noted. In any case, the decision of whether to use **equal** or **eq** is primarily a matter of the significance of a concept's being non-unique, which has not been properly established yet.

underp (concept *concept-1*) (concept *concept-2*)
: This returns **nil** if *concept-1* is not **underp** *concept-2*, as described above. If it is, then the number of **ilk**s separating the two is returned. E.g.,

```
(underp '[[!tao*t 1]*t 2] '[!tao*t 1])
=> 1
```

underp-or-equal (concept *concept-1*) (concept *concept-2*)
: Like **underp**, with a different boundary case.

2.3 Mutual Exclusion

Detection of mutual exclusion allows one to immediately say that two conceptual descriptions are incompatible. The primitive basis for mutual exclusion of two concepts is that they follow two different branches from their closest common superior concept, both with ties of **s**. That is, if

```
[dog = !animal*s dog]
[cat = !animal*s cat]
[tailed-animal = !animal*t tailed-animal]
```

then we can say that any concept **underp-or-equal** to **!dog** is mutually exclusive with any concept **underp-or-equal** to **!cat**. We can not, however, say the same for **!tailed-animal** and either **!dog** or **!cat**. The following routine tests for mutual exclusion in the hierarchy. Note that one can also detect mutual exclusion by recursively detecting mutual exclusion in other descriptions, such as *characterization*. This is discussed later.

mutually-exclusive? (concept *concept-1*) (concept *concept-2*)
This returns **t** if *concept-1* and *concept-2* are mutually exclusive, as determined by seeing if the concepts directly under the **least-common-superior-concept** of *concept-1* and *concept-2* both have ties of **s**.

2.4 Characterization

The term *characterization* is used here in a more restricted sense than it is used in English. In this domain, it is an alternative *ako* relationship; that is, an OWL characterization of *x* as *y* implies that you can say that "*x* is a *y*" or "*x* is characterized as being a *y*", but not "*x* is characterized as *y*". One would not use this meaning of characterization to state that "the ball is red", but one might do so to say that "a person is a corporate entity".

Characterization is represented in OWL by use of the **c** tie and the **c** property. Thus,

```
[person = !animate-entity*s person
   &c !corporate-entity]
```

defines the concept labeled **person**, and also implies that a person is "a kind of" **!corporate-entity**. The uses of characterization are discussed more fully with the pattern matcher.

2.5 Inheritance

A primary reason for having one thing being *ako* another is that the second can *inherit* attributes from the first. The primary inheritance path for attributes is from the superiors, i.e. through the **ilk**. This is not always sufficient. However, a reasonable compromise must be reached between searching through many paths, and having to redundantly specify inherited attributes or characteristics all over the place.

look-for-inherited-properties *concept proplist*

This looks through *concept* and its superiors for any of the properties in *proplist* (i.e., it terminates on a non-null **getpl**). If a *slot frame* is encountered (a concept with a **tie** which satisfies **slot-frame-tiep**, as discussed in the next section), all of the matching **h** properties on the **cue** of the slot frame and its superiors are checked also.

The above routine follows the "canonical" way in which things are inherited in OWL. It is expected that a mechanism will be provided so that one can cause inheritance from the **cue** concept; take, for example, the stereotypical example of the representation of "a can of beans", as in "I opened and ate a can of beans". One might wish to represent it as

[!can**tie* !beans]

for some appropriate *tie*, and thus make use of inheritance from both **!can** and **!beans**. Presumably the sentence means that the can was opened and the beans were eaten. Some common scenarios would be

(1) Inherit from only the **ilk**

(2) Inherit from both the **ilk** and the **cue**, with the **ilk** having precedence

(3) Inherit from both the **ilk** and the **cue**, with the **cue** having precedence

(4) Inherit only those attributes implied by both the **ilk** and the **cue**

Hopefully, a mechanism will soon be provided in OWL which allows one to do this while neither constraining the semantics nor increasing the inefficiency significantly.

2.6 Metacharacterization

Metacharacterization is description for the use and/or maintenance of the knowledge base itself. At present, the only metacharacterization used by the OWL system is the **lexical** keyword, which controls concept binding (section <not-yet-written>, <not-yet-written>). Metacharacterization is indicated by use of the **m** property, which at present is assumed to be a list of either atomic symbols (presumably used like keywords), or concepts.

look-for-metacharacterization (concept *concept*) *metacharacterization*

This returns **t** if *metacharacterization* is found under any of the **m** properties which *concept* inherits (as described under **look-for-inherited-properties**, page 5). **Eq** is used for comparison, as typically metacharacterizations will be atomic symbols used as keywords.

2.7 Slot Frames

Slot frames allow one to associate attributes or substructure with concepts. For example, in the description for a generic person

```
[person = !animate-entity*s person
   &h [!location*u : &c !place]]
```

the **h** property lists a concept, called a *slot frame*, which says (loosely) that a **!person** can have a location. The value of the **!location** slot of a **!person** must be characterized as a **!place**. In this example, the concept **!location** is called a *slot frame name*, and the tie used in such a slot frame is called a *slot frame tie*. One can give a slot frame a value, which will be inherited as the default value of any concepts under the concept the slot frame is attached to, as in

```
[pontiac-engine = !engine*t pontiac-engine
    &h [!manufacturer*u : &v !pontiac]]
```

or one could embed the value in the conceptual hierarchy, as in

```
[chevrolet-engine = !engine*!manufacturer !chevrolet]
```

The latter has more of a flavor that the value of that slot is an integral part of the object than the former example.

The process of finding the value of a slot for a particular concept involves searching through the concept and its superiors for either a concept specifying the value (as in **!chevrolet-engine**) or an **h** property for that slot, as in **!pontiac-engine**. The first such concept found supplies the value. (The slot frame names are compared with **eq**.)

Because of this action, the act of copying a slot frame down from some concept to an inferior involves creating a new slot frame, copying the value(s), and attaching it to the concept, all as an "atomic operation". This also demonstates one of the primary inheritance paths: for example, in

```
[bill = !person*i bill
    &h [!location*qu : &v !boston]]
```

and using the definition of **!person** given at the beginning of this section, **[!location*qu !bill]** inherits from **[!location*u !person]** before it inherits from **!location**.

