MIT/LCS/TM-200

LSB Manual

Glenn Burke

June 1981

LSB Manual

June 1981

Glenn Burke

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Support for this research was provided in part by National Institutes of Health grant number 1 P41 RR 01096-04 from the Division of Research Resources, and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

Abstract

LSB (for Layered System Building) is an integrated set of facilities for aiding in the construction of highly-modular, multi-layered, implementation-independent Lisp systems. It provides for conditional inclusion of source text, documentation production, automated declarations, and "high-level" definitions. Lisp code compiled with LSB in general does not require LSB in its run-time environment. LSB has been in use for some time in PDP-10 Maclisp, is operational in Multics Maclisp and Lisp Machine Lisp, and is being developed for NIL.

Acknowledgments

The rudiments of LSB were first conceived by Lowell Hawkinson in early 1978, and he and John Thompson implemented a prototype version for Maclisp shortly thereafter. Since then, Lowell has provided a constant source of ideas and inspiration that has been invaluable to the design of LSB.

Among those individuals who have contributed strongly to the development of LSB through their use of it are Bill Martin, Peter Szolovits, Bill Long, Ramesh Patil, Harold Goldberger, Ken Church, and Brij Masand. Providing for their many needs and reacting to their assorted experiences has much improved the design and presentation of LSB's facilities. David Moon, Howard Cannon, and Peter Szolovits deserve special thanks for their comments on various specific aspects of LSB.

This manual is dedicated to Bill Martin, who provided most of the ultimate support for the work it presents. Never one for fancy programming tools, he nonetheless embraced LSB because of its contributions to the buildability of large, layered systems — systems that might be made to exhibit intelligence and expertise of a high order.

Note

Any comments, suggestions, or criticisms will be welcomed. Please send Arpa network mail to BUG-LSB@MIT-ML.

Those not on the Arpanet may send U.S. mail to

Glenn S. Burke
Laboratory for Computer Science
545 Technology Square
Cambridge, Mass. 02139

There is also an Arpa network mail distribution list for announcements pertaining to LSB. Contact the author to be placed on it.

Table of Contents

1. Introduction

LSB (for Layered System Building) is an integrated set of facilities for aiding in the construction of highly-modular, multi-layered, implementation-independent Lisp systems. It provides for conditional inclusion of source text, documentation production, automated declarations, and "high-level" definitions. Lisp code compiled with LSB in general does not require LSB in its run-time environment. LSB has been in use for some time in PDP-10 Maclisp, is operational in Multics Maclisp and Lisp Machine Lisp, and is being developed for NIL.

The basic idea is that when a programmer defines something, it should not be necessary to *declare* it separately; after all, the definition itself usually makes a separate declaration redundant. Definitions should be able to be located where style and modularity dictate; their positioning, both with respect to location in a particular source file and the position in the file, should not depend on the basis of the vagaries of the Lisp compiler. Documentation, especially on a local level, is also best positioned in close proximity to what it describes. This allows for ease of updating the definition and the documentation together, reducing the tendency for them to diverge as development progresses.

It is becoming increasingly common for the same source code to be used in different Lisp implementations. The programmer should be able to easily make use of implementation-specific facilities, by having some mechanism for *conditionalizing* the source code. On the other hand, the use of many facilities, particularly in the realm of function definition and type declarations, can be specified at a high level, independent of any particular Lisp implementation.

In LSB, definitions are aggregated into *modules*, and *modules* into *systems*. From the source text for a module, LSB derives the information necessary to compile and run code of that module, of other modules of that system, and of modules in other systems which utilize that system. A sample LSB system can be found in chapter 15, page 78.

1.1 Conventions of this Manual

The reader of this manual is expected to have a working knowledge of Lisp, particularly one of the dialects for which LSB exists (Lisp Machine Lisp [LMMan], Maclisp [Moonual], and NIL (VAX) [NILDoc]). Terms which are in common use in these dialects will be used freely here.

The LSB environment establishes an initial default input radix of decimal. Integers in text and in examples should be so interpreted except where explicitly counterindicated.

In Lisp code examples, the symbol => should be read as "evaluates to", and the symbol ==> as "macro-expands to".

All Lisp code in this manual is written in lower case. Although most of the Lisp implementations LSB runs in canonicalize lower-case symbols to upper-case, Multics Maclisp does not. All LSB routines, macros, and variables are lower case in Multics. In general, in a reasonably standard reading environment, atomic symbols may be typed in just as they are shown in here. All atomic symbols in this manual which are shown with colons in their names, such as `:previously-loaded` or `lsb:token-equal`, are, in Maclisp, actually symbols with exactly that print-name, colon and all. In Lisp implementations with packages, the package prefix part of the name is intended to be interpreted as just that. Thus, the same source code can refer to these names in the same way in different Lisp implementations.

The null value is represented by the symbol nil, and its canonical complement by t. In the NIL implementation, LSB will accept the atomic symbol t in various places as meaning #t or "truth". Most notable of these are as an "argument" to a two-state definition option, such as do-argument-type-checking, and as the predicate of a diversion stream. No such compensation will be guaranteed for the symbol nil: if one is writing source code to be used in NIL, use () instead.

Certain Lisp implementations have missing functionality, which makes it difficult, inefficient, or impossible to implement certain LSB features. These deficiencies are noted in this manual without further explanation. Most notable in this respect is Multics Maclisp, which has a severe lack of functionality in dealing with the Multics file system.

In various places in this manual, atomic symbols may be referred to as *keywords*, and the terms *keyword equality* and *token equality* may be used. LSB token equality is simply case-independent print-name equality; this is used in many contexts to make things independent of packages or case distinctions. *Keyword equality* is token equality extended to allow synonyms (a table of which is given in chapter 16). The internal LSB primitives used for this are documented in section 12.1, page 63.

2. Overview

This chapter is intended as a descriptive explanation of how LSB works and what it can do. It should provide a sufficient basis for casual and simple use of LSB. Precise and much more detailed definitions of the facilities presented here occur in later chapters.

2.1 Derivability

One of the primary precepts of LSB is *derivability*. That is, it is assumed (although not necessarily enforced) that all the information necessary to understand, compile, and execute some code is derivable from a single source text. Thus, LSB provides facilities for

conditional inclusion
: One may specify that parts of the source text are applicable only in certain environments (e.g., certain Lisp implementations), and therefore a single source text may serve as the source in many environments.

text diversion
: Portions of the source text, while treated as comments by Lisp itself, may be copied elsewhere (possibly transformed), perhaps to serve as documentation for the code.

form diversion
: Lisp objects may be sent to various places, to provide information needed for compilation, execution, debugging, etc.

combined definition and declaration
: The definition facilities in LSB use the *form diversion* mechanism to produce the declarations appropriate to the type of object being defined (e.g., routine, macro, variable). All of the information needed for these declarations can be included in the definition syntax itself, obviating the need for separate, and potentially implementation-dependent, declarations.

module and system organization
: *Modules*, corresponding to the source files, are grouped into *systems* and share information which need not be propagated elsewhere. Definitions of *systems* specify what other LSB systems they use, and may be used to customize the processing (e.g., loading, compilation) environment to be used for each module.

2.2 Visibility Classes

LSB associates with every defined object (e.g., routine, macro, or variable) a *visibility class*. This is essentially an indicator of "how far" information about the object should propagate. The three visibility classes provided are:

private	restricted to the module
system	restricted to the system containing the module
public	intended for explicit use by users of this system

The system visibility class used to be named intrasystem. That name is still accepted, and occurs frequently in existing code: define-intrasystem-routine is defined to be the same as define-system-routine, and intrasystem-compilation is synonomous as a keyword with system-compilation, for example.

LSB, using visibility classes, does not attempt to resolve naming conflicts. It makes no use of *packages* or other name resolution techniques, where such exist. A visibility class is used only for informational (e.g., declaration and documentation) purposes. (This may not be the case in future Lisp implementations where package organization is not limited to a strict hierarchy.)

2.3 Definitions

Everything in LSB gets "defined", even special variables. An LSB definition provides a means for specifying all necessary information about what is being defined, and thus allows that information can be localized. This includes such things as default or required initialization (for special variables), argument types (for routines), and the value type (for both).

2.3.1 Routines

Routine definition is LSB's counterpart to Lisp's function definition. In a routine definition, rather than specifying a function name and an argument list, one specifies a *prototype call*. This is a form which describes what a call to that routine looks like. For example,

```
    (print-decimal-number number (optional stream))
```

describes a call to the routine print-decimal-number, which can take one or two arguments. The prototype call tells LSB what variables ("formal parameters") are to be bound, the data types of those variables, how many arguments the routine may take, the data types of those arguments, and how to process a call to the routine.

Call processing is the mapping of a *call* into a lambda expression which takes a fixed number of arguments. Because of the information available to LSB, it is able to decide at what point (from compile time to run time) this mapping may be made most advantageously.

A full definition of print-decimal-number might look like

```
    (define-public-routine (print-decimal-number
                                number (optional stream))
        ((lambda (base *nopoint)
            (princ number stream))
         10. t))
```

print-decimal-number is a *routine*, as opposed to (for example) a *macro*. When only one argument is given to it, *stream* will be bound to nil. The prototype call might have been written as

```
    (print-decimal-number
        number (optional stream standard-output))
```

if it were desired that the value of the variable standard-output should be used in lieu of a missing second argument.

The following defines the routine square$, which operates only on flonums:

```
    (define-public-routine (square$ (flonum n))
        (declarations (value-type flonum))
        (*$ n n))
```

It also shows the declaration format for LSB routine and macro definitions. In general, the cdr of the declarations form is an a-list of keywords and any other information specific to that declaration clause. The value-type declaration says that this routine always returns a flonum. declarations may be abbreviated as dcls. Multiple declarations forms may be used, as long as they all precede any code.

macros may be defined in a form similar to routines; a prototype macro call has the same format as a prototype routine call.

```
      (define-public-macro (foo x)
          (list 'caddar x))
```

defines foo as a macro that is synonymous with caddar.

2.3.2 Variables

In LSB, one can "define" special variables as well as routines. Having a definition provides a distinct locus for giving attributes to the variable. In the definition, one may specify such things as the type of the variable, and its initialization. The expression

```
      (define-public-variable *print-stm-props?
          (default-initialization nil))
```

defines the special variable *print-stm-props?. A variable definition provides for the initialization of the variable, if any, and appropriate declarations are generated to tell the compiler about the variable. The default-initialization (or default-init) declaration says that if the variable is not valued, it should be set to the value of the form specified, in this case nil. One could cause the variable to be unconditionally initialized by using initialization (or init) instead. The type can be specified, as in

```
      (define-system-variable *stack-level
          (value-type fixnum)
          (default-init 0))
```

2.4 Conditional Inclusions

LSB provides a mechanism for conditional inclusion of forms or subforms in a source file, using the characters left brace ({) and right brace (}). When a left-brace is encountered by the Lisp reader, a form (the *inclusion test*) is read. That form should be a list whose car is an atomic symbol (the *inclusion tester*). The inclusion test is interpreted to determine either "success" or "failure". A failing test causes all of the text through the matching right-brace to be discarded, effectively making it disappear from the input stream (as far as Lisp's read is concerned). Otherwise, the text is left alone, and the matching right-brace will be ignored (treated as a blank) when encountered. Thus,

```
      (define-private-routine (make-a-frob size)
          {(only-for Lispm) (make-array size ':type 'art-q)}
          {(only-for Maclisp) (*array nil t size)})
```

is like having

```
      (define-private-routine (make-a-frob size)
          (make-array size ':type 'art-q))
```

on the Lisp Machine, but

```
      (define-private-routine (make-a-frob size)
          (*array nil t size))
```

in Maclisp. The only-for inclusion tester checks for implementation features of the environment, e.g., (status feature Lispm).

Additionally, if the inclusion test is one of the atomic symbols `--` or `-*-`, then the text within the braces is *always* ignored, and acts as a comment. The `--` keyword is an "em-dash", obscurely derived from ADA. The `-*-` is for use in "file property lists", as used on the Lisp Machine, and also by the Emacs editor. Thus, one may put at the very beginning of a file something like

```
{-*-  Mode:Lisp;  Package:Format  -*-
      random text treated as comments}
```

One may use a logical composition of implementation features in place of a simple one. For example,

```
{(only-for (and string Multics)) ...}
```

is the same as

```
{(only-for string) {(only-for Multics) ...}}
```

Any composition of and, or, and not may be used in constructing these "feature predicates". Multiple "arguments" to only-for are treated as an implicit or; thus, (only-for Maclisp Lispm) is equivalent to (only-for (or Maclisp Lispm)).

```
{(except-for Lispm) ...}
```

is the same as

```
{(only-for (not Lispm)) ...}
```

In general, (except-for *frob-1 frob-2 ... frob-n*) is the complement of (only-for *frob-1 frob-2 ... frob-n*). There are some other inclusion testers, similar to only-for and except-for, which allow the use of multiple "feature environments", in order to facilitate cross compilation. These are discussed in chapter 4, page 24.

An inclusion test which is an atomic symbol (and not `--` or `-*-`) is treated as a simple implementation feature test, and interpreted as (only-for *test*). Thus,

```
{Lispm ...}
```

is the same as

```
{(only-for Lispm) ...}
```

Conditional inclusions may conditionalize any number of forms. They may also be nested, to allow successive selections and defaultings, as in

```
(define-private-routine (make-a-frob size)
   {(only-for Lispm)
      (make-array size ':type 'art-q)
      }
   {(except-for Lispm)
      {(only-for NIL)
         (make-vector size)
         }
      {(except-for NIL)
         (*array nil t size)
         }
      }
   )
```

2.5 Diversion Streams

Diversion streams are an abstraction used by LSB to handle *derivation* (see section 2.1). An LSB definition processor partitions the various pieces of information it gathers, according to their intended uses, and then "sends" them to appropriate diversion streams. The effects of this partitioning range from immediate evaluation, to outputting text into a file, to compiling Lisp forms into a file.

In the routine definition

```
(define-public-routine (square$ (flonum n))
    (dcls (value-type flonum))
    (*$ n n))
```

there are two different kinds of information data which need to be dealt with. One is the definition of square$ itself, and the other is the set of declarations needed to compile square$ and calls to square$. Using primitive Maclisp, that would be written out as

```
(declare (*expr square$) (flonum (square$ flonum)))
```

and

```
(defun square$ (n)
    (declare (flonum n))
    (*$ n n))
```

LSB arranges for declarations similar to those in the declare form above to be sent to the compilation-environment and pubdcl diversion streams. The first acts as a no-op when the definition is being processed at run time, but at compile time causes "immediate evaluation", as if the compiler had seen a corresponding declare form. The pubdcl (*public declarations*) diversion stream also does nothing at run time. At compile time, it causes the declaration forms to be written into a file, so that they can be used for the compilation of other modules. There is a declaration diversion stream for each visibility class; complementing pubdcl are sysdcl (*system declarations*) and moddcl (*module declarations*).

In addition to form diversion streams, there are also diversion streams for text. Three standard text diversion streams are pubdoc, sysdoc, and moddoc. Text diversion streams are used in a different manner than form diversion streams: an extension of the conditional inclusion mechanism allows the *excluded* text to be *diverted* to any number of of these streams. Typically, these diversion streams simply output the text to a file. By means of text diversion, a single source can contribute to multiple levels of documentation. The documentation files of the same visibility class for several modules could be recombined to produce (say) a manual. In this manner, one can have the documentation of some code or of a module itself located with the source text it describes. For example, one might include in some module something like

```
{(public-documentation)
.chapter "The Matcher"
        The pattern matcher in FROBOZZ is guaranteed to
solve all of the world's problems.  In order to do this,
blah blah blah.}
```

2.6 Definition Availability

When objects are defined, it is sometimes necessary to specify where the definition is to be diverted to. The most common case is that of a routine which is called by a macro. LSB needs to be told to divert the routine to the same place(s) that the macro goes. The needed-for declaration clause handles this. There may also be random forms (e.g., calls to defprop) which need to be diverted in a similar manner; the forms-needed-for special form will do the same for arbitrary forms.

Consider, for example,

```
(define-private-routine (bar x y)
    ...)
(define-public-macro (foo arg)
    (bar arg t))
```

Each definition type (e.g., define-*visclass*-routine, define-*visclass*-macro) has a default needed-for declaration, which may be overridden, or just added to (by use of the also-needed-for declaration). In this example, the simplest way to achieve the desired effect is to define bar using the also-needed-for declaration

```
(define-private-routine (bar x y)
    (dcls (also-needed-for public-compilation))
    ...)
```

which says that not only do we need the definition of bar when we are running (the default for ordinary routine definition), but also during "public compilation". The way the default declaration could be totally overridden (in this case for the same effect) is

```
(define-private-routine (bar x y)
    (dcls (needed-for running public-compilation))
    ...)
```

The needed-for keywords which may be used are:

running
: This corresponds to the toplevel diversion stream. The definition will be output in the compiled output file, or to the interpreter.

interpretation
: This corresponds to the interpreter diversion stream. The definition will be made at run time, but not in the compiled output file (unless that is implied in some other way).

public-compilation
system-compilation
private-compilation
: These map into *both* the declaration diversion stream of the appropriate visibility class (pubdcl, sysdcl, or moddcl), *and* the compilation-environment diversion stream.

compilation
: Just like the previous three, with the visibility class determined from that of the object being defined.

any diversion stream name
: This is to be used in case the above keywords are insufficient, as they would be if one is defining one's own diversion streams.

A typical situation where also-needed-for and forms-needed-for are called for is

```
(define-system-macro (hack-it a b)
    (do-it-up a b (get a 'hack)))
(define-private-routine (do-it-up x y z)
    (dcls (also-needed-for system-compilation))
    ...)
(forms-needed-for (running system-compilation)
    (defprop foo foo-hacker hack)
    (defprop bar bar-hacker hack)
    (defprop baz baz-hacker hack))
```

Occasionally one may have a routine or variable which may be implicitly referenced at a higher visibility class than it is defined at. Obviously one could change its visibility class, but that may not be the appropriate solution. Typically, the visibility class corresponds most closely to the documentation; it is chosen on the basis of who should know about it. Take the case of a macro hairy-frob, which expands into code which calls the internal routine hairy-frob-internal:

```
(define-public-macro (hairy-frob this that)
    (list 'hairy-frob-internal
          this that t 0 ''hairy-frob))
```

In this situation, the declarations for hairy-frob-internal need to be public. The proper way to achieve this is to use the referenced-at-visibility-class (or reference) declaration in the definition of hairy-frob-internal:

```
(define-private-routine (hairy-frob-internal
                          this that flag
                          start-count caller)
    (dcls (reference public))
    ...)
```

2.7 Modules and Systems

In order to utilize the information derived from other modules during the compilation of one, LSB requires that modules be organized into *systems*. Each system must be defined to LSB, to say what other systems it utilizes, what modules it contains, where to find it in the file system, and environmental options, such as what input radix should be used. One puts a form like

```
(module print xlms)
```

at the front of a module to tell LSB what module and system the file corresponds to; in this case, the print module of the xlms system. (Module and system names are compared by LSB for token equality.) When this form is processed by the Lisp interpreter or compiler, LSB sets up the environment necessary for the interpretation (loading) or compilation of the remainder of the file, as specified by the system definition. If the source code is to be used in a Lisp implementation which supports "file property lists" (such as Lisp Machine Lisp), one should *also* use the lsb option in the file property list:

```
{-*- Mode:Lisp; LSB:Print,XLMS -*- 1-Jun-81
   Copyright (c) 1981 by Grandiose System Building
   and Massachusetts Institute of Technology.
   All rights reserved.}
```

This will then allow many other file property list options to be derived from the LSB system definition; most important of these are the package, readtable, and input radix.

System definitions typically reside each in a separate file; LSB has search and defaulting rules for finding them when necessary, and also has initial knowledge of a large number of commonly used systems. The system definition typically resides on the directory with the same name as the system name, a (first) filename the same as the system name, and a file-type of system. LSB will also look for it on the directory which the source file is being loaded or compiled from. For example,

```
(define-system string
     (directory amber)
     (built-on loop lbind)
     (users-implicitly-need user-hunk)
     (modules (string (needed-for-user-compilation)) strfn char))
```

is the system definition for the string system; it resides on the amber directory, as does the source for all of the modules (string, strfn, and char).

