Portable Utilities for Common Lisp
User Guide & Implementation Notes

Mark Kantrowitz

May 1991

CMU-CS-91-143

School of Computer Science
Cargnegie Mellon University
Pittsburgh, PA 15213-3890
mkant+@cs.cmu.edu

Abstract

One of the most frequent complaints in the Lisp community is the lack of availability of programming tools. This document describes portable implementations of six tools for the development and maintenance of Common Lisp programs: XREF, a Lisp code cross referencer; METERING, a timing and consing code profiler; DEFSYSTEM, a "make" for Lisp; LOGICAL-PATHNAMES, portable pathnames for Lisp; SOURCE-COMPARE, a "diff" for Lisp; and USER-MANUAL, a program which extracts documentation from Lisp programs. All six tools are publicly available via anonymous ftp.

This research was sponsored in part by a Hertz Foundation Research Fellowship Grant. The author is supported by a National Science Foundation (NSF) Graduate Fellowship.

Keywords: programming environments, Common Lisp, program maintenance, programming tools, cross referencing, code profiling, source comparison, system definition, logical pathnames.

1. Introduction

This document describes six portable systems tools which aid programmers in the development and maintenance of Common Lisp programs.

1.1. Why Portable Utilities?

One of the most frequent complaints in the Lisp community is the lack of general availability of tools for system development and management. While some tools are available in particular Lisp environments (most notably on dedicated Lisp Machines, such as Symbolics and Xerox), none are available in every version of Lisp, and those that are available are often incompatible.

For example, many Lisps provide some sort of system definition tool, commonly called ‘defsystem’, but most such tools are incompatible. Some Lisps use simple modules [6, pp. 188-192], perhaps augmented with search lists, while others use a more complicated procedural system-construction tool [3, 4]. Even those with similar functionality have a different definition syntax. Since the tools are often proprietary, one is constrained either to using a particular Lisp or to writing separate system definitions for each and every tool. With today’s heterogeneous programming environments, programmers often use different Lisps on different machines, or even on the same machine, so the former is not a viable option and the latter is a major headache for the program maintainer.

A primary goal of this manual and associated software is to address this issue by providing portable implementations of the most useful utilities. All of the utilities are implemented in Common Lisp[1] and any Lisp implementation-dependent changes are clearly noted, with reasonable defaults if the changes not supplied. Porting the tools to other Lisps is therefore quite painless. Since the tools are publicly available for no fee, one may simply use the tools in whichever Lisps one desires.

1.2. Design Philosophy

Although system development tools can greatly improve programmer productivity, not every programmer has the time and opportunity to write such tools from scratch. On the other hand, given source code for a tool that is close to what they want, most programmers can quickly and easily modify it to meet their needs. Likewise, the programmer who encounters a bug in the tool can fix it himself.

If the incremental enhancements and changes made by the users are then propagated back to the

[1] The utilities have all been tested in Franz Allegro Common Lisp (3.0.1 Decstation 3100), Macintosh Allegro Common Lisp (1.3.2), and CMU Common Lisp (old and new compilers). In addition, many of the utilities have been tested in other Lisps, including Lucid Common Lisp (2.1 Vax, 3.0, and 4.0), Symbolics Common Lisp (7.2 and 8.0), Ibuki Common Lisp, and VAXLisp.

original source, the improvements can snowball, yielding a better program than any individual programmer or team of programmers could have written. This is especially true of programs with a large community of users, such as programming tools. The key is to get the ball rolling.

Thus our philosophy in designing and implementing these utilities has been to concentrate on basic functionality, and rely on the users to add the bells and whistles they want. To provide a good starting point for new features, the program must be written in as generic and portable a manner as possible. Accordingly, we wrote these utilities in "pure" Common Lisp, segregating any implementation-dependent functions, and focused on providing a clean and efficient implementation of the core of the programs.

Even though most of the tools have been available for much less than a year, they have already benefited from this approach. Users have helped port the utilities to other Lisps (often by providing just the implementation-dependent functions), fixed bugs, added features, and made suggestions for other improvements. The tools have become quite popular, and are currently being used by programmers at over 100 sites that we know of.

1.3. Overview

In the following chapters we describe each of the following utilities in detail:

XREF A portable cross referencing tool for determining callers of functions and variables in Lisp programs. Useful for mapping out the structure of a program. Similar to the Symbolics Who-Calls database [3] [8, pp. 183-185] and the Xerox Masterscope program [2]. Includes an interface to Joe Bates' PostScript DAG grapher for drawing call graphs.

METERING
A portable code profiling tool, for gathering timing and consing statistics while a program is running. Monitors the use of functions and macros, calculating the number of calls, CPU time, and storage use. Inclusive and exclusive function call statistics. The METERING system is based on the MONITOR program written by Chris McConnell and the PROFILE program written by Skef Wholey and Rob MacLachlan, with several extensions.

DEFSYSTEM
A portable system definition facility (a "make" for Lisp), similar to the Symbolics system construction tool [3] [4]. Compiles and loads files according to a user-defined file-dependency graph, while trying to minimize extraneous compilations and loads. Includes an interface to LOGICAL-PATHNAMES. XREF includes a tool to assist in building a system definition for a set of files.

LOGICAL-PATHNAMES
A portable implementation of the X3J13 June 1989 specification for logical pathnames, as documented in [7, section 23.1.5]. Useful for portable pathname reference, cross-host access, and pathname aliasing.

SOURCE-COMPARE
A portable tool for finding the differences between source files (a "diff" for Lisp). While it may be used to compare arbitrary text files, it has several features specialized for Lisp, such as the ability to ignore Lisp comments. It uses a greedy algorithm for longest common substring that may not necessarily find the longest common substring, but which

runs in average case linear time and works well in practice.

USER-MANUAL
: A portable program for extracting documentation from Lisp source code. Helps create user guides and program documentation.

This manual describes only the programming utilities. The collection also includes other useful Lisp programs, such as a regular-expression style matcher and FrameWork, a generic frame-based knowledge representation system, as well as a variety of text files of interest to Lisp programmers.

Within each chapter we first give an overview of the basic features of the utility, including basic commands and variables. Next, we describe how to load the software, with a particular emphasis on what implementation-dependent changes may be required. Then come usage notes, if any. After that follows a few examples of how to use the programs and sample output. Finally, we conclude each chapter with a discussion of the implementation, which should help users modify and extend the software.

1.4. Obtaining the Utilities

The utilities are available by anonymous ftp from CMU:

- `ftp` to `a.gp.cs.cmu.edu` [128.2.242.7] or any other CMU CS machine.
- The directory `/afs/cs.cmu.edu/user/mkant/Public/Lisp-Utilities/` contains the files.
- `cd` to this directory in one fell swoop. Do not try to `cd` or `ls` any intermediate directories, since the CMU security mechanisms prevent access to other directories from an anonymous ftp.
- Use `ls` to see what files are available. For users accessing the directory via an anonymous ftp mail server, the file `README` contains a current listing and description of the files in the directory. The file `UPDATES` describes recent updates to the released versions of the software in the directory. The file `COPYING` describes the general license agreement and lack of warranty.

Of course, if your site runs the Andrew File System[2] and you have afs access, you can just `cd` to the directory and copy the files directly.

The following is an example of using ftp to retrieve the software:

```
% ftp a.gp.cs.cmu.edu
Connected to A.GP.CS.CMU.EDU.
```

---

[2]Currently Boston University, Carnegie Mellon University, Chalmers University of Technology, Dartmouth, HP Cupertino, Idaho National Engineering Lab, MIT, Mt. Xinu, Naval Research Lab, NIH, Open Software Foundation, Pittsburgh Supercomputing Center, Rensselaer Polytechnic Institute, Stanford, Superconducting Supercollider Lab, Transarc, Unisys, University of Arizona, University of Michigan, University of Notre Dame, University of Pittsburgh, and University of Southern California/ISI.

```
220 A.GP.CS.CMU.EDU FTP server (Version 4.105 of 10-Jul-90 12:07) ready.
Name (a.gp.cs.cmu.edu:mkant): anonymous
331 Guest login ok, send ident as password.
Password:
230 Filenames can not have '/..' in them.
ftp> cd /afs/cs.cmu.edu/user/mkant/Public/Lisp-Utilities
250 Directory path set to /afs/cs.cmu.edu/user/mkant/Public/Lisp-Utilities.
ftp> ls
200 PORT command successful.
150 Opening data connection for ls (128.2.220.10,3107).
COPYING
README
UPDATES
c-lisp-interfaces.text
cl-x-lisp-interfaces.text
defsystem.lisp
   [...rest of listing deleted...]
226 Transfer complete.
430 bytes received in 0.23 seconds (1.8 Kbytes/s)
```

The following table lists the relevant files, their length in lines of Lisp (excluding comments), and their size in bytes:

File	lines	bytes
defsystem.lisp	1391	88k
framework.lisp	1666	125k
logical-pathnames.lisp	1511	77k
matcher.lisp	113	15k
metering.lisp	714	45k
psgraph.lisp	457	18k
psgraph.doc		5k
source-compare.lisp	640	54k
user-manual.lisp	500	35k
xref.lisp	2109	125k
xref-patterns-for-macl.lisp	76	3k
xref-test.lisp	92	2k
Total	9269	592k

There is a mailing list for notification of major updates, bug-fixes and additions to the Lisp Utilities collection. To be added to the mailing list, send email with your name, email address, and affiliation to `CL-Utilities-Request@cs.cmu.edu`.

Bug reports, comments, questions and suggestions should be sent to `mkant+@cs.cmu.edu`. Also, please send us copies of any changes or improvements you make to the software, so that we may merge them into the originals.

1.5. Acknowledgments

Many users of the tools have contributed enhancements, bug fixes, suggestions and detailed bug reports. I would especially like to thank Anton Beschta, Sean Boisen, Michael Brent, Steve Chanin, Daniel J. Clancy, Matthew Cornell, Rodney Daughtrey, David A. Duff, Ute Gappa, Gabriel Inaebnit, Dick Jackson, Bradford W. Miller, Karsten Poeck, Jean-Francois Rit, William D. Smith, Ralph P. Sobek, Steve Strassmann and Rick Taube.