2.7.1 Slot Frame Ties

The **tie** of a slot frame is used to determine certain attributes about both the slot frame itself, and the process of evaluation of such a slot frame (described in chapter 6). Implementationally, a *slot frame tie* is recognized by being a symbol or concept which has (or inherits) a **slot-frame-tie** property. This property is a list of keywords which name various attributes of slot frames constructed from this tie. These keywords are used primarily for evaluation of slot frames, and the presence of this property distinguishes a slot frame from other concepts. The keywords **multiple-valued** and **single-valued** tell whether the slot frame is allowed to have one, many, or no values. The **read-only** keyword says that the value is not allowed to be modified. The **ilk-evaluable** and **cue-evaluable** keywords tell whether, when a slot frame with this tie is evaluated, the **ilk** and **cue** should be evaluated to find a new slot frame to find the value of (section 6.2.1, page 22). The initially defined slot frame ties, and their **slot-frame-tie** properties, are:

```
(absorb
   [. r &slot-frame-tie
          multiple-valued cue-evaluable]
   [. u &slot-frame-tie
          single-valued cue-evaluable]
   [. ror &slot-frame-tie read-only multiple-valued]
   [. rou &slot-frame-tie read-only single-valued]
   [. v &slot-frame-tie
          ilk-evaluable cue-evaluable]
   [. uv &slot-frame-tie single-valued
                        ilk-evaluable
                        cue-evaluable]
   [. q &slot-frame-tie multiple-valued]
   [. uq &slot-frame-tie single-valued]
   [. roq  &slot-frame-tie multiple-valued read-only]
   [. rouq &slot-frame-tie single-valued read-only]
  )
```

slot-frame-tiep *object*

If *object* is a *slot frame tie*, this returns the **slot-frame-tie** property. *Object* need not be a concept. For convenience in error checking, **slot-frame-tie** is defined as a data type, based on this predicate.

slot-frame-tie-data *slot-frame-tie*

This is like **slot-frame-tiep** but makes the assumption that *slot-frame-tie* is either a symbol or a concept.

2.7.2 Slot Frame Names

Just as certain attributes about a slot frame can be determined from the tie, some can be determined from the slot frame name. One is what **tie** should be used in creating a slot frame when one is being copied down from a superior concept. This also is for use in evaluating and setting slot frames.

When a slot frame is being copied down from a superior concept, a **tie** may have to be chosen for the newly created slot frame. A slot frame name should have (or inherit) a **slot-frame-name** property, which should be a list of two things:

(1) The tie to be used when the superior's slot frame is an ordinary attached slot frame (i.e., its **tie** is a *slot frame tie*, as in the **!pontiac-engine** example)

(2) The tie to be used when the superior's slot frame has its value embedded in the hierarchy (i.e., its **tie** is the *slot frame name*, as in the **!chevrolet-engine** example)

slot-frame-ties (concept *slot-frame-name*)
This searches *slot-frame-name* and its superiors for a **slot-frame-name** property, which is returned. If none is found, **nil** is returned.

For convenience in defining slot frame names, the following concepts are pre-defined:

```
(absorb
  [slot-frame-name = !hsiang*t slot-frame-name
     &slot-frame-name q roq]
  [single-valued-slot-frame-name =
     !slot-frame-name*t single-valued-slot-frame-name
     &slot-frame-name uq rouq]
  )
```

2.7.3 Routines for Manipulating Slots

The routines available, and/or their semantics, may be changed as experience with their use dictates.

make-or-find-slot (concept *slot-frame-name*) (concept *concept*)
(Optional (slot-frame-tie *slot-frame-tie*))
If *concept* has a slot for *slot-frame-name*, that is returned; otherwise, one is created, with a tie of *slot-frame-tie*, which is determined from *slot-frame-name* if **nil** or not specified.

make-slot (concept *slot-frame-name*) (concept *concept-to-make-slot-in*)
(Optional (slot-frame-tie *slot-frame-tie*))
Creates a *slot-frame-name* slot in *concept-to-make-slot-in*. It is an error if one already exists. *Slot-frame-tie* is determined from *slot-frame-name* if **nil** or not specified.

look-for-slot (concept *slot-frame-name*) (concept *concept*)
Looks for a slot of type *slot-frame-name* in *concept*. **Nil** is returned if none is found.

fetch-slot (concept *slot-frame-name*) (concept *concept*)
Like **look-for-slot**, but requires that a slot be found.

find-generic-slot (concept *specific-slot*)
This finds the generic slot which *specific-slot* is the corresponding slot in some individual/instantiation of. That is, if we have

```
(absorb
  [foo = !bar*s foo
     &h [!baz*u :]]
  [foo-1 = !foo*i foo-1
     &h [!baz*u :]]
  )
```

specific-slot should be **[!baz*u !foo-1]**, and this will return **[!baz*u !foo]**. This routine trivially utilizes **find-slot-in-superiors**, below.

find-slot-in-superiors (concept *slot-frame-name*) (concept *starting-with*)
This searches through all the **h** properties of *starting-with* and its superiors for a slot frame of type *slot-frame-name*. A concept with a tie of *slot-frame-name* causes satisfaction, and will be returned if encountered.

slot-existsp (concept *slot*) (concept *in-concept*)
If there exists a slot in *in-concept* (or its superiors) with slot-frame-name of *slot*, that slot frame is returned.

2.8 Online Aids

2.8.1 Undefined Labels

OWL utilizes the ***dummy-label-creator** hook in Brand X. The object created for an undefined label will be a concept whose **ilk** is the concept **!dummy-label-definition**, whose **tie** is **t**, and whose **cue** is the label — typically an atomic symbol. **!dummy-label-definition** is defined as:

```
[dummy-label-definition = !tao*t dummy-label-definition]
```

Thus, if one references the label **foo** before defining it, it will be the concept

```
[foo = !dummy-label-definition*t foo
   &dummy-label t]
```

defugl (Optional *ilk*) (Optional *tie*) (Optional *labels-to-define*)
This routine is useful for defining undefined labels. *Ilk*, if supplied and not **nil**, should be a concept; otherwise it will default to the concept **!default-definition**, defined as

```
[default-definition = !tao*t default-definition]
```

Tie, if not supplied or **nil**, defaults to **t**. If *labels-to-define* is not specified or **nil**, then the result of (**ugl**) is used. Thus,

```
(defugl '!frob nil '(chair table))
```

is like

```
(absorb
  [chair = !frob*t chair]
  [table = !frob*t table])
```

and

```
(defugl '!feline 's '(cat leopard tiger lion))
```

is like

```
(absorb
  [. !feline
    &() [cat = :*s cat]
        [leopard = :*s leopard]
        [tiger = :*s tiger]
        [lion = :*s lion]])
```

except that each of the labels are assumed to already exist but not be defined, and

```
(defugl)
```

is like

```
(defugl '!default-definition t (ugl))
```

2.8.2 Printing the Structure

print-tree (Optional *start*) (Optional (fixnum *start-pos*))

This prints out the tree, starting at *start*, which defaults to **!tao**. One concept is printed per line; inferiors are printed two spaces to the right of their superior. A starting indentation may be specified as *start-pos*. The long form of the concept is printed, but without properties. Note that this can only find unique concepts, since it must get from a concept to those which have it as their **ilk**.

print-structure (concept *concept*) (Optional *number-of-levels-to-consider*)

This shows the structure of *concept* and its superiors. Try it.