2.7.1 Pathnames

LSB allows one to interact with the host file system in such a way that in many cases the same LSB specification suffices in differing and incompatible implementations. This mechanism is based on the assumption that the following "components" of a pathname exist as a superset of all those a particular LSB implementation might handle:

host This is the name of the "host machine" the file is to be obtained from (or sent to). This is necessary in implementations where one may need to reference multiple hosts, such as the Lisp Machine.

device Whatever is meant by a "device" in the host implementation.

directories
LSB has provision for specification of a directory path.

names LSB has provision for multiple filenames.

file-type
The "type" of the file, as used (for example) TOPS-20.

version
The version number of the file.

Thus, the TOPS-20 pathname "PS:<MACLISP>DEFMACRO.FASL.259" has a device of PS, a single directory of MACLISP, a single name of DEFMACRO, a file-type of FASL, and a version of 259. That same pathname referenced from a Lisp Machine might also need to have a host specified. In general, LSB allows these components to be specified or defaulted separately, so that the same LSB specification suffices in different environments when the corresponding components are the same.

3. The System Definition

The system definition is where information not specific to individual routines, macros, or variables is kept. It tells LSB what other systems the one in question is *built on*, where various things (including source files) reside in the file system, what Lisp environmental options should be in effect, and how diversion streams should behave; much of this may be specified either per-module or per-system. For example,

```
(define-system write
         (directory format)
         (built-on loop lbind)
         (modules write))
```

defines the system write, which uses (is *built on*) the systems named loop and lbind, contains a single module named write, and resides on the format directory.

3.1 The System Definition Location

When LSB needs the definition of a particular system (say xlms), it attempts to find it if it is not already known. One way it might know where to look is through the use of the define-system-location macro:

define-system-location *Macro*

(define-system-location *system-name location*) tells LSB that the system definition for *system-name* may be found in the file named by the pathname *location*. For our example, someone could have done

```
(define-system-location xlms "XLMS;XLMS SYSTEM")
```

since "XLMS;XLMS SYSTEM" is the pathname of the file which contains the system definition for xlms on the MIT-ML host. When this form is seen by the compiler (at top-level, like a defun) it is treated such that the location of the system definition is defined at compile time, and also when the compiled output file is loaded (that is, it uses (eval-when (eval compile load) ...) implicitly).

The pathname specified with define-system-location need not be complete; it may contain unspecified components, which will get filled in: LSB has searching and defaulting rules which in many cases obviate the need for define-system-location completely.

3.2 Searching for the System Definition

When LSB does not already know a system definition, it searches for it in a prescribed manner. This searching is done even if a system definition location has been specified, because that system definition may have missing components; for instance, the Brand-X system definition location may only state that the filename to look for is brandx. LSB maintains a dynamic stack of where it should search: each entry corresponds to a directory path (and device and host, where implementationally appropriate). The entries are pushed when LSB recursively looks at systems, for instance when it is establishing the compilation environment for a module; the details of this procedure are described later. In any case, the *first* directory to be looked at will be the directory with the same name as the system. Where appropriate, the device will be the "canonical" device for the implementation (e.g., PS for TOPS-20), and the host will be the current default host, however that is maintained. The last place to be looked at will be the directory (including the device and host components) where LSB is kept. Somewhere in the middle will be the directory (and device and host) where the source file being processed was found. The searching involves iterating down this search list; of the components missing from the system definition location (if there is one), missing directory, device, and host components are filled in from the search list, a missing filename defaults to the system name, and a missing file-type defaults to system. A file with name of lsbsystems is also looked for (the file type will be the canonical "lisp source file" type, e.g. lisp on Multics, ">" on ITS, LSP on TOPS-20). If during this procedure a system definition location is found for the system (and one was not already known), the searching starts over again using that system definition location.

Take, for example, the xlms system, which exists on MIT-ML. The first place LSB will look in the absence of a system definition location will be "DSK:XLMS;XLMS SYSTEM" — this is in fact where the system definition for xlms is kept.

There is one other way in which LSB may be told where to find a system definition: it may be specified with the name of the system.

```
(module print (xlms "XLMS;TEST SYSTEM"))
(define-system write
     (directory format)
     (built-on (loop "LISPM2;") lbind)
     (modules write))
```

A location specified in this way overrides any known system definition location; it is *not* used to aid the defaulting process for finding the definition of that system, but rather to respecify it, if (for example) one wants to test a perturbed system definition or test a new system by that name. For simply supplying a system definition location when one might not be known, the lsbsystems file should be used.

The lsbsystems file can be used for various purposes. It can be used to consolidate all of the system definitions of systems which reside on the same directory. It can also be used to define the system definition locations of systems which systems on that directory are built-on (or otherwise reference). This last is often necessary because otherwise LSB might have no handle on where to look for some system definition, if the referencing module is on a different directory. One solution to where to look is simply to have the system definition location pre-defined to LSB. Systems which may be of general use are welcome additions. The pre-defined systems are in the lsbsystems file on LSB's directory. On ITS, this is the file named "DSK:LSB;LSBSYS >", and on Multics ">udd>Mathlab>LSB>lsbsystems.lisp". These files have identical contents, which are conditionalized for the various implementions LSB runs in.

Due to certain implementation screws pertaining to remembering that a particular file has been loaded, in Lisp implementations with packages, LSB binds the package to the user package when it loads a system definition (or lsbsystems) file. This provides a *default* for the package that file gets loaded into; it may be overridden, if necessary, by the use of the package option in the file property list (since LSB system definition files are not themselves modules).

3.3 Relations between Systems

The most significant relation between systems is the built-on relation. To say that the write system is *built on* the loop system is to say that the code of the modules of the write system utilizes public facilities of the loop system. The built-on relation is not transitive; that is, if x is built on y and y is built on z, it is not assumed to be the case that x is built on z. Of course, the actual dependencies involved depend on what context is being considered: compilation, interpretation, or running compiled code. In general the built-on relation is viewed from the context of compilation.

If the built-on relation *does* need to be transitive, that information may be given to LSB by the use of the users-implicitly-need clause in place of the built-on clause. The use of this clause in the system definition for pretty-print-definition

```
    (define-system pretty-print-definition
        (built-on loop {pdp10 user-hunk})
        (users-implicitly-need write)
        (modules ppdef ppdesc))
```

says that the pretty-print-definition system is built-on the write system, and that systems which are built-on the pretty-print-definition system are also implicitly built-on the write system.

Another possible relation between systems is *sideways extension*. For system hair to be built-beside of system kernel means that hair utilizes the facilities of kernel *just the same as if it were part of the* kernel *system*, but implies that system kernel has no need for the facilities of system hair. Thus, kernel could be a "core system" intended for extension, and hair could be a system which does that extending. What this amounts to is that system hair utilizes both the public and system information derived from system kernel.

3.4 The LSB Processing Environment

The "environment" which LSB establishes for the processing of a module consists of three parts:

simple environmental options
: These include such things as the package, readtable, and input radix. All of these map into variable bindings; it is possible to bind this part of the LSB environment, and to calculate what it should be without actually modifying the current Lisp environment.

processing support
: For compilation, this is typically declarations and macro definitions; all of the stuff obtained by loading the various declaration diversion streams of the systems involved. For interpretation, this is whatever the module or its system says needs to be loaded for interpretation; usually nothing, as LSB's automatic loading is oriented solely towards compilation at this time.

diversion streams
: The diversion stream definitions for the module.

The LSB processing environment may be set up in one of two ways. The most common, and the only one available outside of the Lisp Machine, is driven entirely by the macro processing of the module special form. First, all loading necessary is performed; the *processing support* above. Then, all of the environment options are determined, and the variables are side-effected to their appropriate values. Finally all of the diversion streams appropriate for the module are defined. (These three steps are presented in much more detail in section 3.4.6, page 19.)

On the Lisp Machine (or in any Lisp implementation which interprets and heeds the file property list), there is one difference: the binding environment is determined at the time the file property list is parsed, and so is established around the entire loading or compiling operation. The module special form then knows not to bother. One uses the file property list with LSB by using the LSB option, as in

```
      ; -*- Mode:Lisp; LSB:module-name,system-name -*-
```

Note that the "arguments" are in the same order as they are to the module special form. Use of this option does *not* obviate the need for the module special form at the beginning of the file. Note also that if one uses the LSB system definition to specify the package a file should be compiled or loaded into, it is necessary to use this. It is an error (which will have unpredictable results) to specify any option in the file property list to which LSB provides a default, or which is itself specified in the system definition. All such options which LSB handles are enumerated below.

3.4.1 Environment Options

input-radix *radix*

The value of ibase used for the processing of the module will be *radix*. The default value is decimal.

readtable *readtable-name*

The readtable used will be the readtable associated with *readtable-name*, a symbol; LSB keyword equality is used. The default used is the name standard, which is associated with the readtable current when LSB was loaded/created. There are initially no other choices; the expectation is that large systems with special input syntax will supply special readtables of their own (see section 12.3, page 65).

inclusion-test-readtable *readtable-name*

This specifies the readtable to be used for reading in *inclusion tests* (chapter 4, page 24). *readtable-name* may be unspecified or nil, in which case whatever readtable is current at the point of the conditional inclusion will be used; this is the default.

package *package-name pathname*

This option, which is ignored in Lisp implementations without packages, specifies that the package to be used should be the package associated with *package-name*. If no such package is defined, then a file is searched for similarly to the way a system definition is searched for, using *pathname* (which is optional) as a default. This, in fact, uses the same search list which searching for the system definition does. The default filename looked for will be *package-name*, and the default file-type will be pkg. Also looked for will be the file lsbpackages, analogous to the lsbsystems file used for system definitions. Of course, searching is less likely to be useful in this case because the package option already has at its disposal the default directory, host, and device of the system whose definition the clause appears in. *It is unclear at this time that searching is an appropriate thing to do anyway. It is recommended at this time that the package pathname be given, if it will not default correctly the first time (directory of that given in the* directory *clause of the system definition, filename of the system name, etc.).*

This option is somewhat special for two reasons: it can have side-effects during its interpretation (the loading of the file and creation of the package), and it also does *not* provide a canonical default package if this clause is not given: the package used in the absence of this clause would be just that used if LSB were not present, i.e. one specified in the file property list, or barring that, the current package.

announce *kwd-1 kwd-2 ...*
: Use of this option causes the module special form to produce in its expansion a form which will "announce" the loading of that module. The message printed will contain the module name, system name, and the name of the source file. The only keyword implemented at this time is version: it causes the version of the source file to be put on the version property of the module name. This hack is to allow the announce option to supply a functionality similar to that of the Maclisp herald macro. This functionality fits in poorly both with LSB (since only the module name is used) and with non-Maclisp Lisp implementations, because of packages; outside of Maclisp, both the module name and the property name (version) will be in the keyword package.

do-macro-memoizing *how*
: Controls *macro memoization*. If *how* is not supplied, the default is assumed. Otherwise *how* may be t for the default, nil for none, or some other keyword describing a particular mechanism to use. For full details, see section 5.10, page 39.

type-check-visibility-classes *vis-class-1 ...*
: Routines defined with the named visibility classes will have argument type checking code automatically generated for them. This augments the do-argument-type-checking clause (below), and is discussed more fully in chapter 5.

number-check-visibility-classes *vis-class-1 ...*
: Routines and macros with the named visibility classes will have number-of-argument checking code automatically generated for them. This augments the do-argument-number-checking clause (below), and is discussed more fully in chapter 5.

The following options are all "flag options"; they take a single "argument" which is interpreted only as being t or nil. If the argument is not given, t is assumed.

do-argument-number-checking
do-argument-type-checking
: Control automatic generation of number-of-argument or argument type checking code, for all visibility classes. Obviously, code is only generated when it is actually needed. This defaults to nil, i.e. no checking other than what is implicitly supplied by the Lisp definition LSB produces.

needed-for-user-compilation
needed-for-compilation
: These have to do with specifying whether the module (or all of the modules in the system) are needed for the compilation of systems built on this one. They also control the redundant outputting of certain definitions (e.g., macros) to both a declaration diversion stream and to the compiled output file. This is currently in a bad state, and is being revised.

inhibit-documentation-production
: This turns off documentation production. When one has source code which runs in multiple Lisp implementations, it may be wasteful for each implementation to

redundantly produce the same documentation. The flag this sets also inhibits the recording of information used for producing automatic documentation, as discussed on page 56. This option works by setting a flag which is checked by the lsb:divert-documentation? function (page 51), which is a default diversion-stream predicate for documentation diversion streams. Thus, it only works if the documentation diversion streams actually use that function for their diversion predicate.

3.4.2 The Processing Support Options

Here are the system definition clauses which are interpreted for various reasons such as setting up the compilation environment. Unlike the environment option and diversion stream clauses, these may *only* appear at top-level in the system definition, not in a module specification (a component of the modules clause, below).

built-on *system-spec-1 system-spec-2 ...*
: The built-on clause specifies that this system is *built on* each of the other systems specified.

users-implicitly-need *system-spec-1 system-spec-2 ...*
: users-implicitly-need is like built-on, and additionally specifies that any systems built on this one are implicitly built on all of the systems specified here: it is transitive, whereas built-on is not.

built-beside *system-spec-1 system-spec-2 ...*
built-along-side-of ...
: Sideways system extension, as described in section 3.3, page 13.

files-needed-for-compilation *pathname-1 pathname-2 ...*
: The named files will be loaded in during compilation environment setup, if they have not been already. (This clause used to be named additional-files-needed; that name is accepted as a synonym for files-needed-for-compilation, but that synonymization will be flushed someday.)

users-implicitly-need-files *pathname-1 pathname-2 ...*
: The named files need to be loaded into the compilation environment. This need propagates to systems built on this one. *This should properly be named* users-implicitly-need-files-for-compilation *but that is quite a mouthful...*

modules *module-spec-1 module-spec-2 ...*
: Specifies the modules which comprise the system. A *module-spec* is either the name of a module, or a list whose car is the module name, and whose cdr is a list of clauses similar to those in a system definition. What may appear there is described in section 3.4.5, page 18.

default-user-options *option-clause-1 option-clause-2 ...*
: These specify default values for LSB environment options (previous section) which should be used for systems *built on* this one. The defaulting process is described in section 3.4.6, page 19.

3.4.3 The Diversion Stream Clauses

Diversion stream definition clauses allow one to define new (or redefine existing) diversion streams, for either all modules in a system, or even per-module. In this way, they default similarly to other LSB options discussed above. (Diversion stream definition defaults cannot be inherited from systems being built-on, however.) A list of the keywords used for defining diversion streams follows; they are discussed fully in chapter 8.

> diversion-stream *divstream-name clause-1* ...
> form-diversion-stream ...
> form-divstream ...
> declaration-diversion-stream
> declaration-divstream
> dcl-divstream
> textual-diversion-stream ...
> text-diversion-stream ...
> text-divstream ...
> documentation-diversion-stream
> doc-divstream

3.4.4 Pathname Specification Clauses

The following clauses may be used to specify default pathname components. These are not only accepted in the system definition, but also within a *module spec*, and in various other places where LSB looks for pathnames, such as in diversion stream definitions.

directory *subdir-1 subdir-2* ...
dir *subdir-1 subdir-2* ...

(directory *dir*) says that the default directory to use is *dir*. In most cases this is all that is needed, since typically the "canonical" device of the host is the correct choice, and there is no choice of host in most Lisp implementations. If multiple *subdir*s are specified, then they specify a directory hierarchy path. In the Multics implementation of LSB, one is also allowed to specify the entire path in a single atom, just as one would for the Multics Maclisp namelist format:

```
(directory >udd>Mathlab>LSB>format)
```

is equivalent to

```
(directory udd Mathlab LSB format)
```

It is unfortunately necessary for this entire path to be specified.

device *device-name*

For whatever it is worth, this may be specified. For example, (device arc). This is especially useful for referencing an ITS machine which is not on the Chaos network from a Lisp Machine:

```
(define-system Brand-X
    (host mc)
    (device ml)
    (directory brandx)
    ...)
```

host *host-name*

Specified the host to be used. This is obviously only useful in Lisp implementations which as a matter of course access multiple hosts, namely Lisp Machine Lisp. It will be ignored elsewhere.

pathname *host-dependent-pathname*

This clause is *not* actually a system definition clause; it is however used in many places in conjunction with the directory, device, and host clauses, so is documented here for completeness. It is used in places where a complete pathname is needed, and will have its missing components defaulted from the other three clauses. For example, the dumb-objects system is defined as follows:

```
(define-system dumb-objects
    (directory lsb1)
    (built-on loop)
    (modules (dumb-objects (pathname dumobj)))
    (needed-for-user-compilation))
```

The pathname clause shown says that the dumb-objects module has a filename of dumobj rather than the default (dumb-objects). The directory is defaulted to lsb1. If that particular module were on a different directory (say, test-dir), the modules clause could be re-written as either

```
(modules (dumb-objects (pathname |test-dir;dumobj|)))
```

or as

```
(modules (dumb-objects (dir test-dir) (pathname dumobj)))
```

The last is less dependent on the pathname conventions of the particular host, so may be a reasonable choice if the pathname components will be the same for different hosts or Lisp implementations.

3.4.5 The Module Specification

A *module specification* is a description of a module in the system. It may be either just the name of the module, as in the system definition clause

```
(modules ppcode ppdesc)
```

or a list whose car is the module name, and whose cdr is a list of clauses:

```
(modules (dumb-objects (pathname dumobj)))
```

These clauses describe various attributes about the module, such as where it resides in the file system and what environmental attributes its compilation environment should have. Most of these can, in fact, be defaulted: the name of the file (both source and compiled output) defaults to the name of the module, with the directory, device, and host being taken from those defaults for the system (as either specified with the corresponding clauses or defaulted), and the *extension* or *file-type* for the source and compiled output default to whatever is appropriate for them in the Lisp implementation. These can be selectively overridden by use of the pathname, directory, device, and host clauses, as described above. Most of the clauses described above for system definitions are also applicable to single modules; those that are not are those which describe system relations, listed in section 3.4.2. Thus,

```
(define-system mathematical-hacks
    (built-on loop)
    (modules arithmetic
             (bit-twiddling
                (input-radix 8))))
```

describes the mathematical-hacks system, which consists of the two modules arithmetic and bit-twiddling. Module arithmetic uses an input radix of 10 (decimal), which is LSB's default, but module bit-twiddling uses octal.

3.4.6 Environment Setup -- Details

The LSB environment setup is performed in one of two ways. In general, most if not all of the work will be performed when the module form which should be at the front of the source file is processed, either by the compiler or by being evaluated during loading. In both cases there are three fairly discrete actions:

(1) Loading whatever support code is needed for the type of processing being performed, and making the appropriate declarations (when compiling)

(2) Establishing the binding environment (specified with the various options)

(3) Defining the diversion streams which will be used by that module

The module form first performs step *1*. For compilation, this involves loading all of the pubdcl files of the systems this system is built-on (note the non-transitivity of the built-on relation, and the implications of the users-implicitly-need clause), and any other files they have specified with the users-implicitly-need-files clause. Then the pubdcl and sysdcl files for that system itself are loaded. This is a depth-first operation; take, for example, the systems

```
(define-system write
    (built-on loop lbind)
    (modules write))
(define-system pretty-print-definition
    (built-on loop {pdp10 user-hunk})
    (users-implicitly-need write)
    (modules ppdef ppdesc))
(define-system hacks
    (built-on pretty-print-definition)
    (modules crock kludge))
```

If we are compiling the crock module of the hacks system, the following files will be loaded (assuming they exist), in this order:

```
write pubdcl
ppdef pubdcl
ppdesc pubdcl
kludge pubdcl
kludge sysdcl
```

Additionally, if the special LSB pre-processing compiler interface is *not* being used (see section 10.1, page 59; if available, this is the default action), the pubdcl, sysdcl, and moddcl files of the module itself will be loaded. Note that by default there is no moddcl file produced, but it is provided for in case this sort of forward-reference capability is needed.