The METERING code profiler is derived directly from the work of Chris McConnell on the MONITOR program and the work of Skef Wholey and Rob MacLachlan on the PROFILE program. Many thanks to Chris McConnell and Rob MacLachlan for comments on the result of merging and extending their two programs.

Thanks to Neil J. Calkin for the idea that led to the proof that SOURCE-COMPARE runs in average case linear time.

I am grateful to Peter Lee for supervising this work towards a minor in programming systems.

Finally, I would like to thank my advisor, Joe Bates, for allowing me to become occasionally "distracted" from my research to work on the utilities during the past year.

2. XREF: Cross Referencer

The XREF or List Callers system is a portable Common Lisp cross referencing tool. It grovels over a set of files and compiles a database of the locations of all references to each symbol used in the files. It is similar to the Symbolics Who-Calls and Xerox Masterscope facilities [2] [3] [8].

When you change a function or variable definition, it can be useful to know its callers, in order to update each of them to match the new definition. Similarly, a graphical display of the structure of a program can help make undocumented code more understandable. This code analyzer implements both capabilities.

The database compiled by XREF is suitable for viewing by a graphical browser. Since the call graph is not necessarily a DAG, and many graphical browsers assume a DAG, XREF includes code to convert the graph to a tree-like representation. XREF also includes a simple text-indenting outliner for displaying call graphs on ascii terminals, as well as an interface to Joe Bates' PSGRAPH PostScript DAG grapher.

2.1. Overview

XREF analyzes a user's program, determining which functions call a given function and the locations where variables are bound/assigned and used. The user may retrieve this information for a single symbol, or display the call graph of portions of the program (up to and including the entire program). This helps the programmer debug and document the program's structure.

XREF is primarily intended for analyzing large programs, for which it is difficult, if not impossible, for the programmer to grasp the structure of the whole program. Nothing precludes using XREF for smaller programs, however, where it can be useful for inspecting the relationships between pieces of the program and for documenting the program.

Two aspects of the Lisp programming language greatly simplify the analysis of Lisp programs:[3]

- The syntax of Lisp programs and data are the same. Successive definitions from a file may be read in as list structure.
- The basic syntax of Lisp is uniform. A Lisp program consists of a set of nested forms, where each form is a list whose car is a tag (e.g., function name) that specifies the structure of the rest of the form.

Thus Lisp programs, when read as data, can be thought of as parse trees.[4] Given a grammar of syntax patterns for the language, XREF recursively descends the parse tree for a given definition, computing a set of relations that hold for the definition at each node in the tree. For example, a

[3]Of course, macros and eval complicate the analysis of Lisp programs.

[4]While XREF currently works only for programs written in Lisp, it could be extended to other programming languages by writing a function to generate parse trees for definitions in that language, and a core set of patterns for the language's syntax.

typical relation is that the functions in the body of a definition are called by the defined function. The relations are stored in a database for later inspection by the user.

XREF may operate in either a static or a dynamic mode. In the static mode XREF does a static syntactic analysis of the program, but does not detect references due to the expansion of a macro definition. In the dynamic mode XREF will expand any macros for which it does not have predefined patterns.

The dynamic analysis of a program requires XREF to have some knowledge about the semantics of the program. For example, a macro could call functions defined by the program to do the expansion. This entails either modifying the compiler to record the relationships (e.g., Symbolics Who-Calls Database) or doing a walk of loaded code and macroexpanding as needed (PCL code walker). Since the former is not portable, XREF implements the latter.

In order for XREF to expand macros the code used by the macros must be loaded and in working order. Also, XREF's parameters probably should be set so that it processes forms in their proper packages. If the code is not loaded, XREF will default to operating in the static analysis mode. When XREF operates in dynamic mode it doesn't need any special knowledge about the syntax of macros (excluding the 24 special forms of Lisp). On the other hand, to operate properly in static analysis mode XREF must have patterns defined for all the standard macros of Common Lisp. Thus, even though most Lisps implement `dolist` as a macro, XREF will not call `macroexpand-1` on a form whose car is `dolist` because it will use the predefined template for `dolist` instead.

If macro expansion is disabled, the default rules for handling macro references may not be sufficient for some user-defined macros, because macros allow a variety of non-standard syntactic extensions to the language. In this case, the user may specify additional templates in a manner similar to that in which the core Lisp grammar was specified.

2.2. Loading XREF

XREF runs best when compiled and will issue a warning if the source is loaded instead. It also loads much faster when compiled. To use, load the compiled version of `xref.lisp` and any additional patterns, such as `xref-patterns-for-macl.lisp`. XREF is loaded into the "XREF" package, so prefix all the following functions and variables with an "XREF:".

2.3. Using XREF

This section describes all of the basic XREF commands and the variables which control their behavior. XREF includes functions for creating the reference database, saving the database to file, restoring a saved database, and retrieving information from the database in a variety of formats.

2.3.1. Creating, Saving and Restoring the Reference Database

`xref-files` and `xref-file` are the main functions for creating the reference database. For very large systems of files it can take several minutes to process the code, so writing the database to file may save some time. `write-callers-database-to-file` may be used to save the database to a file, which may then be loaded using `load` to restore the database.

`xref-files` (&rest files) [Function]

> Grovels over the Lisp code located in the specified source files *files*, using `xref-file`.

`xref-file` (filename &optional (clear-tables t) (verbose *xref-verbose*)) [Function]

> Cross references the function and variable calls in *filename* by walking over the source code located in the file. Defaults type of filename to `"lisp"`. If *clear-tables* is `t` (the default), it clears the callers database before processing the file. Specify *clear-tables* as `nil` to append to the database. If *verbose* is `t` (the default), prints out the name of the file, one progress dot for each form processed, and the total number of forms.

`write-callers-database-to-file` (filename) [Function]

> Saves the contents of the current callers database to a file. This file can be loaded to restore the previous contents of the database.

2.3.2. Examining Symbol References

The following functions display information about the uses of the specified symbol as a function or variable.

`list-callers` (symbol) [Function]

> Lists all functions which call *symbol* as a function (function invocation).

`list-readers` (symbol) [Function]

> Lists all functions which refer to *symbol* as a variable (variable reference).

`list-setters` (symbol) [Function]

> Lists all functions which bind/set *symbol* as a variable (variable assignment).

`list-users` (symbol) [Function]

> Lists all functions which use *symbol* as a variable or function.

`who-calls` (symbol &optional how) [Function]

> Lists callers of symbol. *how* may be `:function`, `:reader`, `:setter`, or `:variable`.

`what-files-call` (symbol) [Function]

> Lists names of files that contain uses of *symbol* as a function, variable, or constant.

`source-file` (symbol) [Function]

> Lists the names of files in which *symbol* is defined and/or used.

`list-callees` (symbol) [Function]

Lists names of functions and variables called by *symbol*.

2.3.3. Viewing and Graphing the Reference Database

The following functions are useful for viewing the database and displaying it in a variety of formats.

`display-database` (&optional (database :callers) (types-to-ignore *types-to-ignore*)) [Function]

Prints the name of each symbol and a list of all its callers. Specify *database* as `:callers` (the default) to get function call references, as `:file` to the get files in which the symbol is called, as `:readers` to get variable references, and as `:setters` to get variable binding and assignments. Ignores functions of the types listed in *types-to-ignore*.

`print-caller-trees` (&key (mode *default-graphing-mode*) (types-to-ignore *types-to-ignore*) compact root-nodes) [Function]

Prints the calling trees (which may actually be a full graph and not necessarily a DAG) as indented text trees using `print-indented-tree`. *mode* is `:call-graph` for trees where the children of a node are the functions called by the node, or `:caller-graph` for trees where the children of a node are the functions the node calls. *types-to-ignore* is a list of funcall types (as specified in the patterns) to ignore in printing out the database. For example, `'(:lisp)` would ignore all calls to Common Lisp functions. *compact* is a flag to tell the program to try to compact the trees a bit by not printing trees if they have already been seen. *root-nodes* is a list of root nodes of trees to display. If *root-nodes* is `nil`, displays trees for all the root nodes in the database.

`print-file-dependencies` (&optional (database *callers-database*)) [Function]

Prints a list of file dependencies for the references listed in *database*. This function may be useful for automatically computing file loading constraints for a system definition tool such as defsystem.

The PSGRAPH program (`psgraph.lisp`) must be loaded before using `psgraph-xref`.

`psgraph-xref` (&key (mode *default-graphing-mode*) (output-directory *postscript-output-directory*) (types-to-ignore *types-to-ignore*) (compact t) (shrink t) root-nodes) [Function]

Creates a postscript file for each call-graph in the database. If *shrink* is `t`, shrinks the output to fit on a single page. If *compact* is `t`, will print the tree rooted at a given node only once. *mode* may be `:call-graph` to display the call-graph, `:caller-graph` to display the inverse. If *root-nodes* is `nil`, it tries to find all the root nodes in the database (functions not called by other functions) and display those. Otherwise, *root-nodes* should be a list of root nodes of the trees to be displayed.

2.3.4. XREF Variables

The following variables control the default operation of XREF.

`*xref-verbose*` t [Variable]

When `t`, `xref-file` prints out the names of the files it looks at, progress dots, and the number of forms read.

`*types-to-ignore*` (quote (:lisp :lisp2)) [Variable]

Default set of caller types (as specified in the patterns) to ignore in the database handling functions. `:lisp` is CLtL 1st edition [6], `:lisp2` is additional patterns from CLtL 2nd edition [7].

`*handle-package-forms*` () [Variable]

When non-`nil` and `xref-file` encounters a package-setting form like `in-package`, the form is evaluated to set the current package to the specified package. When done with the file, `xref-file` resets the package to its original value. In some of the displaying functions, when this variable is non-`nil` one may specify that all symbols from a particular set of packages be ignored. This is only useful if the files use different packages with conflicting names.