3. Lisp Extensions

This chapter describes some of the "extensions" to Maclisp which the Interpreter environment makes use of. These are primarily features extracted from the Lisp Machine Lisp dialect. They themselves are not related to Brand X or the Brand X interpreter; in fact, they could be used separately from it.

3.1 Conditions and Signalling

The following two paragraphs are excerpted from the Lisp Machine manual (November 1978 version).

> Programmers often want to control what action is taken by their programs when errors or other exceptional situations occur. Usually different situations are handled in different ways, and in order to express what kind of handling each situation should have, each situation must have an associated name. In Lisp Machine Lisp, there is the concept of a *condition*. Every condition has a name, which is a symbol. When an unusual situation occurs, some condition is *signalled*, and a *handler* for that condition is invoked.
>
> When a condition is signalled, the system (essentially) searches up the stack of nested function invocations looking for a handler established to handle that condition. The handler is a function which gets called to deal with the condition. The condition mechanism itself is just a convenient way for finding an appropriate handler function given the name of an exceptional situation. On top of this is built the error-condition system, which defines what arguments are passed to a handler function and what is done with the values returned by a handler function. Almost all current use of the condition mechanism is for errors, but the user may find other uses for the underlying mechanism.

signal *condition-name* (any-number-of *args*)

Signal searches through all currently established condition handlers, starting with the most recent. If it finds one that will handle the condition *condition-name*, then it calls that handler with a first argument of *condition-name*, and with *args* as the rest of the arguments. If the handler returns **nil**, **signal** will continue searching for another handler; otherwise, **signal** returns whatever the handler returned. If **signal** doesn't find any handler that returns a non-**nil** value, it will return **nil**.

condition-bind *Special Form*

The **condition-bind** special form is used for establishing handlers for conditions. It looks like:

```
(condition-bind ((cond-1 hand-1)
                 (cond-2 hand-2)
                 ...)
    body)
```

each *cond-n* is either the name of a condition, or a list of names of conditions, or

nil. If it is **nil**, a handler is set up for *all* conditions (this does not mean that the handler really has to handle all conditions, but it will be offered the chance to do so, and can return **nil** for conditions which it is not interested in). Each *hand-n* is a form which is evaluated to produce a handler function. The handlers are established such that the *cond-1* handler would be looked at first. All of the *hand-i* forms are evaluated before any of the handlers for the conditions are established.
Example:

```
(condition-bind ((unvalued-concept
                   'my-unvalued-concept-handler)
                 ((lossage-1 lossage-2) lossage-handler))
    (princ '|Hello there.|)
    (do-some-computing))
```

This establishes the function **my-unvalued-concept-handler** as the handler for the **unvalued-concept** condition, the value of the symbol **lossage-handler** as the handler of the **lossage-1** and **lossage-2** conditions, then prints a message and does some computing with those handlers established. **Condition-bind** makes use of ordinary variable binding, so that if the **condition-bind** form is thrown through, the handlers will be disestablished. This may also matter if one is using funargs.

Note that the **signal** formalism makes no demands on what the arguments going with a condition are. An additional layer needs to be built on top of it, tailored to the particular use. For the Brand X Interpreter, **signal** is utilized via the **fail** routine, which provides a default action when the condition is not handled. The failure mechanism is described later.

In this particular Maclisp implementation of **signal**, the condition binding environment is re-bound to the "null" environment in a break loop. (*Explain this? This parallels the Lisp Machine, where entering the error handler switches stack groups, thus changing binding environments.*)

3.2 Multiple Values

Sometimes one has a relatively complicated computation which produces a value, and may happen to produce other information which may be of interest to the caller. In cases like this one often returns some data structure containing the various results, e.g. a list. An alternative is to have a protocol for passing back and receiving multiple values. The facility provided for this is derived from Lisp Machine Lisp, and is documented fully in chapter 8. It is noted here because all of the Brand X Interpreter supports the passing back of multiple values, and this behaviour is part of the description. Thus, if something is described as "returning a list of the values from the evaluation of...", these multiple values are being refered to.

The existence of the multiple value mechanism in the Brand X Interpreter is experimental. If it is deemed to be useless, it may be flushed. Note however that the overhead of having multiple values is very very small when they are not used.

4. The Failure System

The Brand X Interpreter, being essentially an extension of the Lisp environment it exists in, is constrained by the control flow allowable in that environment. It does not support backtracking or continuation passing (although since it supports downward funargs to approximately the same extent that Maclisp does one could inefficiently and painfully do some continuation passing). To compensate for this it uses a failure mechanism which is sort of an extension of the Lisp error system. This uses as a protocol the Lisp Machine derived **signal**ling convention, described in section 3.1.

4.1 Causing Failure

fail *condition-name format-string* (Any-number-of *other-arguments*)

This is the entry which defines the protocol by which the failure system interacts with **signal**. *Condition-name* is **signal**ed with the same arguments given to **fail**. If the condition is not handled, then the a Lisp error occurs, which should informatively describe the condition, the reason for the error, and tell where the failure was signaled from. Note that the mapping from the arguments to **fail** to the arguments to **signal**, and the behaviour when the condition is not handled, is defined by **fail**; **signal** would simply return **nil** if the condition were not handled. The arguments to the condition handler are different for **fail**ure than in the Lisp Machine error system primarily for efficiency reasons; when an unhandled condition is signaled via **fail**, there may be a noticible pause while **fail** figures out where it was called from — hence the "caller" is not included in the arguments to the handler. Thus, if the Brand X Interpreter existed on the Lisp Machine, it would not work to call **fail** with conditions defined by the error system.