Next, the binding environment is set up. If the LSB option was given in the file property list of the source file, and that option was actually used by whatever is doing the processing (e.g., load, the compiler), then this has already been done. It is expected that environments which will be using this will have the support necessary already loaded (this mainly concerns values of defaulted options, or new option definitions); in other environments, this support should have been loaded in by the previous step. This also is why the lsb-load (page 22) function exists—for the loading of an interpreted module, the variables which comprise the processing environment need to be bound so that the module form does not side-effect the global environment. Anyway, LSB sets up the binding environment by calculating all of the variables which will be affected, and then filling them in with their specified values: first any per-module options for the module being processed are looked at, then any for the system as a whole. Then, any options specified in any default-user-options clauses of systems this one is built-on. The order in which options are defaulted in this last manner is not extremely well defined at this time, but it can be guaranteed that, when built-on relations only go one-level deep (as is normal unless the users-implicitly-need clause has been used in one of these systems), it will be the order in which those systems appear in the system definition. Thus, looking at the write system definition given above, if the write module were being compiled, first any default user options of the loop system would be looked at, then those for lbind. Any options which are not either specified or defaulted from other systems will have be set to their canonical default values.

Lastly, the diversion stream environment is established. This may implicitly make use of support loaded during step *1*, and options (variables) set up by step *2*. Essentially, what happens is

(1) All extant diversion streams are "killed", if necessary, which it is not if the diversion stream environment is "bound" (as done by the lsb-load function, or by the use of LSB in the file property list). If this is done, a warning message may be printed, as this could in theory involve closing and deleting files being written. But normally nothing interesting is happening in the interpreter.

(2) All diversion stream definitions specified for the module only are processed

(3) All diversion stream definitions given at top-level in the system definition, which have not already been defined, are processed

(4) All other diversion streams which LSB uses and which have not been defined, are, using default definitions determined from the type of processing being performed.

Most simple systems rely solely on step *4*.

Finally, the module form macro-expands into various potentially interesting things, in an implicit progn. Some of this might involve bootstrap code (to load things up); this has not been worked out yet. This is also the place where any actions dictated by the announce option are performed. And:

***source-file-information** *Variable*

The module special form expands producing a setq of this variable to a disembodied property list containing much information about the module. (At this time, this variable may either not get set or not contain much information when the module is loaded interpretively.) The car of this list is the actual real pathname (truename) of the source file (if it could be determined). The plist part may contain the following properties:

:module

The module name. In Lisps with packages, this will be a symbol interned in the user package. It will be all in one case; lower on Multics, upper elsewhere.

:system

Similar

:lisp-version

The version number of the Lisp the processing was performed in. This will be a fixnum.

:system-version-info

This is a hairier version of :lisp-version. On the Lisp Machine, this will be a string which is the result of calling si:system-version-info. Elsewhere, LSB will make do with what it can get, but in any case, if this property is present, it will be a string (symbol, in PDP-10 Maclisp) containing some descriptive information. See si:system-version-info and related topics in the Lisp Machine manual for more information.

:site Some name for the machine the processing was performed on. Presently, LSB only knows how to do this for PDP-10 Maclisp. On Lisp Machines, it is undecided whether this will be only the Lisp Machine name or possibly some composition of the specific Lisp Machine and *its* site, as in (MIT CADR-6).

:culprit
The name of the user. On the Lisp Machine, this is simply the value of user-id. On the ITS operating system, this is the uname rather than the xuname for reasons having to do with INIT files: in that case, the xuname, which supposedly represents the real user name as opposed to some instantiation of it, will be under the :claims-to-be property.

:claims-to-be
On ITS only, the xuname of the user, if it differs from the uname.

:compiler-version
Some type of description of the compiler version. This is a brief descriptive history on Multics. On the PDP-10 this will simply be the version number of the compiler. This will not be present on the Lisp Machine because the compiler is an integral part of the basic Lisp Machine system, implicitly included in the :lisp-version and :system-version-info properties, and no explicit information is available.

:date Date and time of the processing, in the form
((*year month day*) *hour minute second*)
The year is excess-1900.

LSB also handles a few more mundane things automatically. When setting up a compilation environment in Maclisp, a genprefix is automatically performed on the concatentation of the system name, a ".", the module name, and a "-"; e.g., "mysys.mymod-". In the PDP10 Maclisp implementation, LSB does some hackery to avoid having the garbage collector thrash due to array relocation during compilation environment setup (this being a result of the large number of arrays used by fasloading so many files). If one has a system which causes very many files to be loaded and this seems to be happening (indicating that LSB's default handling is too modest), a (getsp *n*) clause may be included in the system definition. LSB by default does this with *n* of 20000. This mechanism may be improved at some point. Obviously *n* should be significantly smaller than the amount of free memory remaining.

3.5 The LSB Loader

Because of the possibility of an interpreted LSB module side-effecting the environment, which can occur in Lisp implementations which do not utilize the file property list or if the module does not use the LSB option in the file property list, it may be necessary to use a loading function which properly scopes all of the LSB environment variables.

lsb-load *pathname kwd-1 kwd-2 ... kwd-n*

This is the routine LSB uses to load in files. It causes the LSB environment to be scoped around the loading; all of the flag variables are bound, as is the diversion stream environment. Note that all of these things maintain their current values; if the file being loaded is an interpreted LSB module, it is the module form in that file which will reset them. The various keywords are as follows. In general the few keywords provided negate default actions.

noerror
: Do not cause an error if the file is not found. A message stating this may be printed depending on other keywords given; in any case, lsb-load will return nil.

conditional
: If this file has been loaded already, do not load it again. The atom :previously-loaded will be returned. Note that in the Lisp Machine implementation this atom is previously-loaded in the user (keyword) package, but in Maclisp it is an atom with a colon as its first character.

nodefault
: Do not set the pathname defaults used by load (and lsb-load). Normally, they are set to the pathname that gets loaded. Programs that call lsb-load may not wish to change the default pathname on the user without his cognizance, in which case they should use this option.

verbose
: Normally, the verbosity of lsb-load defaults to the value of *lsb-verbose?. This explicitly says that lsb-load should be verbose.

silent
: The opposite of verbose.

uninteresting
: This is primarily useful for searching for a file to load, as is done for system and package definitions. lsb-load will not modify the file defaults, and will not error out if the file is not found. Additionally, it will not even print an error message if the file is not found; however, it may still print a loading message if the file *is* found, dependent on the presence of the verbose or silent keywords, or the value of *lsb-verbose?.

The default behaviour of lsb-load is for it to follow the standard system load function fairly closely, simply augmenting it by "scoping" the LSB processing environment appropriately.

In order for lsb-load to determine whether some file has already been loaded, it maintains a database about loaded files. In the Lisp Machine implementation, such a facility already exists, so lsb-load uses it; elsewhere there is none, so lsb-load is incapable of determining that a file loaded with load or fasload has in fact been loaded. In the common cases where lsb-load is used, such as loading the various declaration files into a compiler, this normally does not matter. The Multics implementation of lsb-load is not able to determine that a more recent file of the same name as one previously loaded has changed.

In the Maclisp implementation, **lsb-load** does **(sstatus uuolinks)** after loading, to avoid redefinition problems. It is not yet smart enough to figure out if it needs to bind **fasload**, however.

4. Conditional Inclusions

Conditional inclusions allow one to conditionally include portions of the source code being processed. This may be done by the use of reader syntax which conditionalizes the input seen by read, or by the use of conditionalization macros to conditionalize code being compiled or executed. Essentially, all forms of conditional inclusion require the specification of an *inclusion test*. An *inclusion test* may be one of the following:

normal inclusion test

A regular *inclusion test* is a list whose car is an atomic symbol, referred to as the *inclusion tester*. The inclusion tester is examined using LSB keyword equality to determine how the form should be interpreted to determine success or failure of the inclusion test.

-- or --*

An inclusion test of either of these tokens fails.

atomic symbol

Any other atomic symbol used as an inclusion test is treated as being shorthand for (only-for *atomic-symbol*).

only-for *Inclusion Tester*

The inclusion test (only-for *x*) interprets *x* as an *implementation feature test*, and succeeds iff that test succeeds. An *implementation feature test* is either the name of an implementation feature, or a logical composition of implementation feature tests, made with and, or, and not. An implementation feature name is tested for by looking in (status features) (but note also section 4.2, page 25); LSB token equality is used. Multiple "arguments" to only-for are treated as an implicit or; thus, (only-for Maclisp Lispm) is equivalent to (only-for (or Maclisp Lispm)), succeeding if either Maclisp or Lispm are "features".

except-for *Inclusion Tester*

(except-for *i1 i2* ...) is equivalent to (only-for (and (not *i1*) (not *i2*) ...)). Thus, it is the complement of (only-for *i1 i2* ...).

A mechanism for extending the composition operators for implementation feature tests is under development.

4.1 Read Time Conditionalization

Read time conditionalization is effected by use of the { reader-macro. The general format is

{*inclusion-test anything-else-with-matching-braces*}

Inclusion-test is read in (by calling read) and interpreted as described above. If the test fails, then the reader-macro gobbles down all of the text up to the matching }; to the Lisp reader, the result is like a space (i.e., an atom delimiter). If the test succeeds, then the reader-macro immediately returns, leaving *anything-else-with-matching-braces* intact in the input stream; the } will behave like a space (an atom delimiter, but otherwise ignored) by the Lisp reader.

When text is being skipped over due to a failing inclusion test, matching open- and close-braces are kept track of. There is no way to "quote" one of them in such a context; what one should do is to match an unmatched brace by placing it in a comment appropriate for the syntax of whatever is within the braces. For example,

```
{(only-for Lispm)
    ; Matching "{"
    (princ "}" error-output)
    }
```

Note that since excluded text is not read by the Lisp reader, it need not be syntactically valid Lisp; thus, this mechanism is good for conditionalizing syntactic constructions which are not valid in all Lisp implementations, as in

```
(member x '(0 0.0 {Lispm 0.0s0}))
```

where 0.0s0 is "small-flonum" zero, a syntax not supported in Maclisp. And of course, the excluded text may be used as comments, by using either `-*-` or `--` as inclusion tests:

```
{-*- Mode:Lisp; Package:PP -*-         1-Apr-84
     This file defines a universal pretty-printer
with capabilities far exceeding any thus-far
etc}
```

When the inclusion test is read, **readtable** is bound to the readtable specified with the **inclusion-test-readtable** system definition option (page 14). If **nil** was specified as the readtable (which is the default), then the current readtable is used. This option exists to allow normal Lisp syntax to be used (if desired) from within some abnormal syntax.

4.2 Multiple Contexts

Thus far, the test for an implementation feature has been said to be based on the membership of the feature name in **(status features)**. This is not strictly true, as very often what is intended is examination of the *target* environment as opposed to the *processing* (e.g. compilation) environment. The inclusion tester **only-for** and **except-for** are intended to refer to the *target* environment, which is normally assumed to be the same as the processing environment. To guarantee that the test is based on the processing environment, one can use:

only-on *Inclusion Tester*
except-on *Inclusion Tester*

These are the same as **only-for** and **except-for**, except that they *always* use **(status features)**, guaranteeing conditionalization based on the environment in which the test is made. Thus,

```
{(only-for tops-20) stuff}
```

includes *stuff* if the code is intended to run in a TOPS-20 Lisp, whereas

```
{(only-on tops-20) stuff}
```

includes *stuff* only when it is being read into a TOPS-20 Lisp.

There is currently no mechanism for specifying multiple "feature environments". See section 12.5.1, page 67 for a description of the current facilities which may be used to implement the above. It is expected to change drastically when a better facility is defined, and is only provided as an interim solution.

5. Defining Operations

For the purposes of discussion, a defined object which is either a routine or a macro will be called an *operation*. A *routine* is one where the body of code in the definition form is to be executed when the call is executed; a macro is one where the body of code is to be executed to produce a form to replace the call. All LSB operation definition forms have the same general format:

```
(define-visclass-definitiontype prototype-call
     (declarations dcl-clause-1 dcl-clause-2 ...)
     form-1 form-2 ...)
```

where *visclass* is the visibility class, i.e. public, system, or private, and *definitiontype* is the type of definition, e.g. routine, macro, optimizer. There may be any number of declarations (abbreviated dcls) forms, but they must precede any of the body forms.

5.1 The Prototype Call

The *prototype call* shows what a call to a routine or macro looks like, defines the mapping from the arguments into the formal parameters, and specifies the data types of both the arguments and the variables. For example,

```
(frobnicate foo (optional bar 'ugh) (any-number-of bletch))
```

shows a prototype call for frobnicate, which takes one required argument, one optional argument, and any number of other arguments. It shows how the call is to be mapped into the formal parameters of frobnicate (foo, bar, and bletch): the first argument is required, and maps into the foo variable. The second is optional; if it is not specified, then bar will be bound to the symbol ugh. All remaining arguments map into the bletch variable, which will be bound to a list of them. This process of mapping from the arguments into the formal parameters is known as *call mapping*, and the keywords such as optional are known as *call mapping keywords*. Other keywords which may be used like any-number-of are one-or-more-of and two-or-more-of which require there to be at least one or two arguments corresponding to that formal parameter. rest and body are synonymous with any-number-of; the latter is intended for macros which take any number of forms to be used in some kind of implicit progn. If there is a minimum number of arguments required for this type of parameter (a *rest parameter*), it is meaningless (and an error) to have any optional arguments. There may only be one rest parameter, and it must come last.

One can also specify the data types of the arguments and variables in the prototype call, as in the example shown previously,

```
(define-public-routine (sqrt$ (flonum n)) ...)
```

For optional and rest parameters, note that the type of the variable and the type of the corresponding argument(s) need not be the same. One can thus have:

```
(my-routine (notype (optional (fixnum count) nil))
            (vector (any-number-of (flonum frobs))))
```

which says that my-routine takes an optional argument, count, which must be a fixnum. The variable count, however, is of type notype, and will be bound to nil if no argument is given. All other arguments are required to be flonums, and are gathered into a vector which frob is bound to. The nil shown is a form to be evaluated to supply a default value for the variable if no corresponding argument is present; this evaluation will be done in an environment where only the variables to the left in the prototype call have been bound to their arguments or default values. In this instance, that nil is superfluous, because the default default value will be chosen on the basis of the *variable* type, notype.

The optional syntax allows for the specification of a variable to be used as a flag for whether or not the corresponding argument was supplied:

```
    (frob-name frob (optional new-name nil new-name-p))
```

which could be the prototype call for the routine frob-name. The variable new-name-p will be bound to t if frob-name receives two arguments, nil if it receives only one. This variable is automatically declared to be of type truthvalue (chapter 7).

The any-number-of syntax allows for the specification of a variable to receive the count of the arguments which were mapped into the formal parameter:

```
    (define-public-routine (average (one-or-more-of numbers count))
        (quotient (apply (function plus) numbers) count))
```

This variable is automatically declared to be fixnum; a type must not be specified for it.

Sometimes an operation takes an argument which is not actually used by the code of the operation. This situation typically arises when the code is not totally complete, but that argument position needs to be "allocated" for future compatibility. Since the author of the code knows of the situation, it is undesirable to have the compiler warn him about it. One may use the unused keyword to tell LSB that the variable so designated does not get referenced by the code, as in this prototype call for hairy-routine:

```
    (hairy-routine
       file-to-be-processed
       file-to-send-output-to
       (optional die-on-errors?)
       (any-number-of (unused keyworded-additional-options)))
```

This option is applicable in all LSB operation definitions. It should *not* be used with the *supplied-p* variable for an optional argument, nor for the *count* variable for a rest argument. LSB will do whatever the Lisp implementation requires to ensure that the compiler will not complain about a lack of reference to the variable. It is an error for the variable to be referenced in the code of the operation when the unused keyword is used.

5.2 Variable Bindings

LSB operation definition forms provide syntax for binding variables within the body of the operation. When a form appearing *at top level only* in the definition body starts with one of the keywords auxiliary-bindings or bindq, the cdr of that form specifies a binding environment to be used around the remainder of the definition form. Some may find this syntax distasteful and prefer to use a form which textually shows the binding scoping (such as lbind and lbind*); others may find it convenient to use, as it allows the bindings to be placed more naturally while not producing deeply nested code. lbind and lbind* are macros which provide similar functionality, but without restrictions on their positioning; they are documented in chapter 13, page 73.

5.2.1 Auxiliary-bindings

An auxiliary-bindings (abbreviated as either aux-bindings or auxs) form binds a set of variables sequentially, allowing the value computed for one to depend on a previous variable. This form looks like

```
    (auxs aux-bind-spec-1 aux-bind-spec-2 ...)
```

where each *aux-bind-spec* may take one of the following forms:

variable
: The variable is bound to nil.

(*variable value*)
: The variable is bound to the value of *value*.

((*data-type variable*) *value*)
: The variable is bound to the value of *value*, and declared to be of type *data-type*.

(*variable*)
((*data-type variable*))
: If a value is not specified, it will default to the initial value for the stated data type.

Thus, the body of the code in

```
      (define-public-routine (frobnicate x)
          (auxs (a (f x)) ((fixnum b)) ((flonum c) (g a)))
          do-this
          (auxs (p (hack a b c)))
          do-that)
```

produces the following binding contour:

```
      ((lambda (x)
          ((lambda (a)
              ((lambda (b)
                  ((lambda (c)
                      do-this
                      ((lambda (p) do-that)
                       (hack a b c)))
                   (g a)))
               0))
           (f x)))
       argument)
```

along with local declarations appropriate for the Lisp implementation. There is actually a bit of optimization performed to try to bind as many variables in parallel as possible; this is done for the sake of Lisp implementations where that may be more efficient, especially if they are special. In the above example, b would be bound in the same lambda as a, since it is being bound to a constant.

5.2.2 Bindq

bindq is an alternative to auxiliary-bindings. In addition, it *always* binds variables in parallel. A bindq form looks like

```
      (bindq varspec-1 value-1 varspec-2 value-2 ...)
```

and can be used in the same places as an auxiliary-bindings form. Each *varspec* may be either the name of a variable, or a list of the data-type for the variable and the name of the variable. The code for

```
      (define-public-routine (frobnicate x)
          (bindq a (f x) (fixnum b) 0 c (g x))
          do-this
          (bindq p (hack a b c))
          do-that)
```

produces the binding contour

```
      ((lambda (x)
          ((lambda (a b c)
              do-this
              ((lambda (p) do-that)
               (hack a b c)))
           (f x) 0 (g x)))
       argument)
```

5.3 Defining Routines

define-public-routine (≡ **defpubr**) *Special Form*
define-system-routine (≡ **defsysr**) *Special Form*
define-private-routine (≡ **defprivr**) *Special Form*

In LSB, one does not define ordinary *functions*; one defines *routines*. The difference is that a Lisp function definition implies a specific implementation, whereas an LSB routine definition simply says how one desires to *call* the routine. The actual implementation of the call is left to LSB, and may differ across Lisp implementations. There are declaration options to give LSB information which may be useful in choosing a particular implementation, and to demand a particular one.

Consider the following routine definition:

```
     (define-public-routine (print-decimal-number
                                   (number n)
                                   (optional stream))
          (bindq base 10. *nopoint t)
          (princ number stream))
```

The functional specification of print-decimal-number ultimately reduces to the primitive lambda expression (lambda (n stream) ...). Somewhere between the processing of a call such as (print-decimal-number *num*) and the evaluation of the forms in the definition body there has to be a mapping made between the call and the application of that primitive lambda expression. One possibility is to turn the call (print-decimal-number *num*) into a new call (print-decimal-number-aux *num* nil), with print-decimal-number-aux defined to take the same arguments as print-decimal-number, except that they are all required. LSB does in fact make this kind of transformation in Lisp implementations which have a significantly less efficient calling sequence for functions of a variable number of arguments.