`*handle-function-forms*` t [Variable]

When `t`, `xref-file` tries to be smart about forms which occur in a function position, such as lambdas and arbitrary Lisp forms. If so, it recursively calls `record-callers` with pattern `'form`. If the form is a lambda, the name `:unnamed-lambda` is used in the database.

`*handle-macro-forms*` t [Variable]

When `t`, if the file was loaded before being processed by XREF, and the car of a form is a macro, it notes that the parent calls the macro, and then calls `macroexpand-1` on the form.

`*default-graphing-mode*` :call-graph [Variable]

Specifies whether we graph up or down. If `:call-graph`, the children of a node are the functions it calls. If `:caller-graph`, the children of a node are the functions that call it.

`*indent-amount*` 3 [Variable]

Number of spaces to indent successive levels in `print-indented-tree`.

2.4. An Example of Using XREF

In this section we give some examples of using XREF to analyze `xref-test.lisp`, a simple nonsense program. The program is listed in Appendix I and tests several aspects of XREF.

Assuming XREF is already loaded, we must first analyze the forms in the file using xref-file:

```
<cl> (xref:xref-file "xref-test.lisp")
Cross-referencing file xref-test.lisp.
......
6 forms processed.
```

If we wish to see which functions call `frowz`, we use the list-callers function:

```
<cl> (xref:list-callers ’frowz)
(BARF TOP-LEVEL)
```

The function print-caller-trees may be used to print the call graph using indentation to show levels:

```
<cl> (xref:print-caller-trees)
Rooted calling trees:
  TOP-LEVEL
     FROB
        FROB-ITEM
           APPEND-FROBS
     BARF
        FROWZ
           PROCESS-KEYS
           SNARF-ITEM
           PROCESS-KEY
              SYMBOL-NAME-KEY
           NODE-POSITION
     FROWZ
        PROCESS-KEYS
        SNARF-ITEM
        PROCESS-KEY
           SYMBOL-NAME-KEY
        NODE-POSITION
```

Note how the tree rooted at `frowz` is repeated, once for each place it occurs. We can eliminate this duplication using the `:compact` keyword:

```
<cl> (xref:print-caller-trees :compact t)
Rooted calling trees:
  TOP-LEVEL
     FROB
        FROB-ITEM
           APPEND-FROBS
     BARF
        FROWZ
     FROWZ
        PROCESS-KEYS
        SNARF-ITEM
        PROCESS-KEY
           SYMBOL-NAME-KEY
        NODE-POSITION
```

This time the tree is printed only once, and only the symbol `frowz` is repeated.

A PostScript version of the call graph may be created using the `psgraph-xref` interface from XREF to PSGRAPH. To use this interface, load PSGRAPH and evaluate the definition of `psgraph-xref` which is commented out in `xref.lisp`. Running `psgraph-xref` then creates a separate PostScript file for each root of a call graph in the database. The file `xref-test.lisp` has only one root, the function `top-level`:

```
* (xref:psgraph-xref)
Creating PostScript file "top-level.ps".
```

Figure 2-1 shows what this graph looks like.

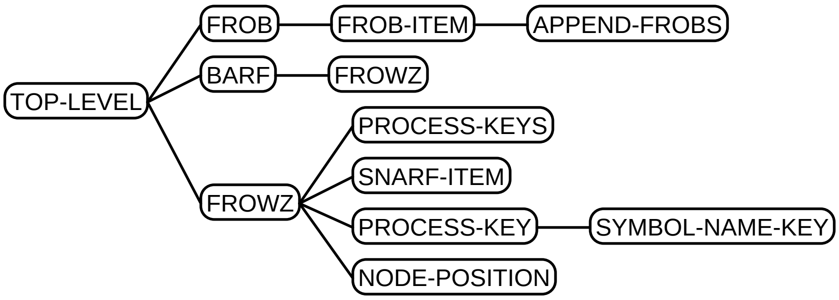

Figure 2-1: Sample PostScript Call Graph

2.5. Extending XREF

As noted in Section 2.1, XREF works by considering the Lisp forms to be parse trees, and matching the parse trees against a grammar for the language. The following macros define new function and macro call patterns. They may be used to extend XREF to handle new definition forms and extensions to Common Lisp.

`define-pattern-substitution` (name pattern) [Macro]

> Defines *name* to be equivalent to the specified pattern. Useful for making patterns more readable. For example, the `lambda-list` pattern is defined as a pattern substitution, thereby making the definition of the `defun` caller-pattern simpler.

`define-caller-pattern` (name pattern &optional caller-type) [Macro]

> Defines *name* as a function/macro call with argument structure described by *pattern*. *caller-type*, if specified, assigns a type to the pattern, which may be used to exclude references to *name* while viewing the database. For example, all the Common Lisp definitions have a caller-type of `:lisp` or `:lisp2`, so that you can exclude references to common Lisp functions from the calling tree.

`define-variable-pattern` (name &optional caller-type) [Macro]

> Defines *name* as a variable reference of type *caller-type*. This is mainly used to establish the caller-type of the variable.

`define-caller-pattern-synonyms` (source destinations) [Macro]

> For defining function caller pattern syntax synonyms. For each name in *destinations*, defines its pattern as a copy of the definition of *source*. Allows a large number of identical patterns to be defined simultaneously. Must occur after the *source* pattern has been defined.

XREF includes pattern definitions for the latest Common Lisp specification, as published in [7].

Patterns may be either structures to match, or a predicate like `#'numberp`. The pattern specification language is similar to the notation used in [7], but in a more Lisp-like format:

(:eq name)	The form element must be eq to the symbol name.
(:test test)	test must be true when applied to the form element.
(:typep type)	The form element must be of type type.
(:or pat1 pat2 ...)	Tries each of the patterns in left-to-right order, until one succeeds. Equivalent to { pat1 | pat2 | ... }.
(:rest pattern)	The remaining form elements are grouped into a list which is matched against pattern.
(:optional pat1 ...)	The patterns may optionally match against the form element. Equivalent to [pat1 ...].
(:star pat1 ...)	The patterns may match against the patterns any number of times, including zero. Equivalent to { pat1 ... }*.
(:plus pat1 ...)	The patterns may match against the patterns any number of times, but at least once. Equivalent to { pat1 ... }+.
&optional, &key, &rest	Similar in behavior to the corresponding lambda-list keywords.
FORM	A random Lisp form. If a cons, assumes the car is a function or macro and tries to match the args against that symbol's pattern. If a symbol, assumes it's a variable reference.
:IGNORE	Ignores the corresponding form element.
NAME	The corresponding form element should be the name of a new definition (e.g., the first arg in a defun pattern is name).
FUNCTION, MACRO	The corresponding form element should be a function reference not handled by form. Used in the definition of the pattern fn which is used in defining the patterns for apply and funcall.
VAR	The corresponding form element should be a variable definition or mutation. Used in the definition of let, let*, etc.
VARIABLE	The corresponding form element should be a variable reference.

In all other pattern symbols, it looks up the symbol's pattern substitution and recursively matches against the pattern. It will automatically destructure list structure that does not include consing dots.

Among the predefined pattern substitution names are:

STRING, SYMBOL, NUMBER	Appropriate :test patterns.
LAMBDA-LIST	Matches against a lambda list.
BODY	Matches against a function body definition.
FN	Matches against #'<function>, '<function>, and lambdas. This is used in the definition of apply, funcall, and the mapping patterns.

See `xref.lisp` for others.

Here are some sample pattern definitions which illustrate the use of `define-caller-pattern`:

```
(define-caller-pattern defun
  (name lambda-list
    (:star (:or documentation-string declaration))
    (:star form))
 :lisp)

(define-caller-pattern funcall (fn (:star form)) :lisp)
```

In general, XREF is intelligent enough to handle any sort of simple funcall. One only needs to specify the syntax for macros that use destructuring (unless `*handle-macro-forms*` is `t` and the files being analyzed are also loaded), for functions with some argument positions that are special (e.g., apply and funcall), or to indicate that the function is of a specific caller type.

2.6. Implementation Notes

The functions `record-callers` and `record-callers*` do the real work in cross referencing a file. `record-callers` processes patterns that are symbols or otherwise atomic, while `record-callers*` processes simple list-structure patterns.

`record-callers` checks if the pattern is one of the known basic patterns. If so, it updates the database appropriately. Otherwise, it is a pattern defined in terms of other patterns, and `record-callers` substitutes the definition of the pattern substitution.

If the pattern is `form`, `record-callers` uses the form's tag (the car of the form) to look up a new pattern from the pattern database, and calls record-callers recursively on the form and the new pattern. If `*handle-macro-forms*` is `t` and the tag is a macro, it expands the macro and calls itself again on the result. Otherwise, `record-callers` assumes that the form is a random function call, and processes it with a default pattern of `(:star form)`.

`record-callers` also handles the special `:eq`, `:test` and `:typep` patterns. If the pattern is a list and not one of these special patterns, `record-callers` asks `record-callers*` to process the form and pattern.

`record-callers` (filename form &optional pattern parent (environment nil) funcall) [Function]

> `record-callers` is the main routine used to walk down the code. It matches the *pattern* against the *form*, possibly adding statements to the database. *parent* is the name defined by the current outermost definition; it is the caller of the forms in the body. *environment* is used to keep track of the scoping of variables. *funcall* deals with the type of variable assignment and determines how the environment should be modified. `record-callers` handles atomic patterns and simple list-structure patterns. For complex list-structure pattern destructuring, it calls `record-callers*`.