The argument convention is that the second argument to **fail** (and thus to the condition handler) is a symbol which can be given to the **format** routine with the remaining arguments, to construct a legible description of the reason for failure. In general, some subset of the third through last arguments are defined to be particular things by the condition handler; for example, the **unvalued-concept** condition handler requires this first "data" argument to be the concept for which a value could not be calculated, as in

```
(fail 'unvalued-concept
      '|The concept ~S is not valued|
      concept-which-is-unvalued)
```

Additional arguments may be given. A severly abbreviated **format** reference is included as chapter page 33; full documentation is available in the Lisp Machine manual, and may be made available separately at some later date if there is sufficient demand.

4.2 Trapping Failure

Sometimes one desires to "trap" certain kinds of failure. Here are some special forms which may be used to do so. They are similar in syntax to the Maclisp ***catch** routine.

***catch** *is not yet documented in the Maclisp reference manual. It has the following syntax:*

```
(*catch tag form-1 form-2 ...)
```

as in the example

```
(*catch 'negative
    (mapcar (function (lambda (x)
                        (cond ((minusp x)
                               (*throw 'negative y))
                              (t (f x)))))
            y))
```

which returns the first negative element of the list y *if one is found, otherwise a list of the applications of* f *to the elements of* y. *Note also that the Maclisp* ***catch** *has different but upwards-compatible syntax from the Lisp Machine* ***catch**, *in that more than one form may be specified, and one may specify a list of tags in place of a single tag.*

failure-trap *Special Form*

(failure-trap *condition-or-conditions*
form-1 form-2 ...)

Condition-or-conditions gets evaluated. It should evaluate to either the condition name, or a list of condition names. Those conditions are enabled with a handler which will **throw** control back to the **failure-trap** which established it, causing **failure-trap** to return **nil**. If that does not occur, **failure-trap** returns a *list* of values returned from the evaluation of the last *form* — where multiple values are not being used, that will be a list of one element, somewhat similar to the behaviour of **errset**. This may then be considered to be a more highly structured and specialized version of **errset**.

failure-trap? *Special Form*

This has syntax identical to **failure-trap**, and similar semantics. The difference is that this returns **t** rather than a list of values if it was exited "normally".

%trace-failure? *Variable*

This variable is examined by the handler established by **failure-trap** and **failure-trap?**. If it is not **nil**, then the handler prints an informative message (similar to that given by **fail** when a condition is not handled) before performing the non-local return to **failure-trap** (or **failure-trap?**). In this way one can "watch" failures which are trapped by **failure-trap**. If the value of **%trace-failure?** is the atom **break**, then a break loop is entered; the variable **args** will have some interesting stuff in it. Note that if this tracing is being done, there may be a noticible pause while the stack is examined to find the routine which caused the failure. Also, if one desires to trace *all* failures, one may use the **trace** debugging facility to trace the **fail** routine.

5. The Matcher

Defined herein is a mechanism for quantifying the closeness of match of a concept variable to its value. The description here also implicitly defines the inheritance of various attributes.

The basic premise is that we can identify *features* of a pattern, and require the value being compared to satisfy those features. In the process, we can "keep score" of how well the match is proceeding; this score is what is eventually returned by the matcher, and is used (admittedly in an ad-hoc manner) to determine whether one value is "closer" to a pattern than another value is.

The features noted are those of *characterization*, *functional restriction*, and *slot values*. The last has two forms; one may either require a slot to just be valued, or one may require it to have a particular value.

The scoring mechanism for comparing a pattern against a value essentially just adds a "point" for each feature found in both. If however the feature is not found, then the match fails. The point score is further modified by allowing some decay based on the proximities of the features; if the features are found far from the original concepts, they are thus less significant than if they had been found close; if the features are far from each other, they are also less significant. This decay is relatively small compared to the original point scored, so it effectively allows an ordering of similar concepts. The scoring is detailed later.

compatible-descriptionp (concept *object*) (concept *value*)
cd *object value*

This attempts to tell of *object* and *value* are compatible. This is used for argument matching, value checking, etc. If they are not compatible, it returns **nil**; otherwise a flonum representing the degree of matching of the two objects. Note that zero, which implies no matching features but also no known incompatibilities, is not returned by this routine; it returns **nil** instead.

possibly-compatible-descriptionp (concept *pattern*) (concept *value*)
pcd *pattern value*

This is just like **compatible-descriptionp** but will also return a zero result, signifying a lack of incompatibility but no intersecting features.

incompatible-descriptionp (concept *pattern*) (concept *value*)

This is like

```
(not (possibly-compatible-descriptionp pattern value))
```

but is slightly faster because it tells the matcher it need not score the results.

5.1 Primary Characterizations

One notion which warrants special attention is that of the *primary characterizations* of a concept. The primary characterizations are calculated as follows. The zero-order characterizations of a concept is the concept itself. After that, the concept and its superiors are searched for **c** properties and **c** ties. Those which have not been gathered already, and are not mutually exclusive (hierarchically, as tested by **mutually-exclusive?**), are collected. This proceeds iteratively, each iteration adding in the characterizations of the concepts and their superiors gathered in the previous iteration. When finally none have been added on a given iteration, the procedure stops. Note that this requires that there not be (directly or indirectly) looped **c** paths.

enumerate-characterizations (concept *concept*)

This enumerates all the characterizations of *concept*, as described above. They are returned in enumeration order; thus, it is guaranteed that the first element of the list will be *concept*.

If we have the small environment

```
(absorb
  [bar = !baz*t bar &c !c2]
  [foo = !bar*t foo &c !c1]
  [c1 = !node-1*s c1]
  [c2 = !node-1*s c2]
  [c3 = !node-1*t c3]
  [example = !foo*c !c3]
  )
```

then, if we presume nothing of interest other than what is shown, the primary characterizations of **!example** are

```
(!example !c3 !c1)
```

Note that **!c2** is not included because it is mutually exclusive with **!c1**, which is encountered first.

5.1.1 Characterization Proximities

When the primary characterizations of a concept are enumerated, the *proximity* of the characterization found to the original concept is noted also. This is used internally by the matcher, hence is not returned by **enumerate-characterizations** (above). The *proximity* of a characterization to the concept it (directly or indirectly) characterizes is essentially just the number of **ilk** operations which are necessary to get to the characterization. Tracing "sideways", via the **c** property or tie, is not counted.

enumerate-characterizations-and-distances (concept *concept*)

This is just like **enumerate-characterizations**, but returns the distances also. For the example in the previous section,

```
(enumerate-characterizations-and-distances '!example)
```

would return

```
(!example 0 !c3 0 !c1 1)
```

Compatible-descriptionp compares the primary characterizations of a pattern and value by requiring that each primary characterization of the pattern except for the first (the pattern itself) be satisfied by some primary characterization of the value, by being **underp-or-equal.** If however one is found which is **mutually-exclusive?**, then the match is terminated. Each match found is scored based on both the total characterization proximity, and the closeness of the satisfying characterization. The scoring mechanism is described later.