The prototype call for a routine may also specify that the argument(s) mapping into a variable be *implicitly quoted*, by use of the quoted keyword wrapper around the variable, as in the prototype call

```
      (foo (quoted x) (optional (quoted y))
           (any-number-of (quoted frobs)))
```

which says that foo takes one or more arguments, and none of them are evaluated. This *implicit quoting* is done as part of the *call processing*. What actually happens is that foo is defined as a macro which quotes the appropriate arguments, producing a call to foo-aux, just as done by the optimization for routines taking a variable number of arguments (in fact, both of these transformations may occur at the same time). The implication of this is that it is not possible in general to apply or funcall a routine which takes quoted arguments. This restriction is based on the "lowest common denominator" of the various Lisp implementations LSB is aimed at; this restriction may be lifted at some point, at least for some of these implementations.

The following declaration options may be of use in defining routines:

value-type *data-type-name*
data-type *data-type-name*
: This specifies that the value returned by the routine will *always* be of type *data-type-name*. If the type and the Lisp implementation so warrant, this may produce appropriate declarations which may affect the calling sequence of the routine.

do-argument-type-checking *flag*
: This turns on argument type checking for this routine if *flag* is non-nil or unspecified, otherwise turns it off. Thus, argument type checking can not only be specified per-module or per-system (with the do-argument-type-checking system definition keyword), but also per-definition.

do-argument-number-checking *flag*
: Like do-argument-type-checking, but enables or disables number-of-argument checking for this routine, and is also applicable to macros.

type-check-arguments *var-1 var-2 ...*
: If there are no *vars* supplied, this is equivalent to (do-argument-type-checking t); otherwise, it enables it for the arguments corresponding to only those variables named.

returnable
: the declaration (returnable) causes the entire definition body to be formed inside a prog, with a return wrapped around the last form. This is convenient for routines which ordinarily would not need this except for a small number of extraordinary cases which must be handled specially and return early. (returnable *name*) causes the prog to be "named" *name*, so that it may be explicitly returned from (even through other prog or do forms) by using the return-from special form, as in

```
      (define-public-routine (hack 1)
           (dcls (returnable george))
           ...
           (do ((11 1 (cdr 11))) ((null 11))
              (cond ((not (numberp (car 11)))
                     (return-from george 'error)))
              ...)
           ...)
```

Since naming the containing prog requires the Lisp implementation to support *named progs*, this extension does not work in Maclisp.

Where *multiple values* are supported, LSB uses multiple-value-return instead of return, and multiple-value-return-from instead of return-from so that they will be passed back properly.

default-definition-from *routine-name*

If this declaration is given, there should be no "body" for the routine. The definition, instead of being created, will be gotten from the definition of *routine-name*, which *must* be defined. It is imperative that the prototype call and declarations for this routine reflect those with which *routine-name* was defined, as they will be used to produce declarations for the new name. This option should not be used lightly; it is designed primarily to save away the definition of an existing routine so that that routine may be redefined.

redefinition

This exists to tell LSB that the routine is a redefinition of some existing routine. LSB will try to keep the Lisp and compiler from complaining about the redefinition. Other than that, you redefine things at your own risk.

primarily-applicable-routine
applicable-routine-only

This declaration tells LSB that the routine is used primarily to be funcalled or applyed, so there is no point in doing sophisticated call processing on it. Appropriate declarations will still be produced, however. It is illegal to have quoted arguments for a routine with this declaration. applicable-routine-only is the old name for this declaration; it should not be used in new programs, as it will be recycled to additionally inhibit diversion of declarations.

slow-and-hairy

This tells LSB that the routine is complex enough that time should be discounted when a space/time tradeoff is made for deciding what (if any) calling sequence optimizations should be performed.

perform-calling-sequence-optimizations

Tells LSB to perform calling sequence optimizations. Useful only if LSB's default action is inappropriate.

inhibit-calling-sequence-optimizations

inhibit-calling-sequence-optimizations is exactly the opposite of perform-calling-sequence-optimizations.

implement-as *how bvl*

This requests LSB to use a specific implementation for the routine being defined, *and* supplies it. *how* should be one of the tokens expr, lexpr, or fexpr, and is used to make the appropriate functional declaration for the routine. Information as to number of arguments and the argument types still comes from the prototype call, however no automatic argument type checking or number-of-argument checking will be performed. The bound variable list for the function definition will be *bvl*, and any items after *bvl* in this declaration form will be prepended to the body of the definition.

assembly-language-definition

If this is present, then the body of the definition is assumed to consist of assembly language code (in Lisp format appropriate to the implementation) rather than Lisp code. LSB will provide the appropriate header and args information to be output, and will automatically default the declarations which would otherwise have to be supplied by such clauses as implement-as and inhibit-calling-sequence-operations. Use of assembly code in this fashion keeps it in textual proximity to the "definition form", and also allows LSB to continue to automatically produce declarations. Additionally, any pre- and post-definition forms implicitly generated by such clauses as redefinition will be properly placed around the actual definition. For an example, see page 42. This is only supported in PDP-10 Maclisp at present.

5.4 Open Coding

define-public-open-codable-routine (≡ **defpubopen**) *Special Form*
define-system-open-codable-routine (≡ **defsysopen**) *Special Form*
define-private-open-codable-routine (≡ **defprivopen**) *Special Form*

These define routines just like define-*visclass*-routine, and in addition tell LSB that compiled calls to such a routine should be open-coded.

open coding means that when a call to a routine is compiled, the body of the routine will be compiled in place of a call to it. That is, if we have

```
        (define-public-open-codable-routine (cube (number n))
            (expt n 3))
```
then the expression
```
        (plus (f x) (cube (g x)))
```
will be compiled as if it were
```
        (plus (f x) ((lambda (n) (expt n 3)) (g x)))
```
The routine cube will be defined just as if it were an ordinary routine, except that no calling sequence optimizations will be performed, since they are obviated by the open-coding.

Defining something as an open-codable routine has some advantages over defining it as a macro. First and foremost is the clarity of the definition. Also, the routine definition typically uses less space in the runtime environment than a corresponding macro definition would. An open-codable-routine, because it is a routine, may be funcalled or applyed if there are no quoted arguments specified in the prototype call. Lastly, the use of an open-codable-routine makes the visible semantics of a call to that routine obvious: if we have

```
        (define-public-open-codable-routine (foo a b)
            (bar b a))
```
then a caller of foo does not have to worry that foo will evaluate its arguments repeatedly or out of order.

Obviously, there will be cases where it is *not* necessary to bind the formal parameters of an open-codable-routine around the body of the code. This is the case in the cube example. Rather than producing (for compilation) an expansion like

```
        ((lambda (n) (expt n 3)) (f x))
```
for (cube (f x)), it is obvious that
```
        (expt (f x) 3)
```
would suffice, for any function (or special-form or macro) f. LSB has two mechanisms to handle this.

The first, *used by default*, is *lambda optimization*. Essentially, the code expanded to be used in place of the original call is re-examined, and wherever possible, lambda-bindings are eliminated. This succeeds in making such transformations as the one shown above for cube. LSB knows not to optimize out the bindings of special variables.

The second is to explicitly tell LSB that the arguments to the routine may be safely textually substituted into the body of the routine. This is done by means of the use-sublis-for-open-coding declaration. For example, cube could have been defined as

```
    (define-public-open-codable-routine (cube (number n))
        (dcls (use-sublis-for-open-coding))
        (expt n 3))
```

Obviously this is unsafe and *not recommended* if the routine references the variables out of order or anything but *exactly once*, as that destroys the implied semantics of function calling. And, since sublis is actually used, there should be no name conflicts. *The use of* sublis *does not mean that the body of the routine may be a pattern to be substituted into; it must still be viable as an ordinary routine.* LSB reserves the right to not actually use sublis if it can preserve the substitution semantics and produce better code in some other manner.

Open-codable-routines may be defined with optional, any-number-of, and quoted arguments. The checking for optional arguments will be performed at compile time, and the argument or default value form inserted into the code as appropriate. any-number-of arguments work, but they can only be implemented as heap-consed lists, so are not recommended; often in this case a macro is called for, or perhaps an ordinary routine.

Occasionally a situation arises where one desires to have a routine open-coded only in certain Lisp implementations. For this, one may give an ordinary routine definition the open-code declaration, which may be placed inside of a conditional inclusion:

```
    (define-public-routine (cube (number n))
        {Lispm (dcls (open-code))}
        (expt n 3))
```

which causes cube to be open-coded only if is being compiled for a Lisp Machine.

A note is in order with respect to the setf special form: since an open-codable-routine is not a macro, setf cannot determine what a call to it will expand to, so if one is being used to create a synonym for some sort of structure reference and is meant to be invertable with setf, either setf must be informed how to make its transformation, or the routine should be defined as a macro. The same is true for the locf special form in Lisp Machine Lisp.

5.5 Defining Macros

define-public-macro (≡ defpubmac) *Special Form*
define-system-macro (≡ defsysmac) *Special Form*
define-private-macro (≡ defprivmac) *Special Form*

Macro definition format in LSB is similar to routine definition format. The same *call mapping keywords* (e.g. optional, any-number-of) may be used, but typed and implicitly quoted arguments may not be. If any type of *rest parameter* is used, no implementation type should be specified for it, as the variable will be bound to a sublist of the original call.

Thus,

```
    (define-public-macro (frob-name frob)
        `(caddr ,frob))
```

effectively defines frob-name to be a synonym of caddr. This particular example is defined as a macro rather than as an open-codable-routine so that the setf special form can invert it:

```
(setf (frob-name x) 'new-name)
        ==> (rplaca (cddr x) 'new-name)
```

The *flag variable* which may be specified to flag whether an optional argument is supplied may be useful with macros as well as with routines:

```
(define-public-macro (frob-name
                       frob (optional val nil val?))
     (cond ((null val?) '(caddr ,frob))
           (t '(rplaca (cddr ,frob) ,val))))
```

And, of course, macros are the best way to write code which changes environments or control flow:

```
(define-public-macro (if predicate consequent
                        (any-number-of else-forms))
     (cond ((null else-forms)
               '(and ,predicate ,consequent))
           (t '(cond (,predicate ,consequent)
                     (t . ,else-forms)))))
(define-public-macro (using-decimal-radix (body body))
     '((lambda (base *nopoint) ,@body) 10. t))
```

5.5.1 Unneeded Macros

Quite often one needs macros around in the runtime environment for running interpreted code (which may or may not be in the same module a macro is defined in) but it is undesirable for them to be present in an environment where all of the code is compiled. This is often necessary in small address-space Lisps, and is often not unreasonable in even large address-space Lisps which do not have packages, such as Multics Maclisp, to avoid cluttering up the runtime environment with definitions which are only needed for running code interpreted, or for debugging.

define-public-xmacro (≡ **defpubxmac**) *Special Form*
define-system-xmacro (≡ **defsysxmac**) *Special Form*
define-private-xmacro (≡ **defprivxmac**) *Special Form*

The *only* difference between this and normal macro definition (define-*visclass*-macro) is that the (compiled) macro definition will be sent to the macros diversion stream rather than output into the compilation output file. The utility of this is that the macros definitions (which are normally not needed if all callers are compiled) will not be present in the compilation output file, but will be in the file written as the macros diversion stream, so may be loaded when needed. This type of macro definition is common in systems which define many (or large) macros which are not normally used, such as pretty-printers. Note that the semantics of this depends on how the macros diversion stream behaves—see page 54.

5.6 Compiler Macros

Sometimes one wishes to define something as a routine (for efficiency or argument checking when it is called from interpreted code, or for its ability to be funcalled), but desires special handling of a call to it when it is compiled, that only a macro can provide. Another scenario is where one is defining a *special form* (section 5.7, page 35), and it is *imperative* to have special handling of the form when it is being compiled. LSB allows one to define macros which *only* are used for the expansion of code being compiled.

define-public-compile-time-macro *Special Form*
define-system-compile-time-macro *Special Form*
define-private-compile-time-macro *Special Form*

These have syntax identical to that of define-*visclass*-macro, but the definition is *only* used for expanding code being compiled.

The definition of cube as an open-codable-routine

```
    (define-public-open-codable-routine (cube (number n))
         (dcls (use-sublis-for-open-coding))
         (expt n 3))
```

is performed by LSB just as if the user had done

```
    (define-public-routine (cube (number n))
         (dcls (inhibit-calling-sequence-optimizations))
         (expt n 3))
    (define-public-compile-time-macro (cube n)
         (sublis (list (cons 'n n)) '(expt n 3)))
```

More examples are given with the descriptions of definition constructs which may also require the use of compile-time-macros.

Note that use of this facility may compete with automatically generated code for calling sequence optimization or open coding; if this is suspected, one should explicitly disable calling sequence optimization as in the above example, by use of the inhibit-calling-sequence-optimization declaration.

5.7 Special Forms

special forms are constructs which do strange non-functional things with their arguments, like eval them. cond is a special form, as are do, prog, and go. In most cases it is best to define special constructs like this as macros, so that only one definition is needed, and so that code analyzers (and the like) need not understand the form specially. Thus, if can be defined as a macro as it is on page 34. There may be situations where there are overriding reasons for using a special form instead, even considering the extra effort of defining a compile-time-macro so that such a form can be properly compiled. One possible scenario (in fact, the one which led to the implementation of special form definition in LSB) is a system (especially in a limited address space Lisp) where there are many special constructs defined, and they get much use. The overhead of expanding and remembering all of those macros, combined with the occasional tendency of macros definitions to take up more space than the corresponding routine definitions, may be just too large a price to pay.

define-public-special-form (≡ **defpubspec**) *Special Form*

Only public special form definition is provided for. Automatic type checking is not handled here. Use of the quoted keyword to specify implicit quoting of arguments is an error; all arguments are implicitly quoted. (This is a special form, after all.)

As an example, we can define if as a special form:

```
(define-public-special-form (if predicate consequent
                                (any-number-of else-forms))
     (cond ((eval predicate) (eval consequent))
           ((null else-forms) nil)
           (t (do ((l else-forms (cdr l)))
                  ((null (cdr l)) (eval (car l)))
                (eval (car l))))))
```

To be properly compiled, if would also need to be defined as a compile-time-macro, in the same way it is defined as a macro on page 34: that is, simply defined with define-public-compile-time-macro instead of define-public-macro.

5.8 Optimization and Transformation

define-public-optimizer (≡ **defpubopt**) *Special Form*
define-system-optimizer (≡ **defsysopt**) *Special Form*
define-private-optimizer (≡ **defprivopt**) *Special Form*

An *optimizer* is similar to a compile-time-macro (section 5.6, page 35), and *additionally* has the option of deciding *not* to expand the call. Although the specific mechanism used to signify that the optimizer "did nothing" differs in differing Lisp implementations, LSB optimizers should indicate this by returning nil; if nil is desired as the actual value, (quote nil) should be returned, as it is entirely equivalent in any context the optimization will be performed in.

When the compiler compiles a call, it will try the optimizers for that call before it tries a macro definition (if any). Additionally, LSB arranges for compile-time-macros (and thus LSB generated calling sequence optimization code) to come last, if it could conflict with any user-defined optimizers (since compile-time-macros and calling sequence optimization may use the optimizer mechanism to do their work). User optimizers will get tried in the order they are defined in.

When an LSB routine or macro is defined, code is generated (as part of the declaration information) to flush any existing optimizers and compile-time-macros. Thus, optimizer definitions should come after the routine or macro definition they are for, and they should appear in the same file. It does not work to mix visibility classes either; that is, if the routine is public, it does not necessarily work for an optimizer to be private or system.

Since a given name can have multiple optimizers associated with it, it is helpful for redefinition purposes to associate some kind of identifier with each particular one. This should be specified in the identifier (abbreviated id) declaration clause:

```
(define-public-optimizer (foo arg1 arg2)
     (dcls (id number))
     (and (numberp arg1) (numberp arg2)
          '(super-foo ,(plus arg1 arg2))))
```

This identifier should be a symbol, and need only be unique with respect to the operation name the optimizer handles. It is highly recommended that the identifier clause be used. If it is not used, in environments such as Lisp Machine Lisp, incremental recompilation of an optimizer will not properly redefine the old one.

Optimizers do not work in Multics Maclisp. Do not try to use them.

5.9 Rest Parameter Implementation

One of the most obvious points of incompatibility between various Lisp dialects is the implementation of *rest parameters*. LSB attempts to compensate for this by providing a consistent and safe default, and by allowing explicit specification of how the data object the formal parameter is bound to is to be implemented.

The default implementation of a *rest parameter* is as a heap-consed list: a list is a simple data representation guaranteed to be common to all Lisp implementations. It is heap-consed to prevent obscure and erratic behaviour when a pointer to it is passed up outside the dynamic scope of the function call.

The single commonality between Lisp dialects is that such an object is a kind of sequence. This means that (in theory at least) one should be able to access components of the object by some sort of indexing routine. One may thus declare in the prototype call that the variable is to be implemented as a sequence:

```
    (define-public-routine (print-items
                              (sequence (any-number-of items n)))
         (terpri)
         (do ((i 0 (1+ i))) ((= i n))
            (princ (elt items i))))
```

where elt is the generic sequence accessor (defined in NIL, but not in Lisp Machine Lisp, nor by default in Maclisp). This particular implementation is still heap-consed, however; it could lead to the production of quantities of garbage free storage. Since most of the time one uses a *rest parameter* one is *only* going to use it within the dynamic scope of the routine it is a parameter of, it is wasteful for the storage used to hold this sequence to *not* be temporary. Thus, LSB provides the concept of the argument-sequence sequence type and rest parameter implementation.

The argument-sequence (abbreviated argseq) rest parameter implementation causes the rest parameter to be implemented as the best method which does not consume free storage provided by the Lisp implementation. There are mechanisms defined for manipulating them, which map directly into the methods actually used in the Lisp implementation. What happens is that in Lisp Machine Lisp, specifying argument-sequence is like specifying pdl-list; in NIL, it is like specifying pdl-vector; and in PDP-10 Maclisp, the variable is bound to a fixnum which incorporates the information which is *implicitly* present in the call to a lexpr, and also acts as a fixnum declaration.

argref *argseq index*
: This fetches the *index*th element of the argument-sequence *argseq*. *Index* is zero-origined. In PDP-10 Maclisp, this is almost the same as a call to the arg function; in Lisp Machine Lisp, it is the same as a similar call to nth, and in NIL, it turns into a vref. In all implementations, this will be inline-coded when compiled.

argset *argseq index val*

This clobbers the *index*th element of the argument-sequence *argseq* to be *val*. In all implementations, this will be inline-coded when compiled.

argseq-length *argseq*

This returns the length of the argument-sequence *argseq*. Note that this is often not necessary, as one can get a variable bound to the length of the argument sequence in the prototype call. This will be inline-coded in all implementations.

argseq-list *argseq*

Returns the elements of *argseq*, in a list. This routine is *not* inline-coded, and its use is discouraged. It is provided so that one can get the elements out of an argument-sequence, as a list; this is primarily for debugging, and primarily for PDP-10 Maclisp, where an argument-sequence is implemented as a data-type which does not print out its components.

It is reasonable to declare a variable which is not a rest parameter to be an argument-sequence. Consider the following:

```
(define-public-routine (print-items (argseq (any-number-of items)))
    (terpri)
    (print-items-aux items))
(define-public-routine (prin1-items (argseq (any-number-of items)))
    (print-items-aux items))
(define-private-routine (print-items-aux (argseq items))
    (do ((i 0 (1+ i)) (n (argseq-length items)))
        ((= i n))
       (prin1 (argref items i))
       (princ " ")))
```

For use with the loop iteration macro [LOOP], LSB defines the argseq-elements (aka argseq-element) sequence iteration path, so that one may iterate over the elements of an argument sequence (or some subset of them):

```
(define-private-routine (print-items-aux (argseq items))
    (loop for x being the argseq-elements of items
          do (prin1 x) (princ " ")))
```

In Lisp implementations where an argument-sequence is a kind of list (Lisp Machine Lisp and Multics Maclisp), the argseq-elements iteration path is guaranteed to produce just a simple iteration over the list in simple cases like the above; that is, in these cases the list will not be repeatedly indexed into.

For those who know what they are doing, the following data-type keywords are recognized as specifying rest parameter implementations:

argseq
argument-sequence
: This is as described above.

list
: A heap-consed ordinary list. In the Lisp Machine implementation, this list will be made with all the elements cdr-next except for the last, which will be cdr-normal rather than cdr-nil—this is presumed to be a reasonable compromise.

vector
: A heap-consed vector. This exists primarily for NIL.

sequence
: A heap-consed sequence of the type appropriate for the implementation: equivalent to

list in Maclisp or Lisp Machine Lisp, **vector** in NIL.