`record-callers*` is a more complex function. It is recursive in both `record-callers` and itself, and also maintains a stack of unprocessed patterns. The stack is needed to process the `:optional`, `:star`, `:plus` and `:rest` patterns correctly. For example, to process a `:optional` pattern element, `record-callers*` first tries matching the form against the rest of the `:optional` pattern element, pushing the other pattern elements onto the stack. If at any point `record-callers*` runs out of pattern elements, it continues from the pattern at the top of the stack. If processing the form with the `:optional` pattern element included fails (returns `nil`), `record-callers*` then tries skipping over the element. The `:star`, `:plus` and `:rest` patterns are similar.

`record-callers*` (filename form pattern parent environment &optional continuation in-optionals in-keywords) [Function]

> `record-callers*` handles complex list-structure patterns, such as ordered lists of subpatterns, patterns involving `:star`, `:plus`, `&optional`, `&key`, `&rest`, etc. *continuation* is a stack of unprocessed patterns, *in-optionals* and *in-keywords* are corresponding stacks which determine whether `&rest` or `&key` has been seen yet in the current pattern.

XREF assumes that the source code is syntactically correct Lisp, and uses `read` to read forms from the file. If `xref-file` drops into the debugger while processing a file, examining the value of `*last-form*` can help determine what went wrong.

`*last-form*` () [Variable]

> The last form read from the file. Useful for figuring out what went wrong when `xref-file` drops into the debugger.

The function `gather-tree` is used to create a list-structure tree representation of the database. Since the database may contain cycles, it stops when a reference is repeated in order to avoid infinite loops. The function `make-caller-tree` does something similar for when the root nodes are not specified. It calls `find-roots-and-cycles` to return a list of the uncalled callers as potential roots. The function `print-indented` tree prints out such trees using indentation to represent child nodes.

`gather-tree` (parents &optional already-seen (mode *default-graphing-mode*) (types-to-ignore *types-to-ignore*) compact) [Function]

> Extends the tree, copying it into list structure, until it repeats a reference (hits a cycle).

`make-caller-tree` (&optional (mode *default-graphing-mode*) (types-to-ignore *types-to-ignore*) compact) [Function]

> Outputs list structure of a tree which roughly represents the possibly cyclical structure of the caller database. If *mode* is `:call-graph`, the children of a node are the functions it calls. If *mode* is `:caller-graph`, the children of a node are the functions that call it. If *compact* is `t`, tries to eliminate the already seen nodes, so that the graph for a node is printed at most once. Otherwise it will duplicate the node's tree (except for cycles). This is useful because the call tree is actually a directed graph, so we can either duplicate references or display only the first one.

`find-roots-and-cycles` (&optional (mode *default-graphing-mode*) (types-to-ignore *types-to-ignore*)) [Function]

Returns a list of uncalled callers (roots) and called callers (potential cycles).

`print-indented-tree` (trees &optional (indent 0)) [Function]

Simple code to print out a list-structure tree (such as those created by `make-caller-tree`) as indented text.

3. METERING: Code Timing and Consing Profiler

The METERING system is a portable Common Lisp code profiling tool. It gathers timing and consing statistics for specified functions while a program is running. The METERING system is the result of a merging of the capabilities of the MONITOR program written by Chris McConnell and the PROFILE program written by Skef Wholey and Rob MacLachlan and extending the resulting program. Portions of the documentation from those programs were incorporated into this chapter.

3.1. Installing METERING

Before using METERING there are a number of small, implementation-dependent macros you may want to customize for your Lisp.

The METERING system will collect timing statistics in any valid Common Lisp. The macro `get-time` is called to find the total number of ticks since the beginning of time. The constant `time-units-per-second` is used to convert ticks into seconds. These default to `get-internal-run-time` and `internal-time-units-per-second`, respectively.

To collect consing statistics, define a `get-cons` macro for your implementation of Lisp. The `get-cons` macro has been defined for CMU Common Lisp, Lucid Common Lisp (3.0), and Macintosh Allegro Common Lisp (1.3.2). If you write a `get-cons` macro for a particular version of Common Lisp, we'd appreciate receiving the code. This macro should return the total number of bytes consed since the beginning of time.

The METERING system works by encapsulating the definitions of the monitored functions. By default, this encapsulation captures the arguments in an &rest arg, and then applies the old definition to the arguments. In most Lisps this will result in additional consing. To reduce the extra consing, when a `required-arguments` function is available we use it to find out the number of required arguments, and use &rest to capture only the non-required arguments (if any). The `required-arguments` function should return two values: the first is the number of required arguments, and the second is non-`nil` if there are any non-required arguments (e.g., &optional, &rest, and &key args). The `required-arguments` function has been defined for CMU Common Lisp, Macintosh Allegro Common Lisp (1.3.2), Lucid Common Lisp (3.0), and Allegro Common Lisp.

Since the encapsulation process creates closures, performance and accuracy are greatly improved if the code is compiled. Accordingly, the user is warned if the source is loaded instead of compiling it first.

3.2. Using METERING

This section describes all of the basic METERING commands and variables which control their behavior. METERING includes functions for monitoring and unmonitoring functions, as well as functions for displaying a report of profiling statistics, including number of calls, CPU time, and storage usage.

3.2.1. Suggested Usage

The easiest way to use the METERING system is to load it and evaluate either

```
(mon:with-monitoring (<name>*) ()
    <form>*)
```

or

```
(mon:monitor-form <form>)
```

The former allows you to specify which functions will be monitored, while the latter monitors all functions in the current package. Both automatically produce a table of statistics. Variations on these functions can be constructed from the monitoring primitives, which are described in Section 3.2.2.

Start by monitoring big pieces of the program, then carefully choose which functions to be monitored next.

If you monitor functions that are called by other monitored functions, decide whether you want inclusive or exclusive statistics. The former includes the monitoring time of inner functions from their callers, while the latter subtracts it. It is important to be aware of what kind of statistics you are displaying, since the difference can be confusing.

If the per-call time reported is less than 1/10th of a second, then consider the clock resolution and profiling overhead before you believe the time. You may need to run your program many times in order to average out to a higher resolution.

3.2.2. METERING Primitives

The `with-monitoring` and `monitor-form` macros are the main external interface to the METERING system.

`with-monitoring` ((&rest functions) [Macro]
(&optional (nested :exclusive) (threshold 0.01)
(key :percent-time))
&body body)

The named functions are monitored, the body forms executed, a table of results printed, and the functions unmonitored. The *nested*, *threshold*, and *key* arguments are passed to `report-monitoring`.

`monitor-form` (form &optional (nested :exclusive) (threshold 0.01) (key :percent-time)) [Macro]

> Monitor the execution of all functions in the current package during the evaluation of *form*. A table of results is printed. The *nested*, *threshold*, and *key* arguments are passed to `report-monitoring`.

The functions `monitor`, `unmonitor`, and `monitor-all` are primitives which are called by `with-monitoring` and `monitor-form`.

`*monitored-functions*` () [Variable]

> List of all functions that are currently being monitored.

`monitor` (&rest names) [Macro]

> The named functions are set up for monitoring by augmenting their function definitions with code that gathers statistical information about code performance. As with the `trace` macro, the names are not evaluated. If a function is already monitored, unmonitors it before remonitoring (useful when a function has been redefined). If a name is undefined, gives a warning and ignores it. If no names are specified, returns a list of all monitored functions. If a name is not a symbol, it is evaluated to return the appropriate closure. This allows the monitoring of closures stored anywhere, such as in a variable, array, or structure. Most other metering packages do not handle this.

`unmonitor` (&rest names) [Macro]

> Remove the monitoring on the named functions. If no names are specified, all currently monitored functions are unmonitored.

`monitor-all` (&optional (package *package*)) [Function]

> Monitors all functions in the specified package, which defaults to the current package.

`monitored` (function-place) [Function]

> Predicate which tests whether a function is monitored.

The following two functions are used to erase accumulated statistics.

`reset-monitoring-info` (name) [Function]

> Resets the monitoring statistics for the specified function.

`reset-all-monitoring` () [Function]

> Resets the monitoring statistics for all monitored functions.

The functions `report-monitoring` and `display-monitoring-results` are used to print a statistical report on the monitored functions. `display-monitoring-results` may be called to view the data created by `report-monitoring` in various ways.

`report-monitoring` (&optional names (nested :exclusive) (threshold 0.01) (key :percent-time) ignore-no-calls) [Function]

Creates a table of monitoring information for the current state of the specified list of functions, and displays the table using `display-monitoring-results`. If *names* is `:all` or `nil`, uses all currently monitored functions.

Takes the following arguments:

- *nested* specifies whether nested calls of monitored functions are included in the times of monitored functions.
  - If `:inclusive`, the per-function information is for the entire duration of the monitored function, including any calls to other monitored functions. If functions A and B are monitored, and A calls B, then the accumulated time and consing for A will include the time and consing of B.[5]
  - If `:exclusive`, the information excludes time attributed to calls to other monitored functions. This is the default.
- *threshold* specifies that only functions which have been executed more than *threshold* amount of the time will be reported. Defaults to 1%. If a threshold of 0 is specified, all functions are listed, even those with 0 or negative running times. See relevant note in Section 3.4.2.
- *key* specifies that the table be sorted by one of the following sort keys:
  - `:function`. Alphabetically by function name.
  - `:percent-time`. By percent of total execution time.
  - `:percent-cons`. By percent of total consing.
  - `:calls`. By number of times the function was called.
  - `:time-per-call`. By average execution time per function.
  - `:cons-per-call`. By average consing per function.
  - `:time`. Same as `:percent-time`.
  - `:cons`. Same as `:percent-cons`.

[5]If a function calls itself recursively, the time spent in the inner call(s) may be counted several times.

`display-monitoring-results` (&optional (threshold 0.01) (key :percent-time) (ignore-no-calls t)) [Function]

Prints a table showing for each named function:

- the total CPU time used in that function for all calls
- the total number of bytes consed in that function for all calls
- the total number of calls
- the average amount of CPU time per call
- the average amount of consing per call
- the percent of total execution time spent executing that function
- the percent of total consing spent consing in that function

Summary totals of the CPU time, consing, and calls columns are printed. An estimate of the monitoring overhead is also printed. May be run even after unmonitoring all the functions, to play with the data.