5.1.2 Taming the Beast

Since in a large database the characterization enumeration process could become quite expensive while producing ever diminishing returns, the following variable exists.

%cd-characterization-iterations *Variable*

This limits the number of iterations performed in enumerating the primary characterizations of a concept. **0** means that the only primary characterization is the concept itself. **1** means the concept itself plus all others (not excluded or already covered) directly attached to it and its superiors. Each additional iteration adds all the characterizations attached to the characterizations (and their superiors) added in the previous iteration. This defaults to **1000.** Setting it to be less than **1** is not reccommended.

5.2 Functional Restrictors

The enumeration of the *functional restrictors* of a concept has similarities to, and utilizes, the enumeration of the *primary characterizations* of the concept. The enumeration steps over the primary characterizations, and for each one, looks at it and its superiors for **f** properties and ties. Each one found which is neither already noted nor mutually exclusive with one already noted, is collected.

enumerate-restrictors (concept *concept*)

This enumerates the restrictors of *concept.* It enumerates the primary characterizations to do so.

enumerate-restrictors-from-characterizations *characterizations*

This enumerates the restrictors from the given primary characterizations. *Characterizations* should be a list as returned by either **enumerate-characterizations** or **enumerate-characterizations-and-distances;** it is able to figure out which.

Compatible-descriptionp requires that each primary restrictor enumerated for the pattern be satisfied by some from the value, by being **underp-or-equal**. Additionally, if one in the value is found **mutually-exclusive?** with one from the pattern, the match is aborted. Each such feature compared is scored based on the proximity of the restrictor to the pattern or value, and the closeness of the two restrictors being compared; this is descibed in detail later.

5.3 Slot Comparison

Compatible-descriptionp offers two methods for matching slots.

5.3.1 Equal Values

The pattern given to **compatible-descriptionp** may contain somewhere along its superiors concepts with ties which are *slot frame names*. For each of these, the value being matched against is required to have a value for that slot which is **equal** to the cue of that concept in the pattern. Thus,

```
[!person*!location !boston]
```

could match against a person whose **!location** slot has a value of **!boston.**

5.3.2 Required Values

If the pattern given to **compatible-descriptionp** is a **arg** instantiation, e.g. **[!person*arg 1]**, then the value being matched against is required to have slots having values for all of the slots on the pattern. Thus, the pattern

```
[!person*arg 1
   &h [!name*u :]]
```

matches against a person that has (or inherits) a **!name** slot value.

5.3.3 Future Developments

The slot sub-matching is crying out for more generality. What is probably wanted is incorporate in the scoring a **compatible-descriptionp** of the values found for the slot; at this time, no such checking is performed at all, primarily for efficiency reasons. What is done now is to simply score a point for such a match, modified only by the distance up the superiors of the value which the slot value is inherited from. Thus, if we are matching the pattern

```
[!person*!name BILL]
```

against

```
[bill = [[!person*f !male]*!name BILL]*i 1]
```

the score is only based on the proximity of 1.

5.4 Scoring

5.5 Debugging the Database

%cd-check-db? *Variable*

Many places in the Brand X Interpreter validate the integrity of the database based on the value of ***rset** (a flag in Maclisp which tends to increase error checking), and some routines check their arguments no matter what. The matcher, however, is a special case; a few small test cases have shown that in (for example) method selection, more than 75% of the time is spent in the matcher. The matcher thus has its own flag for telling it whether it should check to see if things which it would assume are concepts actually are. This includes such things as **c**, **f**, and **h** properties, the **cue**s of concepts with **c** and **f** ties, etc. Setting this variable non-**nil** enables this checking. It defaults to **t**: putting a symbol in place of a concept on a **c** property, for example, could kill the lisp if it were not discovered.

6. Concept Evaluation

When a concept gets evaluated, there are four ways by which a value may be found.

(1) The value may already be known

(2) Some reconstruction or "canonicalization" of the concept, based on the structure of the concept, may have a value

(3) The concept may "match" some other concept in a pool of concepts dynamically maintained just for this purpose

(4) The concept itself, or some canonicalization of it, may have or inherit a means of providing a *default value* for it

6.1 Values and Binding

The "global value" of a concept is maintained on its **v** property. This property is actually a list of values; the evaluation of a concept may return multiple values. The value of a concept may be bound, again to multiple values. The local values are kept on an alist, rather than by some "shallow binding" mechanism, for two reasons: there are actually two alists maintained, so that the bindings of some concepts can be limited by method invocation (i.e., the bindings will be "local" to some procedure, as discussed in *where*), and since these alists are subject to the ordinary Lisp variable binding mechanism, the concept binding environment is properly handled by ordinary Lisp funargs. Because of this alist binding mechanism, it is *not* sufficient to check for a **v** property to see if a concept is valued. The following routines are supplied for this:

concept-boundp *concept*

If *concept* is *bound*, this returns the binding entry — a list, the **car** of which is the concept, and the **cdr** of which is a list of its values. This does *not* check for a global value (**v** property). **Eq** is used for checking, since the binding alist is in effect simulating changes in the **v** property of the concepts bound, and changing that property would not affect an **equal** but not **eq** concept.

concept-valuedp *concept*

If *concept* has a non-**nil** **v** property, this returns *concept*, else **nil**.

concept-bound-or-valuedp *concept*

This is exactly

```
(or (concept-boundp concept)
    (concept-valuedp concept))
```

This is most likely what is wanted, rather than **concept-boundp** or **concept-valuedp** alone.

Note that the above three routines only attack the first case of concept evaluation. They may be phased out and replaced by a more general mechanism for testing to see if a value can be calculated for a concept.

6.2 Concept Reconstruction

This is more the general case of concept evaluation. The **tie** of the concept is examined, and some possibly new concept is constructed in a manner dependent on that, and this *new* concept is evaluated as above. There is one built-in mechanism for doing this, plus a protocol for defining others. What is important, however, is that the canonical reconstruction of a concept be a fixed point — that is, it should re-canonicalize to itself (**eq**).