The following additional implementations exist, but are less general, and primarily exist for one to take advantage of particular Lisp implementation features.

pdl-list
temporary-list
: This says to implement the rest parameter as a stack-allocated list. This is only truly possible on the Lisp Machine; in PDP-10 Maclisp, it causes explicit reclamation of the list on normal exit of the routine, and elsewhere is equivalent to **list**.

pdl-vector
temporary-vector
: The rest parameter is to be implemented as a stack-allocated vector. This is *only* possible in NIL. Elsewhere it *may* attempt to coerce the argument sequence into a heap-consed vector, but that of course will not work unless there is vector support.

Of course, if one is very concerned with efficiency and needs to take advantage of particular implementation features, one can always specify an implementation-dependent **implement-as** clause, such as the following (for the Lisp Machine):

```
(define-public-routine (print-items (any-number-of items))
   (dcls (implement-as expr (&rest items)))
   (do ((l items (cddr l))) ((null l))
      (print (car l)) (prin1 (cadr l))))
```

5.10 Macro Memoization

It is normally the case that any particular call to a macro will expand into the same code. It is thus a significant inefficiency for interpreted macro calls to be repeatedly expanded every time they are evaluated. The term *macro memoization* refers to the process of somehow remembering the expansion of a particular macro call so that it does not need to be repeatedly expanded.

There are various mechanisms for accomplishing this. The easiest and most efficient is to simply clobber the calling form with the expansion, by use of **rplaca** and **rplacd**. Another way is to clobber the form with another macro call which encodes both the original form and the expansion; this intermediate macro form can then be recognized specially by pretty-printers, which can choose to show either the original form or the expansion. If additional information like a definition count is encoded, then the clobbered expansion can be checked for validity and re-expanded when the macro is redefined. Yet another way to perform macro memoization is to store the expansion in a hash table; this has the advantage of not modifying the call at all, and also not showing the expansion in the code. The data stored in the hash table can also encode information like a definition count, so that the form can be re-expanded if the macro definition has changed.

LSB supplies three methods for macro memoization. They differ in the code which will be produced in the macro body; any particular option could additionally be under runtime control. The method chosen may be specified with the **do-macro-memoizing** clause, either in a system definition, module specification, or in the **declarations** of the macro itself.

none If the "argument" given to **do-macro-memoizing** is **nil**, then *no* macro memoization code is produced. Every time a call to such a macro is encountered, the expansion will be re-computed.

the implementation default

This is the default used by LSB. The actual method used varies according to the Lisp implementation; the intent is for the code generated in the macro definition to be runnable in a default environment of that Lisp implementation.

In Multics Maclisp, code is generated to clobber the original form with the expansion by use of rplaca and rplacd. In Lisp Machine Lisp, the displace function is called (q.v.). The PDP-10 and NIL implementations produce slightly more complicated code, calling functions which allow virtually all of the macro memoizing possibilities described above, under runtime control.

displace

This may be specified by use of the keyword displace as the "argument" to do-macro-memoizing. The code generated for the macro definition will call the displace function. Note that this is equivalent to the (current!) default action for Lisp Machine Lisp. If one uses this in Multics Maclisp, one should be sure that a displace function will be available at runtime. Again, the precise runtime semantics of this depend on exactly what displace does.

It is anticipated that a mechanism similar (if not identical) to that used in PDP-10 Maclisp and NIL will be implemented for Multics Maclisp and Lisp Machine Lisp. Even if not supported by the Lisp systems themselves, it would be usable in environments where it could be ensured that the necessary runtime support was loaded, and may aid in problems evident in those Lisp implementations due to redefinition of macros not affecting already-expanded calls to those macros.

5.11 Forward References

declare-routine *Special Form*

```
(declare-routine prototype-call
     dcl-clause-1
     dcl-clause-2
     ...)
```

produces all of the information needed to compile a call to the specified routine, in the current compilation environment, without defining the routine. This is not needed if one is using the LSB compiler interface (described in section 10.1, page 59) which makes a pass over the file extracting all information needed for compilation. It may be needed, however, if one is using LSB on Multics (which does not currently support the LSB compiler interface), or if one has specifically disabled this interface in the PDP-10 implementation.

Consider two routines which call each other:

```
(define-public-routine (foo a (optional b 0))
    (cond ((zerop b) a)
          (t (bar (times a b) (sub1 b)))))
(define-public-routine (bar x (optional y 0))
    (cond ((zerop y) x)
          (t (foo (plus x y) (sub1 y)))))
```

If the definition of bar has not been processed when foo gets compiled, the compiler will make default assumptions about bar when it compiles the call to it. These assumptions, if incorrect, could make the call less efficient; they might even make the compiler error out, or generate incorrect code.

The fix for this example is to put

```
        (declare-routine (bar x (optional y 0)))
```

before the definition for foo. LSB will extract from this the very same declaration information which it extracts from the definition of bar, *including* any code needed to perform calling sequence optimizations.

A declare-routine form should be constructed from the same prototype call and declaration clauses which are used in defining the routine, with the exception of any declaration clauses pertinent only to real definitions. The clauses which may be used are implement-as, value-type, primarily-applicable-routine, slow-and-hairy, called-as-lexpr, and perform-calling-sequence-optimizations.

5.12 Definitionless Routine Definitions

Sometimes one would like to use LSB to propagate declaration information about a routine, but either the routine is defined elsewhere (possibly without LSB) or it is not defined in Lisp. For this, one can simply omit the body of the definition. It is the responsibility of the user to ensure that the calling sequence LSB determines for the routine to in fact be identical to what it actually is, by use of the appropriate declarations. For example, the following defines for the PDP-10 a flonum-only + function which checks for overflow:

```
    (define-public-routine (f+ (flonum x) (flonum y))
        (dcls (value-type flonum) (implement-as expr))
        )
    (lap-a-list
      '((lap f+ subr)
        (args f+ (nil . 2))
            (push p (% 0 0 float1))
            (move tt 0 a)
            (jrst 2 @ (% 0 0 foo))
        foo (fadr tt 0 b)
            (jsp f (* 1))
            (tlnn f 40000) ; this is octal
             (popj p)
            (lerr 0 (% sixbit |FLOATING-POINT OVERFLOW!|))
        nil))
```

Actually, this example is simple enough that it would be best written with the assembly-language-definition clause, as follows:

```
        (define-public-routine (f+ (flonum x) (flonum y))
            (dcls (value-type flonum) (assembly-language-definition))
                (push p (% 0 0 float1))
                (move tt 0 a)
                (jrst 2 @ (% 0 0 foo))
            foo (fadr tt 0 b)
                (jsp f (* 1))
                (tlnn f 40000) ; this is octal
                 (popj p)
                (lerr 0 (% sixbit |FLOATING-POINT OVERFLOW!|)))
```

Of course, not all such cases are this simple, and it may not be possible to share code between various routines when the assembly code for them is in separate LSB definition forms.

5.13 Defining Functional Properties

Very often it is necessary to put a "function" on some property of a symbol. Lisp has the syntax

```
        (defun (foo propname) (this that) ...)
```

such that one may do

```
        (funcall (get 'foo 'propname) this that)
```

in order to invoke this function. LSB supports a similar syntactic construct.

For *routines*, one may simply use a list of the symbol and the property in place of the name of the routine, as in

```
        (define-public-routine ((foo propname) this that) ...)
```

which is the LSB way of doing the previous example. Although it may seem that there is no call for a visibility class in such a definition, there is: such a definition might need to be documented, and a visibility class is needed to determine where the documentation may need to be sent to. There is unfortunately no mechanism for remembering this information at this time, however, so one should not attempt to use document-routine (page 55) on such a thing.

For such a definition, LSB automatically inhibits declaration production and calling-sequence optimizations.

In PDP-10 Maclisp, one may also use the "three-list" format:

```
        (define-public-routine ((foo hackexpr hacksubr) a b) ...)
```

being essentially the same as

```
        (defun (foo hackexpr hacksubr) (a b) ...)
```

which puts the interpreted definition on the hackexpr property, but the compiled subr pointer on the hacksubr property.

If one considers *macros* to be simply a way to perform a mapping from one call-like form to some other form, then this extension is applicable to them also.

```
        (define-public-macro ((foo frobnicate) x y)
          (list 'cons x y))
```

puts a function on the frobnicate property of foo such that

```
        (funcall (get 'foo 'frobnicate) '(foo 1 2))
          => (cons 1 2)
```

Macro memoization (section 5.10, page 39) is turned off by default for this type of construct.

The syntax for this is both cumbersome and moderately unaesthetic. It is expected, however, that such constructs will rarely be written out, but rather constructed by macros which can hide the property-list implementation:

```
    (define-system-macro (define-frobnicator name bvl (body forms))
      `(define-private-routine ((,name frobnicator) ,@bvl)
          (dcls (also-needed-for public-compilation))
          ,@forms))
```

5.14 Prototype Call Summary

A variable specification in a prototype call has the full form shown below.

varspec ::= *simple-varspec* | *optional-varspec* | *rest-varspec*

simple-varspec ::= *typed-variable* | (quoted *typed-variable*)
| (unused *typed-variable*) | (unused (quoted *typed-variable*))
| (quoted (unused *typed-variable*))

The quoted option may only be used with *routines*, and is described on page 30. The unused is applicable to any type of LSB operation definition, and is described on page 27. If a *simple-varspec* appears in a prototype call to the right of an *optional-varspec*, it is interpreted as if it were (optional *simple-varspec*).

typed-variable ::= *variable-name* | (*data-type-kwd variable-name*)

All pre-defined data type keywords are enumerated in chapter 7, page 47.

optional-varspec ::=
simple-optional-varspec | (*data-type-kwd simple-optional-varspec*)

simple-optional-varspec ::=
(optional *simple-variable*)
| (optional *simple-variable default-value-form*)
| (optional *simple-value default-value-form variable-name*)

The uses of *default-value-form* and *variable-name* are explained in section 5.1, page 26.

rest-varspec ::=
simple-rest-varspec | (*rest-variable-implementation-type simple-rest-varspec*)

simple-rest-varspec ::=
(*rest-varspec-kwd simple-variable*)
| (*rest-varspec-kwd simple-variable variable-name*)

The specifics of the optional *variable-name* are discussed in section 5.1, page 26.

rest-varspec-kwd ::= a rest-implementation keyword

These are fully discussed in section 5.9, page 37.

rest-variable-implementation-type `::=`

```
any-number-of | one-or-more-of | two-or-more-of | body | rest
```

The specific keyword used implies the minimum and maximum number of arguments which are to be mapped into the particular rest-variable. There are none currently defined which specify a maximum. **one-or-more-of** and **two-or-more-of** specify one or two arguments as a minimum; **any-number-of**, **body**, and **rest** have no restriction.

6. Defining Variables

An LSB variable definition allows for all declaration and initialization information normally needed. The definition form thus provides a distinct locus in the source text for that information and the documentation. Defining a variable constitutes declaring it to be *special*; the specification of type information for variables which are not special (i.e., *local* or *lexical* variables) is handled by the constructs with which the binding is specified, as discussed in the previous chapter, and chapter 13.

`define-public-variable` (≡ **`defpubvar`**) *Special Form*
`define-system-variable` (≡ **`defsysvar`**) *Special Form*
`define-private-variable` (≡ **`defprivar`**) *Special Form*

These are the special forms with which one defines variables. As with all LSB definition forms, they should only appear "at top level" in a module, to be processed by the compiler or interpreter.

The general format of variable definition forms is

```
(define-visclass-variable variable-name
     clause-1
     clause-2
     ...)
```

as in

```
(define-public-variable *maximum-line-length
     (value-type fixnum)
     (default-init 78))
```

which declares *maximum-line-length to be special, says its value is always fixnum, and will initialize it to 78 if it is does not already have a value.

The clauses which may be supplied in addition to the common definition clauses (section 2.6, page 8) are:

value-type *type-name*
data-type *type-name*
: Asserts that the value of the variable is always of this type. Appropriate type declarations may be produced if warranted in the Lisp implementation.

initialization *initialization-form*
init *initialization-form*
: When the module is loaded, the variable will be unconditionally initialized to the value of *initialization-form*.

default-initialization *initialization-form*
default-init *initialization-form*
: Like the initialization clause, but only sets the variable if it is not already valued.

divert-reinitialization-to *divstream-1 divstream-2 ...*
divert-reinit-to *divstream-1 divstream-2 ...*
: This clause causes a setq form of the variable to its initialization to be output to each of the named diversion streams. It thus may only be specified if either the initialization or default-initialization clauses are given. Note that the reinitialization is always unconditional. This can be used to produce a file which when loaded will

reset a collection of variables to their initial states.

also-divert-reinitialization-to *divstream-1 ...*
also-divert-reinit-to *divstream-1 ...*

Although this probably is not needed because there are no default *reinitialization* diversions, it is included for uniformity. It could conceivably be of use if one had a macro which produced an LSB variable definition which provided for reinitialization diversion and also passed along declaration clauses.

The LSB variable definition facility may also be used purely for declaration purposes. This is useful in cases where either forward references occur (but see section 10.1, page 59), or where the variable is not really a part of the module but still needs to be declared for some reason.

declare-variable *Special Form*

This is a variant of the variable definition special forms which can be used for implementation-independent declaration purposes. It *only* accepts the value-type declaration clause, as none of the others are applicable.

Example:

```
(declare-variable *count*
    (value-type fixnum))
```

7. Data Types

LSB provides a scheme whereby one may symbolically specify the *data type* of something. This data type is used to provide variable and function value declarations (when appropriate to the Lisp implementation) and also to default the initial values of bound variables and unspecified optional arguments. It is also used to provide automated type checking for arguments.

Here are the data type keywords which LSB defines initially.

notype
: This essentially means "untyped". This is what you get when no data type keyword is specified, and there is no default for the particular context.

fixnum A limited-precision integer.

integer
: Any size integer.

number
: Any number.

flonum
: A flonum. This in general corresponds to the Lisp object with **typep** of **flonum**.

small-flonum
: Similar to **flonum**. This exists only in Lisp implementations which have such a data-type, such as Lisp Machine Lisp.

character-code
: In practice, this is equivalent to **fixnum**. In theory, it might cause special storage strategies to be used because of the limited range.

truthvalue
: In practice, this is the same as **notype**. In theory, it might be used to optimize returned-values of conditionals (and the like). It essentially states that only the t-or-nil-ness of the value is of interest.

argseq
argument-sequence
list
temporary-list
pdl-list
vector
temporary-vector
pdl-vector
sequence
: These data type keywords are all specially recognized as specifying the implementation of *rest parameters*. This is fully discussed in section 5.9, page 37.

7.1 Defining Data Types

One may define a data-type keyword to LSB in terms of an already defined type it is a specialization of.

define-public-data-type *Special Form*
define-system-data-type *Special Form*
define-private-data-type *Special Form*

```
(define-public-data-type data-type-keyword
       clause-1
       clause-2
       ...)
```

defines *data-type-keyword* to LSB. The information is declared in the current environment, in the compiled output file, and in the declaration file appropriate for the visibility class. All the normal definition keywords may be used in the clauses. One may additionally use the following clauses:

continue-with *data-type-keyword*
: This says we should get other information from the *data-type-keyword* data-type. This defaults to **notype.**

predicate *routine-name*
: This says that *routine-name* is a predicate of one argument which defines this data-type.

initial-value *initial-value*
: Specifies the default initial value to be used for this data-type.

For example, the Brand-X system defines the following data-types:

```
(define-public-data-type Brand-X-object
     (predicate Brand-X-objectp))
(define-public-data-type triple
     (predicate triplep)
     (continue-with Brand-X-object))
```

Each data-type which has a predicate associated with it can also have automatic argument type checking performed for an argument of that type. When one defines a data-type, the code to do this is automatically generated. The way in which this is done varies in different Lisp implementations: in Maclisp, the (output from) define-*visclass*-data-type is needed for this argument checking to be performed, but on the Lisp Machine it is not (although the predicate is if it will not be open-coded). For example, Brand-X gets automatic type checking for the argument given to the ilk routine, which is defined:

```
(define-public-routine (ilk (triple x))
     (dcls (check-args))
     ...)
```

8. Diversion Streams

Diversion streams are used by LSB to implement *derivability* (section 2.1). They can be loosely divided into two categories, depending on the kinds of objects and operations they handle: *form diversion streams*, which are used generally for Lisp code or forms, and *textual diversion streams*, which handle text. Within each of these broad divisions there are various additional types which determine how the diversions are to take place, and the transformations to be made on the objects diverted. For example, a *declaration diversion stream* is intended to divert *declarations*—that is, forms to be executed and operations to be defined, to tell the compiler how to compile things. The forms diverted to a declaration diversion stream will be compiled into a file. A *form diversion stream* is similar, but does not imply that the contents are for the use of only the compiler. A *textual diversion stream* is one which accepts text—it is copied directly to a file. A *documentation diversion stream* is similar, but additionally implies that the text is some form of documentation, and transformations may be made on it in the diversion process. There are also types of form diversion streams which cause the diverted forms to appear "at top level", as if they had not been diverted (since all form diversions can only occur by means of toplevel special forms), and those which cause the diverted forms to be immediately evaluated.

The definition or redefinition of diversion streams, if it becomes necessary, may be done by using the diversion-stream clause in the system definition. This clause has the format

```
    (diversion-stream name clause-1 clause-2 ...)
```

as in

```
        (diversion-stream interpreter
             (type toplevel)
             (predicate (lsb:not-compiling?)))
        (diversion-stream toplevel
             (type toplevel)
             (predicate t))
```

which are the default definitions used for by LSB for the **interpreter** and **toplevel** diversion streams. In general, every diversion stream has a *type*, which determines how it handles data sent to it (and the restrictions on that data, i.e. forms versus text), and a predicate which is evaluated every time an attempt is made to divert something to that diversion stream, to see if the diversion should be performed. The type **toplevel** handles only forms, and says that they should be treated as if they were "seen at toplevel" in the module.

The types of diversion streams are:

toplevel
: This handles forms only, and makes them "appear at toplevel".

form
: A diversion stream of this type saves the forms in a file. This is done by compiling them.

declaration
: Currently the same as **form**. This is for saving information needed for compilation of things (like declarations and macros).

text
: Handles text. The text is copied to a file.

documentation
: Like **text**, but additionally implies that the text is documentation; transformations may be made on it when it is diverted. See section 8.2.1, page 53.

eval Forms diverted to such a diversion stream are immediately **evaled**.

non-existent

Attempting to divert anything to a diversion stream of type **non-existent** is an error. Attempting to "load" one, if for example the **pubdcl** diversion stream of some module is of this type, does nothing. This is used by some LSB systems which are not actually implemented with LSB.

The other clauses of a diversion stream definition are only pertinent to diversion streams which produce files; these are the same clauses which may be specified for a module: **host**, **device**, **directory**, and **pathname**. When LSB constructs the pathname for a diversion file, missing components default to the corresponding components of the pathname of the module (as LSB calculates it to be from the system definition), except for the file-type and the version. The version defaults to the version of the module being read or compiled. The file-type defaults to the name of the diversion stream, with some exceptions dependent on the host file system:

Diversion Stream	LispM	NIL	Three-character
pubdcl	pubdql	pubdvl	pdc
sysdcl	sysdql	sysdvl	sdc
moddcl	moddql	sysdvl	mdc
pubdoc	pubdoq	pubdov	pdo
sysdoc	sysdoq	sysdov	sdo
moddoc	moddoq	moddov	mdo
macros	maqros	mavros	mac

These exceptions exist either (as in the case of TOPS-10) to compactify the name into 3 characters, or (more commonly) because corresponding diversion files for different Lisp implementations will be kept on the same file system. If for some reason these name defaults need to be hacked, see **lsb:*diversion-fn2s**, page 67.