3.3. An Example of METERING Output

The following is an example of what the table looks like:

```
                                                 Cons
                  %     %                        Per      Total   Total
Function          Time  Cons  Calls  Sec/Call    Call     Time    Cons
----------------------------------------------------------------------
FIND-ROLE:        0.58  0.00    136  0.003521       0  0.478863      0
GROUP-ROLE:       0.35  0.00    365  0.000802       0  0.292760      0
GROUP-PROJECTOR:  0.05  0.00    102  0.000408       0  0.041648      0
FEATURE-P:        0.02  0.00    570  0.000028       0  0.015680      0
----------------------------------------------------------------------
TOTAL:                         1173                    0.828950      0
Estimated total monitoring overhead: 0.88 seconds
```

3.4. Usage Notes

This section comments on some aspects of the implementation that may affect the accuracy of the statistics.

3.4.1. Clock Resolution

On most machines, the length of a clock tick is much longer than the time it takes a simple function to run. For example, on an IBM RT-APC the clock resolution is 1/100th of a second, on a Decstation 3100 it is 1/1000th of a second, and on a Symbolics 3640 it is 1/977th of a second. This means that if a function is called only a few times, then only the first few decimal places are really meaningful.

3.4.2. Calculating Monitoring Overhead

Every time a monitored function is called, the added monitoring code takes some amount of time to run. This can result in inflated times for functions that take little time to run. Also, in many Lisps the function `get-internal-run-time` conses, which can affect the consing statistics. Accordingly, an estimate of the overhead due to monitoring is subtracted from the time and storage reported for each function.

Although this correction works fairly well, it is not completely accurate. This can result in times that become increasingly meaningless for functions with shorter runtimes. For example, subtracting the estimated overhead may result in negative times for some functions. This should only be of concern when the estimated profiling overhead is many times larger than the reported total CPU time.

If you monitor functions that are called by monitored functions, in `:inclusive` mode the monitoring overhead for the inner functions are subtracted from the CPU time for the outer function.[6] In `:exclusive` mode this is not necessary, since we subtract the monitoring time of inner functions, overhead and all.

Otherwise, the estimated monitoring overhead is not counted in the reported total CPU time. The sum of total CPU time and the estimated monitoring overhead should be close to the total CPU time for the entire monitoring run (as reported by the `time` macro).

The timing overhead factor is computed at load time. This will be incorrect if the monitoring code is run in a different environment than that in which the file was loaded. For example, saving a Lisp image on a high performance machine and running it on a low performance one will result in an erroneously small overhead factor.

If the statistics vary widely, possible causes are:

- Garbage collection. Try turning it off and then running the code. Be forewarned that running an encapsulated function results in some extra consing, and that `get-internal-run-time` will probably cons as well.
- Swapping. The time it takes to swap your function into memory can affect the reported statistics. If you have enough memory, try executing your form once before monitoring it so that it will be swapped into memory.
- Resolution of `internal-time-units-per-second`. This value is rather coarse in many Lisps, as noted in Section 3.4.1. If this value is too low, the timings become wild. Try executing your test form more times or for a larger number of iterations.

---

[6]This is accomplished by counting for each function not only the number of calls to the function itself, but also the number of calls to monitored functions. This can become rather confusing for recursive functions.

3.5. Implementation Notes

The overhead is calculated by monitoring `stub-function` and running it for a large number of times (`overhead-iterations`), storing the timing and consing overhead into the variables `*monitor-time-overhead*` and `*monitor-cons-overhead*`, respectively. Since `stub-function` is a null function, this results in a fairly accurate estimate for the overhead of monitoring a function. If you suspect that these values are inaccurate, try running `set-monitor-overhead` again.

`*monitor-time-overhead*` () [Variable]

> The amount of time an empty monitored function costs.

`*monitor-cons-overhead*` () [Variable]

> The amount of cons an empty monitored function costs.

`overhead-iterations` 5000 [Constant]

> Number of iterations over which the timing overhead is averaged.

`stub-function` () [Function]

> A null piece of code run monitored to estimate monitoring overhead.

`set-monitor-overhead` () [Function]

> Determines the average overhead of monitoring by monitoring the execution of an empty function many times.

The key idea behind METERING is to replace the definition of the monitored function with a closure that records the monitoring data and updates the data with each call to the function. As noted in Section 3.1, we can reduce the amount of consing done by the &rest arg in each lambda by using the &rest arg to capture only the non-required arguments. The function `make-monitoring-encapsulation` returns a lambda expression which, when called with a function name, encapsulates it with a closure that has the right number of required arguments. To create these closures efficiently, we precompute the encapsulation-creating functions for up to `precomputed-encapsulations` number of required arguments (with and without optional arguments) and store them in a hash table for later retrieval by `monitoring-encapsulate`. If, when encapsulating a function, the encapsulation-creating function is not found in the hash table by `find-encapsulation`, a new function is added to the table. Since we're precomputing closure functions for common argument signatures, there is no need to call `compile` for each monitored function.

`make-monitoring-encapsulation` (min-args optionals-p) [Function]

> Makes a function which will appropriately encapsulate any function with *min-args* required arguments.

`precomputed-encapsulations` 8 [Constant]

> We create precomputed encapsulations for up to this number of required arguments. Any others will be computed as needed.

`*existing-encapsulations*` (make-hash-table :test (function equal)) [Variable]

Hash table which maps from common argument signatures (required-args . optionals-p) to appropriate precomputed closure functions.

`find-encapsulation` (min-args optionals-p) [Function]

Used to find the appropriate precomputed encapsulation function if it exists, or create (and save) a new one if necessary.

`monitoring-encapsulate` (name &optional warn) [Function]

Monitors the function Name. If already monitored, unmonitor first.

`monitoring-unencapsulate` (name &optional warn) [Function]

Removes monitoring encapsulation code from around Name.

The variables `*monitor-results*` and `*no-calls*` are associated with the functions that create and display monitoring statistics.

`*monitor-results*` () [Variable]

A table of monitoring statistics is stored here.

`*no-calls*` () [Variable]

A list of monitored functions which weren't called.

4. DEFSYSTEM: A "make" for Lisp

The DEFSYSTEM program is a portable system definition facility for Common Lisp.[7] It is useful for managing programs which consist of several files, and provides a convenient way to describe dependencies between files in the system and dependencies of the system on other systems.

A system is defined as hierarchical layers of modules, with an optionally matching directory structure. In contrast with the Symbolics defsystem, systems are described solely in terms of their structure; the user does not need to worry about procedural matters such as compilation order. For example, the components of a system may be listed in any order the user desires, because the `defsystem` macro reorganizes them according to the file-dependency constraints specified by the user. Since it accomplishes this by performing a topological sort of the constraint graph, cyclical file dependencies are not supported (i.e., the file-dependency graph must be a DAG).

DEFSYSTEM includes many of the basic features, such as minimizing the amount of compilation and loading that must be done when some part of the system is changed. Selective recompilation occurs only when the binary file either does not exist or is older than the corresponding source file, or when the file depends on other files that needed to be recompiled. Of course, the user can decide to override this behavior and require that all files be recompiled, even those whose binary files are up to date.

Only two operations on systems are currently defined (compile and load). The interface for defining new operations on systems, however, is simple and straightforward.

DEFSYSTEM does not currently support patching.

4.1. Installing DEFSYSTEM

Before using DEFSYSTEM, decide if you want to have one or more central directories where system definition files will be kept. If so, modify the value of `*central-registry*` to contain a list of the pathnames of those directories.

Verify that the value of `*filename-extensions*` includes source and binary extensions for your Lisp; if not, add them.

Set the variable `*dont-redefine-require*` to `t` if you want to prevent DEFSYSTEM from redefining `require`. This is useful for Lisps that treat `require` specially in the compiler. (For example, some Lisps treat `require` as if an `(eval-when (compile load eval) ...)`

[7] Though home-grown, it was inspired by fond memories of the defsystem facility on Symbolics 3600 Lisp Machines [3] [4]. The exhaustive list of filename extensions for various Lisps was initially taken from Xerox Corporation's PCL miniature defsystem facility. The idea to have one `operate-on-system` function in addition to separate `compile-system` and `load-system` functions was also taken from PCL.

were wrapped around the form, and lose this special treatment when `require` is redefined. While we provide an alternate redefinition of `require` as a macro to work around this problem, some users may find it simpler to not redefine `require`, especially if they don't need the backward compatibility.)

If you intend to use logical pathnames in your system definitions, the LOGICAL-PATHNAMES package must be loaded before compiling or loading DEFSYSTEM.

Save a Lisp image with DEFSYSTEM loaded, so that you won't have to reload it each time you restart Lisp.

4.2. Overview

DEFSYSTEM is loaded into the "MAKE" package, so prefix all the following functions and variables with a "MAKE:" or the nickname "MK:". This name was chosen to avoid naming conflicts with various Lisps, many of which already have a "DEFSYSTEM" package for their own particular system construction tool.

The external interface to the defsystem facility consists of the `defsystem` macro and the `operate-on-system` function. `defsystem` is used to define a new system and `operate-on-system` to compile it and load it. The functions `compile-system` and `load-system` are provided as an alternate way of compiling and loading a system. They call `operate-on-system` with appropriate arguments. The definition of `require` has been modified to mesh well with systems defined using `defsystem`, and is fully backward-compatible.

To use DEFSYSTEM,

1. Write a `defsystem` form for your system, and save it in a file of type `"system"`. If the name of your system is `foo`, the file should be named `"foo.system"`. You may want to move the file into one of your central registry directories.
2. Use the function `operate-on-system` (or `compile-system` and `load-system`) to do things to your system. For example evalutating `(operate-on-system "foo" 'load)` will load the system, while evalutating `(operate-on-system "foo" 'compile)` will compile it. [If you are going to load the system and not compile it, you can also use `(require "foo")` to load it.]