6.2.1 Slot Frame Evaluation

If the **tie** of a concept being evaluated is a *slot frame tie*, then this evaluation procedure occurs. The intent is to find the value of some slot in some concept. The *slot frame name* searched for will be the **ilk** of the original concept if the **ilk-evaluable** keyword is not present in the **slot-frame-tie** property of the tie, otherwise the evaluation of the ilk. The concept on which the slot will be searched for will be the **cue** of the original concept if the **cue-evaluable** keyword is not present, otherwise the evaluation of the cue.

The concept which is the canonical representation of this is the slot frame on the (possibly evaluated) cue which has an *ilk* of the (possibly evaluated) ilk. That slot frame need not exist; the value can be inherited, as discussed earlier, and if the canonical representation is needed, that slot frame will be created if it does not exist. If we have

```
[person = !animate-entity*s person
   &h [!location*u :]]
[bill = !person*i bill]
```

then an attempt to find the canonical representation of

```
[!location*q !bill]
```

will cause a new concept to be created and attached to **!bill**:

```
[bill = !person*i bill
   &h (!location*uq :)]
```

The **tie** used for the creation of this new concept is determined from the **slot-frame-name** property of **!location**. The slot frame tie **uq** results in a fixed-point because it has neither the **ilk-evaluable** nor **cue-evaluable** keywords in its **slot-frame-tie** property, thus causing a re-canonicalization of it to refer to itself by refering to the **!location** slot of **!bill**.

6.2.2 Programmable Evaluation

Discussion defered; under development.

6.3 Available Objects

This is an experimental mechanism by which one can cause a concept to be evaluated finding a "best match" out of a pool of concepts dynamically maintained for that purpose. It is used for concepts which do not have "special evaluation" indicated by their **ties**, and which do not already have values. There are two different ways in which one can add or delete concepts from this pool. One way is specific to evaluation of a form whose **car** is a concept; discussion of this is defered. The other is to use one of the following special forms for the purpose. Note that all four of these special forms evaluate their first argument to determine the object or list of objects they will hack.

using-object *Special Form*
using-objects *Special Form*

```
(using-object object
    form-1 form-2 ...)
(using-objects list-of-objects
    form-1 form-2 ...)
```

evaluates each of the *form-i* in an environment where the *object*(s) specified are made "available for use".

inhibiting-use-of-object *Special Form*
inhibiting-use-of-objects *Special Form*

```
(inhibiting-use-of-object object
    form-1 form-2 ...)
(inhibiting-use-of-objects list-of-objects
    form-1 form-2 ...)
```

evaluates each of the *form-i* in an environment where the object(s) are no longer "available for use".

One may then, for instance, do

```
(using-object '!knife-1
    compute)
```

and presumably if *compute* attempts to evaluate **!knife**, it would evaluate to **!knife-1** rather than causing an unvalued-concept error (assuming, of course, that **!knife** does not already have a value!).

This mechanism is subject to change. At this time it is unclear whether it is reasonable the "use" of an "object" to be scoped as it is. It is also possible that a more specific mechanism, such as "offering" objects to *specific procedures* rather than to "just anyone" may be a better strategy.

6.4 Default Values

If a concept or its canonical reconstruction has no value, has no match from the *available object* mechanism (previous section), and inherits (via **look-for-inherited-properties**) a **get-default-values** property, then the first element of that property should be a Lisp function. That function is called with two arguments: the concept (or reconstruction), and the **cdr** of the property (which could be additional information for the function). It should return a *list* of the values of the concept. If it so desires, it could give the concept a value using the **cset** special form (page 24).

This hook could therefore be used for driving question asking, or simply to provide a default value.

6.5 Modifying Values

Here is where the fixed-point of concept canonicalization comes in. If it is desired to change the value of a concept, it is necessary to find the concept whose value is to be changed. The mechanism by which this is done is to pseudo-evaluate, or canonicalize, the concept. This result is then the concept whose value is to be modified. The reason a fixed point is desired is that this canonicalized concept may be passed around and get recursively re-processed, by perhaps **cset** from within a routine which asks for the value.

csetf *Special Form*

(**csetf** *concept value*)

This changes the value of *concept* to be *value*. Specifically, this finds the canonical representation of *concept*, which is *not* evaluated other than subevaluations performed for canonicalization purposes, and sets its bound or global value (as appropriate) to *value*. If *value* returns multiple values, all of them will be transfered.

```
(csetf [!location*q !bill] '!boston)
```

Note the use of the **q** tie to inhibit any subevaluation of **!location** or **!bill**.

cset *Special Form*

(**cset** *concept value*)

This is just like **csetq**, but the *concept* form is evaluated in order to get the concept which is to be canonicalized. That is, the example under **csetq** could be re-written as

```
((lambda (var)
    (cset var '!boston))
  '[!location*q !bill])
```

6.5.1 Checks

When a concept is being assigned a value or values, some checks are made on the assignment. One of the checks made is whether or not the concept may have as many values as it is being assigned. Currently the only time this check actually gets made is when the concept is a slot frame; in that case, the number is determined from the **slot-frame-tie** property of the tie of the (canonicalized) concept. The only distinctions made are none, one, or many, as determined from the presence or absence of the **single-valued** and **multiple-valued** keywords. If the number is incorrect, then the **wrong-number-of-values** condition will be signalled (via **fail**, as always); that condition gets extra arguments of the concept being assigned, and the list of values. Another check made, also only to slot frames, is whether or not the concept is read-only. If this keyword is present, then the **assignment-to-read-only-concept** is signalled, in the same manner. Both of these checks are of questionable utility by being applicable only to slot frames, and may be flushed in favor of a more comprehensive mechanism.

check-value-compatibilities (concept *concept*) *value-list* (Optional *sum-up-matches?*)

Another check made during concept assignment (and binding) is that of value compatibility. This routine attempts to determine if each of the values in *value-list* is compatible with *concept*. Additionally, if *sum-up-matches?* is not **nil**, it attempts to quantify the closeness (ala **compatible-descriptionp**). If the concept inherits a **value-restrictions** property, then that alone is used to determine if each of the values is compatible with the concept. The property should be a list of functions, which will be called in order on three arguments:

concept
: The (canonicalized) concept which is being assigned

value The value (or one of the values) it is being given

sum-up-matches?
: This flag is passed along. If it is given and it is not obvious what should be returned, **1.0** is a good default.