There are additional system definition (and module specification) keywords which define diversion streams, and additionally default the type and predicate of it:

textual-diversion-stream
text-diversion-stream
text-divstream

This defaults the diversion stream to be of type **text**. The predicate defaults to **(lsb:compiling-to-file?)**; the text will only be diverted if a module is being fully compiled.

documentation-diversion-stream
doc-divstream

This defaults the diversion stream to be of type **documentation**. The predicate defaults to **(lsb:divert-documentation?)**.

form-diversion-stream
form-divstream

The diversion stream will be by default of type **form**. Forms (lisp code) diverted to it will be compiled into a file. The predicate defaults to **(lsb:compiling-to-file?)**.

declaration-diversion-stream
dcl-divstream

The diverstion stream will be by default of type **declaration**. Forms diverted to it will be compiled into a file; the predicate used by default is **(lsb:divert-declarations?)**. This type of diversion stream is distinct from **form** because it may do additional processing or setup on the forms, such as implicitly diverting some kinds of setup forms first; this is not done yet however.

Here are some predefined predicates for use with diversion stream definitions.

lsb:compiling?
This returns t if evaluated during a compilation, nil otherwise. It will return nil if called during the loading of a file (by lsb-load only, sorry), even if during a compilation.

lsb:not-compiling?
Equivalent to (not (lsb:compiling?)).

lsb:compiling-to-file?
This returns t if evaluated during the compilation of a file to a file, nil otherwise (and during loading of a file). This is the default for randomly defined diversion streams.

lsb:divert-documentation?
This returns t if lsb:compiling-to-file? would, *and* if the inhibit-documentation-production flag was not set by specification of that clause in the system definition or module specification. This is the default diversion stream predicate for all diversion streams defined with the textual-diversion-stream clause. Note that if one uses the diversion-stream clause but specifies a type of text or documentation (as explained below), the default predicate is still (lsb:compiling-to-file?).

lsb:divert-declarations?
This is the default predicate used for *declaration diversion streams*. It is currently equivalent to lsb:compiling-to-file? (q.v.), but may become more complex as facilities become better adapted to the use of a Lisp environment with a resident compiler, such as Lisp Machine Lisp.

8.1 Form Diversion Streams

When an LSB definition is processed, the information from it is partitioned up on the basis of what needs to be known where, and the forms generated are "sent" to various form diversion streams. An example of this is given in section 2.5, page 7:

```
(define-public-routine (square$ (flonum n))
     (dcls (value-type flonum))
     (*$ n n))
```

This says that square$ is a routine of one argument (*n*, a flonum) which always returns a flonum result. What is actually produced from this definition is something more on the order of

```
(divert-forms-to (pubdcl compilation-environment)
     declarations for square$)
(divert-forms-to (toplevel)
     (defun square$ (n)
          local-declarations if needed
          (*$ n n)))
```

That is, the definition of square$ is sent to the toplevel diversion stream, which is like having it specified at toplevel in the source file. The declarations for square$, however, are sent to the pubdcl and compilation-environment diversion streams: the former is a *declaration diversion stream* for public declarations, and the latter is *eval diversion stream* which has a predicate such that the forms will be evaluated immediately, but only during compilation.

divert-forms-to *Special Form*

```
        (divert-forms-to diversion-stream-names
            form-1 form-2 ...)
```

This is the primitive macro for initiating form diversions. It is only valid as a "toplevel form" in a module (similar to defun and eval-when). One may not nest this construct, and the behaviour of eval-when and declare inside of it is not defined. What is usually more convenient to use than this is forms-needed-for:

forms-needed-for *Special Form*

```
        (forms-needed-for needed-for-keyword-list
            form-1 form-2 ...)
```

This is similar to divert-forms-to, but accepts needed-for keywords, as described in section 2.6, page 8. If a keyword implicitly needs a visibility class, private is assumed; hence, in this context, the compilation keyword is equivalent to private-compilation.

8.2 Textual Diversion Streams

textual diversion streams are diversion streams whose primary operation is manipulation of text, rather than forms. There are two types of diversion streams defined which handle text: text, which transcribes the diverted text literally, and documentation, which may perform some transformations in the process.

Text diversion is effected by an extension of the read-time conditional inclusion mechanism: the *exclusion* of text by the { reader macro allows its *inclusion test* to specify what diversion streams the excluded text should be diverted to. There is additionally provision for actions to be taken before, after, and during the exclusion process.

divert-to *Inclusion Tester*

(divert-to *divstream-1 divstream-2* ...) is the simplest inclusion test for diverting text. It is only for use in read-time conditional inclusions, as there is no other conditional inclusion mechanism which can provide a source of "text". Thus,

```
        {(divert-to pubdoc)
        This is some documentation.
        }
```

diverts all of the text from the first ")" to the last "}" to the pubdoc diversion stream.

There are several other inclusion tests for use with diverting documentation in more complicated ways, rather than just a literal transcription as divert-to does. It is therefore possible for tranformations on the diverted text to occur both as a result of the way the diversion stream handles the diversion operation, and as a result of the way the text is sent to the diversion stream. These complications are described in chapter 9, page 55. Many of them only function on documentation diversion streams, as they implicitly perform higher-level formatting operations which only documentation diversion streams can supply.

8.2.1 Documentation Diversion Streams

Documentation diversion streams differ from plain text diversion streams mainly by extension. A documentation diversion stream has a documentation-type, which may be specified with the documentation-type clause in the diversion stream definition, and it may be defaulted for all documentation diversion streams by being used as a system definition or module specification clause. There are two documentation types currently defined: bolio, the default, and tex. This normally only manifest themselves when the more complicated documentation diversion inclusion tests are used; these are documented fully in chapter 9.

```
(define-system hairily-documented
    (built-on this that the-other-thing)
    (documentation-type tex)
    (documentation-diversion-stream extra-doc
       (documentation-type R))
    (modules foo
             (bar (documentation-type bolio))
             (baz (documentation-diversion-stream pubdoc
                    (documentation-type bolio)))))
```

The hairily-documented system has a default documentation-type of tex; this is implied by the (documentation-type tex) at top level in the system definition. All documentation diversion streams which are not otherwise specified will default this way. For the bar module, however, the default documentation type is bolio. Note that the type of defaulting going on here does not affect the extra-doc diversion stream, since the documentation type for that never gets defaulted; for all modules, it will be of documentation type R. Likewise, for the baz module, all the documentation diversion streams will have documentation type tex except for extra-doc and pubdoc, which will be R and bolio. Note that there is no R documentation type presently.

8.3 Pre-Defined Diversion Streams

Here are the diversion streams initially defined in an LSB environment.

pubdoc
: A documentation diversion stream, intended for public documentation. Its predicate causes diversion to occur only when the containing module is being compiled and documentation diversion is enabled: it uses the lsb:divert-documentation? predicate.

sysdoc
: Similar to pubdoc, but for system documentation.

moddoc
: A documentation diversion stream, intended for private documentation. By default this diversion stream has a predicate of nil, so text sent to it goes nowhere; that predicate may however be modified in the system definition.

info
: Like pubdoc. This is provided somewhat spuriously. It could be used for such things as online documentation.

pubdcl
: This is a declaration diversion stream, for public declarations. Its predicate causes diversion to occur only when the module is being compiled.

sysdcl
: Like pubdcl, for system declarations.

moddcl

For **private** declarations. This normally has predicate of **nil**, causing no diversions. That may be changed by the user if it is found to be needed for some obscure forward-reference problem.

compilation-environment

An **eval** diversion stream, with a predicate that causes the diversion (and hence evaluation) to occur only in the compiler.

readtime-environment

In a compilation environment, this is defined as an **eval** diversion stream, causing the forms diverted to be immediately evaluated; otherwise, it is a **toplevel** diversion stream, thus being equivalent to **interpreter**. The result of this is that the "diversion" occurs in the processing environment, so may be used to modify the LSB environment, or the reading environment. This may be renamed to **processing-environment**.

interpreter

This is a diversion stream of type **toplevel**, defined with a predicate which causes no diversion when being processed by the compiler: thus,

```
(divert-forms-to (interpreter) ...)
```

acts like

```
(eval-when (eval) ...)
```

compiler-toplevel

A **toplevel** diversion stream with a predicate complementary to that of the **interpreter** diversion stream.

toplevel

A **toplevel** diversion stream with a predicate of t. This is useful if the forms to be diverted not only should be processed "at top level" in the module, but *also* sent somewhere else (e.g., to the **pubdcl** diversion stream).

macros

This diversion stream is used primarily for macro definitions which are not needed in a totally compiled system (see section 5.5.1, page 34). In Maclisp, it is by default a **form** diversion stream which will compile its forms into a file, when used in the compiler, and do nothing in the interpreter (like **pubdcl**). In Lisp Machine Lisp and NIL, it will by default be a **toplevel** diversion stream, so that the forms in it *are* put in the compiled output file.

9. Documentation Diversion

Section 8.2 described *textual diversion streams*, and how text may be sent to them. This chapter discusses more advanced facilities for diverting text and producing documentation.

public-documentation *Inclusion Tester*
system-documentation *Inclusion Tester*
private-documentation *Inclusion Tester*
online-documentation *Inclusion Tester*

These routines are for use as inclusion tests. They are equivalent to (divert-documentation-to *divstream*) for the appropriate diversion stream: pubdoc, sysdoc, moddoc, or info.

divert-documentation-to *Inclusion Tester*

(divert-documentation-to *divstream-1 divstream-2* ...) is an inclusion test which always fails, and causes the text within curly-brackets to be interpreted as documentation and sent to the specified diversion streams. Example:

```
{(divert-documentation-to pubdoc)
.chapter "Hacking Around"
          This is a test of the emergency broadcast system.
It is only a test.  Had it been a real emergency you
would have run out of list storage.
}
```

This inclusion test is *not* equivalent to divert-to (page 52). The enclosed text is output within a "documentation block", which means that it will be preceded by a blank line; this is irrespective of whether a newline immediately follows the inclusion test, as such a newline is ignored.

When LSB defines operations or variables, it records various attributes of them in the environment (either compiler or interpreter). This information is then used by the following routines to supplement user-supplied documentation. For operations (routines, macros, and special-forms, but *not* compile-time-macros), this information includes such things as the type of definition, information about the prototype call, and the value-type. For both, most importantly, it includes the diversion stream(s) to which documentation about the object defined is to be sent to. The following inclusion tests utilize this information in order to figure out where to send the excluded text.

document-routine *Inclusion Tester*
document-routines *Inclusion Tester*

These two inclusion tests are identical; both names are provided for euphony. (document-routine) as an inclusion test will document the most recently defined routine; (document-routine a) will document a, and (document-routines a b c) will document a, b, and c as a group, for an effect similar to that in this text here. For example, one might do

```
(define-public-open-codable-routine (square (number n))
    (dcls (value-type number))
    (times n n))
{(document-routine)
ε3squareε* returns the square of its argument.
}
(define-public-open-codable-routine (square$ (flonum n))
    (dcls (value-type flonum))
    (*$ n n))
(define-public-open-codable-routine (square& (fixnum n))
    (dcls (value-type fixnum))
    (* n n))
{(document-routines square$ square&)
ε3square$ε* and ε3square&ε* are the flonum-only
and fixnum-only versions of ε3squareε*.
}
```

What happens is the diverted text is output between stuff computed from the definition information, to produce a special text-justifier construct for the particular type of definition. What is actually produced depends on the documentation-type of the diversion stream(s); this is described later in this chapter.

document-variable *Inclusion Tester*
document-variables *Inclusion Tester*
These is similar in form and function to document-routine.

```
(define-system-variable *frobozz*
    (default-init (create-a-crock)))
{(document-variable)
This is a disgusting crock.
}
```

When one is utilizing the same source text in different Lisp implementations, it is often unnecessary to redundantly produce documentation from both. The default predicate used by documentation diversion streams (lsb:divert-documentation?, page 51) checks the flag set by inhibit-documentation-production system definition option, which says that documentation should not be diverted. For example, the pretty-print-definition system is defined:

```
(define-system pretty-print-definition
    (directory format)
    (built-on loop sharpsign backquote {PDP-10 user-hunk})
    (users-implicitly-need write)
    {(except-for PDP-10) (inhibit-documentation-production)}
    (modules ppdef ppdesc)
    )
```

When address space is a consideration and the above operation and variable documentation facilities are not being used, one should also use the inhibit-documentation-production option, because in addition to inhibiting the diversion of documentation, it tells LSB not to record information about the definitions.

9.1 The Bolio Documentation Type

Bolio is a text justifier written in PDP-10 Maclisp. It comes with predefined operators and conventions for documenting Lisp programs; because of this, it is the default documentation type. Bolio was used to produce the Lisp Machine Manual, and this document.

The output produced for Bolio by

```
    (define-public-open-codable-routine (square (number n))
         (dcls (value-type number))
         (times n n))
    {(document-routine)
    ε3squareε* returns the square of its argument.
    }
```

looks like

```
    .defun square ε1(numberε*∀nε1)ε*
    ε3squareε* returns the square of its argument.
    .end_defun
```

All of that randomness after square on the .defun line is font switching and spacing so that Bolio does not need to do any parsing of the argument descriptions. The output produced by

```
    (define-public-open-codable-routine (square$ (flonum n))
         (dcls (value-type flonum))
         (*$ n n))
    (define-public-open-codable-routine (square& (fixnum n))
         (dcls (value-type fixnum))
         (* n n))
    {(document-routines square$ square&)
    ε3square$ε* and ε3square&ε* are the flonum-only
    and fixnum-only versions of ε3squareε*.
    }
```

looks like

```
    .defun square$ ε1(flonumε*∀nε1)ε*
    .defun1 square& ε1(fixnumε*∀nε1)ε*
    ε3square$ε* and ε3square&ε* are the flonum-only
    and fixnum-only versions of ε3squareε*.
    .end_defun
```

Different types of definitions produce different documentation operators, in a similar format, differing only in the text-justifier commands used and the argument descriptions: only routine definitions, producing .defun, output any argument descriptions. Variable definitions use .defvar, .defvar1, and .end_defvar, macro definitions use .defmac, .defmac1, and .end_defmac, and special forms (or routines or macros which have the document-as-special-form option specified in their declarations) use .defspec, .defspec1, and .end_defspec.

Except on Multics (which does not do case-conversion on input), variable, argument, data type, and operation names are converted to lower case when output. Call mapping keywords, such as optional, are capitalized. In Lisp implementations with packages, some heuristics are used to attempt to determine how the defined object's name should be printed. For routines, the names of the arguments will always be output without any package information.

9.2 The TEX Documentation Type

The output produced by the tex documentation type for the square example of the previous section looks like

```
        \defun SQUARE \argtype{number}{n}.
        user-text
```

Note that no indication of the end of the text is given, although there will be a blank line there. The tex format in general produces calls like argtype above rather than pre-formatting the type and call-mapping keyword information; e.g., the routine

```
        (define-public-routine (foo (quoted a) (optional (flonum b)))
            ...)
```

would produce the \defun header

```
        \defun FOO \quoted{quoted}{a}
                        \optional{optional}{\argtype{flonum}{b}}.
```

except that it would all be on one line. The reason quoted and optional appear to be duplicated above is that the name in braces is the actual keyword used in the definition, which may be different from the macro name. Currently, the routine, macro, or variable name is *not* lower-casified, although the variable names and keywords are. Like bolio, the tex documentation type can produce multiple \defuns in a block, by using \defun1. It will also produce calls to \defvar, \defmac, and \defspec, and \defvar1 etc. Unlike bolio, all of the operation documenting macro calls produced will contain the argument information from the prototype call.

The full list of argument descriptor macros used is:

\optional{*kwd*}{*innards*}
\rest{*kwd*}{*innards*}
: where *kwd* is the actual call-mapping keyword used, and *innards* is the remainder of the argument description.

\quoted{*kwd*}{*innards*}
: Similar to the above.

\argtype{*kwd*}{*variable*}
: The \argtype will be the most deeply nested macro call if it is present, so its second "argument" can only be the variable name.

Thus the output produced for the variable args in the prototype call (foo (any-number-of (quoted (fixnum args))) looks like

```
        \rest{any-number-of}{\quoted{quoted}{\argtype{fixnum}{args}}}
```

There is no existing package of TEX macros to do anything with this output, yet.

10. Getting LSB

10.1 The LSB Compiler

LSB provides its own standard compiler interface. It is very similar to whatever standard compiler interface is normally provided, but offers one option (as a default) which none do: it reads and macro-processes the input file fully before it begins compilation. That is, it incrementally reads in the file, expanding toplevel macros, LSB definitions and diversions, declares, eval-whens, and includes, but instead of immediately outputting or compiling the resultant forms as an ordinary compiler would, they are buffered up. Only when all of the input has been processed are the forms compiled.

This is a very useful action, due to the way LSB works. Because all of the declarative information about defined objects in LSB is derived from the definition form, one does not declare everything that needs declaring at the front of the file; thus, this first pass allows LSB to extract all of the declarations (and macro definitions) which will be needed for the compilation before the compilation starts. Note that the reading of the file is done incrementally with this form processing: one may, in something like an eval-when or in a toplevel macro call (which the module form at the front of the file is) modify reader attributes, such as syntax and input radix. This also obviates the need for declare-routine (page 40) and declare-variable (page 46) in many cases.

Having the file processed in this manner does not solve all forward-reference problems; only those "one level deep", if that much. LSB makes use of some declaration information when it expands out definitions, and information it needs should be around before one of its forms is to be processed. Thus, if one does something like

```
(define-public-routine (ilk (triple x))
    (dcls (type-check-argument x))
    ...)
(define-public-data-type triple
    (predicate triplep)
    (continue-with brand-x-object))
```

the ilk routine will *not* have type checking performed for it, because at the time LSB creates type-checking code the triple data-type (and the associated mechanism for doing argument type checking for that type) has not been defined. The same could be true for top-level calls to macros which are defined later in the file:

```
(define-a-frob foobar)
(define-public-macro (define-a-frob name)
    ...)
```

Whether or not the ordering matters in this last case depends on what (define-a-frob foobar) expands into; for safety, constructions like this should be ordered properly. Note also that this ordering constraint is in fact that which would be necessary to load the code interpreted anyway, since there is no pre-pass made when a source file is loaded into a lisp.

10.1.1 The Maclisp Compiler

The PDP-10 LSB compiler is a normal PDP-10 compiler with LSB in it, running the LSB compiler interface. The command interface to it is the same as the ordinary PDP-10 Lisp compiler, although it will run the LSB file-processing interface as described above. If for some reason this LSB interface is undesirable, say the file is particularly large and does not fit into the compiler, then one can disable this mode by negating the "L" compiler switch: for example, to the compiler's command-processing loop, saying:

```
        myfile (t-1)
```

On the ITS operating system, the LSB compiler may be invoked with the :LSBCL command. There is currently no LSB compiler available on non-ITS operating systems.

In Multics Maclisp, there is also currently no saved LSB compiler available. The special LSB file-processing interface is not available either. One may, however, bootstrap up an LSB in an ordinary compiler by placing the following form at the front of the source file, *before* the **module** form:

```
        (eval-when (compile)
            (or (status feature LSB)
                (load ">udd>Mathlab>LSB>compilation-environment.lisp")))
```

This form will thus work both in a compiler without LSB, and in a saved LSB compiler if and when one becomes available. One *must* use a compiler/lisp which understands the **eval-when** special form; LSB depends on it. If the source file is also to be used in Lisp implementations other than Multics, the **Multics** feature should be checked for too:

```
        (eval-when (compile)
           (and (status feature Multics)
                (not (status feature LSB))
                (load ">udd>Mathlab>LSB>compilation-environment.lisp")))
```

10.1.2 On the Lisp Machine

At some future time, LSB should exist saved on a disk band. Currently, one may cause LSB to be loaded by doing

```
        (load "MC:LSB;LISPM LOAD")
```

which loads *everything*, and is thus a bit time consuming.

To compile an LSB module, do *not* use **qc-file**:

lsbcom *infile* &optional *outfile package-spec*

This is similar to **qc-file**, but runs the LSB compiler interface. The arguments **lsbcom** takes are interpreted the same way **qc-file** interprets them (q.v.).