DEFSYSTEM checks for an appropriately named system definition file first in your current directory, then in the central registry directories in the order in which they are listed in the variable `*central-registry*`. If it finds a match, it will reload the system definition file if it has changed since you last loaded the system definition. If the system definition file is located in neither the current directory nor one of the central registry directories, you must explicitly load the system definition file.

4.3. Using DEFSYSTEM

This section describes all of the basic DEFSYSTEM commands and the variables which control their behavior. DEFSYSTEM includes functions for defining new systems, compiling systems, and loading systems.

4.3.1. Defining a System

A system is a set of *components* with associated *properties*. The properties include the type of the component, its name, source and binary pathnames, its package, initializations and finalizations, and the component dependencies, as well as the components of the component.

The `defsystem` macro is used to define new systems.

`defsystem` (name &rest definition-body) [Macro]

> Defines *name* to be the name of the system described in *definition-body*. This name is used for all operations on the system. The definition body consists of a sequence of keyword-value pairs, where the keywords correspond to the properties described below. These properties determine what files are included in the system, what files depend on other files, and any features of the overall system, such as its directory or package. *name* may be a symbol or a string; if a symbol, the symbol-name is used. The format of the top level `defsystem` definition parallels that of components, except the component type is replaced with the symbol `defsystem`. Once a system is defined,[8] certain operations such as loading and compilation may be applied to it.

4.3.1.1. Component Types

There are five types of components, `:system`, `:subsystem`, `:module`, `:file`, and `:private-file`.

- Components of type `:system` have absolute pathnames and are used to define a multi-system system. The toplevel system defined by the `defsystem` macro is implicitly of type `:system`.
- Components of type `:subsystem` have relative pathnames and are used to define subsystems of a system.
- Components of type `:module` have pathnames that are relative to their containing system or module, and may contain a set of files and/or modules. This enables one to define modules, submodules, and so on.
- Components of type `:file` represent files with relative pathnames.
- Components of type `:private-file` also represent files, but with absolute pathnames. Components of type `:private-file` are useful for having private copies of one or two files of a system without having to rewrite the entire system definition or duplicate the entire system directory tree.

---

[8] System definitions may be automatically loaded if not defined. See Section 4.5.2.

4.3.1.2. Component Names

The name of a component is refered to by other components to indicate that they depend on it. The name of a component may also be used as the name of the subdirectory or file associated with the component. See Section 4.3.1.3 for details.

The name of a component may be a symbol or a string. For ease of access the definition of a system (its component) is stored in a hash table entry corresponding to an uppercase version of the string or symbol name. If the system name is a symbol, for all other purposes the name is converted to a lowercase string (system names that are strings are left alone). A system defined as `'foo` will have an internal name of `"FOO"` and will be stored in the file `"foo.system"`. A system defined as `"Foo"` will have an internal name of `"FOO"` and will be stored in the file `"Foo.system"`.

4.3.1.3. Component Pathnames and File Types

The absolute pathnames (for components of type `:system` and `:private-file`) and relative pathnames (for all other components) of the binary and source files may be specified using the `:source-pathname` and `:binary-pathname` keywords in the component definition.[9] The pathnames associated with a module correspond to subdirectories of the containing module or system. If no binary pathname is specified, the binaries are distributed among the sources. If no source pathname is given for a component, it defaults to the name of the component. Since the names are converted to lowercase, pathnames must be provided for each component if the operating system is case sensitive (unless the pathnames are all lowercase). Similarly, if a module does not correspond to a subdirectory, a null-string pathname (`""`) must be provided. One may change this behavior by modifying the variable `*source-pathname-default*`. For example, one could set it to `""` instead of `nil` to avoid having to specify `:source-pathname ""` in every module if the files are kept in a single flat directory.

File types (e.g., `lisp` and `fasl`) for source and binary files may be specified using the `:source-extension` and `:binary-extension` keywords. These file types are inherited by the components of the system. If the file types are not specified or given as `nil`, DEFSYSTEM makes a reasonable choice of defaults based on the machine type and underlying operating system.

At system definition time, every relative directory is replaced with the corresponding cumulative absolute pathname with all the pathname-components specified.

---

[9] Macintosh pathnames are not fully supported at this time. For example, trailing colons must be included in the pathnames of each module. For system definitions to be portable between UNIX Lisps and Macintosh Common Lisp, one must use the LOGICAL-PATHNAMES package.

4.3.1.4. Segregating Binaries for Different Lisps

The user may wish to maintain different subdirectories for the binaries of different Lisps. The function `afs-binary-directory` has been provided to imitate the behavior of the @sys feature of the Andrew File System on systems not running AFS. The @sys feature allows soft links to point to different directories depending on which platform is accessing the files. A common setup would be to have the bin directory soft linked to `.bin/@sys` and to have subdirectories of `.bin` corresponding to each platform (`.bin/vax_mach`, `.bin/unix`, `.bin/pmax_mach`, etc.).

`afs-binary-directory` (root-directory) [Function]

> Returns the appropriate binary directory for use as the `:binary-pathname` argument in the `defsystem` macro. For example, if we evaluate `(afs-binary-directory "foodir/")` on a vax running the Mach operating system, `"foodir/.bin/vax_mach/"` would be returned.

The functions `machine-type-translation` and `software-type-translation` are used to define the directory components corresponding to the values of `(machine-type)` and `(software-type)` for particular Lisps.

`machine-type-translation` (name &optional dir-component) [Function]

`software-type-translation` (name &optional dir-component) [Function]

4.3.1.5. Including Foreign Systems

Systems defined using some other system definition tool may be included by providing separate compile and load forms for them (using the `:compile-form` and `:load-form` keywords). These forms will be run if and only if they are included in a module with no components. This is useful if it isn't possible to convert these systems to the `defsystem` format all at once.

4.3.1.6. Component Packages, Initializations and Finalizations

One may also specify the package to be used and any initializations and finalizations. Package usage (specified with the keyword `:package`) remains in force until the `*package*` variable reverts to its old value at the end of the operation on the component. Initializations (specified with the keyword `:initially-do`) are evaluated before the system is loaded or compiled, and finalizations (specified with the keyword `:finally-do`) are evaluated after the system is finished loading or compiling. The argument to the keyword is a form which is evaluated. Multiple forms may be evaluated by wrapping a progn around the forms.

4.3.1.7. Component Dependencies

The dependencies of a system, module or file are specified with the `:depends-on` keyword, followed by a list of the names of the components the system, module or file depends on. The components referred to must exist at the same level in the hierarchy as the referring component. This enforces the modularity of the system definition. If module A depends on a file contained within module B, then module A depends on module B and should be specified as such. This requirement is not enforced in the software, but any use contrary to it will produce unpredictable results.

Thus the only requirement for how the files are to be organized is that at the level of each module or system, the dependency graph of the components must be a DAG (directed **acyclic** graph). If there are any dependency cycles (i.e., module A uses definitions from module B, and module B uses definitions from module A), the `defsystem` macro will not be able to compute a total ordering of the files (a linear order in which they should be compiled and loaded). Usually `defsystem` will detect such cycles and halt with an error.

If no dependencies are provided for the system, modules and files, it may load them in any order. There is no guarantee of loading them in any particular order. Currently, however, it loads them in serial order, because the topological-sort it uses is a stable sorting method.

The algorithm topologically sorts the DAG at each level in the hierarchy (system level, module level, submodule level, etc.) to ensure that the system's files are compiled and loaded in the right order. This occurs at system definition time, rather than at system use time, because it probably saves the user some time to do it this way.

4.3.1.8. Load-only and Compile-only Components

One may define components that are load-only and compile-only using the keywords `:load-only t` and `:compile-only t`.

Load-only components are not compiled during operation `:compile`. For such components, loading the component satisfies any demand for recompilation.

Compile-only components are not loaded during operation `:compile`. The component is either loaded or compiled, but not both. For such components, compiling the file satisfies the demand to load it. This isn't as strange as it seems at first. For example, PCL `defmethod` and `defclass` definitions wrap an `(eval-when (compile load eval) ...)` around the body of the definition, making it pointless in some Lisps to compile and load a file containing only class definitions.

4.3.1.9. Component Definitions

The components of a system, module or file are specified with the `:components` keyword, and are defined in a manner analogous to the way in which a system is defined.

The general format of a component's definition is:

```
<definition> ::= (<type> <name> [:host <host>] [:device <device>]
                                [:source-pathname <pathname>]
                                [:source-extension <extension>]
                                [:binary-pathname <pathname>]
                                [:binary-extension <extension>]
                                [:package <package>]
```

```
                                 [:initially-do <form>]
                                 [:finally-do <form>]
                                 [:components (<definition>*)]
                                 [:depends-on (<name>*)]
                                 [:load-only t]
                                 [:compile-only t]
                                 [:compile-form <form>]
                                 [:load-form <form>])
<type> ::= :system | :module | :file | :private-file
```

The top level defsystem form does not specify a type, replacing it with the symbol `defsystem`.

In addition, component definitions which are strings or lists whose first element is not a valid type are assumed to be of type `:file`. This allows the user to specify a list of files as a list of the filenames.

Here are three examples of various component definitions:

```
(:system test
  :source-pathname "/afs/cs.cmu.edu/user/mkant/Defsystem/test/"
  :source-extension "lisp"
  :binary-pathname nil
  :binary-extension nil
  :components ((:module basic
                        :source-pathname ""
                        :components ((:file "primitives")
                                     (:file "macros"
                                            :depends-on ("primitives"))))
               (:module graphics
                        :source-pathname "graphics"
                        :components ((:file "macros"
                                            :depends-on ("primitives"))
                                     (:file "primitives"))
                        :depends-on (basic))))

(:module graphics
   :source-pathname "graphics"
   :components (("macros" :depends-on ("primitives"))
                (:private-file "primitives")))

(:module graphics
   :source-pathname "graphics"
   :components ("primitives" "macros" "scanning"))
```

Thus one would define a system named `foo` that depends on systems `bar` and `bletch` as follows:

```
(defsystem foo
  :source-pathname "/afs/cs.cmu.edu/user/mkant/foo/"
  :source-extension "lisp"
  :binary-pathname nil
  :binary-extension nil
  :components ((:module graphics
                        :source-pathname "graphics"
                        :components ("primitives" "macros" "scanning")))
```

```
    :depends-on (bar bletch))
```

This system would load the `bar` and `bletch` systems before loading any files of the `foo` system. Note that the modularity restrictions require that `bar` and `bletch` and not modules or files. Components can depend only on components of the same complexity; thus systems can depend only on systems.