The function should return **nil** if the value is not compatible with the concept, otherwise something else, which should be a flonum if *quantify-match?* was not **nil**. *There should be some initially supplied value-restriction functions for common cases such as fixnum, flonum, number.*

If a **value-restrictions** property was not found, then each value must be a concept, and **possibly-compatible-descriptionp** (or **incompatible-descriptionp**, depending on *sum-up-matches?*) is used.

If an incompatibility is found, then the **incompatible-description** condition is signalled. This gets extra arguments of the value which is incompatible, and the concept.

7. Interpretation

When a form (either a list, clist, or ulist) is encountered by the Lisp evaluator and its **car** is a concept, control is passed to the concept evaluator. The concept evaluator then evaluates the arguments (if any) to the concept, and attempts to find a procedure which is suitable for both the arguments, and the external environment. The mechanism for this is accomplished by having *partial definitions* or *methods* for the concept being applied, each of which can have multiple *procedures* with *prerequisites*. The prerequisites, in turn, can optionally have *subgoals* which are to be recursively evaluated to attempt to satisfy the prerequisite if it is not already true.

7.1 The General Behaviour

```
(define-methods !hit
  ([!hit !physical-object]
     (require requirement-1 requirement-2 ...)
     (prerequisites prereq-1 prereq-2 ...)
     (steps step-1 step-2 ...))
  ([!hit !person]
     (require requirement-1 requirement-2 ...)
     (prerequisites prereq-1 prereq-2 ...)
     (steps step-1 ...))
  ...)
```

The evaluator looks at the concept being applied and its superiors. At each level, it selects those partial definitions (methods) whose formal parameters "match" the argument and whose **require**ments are satisfied, and orders them according to an estimation of the number of matching features in common.

The **require**ments of a method are simply concepts which must be evaluable for this method to be viable. This test effectively uses the normal concept evaluation procedure, described earlier. The "closeness of fit" of the value(s) found to the required concepts is quantified and aids in selecting the "best" method.

Once the methods have been pruned and ordered by this matching procedure, they are tried in order. A binding environment is established, pairing the elements of the formal parameter list to the calling concept and its arguments. Then, each of the **prerequisites** to the method are evaluated, in order. A prerequisite is simply a form to be evaluated as a predicate; if it evaluates to **nil** then it is not satisfied, and that method will be aborted. If however the prerequisite has a **subgoals** property, then that is a list of forms which should be evaluated in order to "satisfy" the prerequisite. They will be evaluated, and the prerequisite evaluated again to see if it has indeed become satisfied. During the evaluation of the subgoals of a prerequisite, the **prerequisite-failure** condition is enabled such that if **fail** is called with that condition, the subgoal evaluation will be aborted, and the method being tried will be punted.

Finally, when all of the prerequisites of a method have been satisfied, the **steps** are evaluated, in order. If all goes well, the value of the last step will be returned as the value of the original application. During the evaluation of the steps, the **method-failure** condition is enabled. If it is signaled, then that method will be aborted and the next method tried. Note that it is up to the user to restrain the usage of this condition so that harmful side-effects do not occur before it is used in a method body.

7.2 Matching and Selection

An argument is said to match a formal parameter in a partial definition if it satisfies **compatible-descriptionp** (page 16) of that formal parameter. A method is considered to match the call if all of the formal parameters of the method match the calling concept and its arguments, respectively. It is deemed not to match if the number of formal parameters is not the same as the number of arguments. It also does not match if some of the **required** concepts could not be evaluated. The quantification of the fit of the method to the application is the sum of all of the **compatible-descriptionp**s performed:

(1) The first element of the formal parameter list against the calling concept

(2) Each of the arguments of the application against the corresponding remaining elements of the formal parameter list

(3) The value(s) of each of the **required** concepts against that concept.

The methods thus pruned are ordered by this sum.

Note also that the above occurs for each superior of the concept being applied until a method gets successfully executed. Possibly what should happen is that the entire set of partial definitions of all the superiors will be considered at once.

In order to allow partial definitions which need to contain "identical" formal parameters for matching, the formal parameters may be individualized by making a new concept with a **tie** of the atom **arg**. That is, if one needs a partial definition of the form

```
[!frobnicate !object !object]
```

one may use instead

```
[!frobnicate [!object*arg 1] [!object*arg 2]]
```

The matching (proximity calculation) will be performed on **!object**, but the concepts bound will be **[!object*arg 1]** and **[!object*arg 2]**.

7.3 Events

The interpreter maintains its own "stack" in a structure known as the **event structure.** An event is a concept; its **ilk** is the "previous" or "calling" event, i.e. the next frame up the stack. The tie is a concept which describes what that stack frame was created for, and the cue is data the interpretation of which depends on the tie.

At present, events are created (pushed) at the following times:

(1) When a combination is evaluated. The tie is the concept **!call**, and the cue is the form being evaluated.

(2) When a method (partial definition) is being tried. The tie is the concept **!application**, and the cue is a cons of the partial definition, and the (evaluated) argument list. This event formation also corresponds to a change in the binding environment; the event will have some property or properties from which the concept binding environment may be re-created.

(3) When a procedure is tried. The tie is the concept **!procedure**, and the cue is the procedure being tried.

(4) Various other operations may produce special-purpose event frames. Eventual design decisions will determine whether these should be flushed or documented, or whether access to the event structure should be provided only through special routines.

A point which needs to be settled is whether or not the event structure should be susceptible to **funarg**ing. One can currently create and use downward funargs; they will correctly use the concept binding environment in effect at the time of creation of the funarg. They will also use the event structure in effect then also (and whatever else is attached to the event structure, such as "available objects", section 6.3, page 23). Since the event structure is intended to correspond to a "control stack" rather than the binding environment, this is of debatable correctness. This is fixable (by using Lisp's **unwind-protect**), but not without impairing efficiency by some unknown amount. Whether this is worthwhile is dependent, like the handling of failure, on how the event structure is to be used.

It is possible that the interpretation process as seen through the event structure will become more finely divided, in order that the process of procedure selection may be remembered and continued at some later point. Currently such developments are speculative, and are dependent on decisions about what is useful and reasonable for the evaluation process.

7.4 Side Effects

Attempts to handle side effects in view of failure are being punted. (At this time the only operation safe from side effects before failure is the binding of the formal parameters of a partial definition.) It is probably impracticable to handle them correctly in general anyway. Note, however, that the use of prerequisites rather than failure inside of a procedure can be used as an optimization for the prevention of side effects, if the convention is assumed that evaluation of a prerequisite has no side effects. This is unfortunately not the case if the prerequisite has subgoals; more thought is needed here.