10.1.3 On the VAX

To be written when developed. It is suspected that it may not be possible to compile LSB on the 10 for the VAX because of address-space limitations. LSB use on the VAX will probably not differ drastically from that on the PDP-10.

10.2 Interpreted LSB

On ITS, LSB is available as a dumped environment under the name LSB. This environment contains some things which are not strictly a part of LSB but which are commonly used by most current LSB users. If demand indicates, this can be cleaned up.

Dumped subsystems on ITS very often will need LSB in them if any code is going to be run interpreted in them, but for production purposes this may be undesirable. It is possible to create versions of dumped subsystems which do and do not contain LSB, and which share the portion of the subsystem not containing LSB. This is, in fact, a general feature of Maclisp on ITS, and has nothing to do with LSB; it is documented elsewhere. The file LSB FILES on the LSB directory will, when loaded, load in those parts of LSB normally needed for running interpreted code, and set up autoload properties for some others which are only rarely used. All LSB autoload properties in PDP-10 Maclisp are of the form `((lsb) ...)`, so on a non-ITS system the atom `lsb` may be given a `ppn` property if needed. Much LSB code automatically defaults the `lsb` `ppn` property to that for `lisp`.

On Multics, the file `>udd>Mathlab>LSB>lsb-loader.lisp` is equivalent to the LSB FILES file of PDP-10 Maclisp—it loads only those parts of LSB normally needed for interpretation.

The Lisp Machine programming environment is such that getting LSB for interpretation is the same as getting it for compilation.

11. Coming Attractions

These are random notes on things which are either under development or are being considered.

In PDP-10 Maclisp, it is possible for the file property list to be parsed and used in a LEDIT/EMACS combination, to allow the proper binding environment to be established when code is transferred from the EMACS to the LISP. This can use the LSB option of the file property list. An experimental version of this has been tried, but the minimal hooks necessary do not yet exist with the system-supplied LEDIT.

One problem with using LSB is that things which are logically built-on it are then required to use LSB if they are to properly have their compilation (for example) environment established. In most cases, however, loading the various pubdcl files of the system(s) involved will suffice. A relatively small amount of code would be needed to support the loading of these files into a bare compiler. This is mainly applicable to Lisp environments like Maclisp where the compilation environment is distinct from the runtime environment. For the Lisp Machine, the potential exists for either causing the contents of the declaration diversion files to be "expanded out" so that they do not utilize LSB, or again, to simply have some special code to allow them to be loaded. The latter would require there to be an LSB package. Yet another alternative for the Lisp Machine (or similar) implementation is to cause *all* of the declaration information to be output into the compilation output file; this differs from splitting it into multiple (e.g., qfasl, pubdcl, sysdcl) files in that there would be no duplication of code. Again, this might possibly be done either by having some bootstrap LSB code around at load time, or by convincing LSB to "open-code" the declaration info it outputs; this last is only moderately space-consuming, as much of the stuff output involves declaration info which is redundant with the runtime environment, and error checking.

It is possible to compile portions of an LSB module "out of context". All that needs to be done is to run the LSB pre-processing step (section 10.1, page 59) over the *original* file to extract all private declarations, and then compile the file as if it were that module itself. Appropriate fudging of the diversion streams is necessary to ensure that erroneous diversions are not created for that module, but is not difficult. An experimental version of this has been tried, and all that is necessary is to put a patch-module form at the front of the file, instead of using module. This facility is not available by default yet, but probably will be soon.

12. Extending LSB

This chapter describes various methods and conventions which may be used to extend or customize LSB in some way. The contents are somewhat haphazardly organized, and in many cases there is missing description of how to do things, but it is suitable as a reference for relatively stable but internal facilities of LSB. Nothing in this chapter should be used frivolously; it is primarily compensation for lack of better "public" facilities. Also, any changes to the contents of this chapter would warrant a warning to the INFO-LSB mailing list, so it is safer to use what is listed here than just anything you might find in the source code.

If you use any facilities documented here, it is recommended that the system using them be built-on the lsb system. Although use of some things here does not require this, not all macro definitions and declarations may be pre-loaded in all Lisp implementations.

12.1 LSB Keyword Comparison

Here are the various routines for comparing symbols for LSB keyword or token equality. Note that in all cases the "keywords" being compared are expected to be interned symbols; this may matter in some Lisp implementations.

`lsb:token-equal` *token1 token2*
: implements *token equality* testing.

`lsb:token-member` *token list*
`lsb:token-assoc` *token a-list*
`lsb:token-lookup` *token a-list*
: are analogous to member and assoc. lsb:token-lookup is like using the Lisp Machine function memass; it returns the sublist whose car is what would be returned by lsb:token-assoc. That is,

```
(defun lsb:token-assoc (token a-list)
   (car (lsb:token-lookup token a-list)))
```

`lsb:kwd-equal` *token1 token2*
`lsb:kwd-member` *token list*
`lsb:kwd-assoc` *token a-list*
`lsb:kwd-lookup` *token a-list*
: The versions of the above predicates which check using *keyword equality*.

`lsb:kwd-bassoc` *token a-list*
: This is like lsb:kwd-assoc but bubbles an entry found forward in *a-list*. This should only be used for things which may be safely modified.

Many LSB "tables" are implemented as association lists. Since the keys of the entries cannot necessarily be compared with eq or equal, the following macros may be used to push new entries on.

lsb:push-pair *Macro*
```
        (lsb:push-pair (displace . lsb:displace-macmem)
                       lsb:*macro-memoizers)
```
pushes the entry (displace . lsb:displace-macmem) onto the list lsb:*macro-memoizers. If the variable is not bound, it will be set to nil first. If there is already an entry for displace there, it will be removed.

This macro is defined such that its expansion can be run in a Lisp without LSB present. Therefore it does not actually use *keyword equality*, but cheats and only uses *token equality*. For that reason, it is imperative that only the "canonical" form of a keyword be used in this manner. Note that if the compiler puts the call to lsb:push-pair rather than its expansion in the compiled output file, then LSB will need to be around when the file is loaded. The PDP-10 Maclisp compiler normally will completely macro-expand forms before stuffing them into its output file.

lsb:push-sym *Macro*

This is just like lsb:push-pair, only it *does* use eq for comparing "keys". It may thus be used for adding entries to association lists of (say) variable names. Qualifications for lsb:push-pair about runtime support apply here also.

12.2 Defining System Definition Options

This section documents some facilities which may be used for defining LSB options, which are specifiable in system and module definitions. It may be safely skipped by those who are not interested in defining their own. The facilities here should not be used frivolously; they are intended to be used by the maintainers of systems which need to provide special processing environments for their users.

An LSB option is essentially a state which can be encoded in some variable(s). It has a routine to determine the value(s) implied by the option clause, and each variable has a default value which is used in the absence of a specification.

define-lsb-option *Macro*

(define-lsb-option *option-keyword interpretation-fn*
    varspec-1 varspec-2 . . .)

defines *option-keyword* to be an option for inclusion in a system definition or module specification. *interpretation-fn* is a function of one argument, the clause, which should return an association list of the variables to be modified and their values. Each of the *varspec-i* describes the variables which may be modified (and thus may need to be bound to set up an LSB environment); it may be either just the variable, in which case nil is used as the default value, or a list of the variable and a form to be evaluated to get the value. For example,
```
        (defun hack-input-radix (clause)
            (list (cons 'ibase (cadr clause))))
        (define-lsb-option input-radix hack-input-radix
            (ibase 10.))
```

define-lsb-flag-option *Macro*

This is really a special case of define-lsb-option. The variable(s) will take on only t or nil as values. For example,

```
        (define-lsb-flag-option do-argument-type-checking
            lsb:*type-check?)
```

defines do-argument-type-checking such that either of the clauses

```
        (do-argument-type-checking)
        (do-argument-type-checking t)
```

turns *on* type checking (by setting lsb:*type-check? to t), and

```
        (do-argument-type-checking nil)
```

turns *off* type checking. The syntax to define-lsb-flag-option is the same as that to define-lsb-option, minus the function.

Note that if one desires an option defined with define-lsb-option or define-lsb-flag-option to take effect in the compilation environment, one must explicitly use a forms-needed-for form, like

```
    (forms-needed-for (running public-compilation)
       (defun hack-input-radix ...)
       (define-lsb-option input-radix ...))
```

unless the module is needed-for-user-compilation and the option is not used by the system which defines it.

By special dispensation, in Maclisp it is possible to load compiled calls to define-lsb-option and define-lsb-flag-option into a Lisp which does not have LSB present. If LSB is loaded in at a later time, these options will be in effect (unless otherwise redefined).

12.3 Defining New Readtables

lsb:*readtables *Variable*

This is an association list of keywords and the readtables they represent. Each "readtable" itself is allowed (and in fact recommended) to be a symbol whose value is the readtable to be used. The initial value of this variable is

```
        ((standard . lsb:*standard-readtable))
```

lsb:*standard-readtable *Variable*

The value of this is used as the "standard" and default readtable by LSB. It is initialized to the readtable current when LSB is loaded in; it will typically be the one and only readtable in the Lisp environment.

Thus, one might define a new readtable to LSB by doing

```
    (lsb:push-pair (readtable . Brand-X-Readtable)
                   lsb:*readtables)
```

in some appropriate place.

12.4 Playing with the System Definition

lsb:establish-sysdef *system-spec*

system-spec is exactly what might be specified inside of (say) a built-on clause, as described on page 12; either the name of a system, or a list of the name of a system, and a pathname suggesting where to find the system definition. This performs all the actions associated with searching for a system definition (if it is not already known!) described early in this manual. It returns a list, the car of which is the canonical name for the system, and the cdr of which is the system definition body.

lsb:*sysdefs *Variable*

An a-list of all known system definitions. The car of each entry is the name of the system (which should be compared using LSB token equality), and the cdr is, if non-atomic, the definition. If the cdr is atomic, then it is the name of another system whose definition should be used instead ("indirected to").

lsb:*syslocs *Variable*

An a-list of system names and locations. The location here is exactly that supplied with define-system-location.

lsb:determine-module-file-group *module-spec system-def*

system-def is a system definition, of the form returned by lsb:establish-sysdef. *module-spec* is the entry for the appropriate module out of the modules clause of the system definition. This returns a representation for the pathname of the module which does *not* include a file-type or version; this is used for such things as finding either the source or compiled output file, or for defaulting the pathname for a diversion stream associated with that module.

lsb:determine-diversion-filename *divstream-clause module-spec system-def version?*

This determines the actual pathname for the diversion stream specified by *divstream-clause*, with respect to *module-spec* and *system-def*, using a version of *version?*. *module-spec* and *system-def* are the same as for lsb:determine-module-file-group. *divstream-clause* is the clause defining a diversion stream. *version?*, if not nil, should be the version to be used in the generated filename.

When LSB itself calls this to determine the *output* pathname for a diversion stream, *version?* is the version of the module source file, *module-spec* is the module-spec of the module being compiled, and *system-def* is the system definition of the system the module is a part of. *divstream-clause* is whatever lsb:find-divdef would return for *module-spec* and *system-def*.

When LSB calls this to determine the *input* pathname for a diversion stream (say a pubdcl diversion stream to be loaded), *version?* is nil. In theory, either it could be the actual version of the "installed" source for the module in question, or some symbolic indicator that that is what should be used. For input it is assumed that an unspecified version does something reasonable (typically, retrieving the "most recent" one).

lsb:find-divdef *divstream-name module-spec system-def*

This looks up the diversion stream definition for *divstream-name* for module *module-spec* in *system-def*. If there is a diversion-stream defining clause for *divstream-name* in *module-spec*, that is returned; otherwise, if there is one in *system-def*, that is returned; otherwise, if there is an LSB default for *divstream-name*, that is returned, otherwise nil.

lsb:*diversion-fn2s *Variable*

This is an association list of diversion stream names and their default file-types. It *may* need to be hacked if cross-compilation is being done. Its value on the Lisp Machine, for example, is:

```
((pubdoc . pubdoq) (sysdoc . sysdoq) (moddoc . moddoq)
 (pubdcl . pubdql) (sysdcl . sysdql) (moddcl . moddql)
 (macros . maqros))
```

The diversion stream names are looked up, as always, using LSB keyword equality.

12.5 Inclusion Tests

A non-atomic inclusion test has a routine associated with it. When the inclusion test is performed, this routine should return nil if the text enclosed in curly-brackets is to be skipped over, non-nil if it is not.

12.5.1 Simple Inclusion Tests

The variables, routines, and macros described here should be sufficient to define simple inclusion test routines, such as only-for, except-for, only-on, and except-on.

lsb:*implementation-features *Variable*

This variable is normally nil. If it is set non-nil, then it is used as the set of "destination features" used by the only-for and except-for inclusion tests, instead of the result of (status features). Note that the only-on and except-on inclusion tests *always* use (status features).

lsb:perform-implementation-feature-tests *implementation-feature-tests return-first-null-result? return-first-non-null-result? features-to-consider?*

This is the routine used to parse *implementation feature tests*, like those given to the only-for inclusion test. If *features-to-consider?* is nil, then lsb:*implementation-features is used if that is not nil, otherwise (status features). See define-inclusion-test, below.

define-inclusion-test *Macro*

```
(define-inclusion-test name bvl
      form-1 form-2 ...)
```

At this time, the function defined for an inclusion test gets exactly one argument, the cdr of the inclusion test. At some future date it is anticipated that bvl will be treated in some other manner so that there can be automatic number-of-argument checking. The only-for inclusion test is defined as:

```
(define-inclusion-test only-for (tests)
   (lsb:perform-implementation-feature-tests
      tests () 't ()))
```

12.5.2 Environment Modifying Inclusion Tests

LSB keeps a stack of data which is used around succeeding conditional inclusions. This is used both for recording the location of the initiating left-curly-bracket, and for possibly performing some specific cleanup action when the right-curly-bracket is encountered.

lsb:input-file-status

Returns some information about the name and position of the current input file. Normally this will be a list of the name of the file, and the current file position.

lsb:*asynchronous-environment-stack *Variable*

This is the stack of information on how to deal with all currently unmatched left-curly-brackets. Each entry on the stack is a cons of a description of the left-curly-bracket, and how to undo it. The latter if not nil, is a cons of a function to apply to do the cleanup action, and the arguments to apply it to. The default action performed for a succeeding conditional inclusion is

```
(push (list (list* (lsb:input-file-status)
                   "conditional inclusion test"
                   the-inclusion-test))
      lsb:*asynchronous-environment-stack)
```

If an inclusion test desires to manipulate lsb:*asynchronous-environment-stack, it may do as as shown below, and instead of returning just any non-nil value, it should return the atom lsb:*asynchronous-environment-stack to tell the caller that it has already performed that action. For example, the following defines the gross-hack inclusion test (which for simplicity ignores its arguments) to make the variable *gross-hack* t for the duration of the curly-brackets:

```
(define-inclusion-test gross-hack (ignore)
   (push (list (list (lsb:input-file-status) "Gross Hack")
               #'(lambda (val) (setq *gross-hack* val))
               *gross-hack*)
         lsb:*asynchronous-environment-stack)
   (setq *gross-hack* t)
   'lsb:*asynchronous-environment-stack)
```

12.5.3 Text Diverting Inclusion Tests

Text diverting inclusion tests are inclusion tests which always fail, and which also manage to state where the excluded text should be diverted to. The simplest way for this to be done is with the following routines:

lsb:divert-to-1 *list-of-diversion-stream-names*

This should *only* be called from within an inclusion test which is going to return nil. lsb:divert-to-1 itself returns nil so that it may be used as the last form of an inclusion test routine. It causes the excluded text to be transcribed verbatim to the named diversion streams. Multiple calls may be made to lsb:divert-to-1 if necessary; specifying a diversion stream multiple times will have no effect. For example,

```
(define-inclusion-test divert-my-text (ignore)
   (lsb:divert-to-1 '(mydoc)))
```

defines the inclusion test routine divert-my-text such that (divert-my-text) is equivalent to (divert-to mydoc).

lsb:divert-doc-1 *list-of-diversion-stream-names*
This is just like lsb:divert-to-1, but additionally defaults lsb:*diversion-routine (see below) to a routine which (1) starts the diversion output on a fresh line while (2) flushing the initial newline (if any) at the start of the diverted text. Thus, the public-documentation inclusion test could have been defined by

```
        (define-inclusion-test public-documentation (ignore)
            (lsb:divert-doc-1 '(pubdoc)))
```

Note: the "-1" suffix on the preceding two routines is vestigial, and is expected to disappear, someday.

If one is doing complicated textual diversions, such as those done by document-routine, the following variables may be hacked by the inclusion test routine:

lsb:*diversion-bindings *Variable*
This is an a-list of variables and values they should be bound to. These bindings are established around the diversion of the text.

lsb:*diversion-setup-forms *Variable*
A list of forms to be evaled before the diversion starts. This is done inside the binding environment specified by lsb:*diversion-bindings.

lsb:*diversion-cleanup-forms *Variable*
A list of forms to be evaled after the diversion finishes. This also is done inside the binding environment specified by lsb:*diversion-bindings.

lsb:*diversion-routine *Variable*
If this is not nil, it is a function to be called with no arguments. It has sole responsibility for reading and diverting the text between matching curly-brackets.

lsb:diversion-tyo *character-code*
Diverts *character-code* to all of the diversion streams currently being diverted to. This is a special case of, and is slightly faster than doing (lsb:diversion-operation ':tyo *character-code*).

lsb:diversion-operation *operation* (Any-number-of *args*)
Sends the *operation* message to all of the diversion streams currently being diverted to. The operations which may be of interest inside a textual diversion are:

:tyo *character-code*
Outputs the single character.

:princ *object*
Does the obvious.

:prin1 *object*
This too.

:terpri
This too.

:fresh-line
Performs a :fresh-line operation on the stream: if the stream is not at the start of a line, then a newline is output.

Other special-purpose operations may be defined for documentation purposes, such as **:start-operation-documentation**; see the discussion on documentation types, below.

12.6 Defining Documentation Types

The squeamish and those prone to heart attacks read this section at their own risk.

Internally, diversion streams use a relatively simple message-passing mechanism. A documentation type has associated with it a function which can field some subset of the messages which get sent to documentation diversion streams; if it does not support some message, the default behaviour (that provided by simple **text** diversion streams) will be obtained. The function gets a first argument of the operation name, a second argument of the output stream, and remaining arguments which depend on the operation. It should thus be prepared to accept any number of arguments. Note that this calling convention is that produced by a function defined with the Lisp Machine **defselect** macro.

Here are some of the messages a documentation-type handler should support:

:which-operations
: The handler should return a list of the operations it supports. That list need not include **:which-operations**, but **:which-operations** *must* be handled. The result of this is cached by the diversion stream for efficiency, so it cannot dynamically change. This operation is special in that the handler may be called on it before the stream has been created, in which case the stream argument will be **nil**.

:start-operation-documentation *data-list*
: We are starting to produce some operation documentation, as gotten from **document-routine**. *data-list* is a list of datastructures which contain the information for each of the operations being documented; see <not-yet-written> for using these datastructures. The handler should output whatever is necessary to start the documentation; for Bolio, this is the **.defun** and **.defun1** lines.

:end-operation-documentation *data-list*
: *data-list* is the same as for **:start-operation-documentation**. The handler should "finish up" the documentation block.

:start-variable-documentation *data-list*
: Like **:start-operation-documentation**, but for variables. The elements of *data-list* are in a different format; see <not-yet-written>.

:end-variable-documentation *data-list*
: Analogous to **:end-operation-documentation**.

More operations like the above may be added in the future.

The messages listed above are exactly those supported by the **bolio** and **tex** documentation types. It is possible, however, for the documentation type handler to have much finer control over the output which is produced. For this, it needs to field most if not all messages which are passed on to diversion streams. These are:

:tyo *character-code*
: The character should be output to the stream. The information on whether the documentation type handler fields this message is cached specially for efficiency.

:princ *object*
The object should be princed to the stream.

:prin1 *object*
The object should be prin1ed to the stream.

:terpri A newline should be output to the stream. Note that newlines in text being diverted do not get converted to terpri operations, but are left as whatever character(s) they were read in as. The documentation producing routines, however, should not generate newline character sequences, but rather use the :terpri operation.