Also worth stressing is the fact that systems in DEFSYSTEM are structural, unlike the procedural systems in the Symbolics `defsystem`. So while specifying `(:system bar)` in the body of a Symbolics `defsystem` would include the `bar` system at that point, in DEFSYSTEM it would be saying that the `bar` system has no files. To achieve the same effect one must include the `bar` system in the dependency list of the system.

4.3.2. Describing a System

The function `describe-system` may be used to print a description of a system.

`describe-system` (name &optional (stream *standard-output*)) [Function]

> Prints a description of the system to *stream*. If *name* is the name of a system, gets it and prints a description of the system. If *name* is a component, prints a description of the component.

The function `defined-systems` may be used to get a list of all currently defined systems.

4.3.3. Removing a System

The function `undefsystem` may be used to remove the definition of a system.

`undefsystem` (name) [Function]

> Removes the definition of the system named *name*.

4.3.4. Loading and Compiling a System

The function `operate-on-system` is used to compile or load a system, or do any other operation on a system. At present only compile and load operations are defined, but other operations such as edit, hardcopy, or applying arbitrary functions (e.g., enscript, lpr) to every file in the system could be easily added.

The syntax of operate-on-system is as follows:

`operate-on-system` (name operation &key force (version *version*) [Function]
(test *oos-test*) (verbose *oos-verbose*)
(load-source-instead-of-binary *load-source-instead-of-binary*)
(load-source-if-no-binary *load-source-if-no-binary*)
(bother-user-if-no-binary *bother-user-if-no-binary*)
(compile-during-load *compile-during-load*)
dribble (minimal-load *minimal-load*))

- *system-name* is the name of the system and may be a symbol or string. If the system is not defined, it will be loaded from a file with extension `"system"` and name the same as the system, located either in the current directory or in the central registry, if such a file exists. Otherwise an error will be signalled.
- *operation* is `'compile` (or `:compile`) or `'load` (or `:load`) or any new operation defined by the user. If no such operation is defined, an error will be signalled.
- *force* determines what files are operated on:
  - `:all` (or `t`) specifies that all files in the system should be used
  - `:new-source` If the operation is `'compile`, compiles only those files whose sources are more recent than the binaries. If the operation is `'load`, loads the source if it is more recent than the binaries. This allows you to load the most up to date version of the system even if it isn't compiled.
  - `:new-source-and-dependents` uses all files used by `:new-source`, plus any files that depend on the those files or their dependents (recursively).
  - *force* may also be a list of the specific modules or files to be used (plus their dependents).

  The default for `'load` is `:all` and for `'compile` is `:new-source-and-dependents`.
- *version* indicates which version of the system should be used. If `nil`, then the usual root directory is used. If a symbol, such as `'alpha`, `'beta`, `'omega`, `:alpha`, or `'mark`, it substitutes the appropriate (lowercase) subdirectory of the root directory for the root directory. If a string, it replaces the entire root directory with the given directory. (default `*version*`, which is `nil`)
- *verbose* is `t` to print out what it is doing (compiling, loading of modules and files) as it does it. (default `nil`)
- *test* is `t` to print out what it would do without actually doing it. If test is `t` it automatically sets verbose to `t`. (default `nil`)
- *compile-during-load* is `t` to compile source files when loading a system if the binary files are missing or old. If `nil` it doesn't compile them, but loads either the old binaries or the sources. If `:query` (the default), it will ask the user whether the files should be compiled.
- *dribble* should be the pathname of a dribble file if you want to keep a record of the compilation. (default `nil`)
- *minimal-load* is `t` to only load those files which haven't already been loaded yet, as judged by the file-write-dates of the files. Note that DEFSYSTEM will notice when files change even if a different user compiles the files. (default `*minimal-load*`, which is `nil`)
- *load-source-instead-of-binary* is `t` to force the system to load source files instead of binary files. (default `nil`)

- *load-source-if-no-binary* is `t` to have the system load source files if the binary file is missing. (default `nil`)
- *bother-user-if-no-binary* is `t` to have the system bother the user about missing binaries before it goes ahead and loads them if `load-source-if-no-binary` is `t`. (default `t`) Times out in 60 seconds unless `*use-timeouts*` is set to `nil`.

The `compile-system` and `load-system` functions are just like `operate-on-system`, except the operation is hard-coded as `:compile` and `:load`, respectively, so there is no *operation* argument. Some users find this interface easier to understand. The function `oos` is defined as a synonym for `operate-on-system`.

For example, one would compile all the changed files in a system named "foo" by typing `(mk:compile-system "foo" :force :new-source :minimal-load t)`. Or one could selectively compile changed files in the system when loading the system from scratch by invoking `(mk:load-system "foo" :compile-during-load :query)`. To load all the files in the system, type `(mk:load-system "foo")`. To compile all the files in the system, type `(mk:compile-system "foo")`.

An implicit assumption is that if we need to load a file for some reason, then we should be able to compile it immediately before we need to load it. This obviates the need to specify separate load and compile dependencies in the modules.

Note that under this assumption, the example given in the PCL defsystem becomes quite ludicrous. Those constraints are of the form:

1. C must be loaded before A&B are loaded
2. A&B must be loaded before C is compiled

When you add in the reasonable assumption that before you load C, you must compile C, you get a cycle.

One case is which this might not be true is in a system which worked on the dependency graph of individual definitions. But we have restricted ourselves to file dependencies and will stick with that. (In situations where a file defining macros must have the sources loaded before compiling them, most often it is because the macros are used before they are defined, and hence assumed to be functions. This can be fixed by organizing the macros better, or including them in a separate file.)

Files which must not be compiled should be loaded in the initializations or finalizations of a module by means of an explicit load form, or be specified as `:load-only t`.

It is a known bug that DEFSYSTEM may report loading or compiling a system or module even if it doesn't do anything with the files. So if DEFSYSTEM reports loading a module, but doesn't report loading any files in the module, it hasn't touched the files in the module. In a future version of DEFSYSTEM we may change the message to say that it is checking the system or module.

4.3.5. Other Operations on Systems

To define a new operation, write a function with parameters *component* and *force* that performs the operation. The function `component-pathname` may be used to extract the source and binary pathnames from the component. (`component-pathname` takes parameters *component* and *file-type*, where *file-type* is either `:source` or `:binary`, and returns the appropriate pathname.) If the component has "changed" as a result of the operation, `t` should be returned; otherwise `nil`. See the definition of `compile-file-operation` and `load-file-operation` for examples.

Then install the definition using `component-operation`, which takes as parameters the symbol which will be used to name the operation in `operate-on-system`, and the name of the function. For example, here are the definition of the `'compile` and `:compile` operations:

```
(component-operation :compile  'compile-and-load-operation)
(component-operation 'compile  'compile-and-load-operation)
```

The user could define operations such as `'hardcopy` and `'edit` in this manner.

4.3.6. Changes to Require

This defsystem interacts smoothly with the `require` and `provide` facilities of Common Lisp. `operate-on-system` automatically provides the name of any system it loads, and uses the new definition of `require` to load any dependencies of the toplevel system.

One may prevent DEFSYSTEM from redefining `require` by setting the variable `*dont-redefine-require*` to `t` before compiling DEFSYSTEM.

DEFSYSTEM adds three new optional arguments to `require`. Thus the new syntax of `require` is as follows:

`new-require` (system-name &optional pathname definition-pname default-action (version *version*)) [Function]

> If pathname is provided, the new `require` behaves just like the old definition. Otherwise it first tries to find the definition of the *system-name* (if it is not already defined it will load the definition file if it is in the current-directory, the central-registry directory, or the directory specified by *definition-pname*) and runs `operate-on-system` on the system definition. If no definition is found, it will evaluate the *default-action* if there is one. Otherwise it will try running the old definition of `require` on just the system name. If all else fails, it will print out a warning.

4.3.7. DEFSYSTEM Variables

The following variables control the default operation of DEFSYSTEM. Many of the program parameters set by modifying these variables can also be changed by specifying keyword arguments to DEFSYSTEM functions.

`*defsystem-version*` "v2.4 22-MAY-91" [Variable]

Current version number/date for DEFSYSTEM.

`*central-registry*` () [Variable]

Central directory of system definitions. May be either a single directory pathname, or a list of directory pathnames to be checked after the local directory.

`*bin-subdir*` ".bin/" [Variable]

The subdirectory of an AFS directory where the binaries are really kept.

`*tell-user-when-done*` () [Variable]

If `t`, system will print `"...DONE"` at the end of an operation.

`*oos-verbose*` () [Variable]

If `t`, `operate-on-system` describes what it is doing as it does it.

`*oos-test*` () [Variable]

If `t`, `operate-on-system` runs in a test mode where it describes what it would do, but doesn't actually do it.

`*load-source-if-no-binary*` () [Variable]

If `t`, system will try loading the source if the binary is missing.

`*bother-user-if-no-binary*` t [Variable]

If `t`, the system will ask the user whether to load the source if the binary is missing.

`*load-source-instead-of-binary*` () [Variable]

If `t`, the system will load the source file instead of the binary.

`*minimal-load*` () [Variable]

If `t`, the system tries to avoid reloading files that were already loaded and up to date.

`*operations-propagate-to-subsystems*` t [Variable]

If `t`, operations like `:compile` and `:load` propagate to subsystems of a system that are defined either using a component-type of :system or by another defsystem form.