7.5 Defining Methods

Two routines are provided for defining methods for a concept.

define-method *Special Form*

```
(define-method on-concept method
    (require require-1 require-2 ...)
    (prerequisites prereq-1 prereq-2 ...)
    (steps step-1 step-2 ...))
```

The **require** and **prerequisites** clauses are optional, and none of the "arguments" are evaluated. Essentially, this gives *method* **require**, **prerequisites**, and **steps** properties of the corresponding items (or removes them if the clause is absent), and adds *method* (using **addp**) to the **methods** property of *on-concept*. Because of this, it is probably a good idea for *method* to be unique (if not canonical).

define-methods *Macro*

This allows one to define several methods at once; the format is shown in the example on page 26.

8. Multiple Values

The Brand X interpreter contains a module simulates a scheme for allowing multiple value returns, as implemented on the Lisp Machine. It works properly as long as the following discipline is maintained:

(1) One only attempts to retrieve multiple values from something which returns multiple values

(2) Expressions which *ever* return multiple values should *always* exit via a special form (such as **values**) which causes return of multiple values.

The reason for the latter is that ordinary Lisp evaluation is totally transparent to the passing back of multiple values (in this particular implementation), so in cases where (say) a function may return one or two values, the one value case needs to explicitly reset the passed back extra values. There are, of course, special cases where one cannot know; a mechanism for handling this is discussed in the last section of this chapter, but that is primarily for use by the evaluator itself, or the construction of special forms.

8.1 Passing Back Multiple Values

values *first-form* (Any-number-of *other-forms*)
Values takes one or more arguments, and returns them all as multiple values.

mv-return *required-form* (Any-number-of *additional-forms*)
This takes one or more arguments, and **returns** them as values from the current **prog**, **do**, or **loop**.

multiple-values-from-list *list*
This returns the elements of *list*, which must have at least one element, as multiple values.

return-list *list*
This returns the elements of *list*, which must have at least one element, as values from the current **prog**, **do** (or **loop**).

8.2 Receiving Multiple Values

multiple-value-list (Unquoted *form*)
`(multiple-value-list` *form*`)`
returns a list of all the values returned by the evaluation of *form*.

multiple-value *Macro*

(multiple-value *set-list form*)

is one way to retrieve multiple values from the evaluation of *form*. *Set-list*, which does *not* get evaluated, is a list of atomic symbols or concepts; they will be set to the corresponding values returned by the evaluation of *form*.

mv-bind *Macro*

(multiple-value-bind *bind-list expression*
form-1 form-2 ... form-n)

This is similar to **multiple-value**, except that the elements of *bind-list* are bound to the values returned by the evaluation of *expression*, and each of the *form*s evaluated in that environment. The value (or values) returned by the last form is returned. By special dispensation, one may specify the data type of an atomic symbol in *bind-list* by listing the data type name (which *must* be an atomic symbol) and the variable. For example,

```
(multiple-value-bind ((fixnum quo) (fixnum rem))
                     (quorem foo bar)
    (compute-a-lot))
```

This is useful for declaration purposes.

8.3 Other Special Forms

mv-prog1 (Unquoted *first-form*) (Any-number-of (Unquoted *other-forms*))

This is like **prog1**, but correctly passes back the all of the values returned by the first form. **Prog1** should not be depended on to do so.

mv-prog2 (Unquoted *first-form*) (Unquoted *second-form*)
(Any-number-of (Unquoted *other-forms*))

Like **mv-prog1**.

mv-impasse (Any-number-of (Unquoted *forms*))

(mv-impasse *form-1 form-2 ...*)

is just like **progn** but guarantees that only one value will be returned.

8.4 Multiple Value Pairs

Here are some routines and macros for handling cases where one needs to efficiently remember and then transparently pass back multiple values. The idea is that we collect the multiple values in two places: the first is passed back as the value (of **multiple-value-pair**), and the remainder are consed in the heap and remembered by side-effect. Thus there is no heap consing in the common case where only one value is returned. These routines are primarily useful to things like the evaluator itself, and to implement such things as **mv-prog1**.

multiple-value-pair *Macro*

(multiple-value-pair *form setf-form*)

evaluates *form*, returning its first value, and as a side effect stores a list of the remaining values into *setf-form*, using the **setf** special form. Typically *setf-form* will simply be a variable name. The type of datum it will receive should not be depended on.

multiple-values-from-pair *first-value remaining-values*

This returns multiple values as collected by **multiple-value-pair.**

return-pair *first-value remaining-values*

This effects a **mv-return** of multiple values as collected by **multiple-value-pair.**

9. Format

This chapter is included in lieu of a separate, more comprehensive document. It is not intended to be complete or proper documentation of **format**, but is included so that minimal use of it may be made using **signal** via **fail**.

format *destination control-string* (Any-number-of *args*)

Format, in the simplest case, outputs the characters of *control-string* to *destination*. When a tilde character is encountered, the following character is taken as a dispatch command, to produce special output, possibly utilizing the extra *args*. (Special flags and parameters may be inserted between the tilde and the dispatch character; they will not be considered here.) *Destination* may be:

nil Rather than performing I/O, the characters which would have been output are returned as a string (uninterned symbol in Maclisp).

t t as an output file specification means the terminal in Maclisp; however to **format**, it means "the default", i.e. what you get when you **print** something with no file specification argument.

any other valid Maclisp output file specification
Just outputs the characters there.

Here are a few of the common **format** dispatch characters:

~A **princ**s the next argument.

~S **prin1**s the next argument.

~D Outputs the next argument in decimal (but with no decimal point after integers).

~% Outputs a newline

~& Does a **fresh-line** operation — if the destination stream of **format** is already at the start of a new line, does nothing, otherwise outputs a newline.

Examples:

```
(format nil '|Here are three dots: ~A and four: ~S|
    '|...| '|....|)
  => |Here are three dots: ... and four: /|.../||
(format t '|~&The number ~D was not equal to ~D.~%|
    259. 403.)
```

prints, at the start of a new line,

```
The number 259 was not equal to 403.
```

and follows it with a newline

Thus,

```
(fail 'unvalued-concept
      '|The concept ~S is not valued|
      '!person)
```

could result in the following message (possibly embedded in other text):

```
The concept !PERSON is not valued
```

Index