:fresh-line
A newline should be output to the stream iff it is not at the beginning of a line.

:tab-to *destination* (Optional *increment*)
see documentation on the ~T *operator of* format

:close The stream should be "closed". This operation is only used for successful completion.

:kill The stream should be "closed", and aborted. What normally happens here is the stream is closed and the partially written file deleted.

:open *pathname*
The handler can support this if it desires to produce a non-standard I/O stream; normally, open would be called. Note that if the :open message is passed to the handler, no stream has yet been created, so the stream argument to the handler will be nil. The handler should return an output stream; this will be cached by the diversion stream mechanism, and passed in to the handler for all subsequent operations.

If the handler is going to field character-level stuff, i.e. the :tyo operation, it *must* also handle any of the other operations which may produce output (such as :princ and :terpri); the default action in these cases does *not* involve breaking the high-level operation into its components and passing them back to the handler. In Lisp implementations where general I/O streams are available, it is probably better for the handler to field the :open message and return an I/O stream which will then handle the output operations appropriately.

Finally, if one is willing to wade through all of this and needs to associate a function with a documentation type name:

lsb:*documentation-types *Variable*
The value of this is an association list of documentation type names and their handler functions. The documentation type names are compared using LSB keyword equality.

12.7 Macro Memoization

lsb:*macro-memoizers *Variable*
This is an a-list of keywords acceptable to the do-macro-memoizing system definition (and macro declaration) clause. The cdr of each entry is a function which will be called to produce the macro memoization code, and any setup code which must come before the definition. The arguments are the macro name, the name of the variable which will have as its value the original call form, and the form which will need to be evaluated to produce the expansion. The function should return a list of forms. The first form will be used as the body of the macro, and LSB will arrange for the remaining forms to precede

the macro definition itself. The original value of lsb:*macro-memoizers looks like
```
((t . lsb:standard-macmem) (displace . lsb:displace-macmem))
```
and the displace method of macro memoization is defined by
```
(defun lsb:displace-macmem (macro-name
                            original-form-var
                            new-form-form)
  macro-name ; maybe unused
  `((displace ,original-form-form ,new-form-form)
    {(only-for (or PDP-10 NIL))
       ; This is so that any previous form of
       ; macro-memoization will revert so we can undo it.
       (flush-macromemos ',macro-name ())}
    ))
```

13. Variable Binding - LBIND

The LBIND module, although not a part of the default LSB environment, is closely related to it, and quite useful with LSB, both to supplant the use of the bindq and auxiliary-bindings forms in operation definitions, and in places where those forms are not valid. It also is extremely useful for writing implementation independent code, as the data types of the variables may be specified using the standard LSB data type keywords; no implementation dependent declarations are needed. If it is used, the lbind system should be noted in the built-on clause of the system definition.

lbind *Macro*

lbind provides let-like syntax for binding variables. The syntax provides for the specification of data types rather than destructuring. The general syntax is

```
(lbind (bind-spec-1 bind-spec-2 ...)
   form-1
   form-2
   ...)
```

A *bind-spec* is one of the following:

variable

The variable will be bound to nil, and declared to be of type notype.

(*variable value-form*)

The variable will be bound to the value of *value-form*, which is evaluated outside the binding environment of the variables. It is declared to be of type notype.

((*data-type variable*) *value-form*)

The variable will be bound to the value of *value-form*, and declared to be of type *data-type*.

(*variable*)
((*data-type variable*))
((*data-type variable*) nil)

An unspecified *value-form* is seen simply as being a value-form of nil, which causes lbind to determine the initial value from *data-type*. Thus,

```
(lbind (((fixnum foo) nil)) foo)
  => 0
```

because foo gets bound to 0 rather than nil.

It should be noted that a *bind-spec* for lbind has identical syntax to that of the auxiliary-bindings form recognized in operation definition bodies (section 5.2.1, page 28). Thus, the form

```
(lbind (((fixnum foo) (mumble)) ((flonum bar)) baz)
   ...)
   ==>
((lambda (foo bar baz) ...)
 (mumble) 0.0 nil)
```

along with local declarations for the typed variables appropriate to the Lisp implementation.

In many cases, one would like to compute a variable's value as some function of other variables' value. For this, there is the lbind* macro:

lbind* *Macro*

lbind* has syntax identical to lbind. The bindings are nested, however. That is,

```
(lbind (((fixnum foo) (mumble)) ((flonum bar)) baz)
  ...)
 ==>
((lambda (foo)
    ((lambda (bar)
        ((lambda (baz) ...)
         nil))
     0.0))
 (mumble))
```

14. Useful LSB Systems

There are a number of pre-defined systems in LSB which may be of general interest. Many of these are not written using LSB (in fact, LSB may require them to be bootstrapped), but some have LSB-style definition extensions.

14.1 Pre-Defined Systems

Here are some of the systems which are pre-defined (either with define-system or define-system-location) to LSB. This list is incomplete; the full list is in the file ML:LSB;LSBSYS >.

format The Maclisp implementation of the format function is written using LSB. This same source will be used to bring up format in NIL. The Lisp Machine format is a totally different implementation, although the major public definition of it is compatible. format is documented in the Lisp Machine Manual [LMMan], and documentation of the Maclisp version is in preparation.

Brand-X-Triple
Brand X is a low-level extension to Lisp for use in building knowledge bases [BrandX]. It is written using LSB, and runs in PDP-10 Maclisp and Lisp Machine Lisp.

defstruct
defstruct is a structure defining facility which operates compatibly in Maclisp and Lisp Machine Lisp (NIL?). There are LSB definition extensions to defstruct, described later in this chapter.

loop loop is a hairy iteration macro already mentioned [LOOP]. It, like defstruct, has LSB definition extensions which are described later. loop is not written using LSB; being (built-on loop), however, enables a system to not only use loop whether or not loop is accessible by default in the given Lisp implementation, but also gives the system access to the LSB definition extensions.

backquote
sharpsign
These are for the backquote (`) and sharpsign (#) reader macros. These exist as systems solely for the Multics implementation of LSB; they are available in PDP-10 Maclisp, Lisp Machine Lisp, and NIL by default, in which case being built-on either of them is a fast no-op.

Mathlab-Macros
This, like backquote and sharpsign, exists only for the Multics implementation of LSB. Essentially, it comprises all of the various utility files used by the Mathlab group on Multics, which are not covered by some other LSB system definition (such as backquote, sharpsign, and defstruct). It includes such macros as setf, if, push, and pop—most of the things which are obtainable by default in PDP-10 Maclisp, Lisp Machine Lisp, and NIL.

LSB LSB itself has a system definition so that certain internal facilities may be accessed by users if necessary. See chapter 12, page 63.

user-hunk
: A low-level PDP-10 Maclisp interface to hooks in the Lisp for treating hunks as extended-type objects [LSBUtil].

ttyscan
: Fancy parser-driven rubout processing, for PDP-10 Maclisp only. (Only runs on ITS and TOPS-20 systems.) [LSBUtil]

write
: A protocol for performing text output which can be used to build such things as pretty-printers [pp/write].

pretty-print-definition
: A layer built on write for pretty-printing Lisp objects [pp/write].

pretty-print
: A layer built on pretty-print-definition for pretty-printing Lisp code [pp/write].

14.2 LOOP

LSB defines the following extensions to the **loop** iteration macro, for defining iteration paths.

define-public-loop-path *Special Form*
define-system-loop-path *Special Form*
define-private-loop-path *Special Form*
These are all essentially equivalent to define-loop-path, but arrange for the information to be sent to the diversion streams necessary for appropriate compilation and runtime support.

define-public-loop-sequence-path *Special Form*
define-system-loop-sequence-path *Special Form*
define-private-loop-sequence-path *Special Form*
These hack define-loop-sequence-path similarly.

One might thus do the following to define a public loop iteration path:

```
(define-private-routine (parse-my-loop-path ...)
   (dcls (also-needed-for public-compilation))
   ...)
(define-public-loop-path my-loop-path parse-my-loop-path ...)
```

Because of the simplicity of the define-loop-path and define-loop-sequence-path forms, if it is necessary to divert a loop path definition somewhere else, the forms-needed-for special form may be used

```
(forms-needed-for (running public-compilation hacks)
  (define-loop-path ...))
```

14.3 DEFSTRUCT

defstruct is documented in the Lisp Machine Manual, and documentation of the Maclisp version is in preparation. On the ITS systems there is also online documentation in the file LIBDOC;STRUCT >.

define-public-structure *Special Form*
define-system-structure *Special Form*
define-private-structure *Special Form*

These are the LSB variants of defstruct. They will cause the structure definitions to be available at runtime, and in the compilation environment (propagated according to the visibility class). In this respect, structure definition is similar to macro definition.

The arguments to these forms are the same as those to defstruct, with the addition of optional LSB declarations clauses, as in the example

```
(define-system-structure (matrix
                          (named)
                          (default-pointer)
                          (size-macro matrix-structure-size))
     (dcls (reference public))
     matrix-array
     matrix-type
     matrix-ncols
     matrix-nrows)
```

which says that although the matrix structure has a visibility class of system, it will be needed for public compilation. This might be because there is an open-codable-routine or a macro which expands into a reference to the matrix structure. The declarations allowable here are those common to all LSB definitions: the definition availability declarations (e.g., needed-for and reference,), and other similar ones like divert-documentation-to.

define-public-xstructure *Special Form*
define-system-xstructure *Special Form*
define-private-xstructure *Special Form*

These are just like the corresponding define-*visclass*-structure forms, but divert the structure definition the way the define-*visclass*-xmacro forms do (section 5.5.1, page 34). That is, when compiled, the definition will be sent to the macros diversion stream instead of the compiled output file.

15. An Example System

This is an example system written using LSB. Two modules are included here, with some deletions for brevity.

15.1 The System Definition

This is the contents of the system definition file for the **stats** system, which is in the file ML:STATS;STATS SYSTEM. The file ML:STATS;STATS PKG contains the package definition for the **statistics** package, which is not particularly interesting. The source code is assuming only that it will run in Lisp Machine Lisp or Maclisp. Note that the package refname **stats** is equivalent to **statistics**.

```
      (define-system stats
          {(only-for Lispm)
             (host mc)
             (device ml)
             }
          {(except-for Maclisp)
             (package statistics)
             }
          (directory stats)
          (built-on loop)
          (type-check-visibility-classes public)
          (modules interpolate ttable ctable ftable normal))
```

15.2 The INTERPOLATE Module

This is the contents of the interpolate module, which is in the file ML:STATS;INTERP >.

```
{-*-  Mode:Lisp;  LSB:interpolate,stats  -*-   26-Jun-81

   Copyright (c) 1981 by Grandiose System Building
   and Massachusetts Institute of Technology.  All rights reserved.}

(module interpolate stats)

{(system-documentation)
        The ↑3interpolate↑* module defines common routines to allow
trivial definition of two-parameter statistical functions.
}
```

```
(define-system-xmacro (make-stat-table dimension-list init-list)
    (and (atom dimension-list)
         (setq dimension-list (list dimension-list)))
    (bindq (fixnum implied-size) (length init-list)
           (fixnum actual-size) (apply '* dimension-list))
    (and (not (= implied-size actual-size))
         (error (if (< implied-size actual-size)
                    '|Initialization list has too few entries|
                    '|Initialization list has too many entries|)
                (list 'make-stat-table dimension-list)))
    {(only-for Maclisp)
       '(fillarray (*array nil 'flonum . ,dimension-list) ',init-list)
       }
    {(only-for Lispm)
       '(fillarray (make-array ',dimension-list ':type 'art-q)
                   ',init-list)
       }
    )

{(document-routine)
.lisp
(make-stat-table ε2dimension-listε* ε2init-listε*)
.end_lisp
expands into a form which will create a table (implemented as an
array) which will contain flonum components and be initialized with
the elements from ε2init-listε*.  ε2init-listε* is required to
contain exactly the number of elements required to fill the table.
The array will be of type ε3flonumε* in Maclisp, but a normal array
(ε3art-qε*) on the Lisp Machine so that accessing does not do
additional number consing.
}

(define-system-xmacro (stats:tabref table i j)
  {(only-for Maclisp) '(arraycall flonum ,table ,i ,j)}
  {(only-for Lispm) '(aref ,table ,i ,j)})

{(document-routine)
.lisp
(stats:tabref ε2tableε* ε2iε* ε2jε*)
.end_lisp
accesses a two-dimensional table created by ε3make-stat-tableε*.
}
```

```
(define-system-routine (table-interpolate
                          (fixnum x1) (flonum v1)
                          (fixnum x2) (flonum v2)
                          (fixnum xprime))
     (declarations (value-type flonum))
     (+$ v1 (*$ (//$ (-$ v2 v1) (float (- x2 x1)))
                (float (- xprime x1)))))

{(document-routine)
This calculates the value corresponding to ε2xprimeε* by linear
interpolation, given ε2x1ε* and ε2v1ε*, and ε2x2ε* and ε2v2ε*.
}

(define-system-routine (table-2dim-lookup
                          (fixnum n) (flonum cf) cf-list basis-size
                          more-n-values infinity-index? table who)
     (dcls (value-type flonum))
     ...
     )

{(document-routine)
This implements the basic two-parameter lookup.  ε2nε* is the number
etc
}
```

15.3 The TTABLE Module

```
{-*-  Mode:Lisp;  LSB:ttable,stats  -*-          9-Jun-81

   Copyright (c) 1981 by Grandiose System Building
   and Massachusetts Institute of Technology.  All rights reserved.}

(module ttable stats)
```

```
(define-system-variable *basic-T-table
   (init (make-stat-table (34. 8.)
  ;; 60%  75%     90%    95%     97.5%    99%     99.5%   99.95%
  (0.325  1.0     3.078  6.314  12.706   31.821  63.657  636.619  ;n=1
   0.289  0.816   1.886  2.920   4.303    6.965   9.925   31.598  ;n=2
   0.277  0.765   1.638  2.353   3.182    4.541   5.841   12.924  ;n=3
   etc. etc.
   0.256  0.683   1.310  1.697   2.042    2.457   2.750    3.646  ;n=30
   0.255  0.681   1.303  1.684   2.021    2.423   2.704    3.551  ;n=40
   0.254  0.679   1.296  1.671   2.000    2.390   2.660    3.460  ;n=60
   0.254  0.677   1.289  1.658   1.980    2.358   2.617    3.373  ;n=120
   0.253  0.674   1.282  1.645   1.960    2.326   2.576    3.291  ;n=inf
   ))))

(define-public-routine (stats:t-table (fixnum n) (flonum cf))
   (declarations (value-type flonum))
   (table-2dim-lookup
      n cf '(6000. 7500. 9000. 9500. 9750. 9900. 9950. 9995.)
      30. '(40. 60. 120.) 300. *basic-T-table 'stats:t-table))

{(document-routine)
This implements a T-distribution lookup for degrees-of-freedom ε2nε*
and confidence-factor ε2cfε*.  ε2cfε* may be either a fraction or
a percentage;  they can be distinguished because the former must be
less than ε31.0ε*.  It may range from 60% to 99.95%.
}
```

15.4 The Documentation Produced

Here is what the system documentation for the interpolate module looks like when formatted.

The interpolate module defines common routines to allow trivial definition of two-parameter statistical functions.

make-stat-table *Macro*

`(make-stat-table` *dimension-list init-list*`)`

expands into a form which will create a table (implemented as an array) which will contain flonum components and be initialized with the elements from *init-list*. *init-list* is required to contain exactly the number of elements required to fill the table. The array will be of type flonum in Maclisp, but a normal array (art-q) on the Lisp Machine so that accessing does not do additional number consing.

stats:tabref *Macro*

`(stats:tabref` *table i j*`)`

accesses a two-dimensional table created by make-stat-table.

table-interpolate (fixnum *x1*) (flonum *v1*) (fixnum *x2*) (flonum *v2*) (fixnum *xprime*)
This calculates the value corresponding to *xprime* by linear interpolation, given *x1* and *v1*, and *x2* and *v2*.

table-2dim-lookup (fixnum *n*) (flonum *cf*) *cf-list basis-size more-n-values infinity-index? table who*
This implements the basic two-parameter lookup. *n* is the number of degrees of freedom desired. *cf* is the confidence-factor; if it is less than `1.0` then it is assumed to be a fraction, otherwise it is assumed to be a percentage. *table* should be a two-dimensional table created by `make-stat-table`. The first dimension indexes different *n* values, and the second different *cf* values. It is assumed that some low range of *n* values are complete, and are in the (1 - *n*)th components of the table. *basis-size* is the greatest *n* for which these contiguous entries exist. Other *n* values may be sparse; *more-n-values* is a list of the other *n* values. If *infinity-index?* is not `nil`, then it should be the value of *n* which is considered to approximate infinity; in this case, *table* should have one additional *n* row (not accounted for by *basis-size* and *more-n-values*) which contains the *cf* values for *n* = infinity. Values of *n* below *infinity-index?* will be linearly interpolated. *who* is simply used for generating errors, and should be the name of the caller. For example: if we have data points for *n* from 1-30, 40, 60, 120, and infinity, then the first dimension of table should be of size `34`, *basis-size* should be `30`, and *more-n-values* should be `(40 60 120)`. *infinity-index?* should be an *n* value for which the data are not significantly different from those for infinity. If this table contains data for confidence-factors of 60%, 75% 90%, 95%, 97.5%, 99%, 99.5%, and 99.95%, then *cf-list* should be `(6000 7500 9000 9500 9750 9900 9950 9995)`. These parameters are in fact those used by the `stats:t-table` function, q.v.

This routine always returns a flonum, obtained by linear interpolation from the points surrounding the desired point.

16. Table of Abbreviations

Here are all of the predefined keyword synonyms defined. Reasonable suggestions for additions are solicited.

also-divert-reinitialization-to	also-divert-reinit-to
argument-sequence	argseq, arg-sequence, arg-seq
auxiliary-bindings	auxs, aux-bindings
built-along-side-of	built-with, built-beside
declaration-diversion-stream	dcl-divstream
declarations	dcls
default-initialization	default-init
device	dev
directory	dir
diversion-stream	divstream
divert-documentation-to	divdoc, divert-doc-to
divert-reinitialization-to	divert-reinit-to
document-routines	document-routine
document-variables	document-variable
documentation-diversion-stream	doc-divstream
files-needed-for-compilation	additional-files-needed
form-diversion-stream	form-divstream
identifier	id
initialization	init
number-check-visibility-classes	number-check-visibility-class
pathname	filename
private-documentation	module-documentation

quoted	unquoted
referenced-at-visibility-class	reference
system	intrasystem
system-compilation	intrasystem-compilation
system-documentation	intrasystem-documentation
textual-diversion-stream	text-diversion-stream, text-divstream
type-check-arguments	type-check-argument, check-arg, check-args
type-check-visibility-classes	type-check-visibility-class
value-type	data-type

References

[BrandX]
Szolovits, Peter, and Martin, William A., *Brand X Manual*, MIT Laboratory for Computer Science Technical Memo 186 (November 1980).

[LMMan]
Moon, David A., and Weinreb, Daniel L., *Lisp Machine Manual*, MIT Artificial Intelligence Laboratory publication, March 1981.

[LOOP]
Burke, Glenn S., and Moon, David A., *LOOP Iteration Macro*, MIT Laboratory for Computer Science Technical Memo 169 (July 1980, revised January 1981.) LOOP is also documented in the March 1981 version of the Lisp Machine Manual.

[LSBUtil]
Burke, Glenn S., et al., *LSB Utilities Reference*, MIT Laboratory for Computer Science Technical Memo (in preparation). Documentation on various independent facilities which either extend or are convenient to use with LSB.

[Moonual]
Moon, David A., *Maclisp Reference Manual*, MIT Lab. for Comp. Sci., Cambridge, Mass. (1974). Out of print; updated chapters may be available, revision in preparation.

[NILDoc]
Unwritten documentation on NIL.

[pp/write]
Documentation on the write system, by Lowell Hawkinson, in preparation.
Documentation on the pretty-print-definition extension to the write facility, by Glenn Burke, in preparation.

Index of Tables

Index