`*filename-extensions*` (car [Variable]
(quote
(("lisp" . "fasl") ("lisp" . "lbin"))))

Filename extensions for Common Lisp. Each is a read-time conditionalized cons of the form (Source-Extension . Binary-Extension). If the Lisp is unknown (as in `*features*` not known), defaults to `lisp` and `lbin`.

`*system-dependencies-delayed*` t [Variable]

If `t`, system dependencies of top-level systems are expanded at run time. There is little support for not delaying the expansion of top-level system dependencies, so this variable should not be set to `nil`.

`*providing-blocks-load-propagation*` t [Variable]

If `t`, if a system dependency exists (was provided using `provide`) in `*modules*`, it is not loaded.

4.4. An Example of Using DEFSYSTEM

This section gives an example of using `defsystem` for the files in the following directory structure:

```
% du -a test
1       test/fancy/macros.lisp
1       test/fancy/primitives.lisp
3       test/fancy
1       test/macros.lisp
1       test/primitives.lisp
1       test/graphics/macros.lisp
1       test/graphics/primitives.lisp
3       test/graphics
1       test/os/macros.lisp
1       test/os/primitives.lisp
3       test/os
12      test
```

First we define the system `test` to correspond to the file dependency structure:

```
(defsystem test
  :source-pathname "/afs/cs.cmu.edu/user/mkant/Defsystem/test/"
  :source-extension "lisp"
  :binary-pathname nil
  :binary-extension nil
  :components ((:module basic
                        :source-pathname ""
                        :components ((:file "primitives")
                                     (:file "macros"
                                            :depends-on ("primitives"))))
               (:module graphics
                        :source-pathname "graphics"
                        :components ((:file "macros"
                                            :depends-on ("primitives"))
                                     (:file "primitives"))
                        :depends-on (basic))
               (:module fancy-stuff
                        :source-pathname "fancy"
                        :components ((:file "macros"
                                            :depends-on ("primitives"))
                                     (:file "primitives"))
                        :depends-on (graphics operating-system))
               (:module operating-system
                        :source-pathname "os"
                        :components ((:file "primitives")
                                     (:file "macros"
                                            :depends-on ("primitives")))
                        :depends-on (basic)))
  :depends-on nil)
```

Then we may use `operate-on-system` to compile and load the system.

```
<cl> (operate-on-system 'test 'compile :verbose t)

; - Compiling system "test"
;   - Compiling module "basic"
```

```
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/primitives.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/primitives.fasl"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/macros.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/macros.fasl"
;    - Compiling module "graphics"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/primitives.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/primitives.fasl"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/macros.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/macros.fasl"
;    - Compiling module "operating-system"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/primitives.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/primitives.fasl"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/macros.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/macros.fasl"
;    - Compiling module "fancy-stuff"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/primitives.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/primitives.fasl"
;      - Compiling source file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/macros.lisp"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/macros.fasl"
; - Providing system test
NIL

<cl> (operate-on-system 'test 'load :verbose t)

; - Loading system "test"
;    - Loading module "basic"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/primitives.fasl"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/macros.fasl"
;    - Loading module "graphics"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/primitives.fasl"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/graphics/macros.fasl"
;    - Loading module "operating-system"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/primitives.fasl"
;      - Loading binary file
;        "/afs/cs.cmu.edu/user/mkant/Defsystem/test/os/macros.fasl"
;    - Loading module "fancy-stuff"
```

```
;     - Loading binary file
;       "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/primitives.fasl"
;     - Loading binary file
;       "/afs/cs.cmu.edu/user/mkant/Defsystem/test/fancy/macros.fasl"
; - Providing system test
NIL
```

4.5. Implementation Notes

In this section we discuss some issues relating to the implementation of DEFSYSTEM.

4.5.1. Structural vs. Procedural System Construction Tools

There are two major types of system construction tools, procedural and structural. Procedural tools define a system as a sequence of explicit construction steps, perhaps augmented with some description of structural dependencies. The UNIX `make` [1] and Symbolics `defsystem` [3] [4] are examples of this kind of tool. Structural tools define a system in terms of its structure. Instead of describing how modules are to be constructed, a structural definition describes how the modules reference each other, and infers the order of construction operations from the reference graph. The BUILD system [5] is an example of a structure-based system definition tool.

As noted by Robbins in [5], a procedural definition of a system is harder to understand than a structural definition. In addition, there are several benefits to the separation of construction knowledge from systems knowledge that occurs in structural system construction tools:

- Such tools can be extended by adding new operations on systems without altering existing system definitions. Since the tool is not constrained to a particular set of embedded tasks, the users are free to define new operations.
- When defining a new operation, many low level details (e.g., compilation order) are hidden from the task definer, simplifying the definition of new operations.
- Structural tools are a more natural way for users to describe systems, allowing them to concentrate on the overall structure of the system. Users can ignore low level details of the construction operations when writing a system. The explicit declaration of high level system relationships is also much easier to understand.
- It is much easier to automatically generate structural descriptions of systems. For example, XREF includes tools to assist the user in creating a system definition by producing the file dependency graph.

Accordingly, we chose to design DEFSYSTEM as a structure-based system construction tool.

The user supplies DEFSYSTEM with a description of the structure of the system, and DEFSYSTEM infers the compilation steps. The system definition describes how modules reference each other instead of how they are constructed. From a structural description (module A refers to module B) it can infer the procedural requirements (a change to module B implies that module A should be recompiled, but a change to module A does not imply that module B should be recompiled).

Unfortunately, DEFSYSTEM's mechanism for describing operations is not as elegant or as general as BUILD's. The knowledge about Lisp compilation and loading, although largely isolated into separate operation definitions, is still partially embedded in the definition of `operate-on-system`. So if a new operation is sensitive to reference types other than those provided for the compile and load operations, it may require revising the definition of `operate-on-system` as well as the system definitions.[10] However, DEFSYSTEM is sufficient for at least compilation and loading in Lisp, which is where the major need lies.

4.5.2. Retrieving System Definitions

It is desirable that a system definition be automatically loaded if not already present when its name is referenced by the user or a system definition. `find-system` implements this behavior, loading the definition of the system `foo` from the file `"foo.system"` in the central registry. `find-system` calls `compute-system-path` to determine the pathname of the file containing the system definition.

`find-system` (system &optional (mode :ask) definition-pname) [Function]

> Returns the system named *system*. If the system was not previously defined or the version on disk is newer, `find-system` tries to load the system definition. This allows `operate-on-system`, `compile-system`, and `load-system` to work on non-loaded as well as loaded system definitions. *definition-pname* is the pathname for the file containing the system definition, if provided. Otherwise `find-system` checks for a file matching the system name first in the current directory and then in the central registry directories. If the variable `*reload-systems-from-disk*` is `nil`, `find-system` will not reload the system definition of a defined system from disk if the version on disk is newer.

4.5.3. Appending Directories

The `append-directories` function is used to tack a subdirectory onto a pathname. Sadly, Common Lisp lacks a primitive to do this. Our definition will work for all Lisps that conform to the conventions on structured directories [7, Section 23.1.3]. Minor incompatibilities with the standard are fixed using read-time conditionalization. Major aberrations are handled either using special purpose code, or using

```
(namestring (merge-pathnames (or absolute-directory "")
                             (or relative-directory "")))
```

which seems to work surprisingly well in VMS-based VaxLisp.

The output from the function `test-new-append-directories` may be useful for verifying correct operation of this primitive when porting it to new Lisps.

---

[10] For example, we currently assume that compilation-load dependencies and if-changed dependencies are identical. However, in some cases this might not be true. For example, if we change a macro we have to recompile functions that depend on it, but not if we change a function. Splitting these apart (with appropriate defaulting) would be nice, but not worth doing immediately since it may save only a couple of file recompilations, while making the defsystem much more complex.

See Appendix II for a discussion of this and other problems with Common Lisp.

`append-directories` (absolute-directory relative-directory) [Function]

> There is no Common Lisp primitive for tacking a subdirectory onto a directory. We need such a function because `defsystem` has both absolute and relative pathnames in the modules. We assume that *absolute-directory* is a directory, with no filename stuck on the end. *relative-directory*, however, may have a filename stuck on the end.

4.5.4. Defining a System

Defining a system invokes several functions. `create-component` is the main routine for creating a representation. It takes care of inheriting appropriate attributes from parent components, initializes the component's pathnames using `create-component-pathnames` and `generate-component-pathname`, recursively creates any child components using `expand-component-components` and `expand-component-definition`, ties together the dependency graph using `link-component-depends-on`, and topologically sorts the dependency graph using `topological-sort`.

4.5.5. Operating on a System

`operate-on-system` calls `operate-on-component` to apply the operation to the system and its components. `operate-on-component` sets up the component's package, propagates the operations to the system's dependencies if `*operations-propagate-to-subsystems*` is `t` using `operate-on-system-dependencies`, and does the component's initializations. Then, if the component is of type `:file` or `:private-file` it applies the operation directly to the component. Otherwise, it calls `operate-on-components` to work on the components of the component. Finally, it does the component's finalizations and provides the system.

The function `compile-and-load-operation` corresponds to the `:load` operation while the function `load-file-operation` corresponds to the `:compile` operation. They use `needs-compilation` and `needs-loading` to determine if the component needs to be compiled or loaded based on its compile and load times. The compile time is checked by comparing the file-write-date of the binary file with that of the source file, while the load-time is cached in the component itself. The `delete-binaries-operation` function corresponds to the `:delete-binaries` operation, which deletes all the binary files associated with a system.

4.5.6. Querying the User with Timeouts

Since compiling and loading large systems can take a considerable amount of time, some users would prefer to avoid having to babysit the compilation. DEFSYSTEM includes a function `y-or-n-p-wait` which is similar to the Common Lisp `y-or-n-p` but which will time out after a specified interval of time. All queries from DEFSYSTEM to the user are through `y-or-n-p-wait` with reasonable defaults, allowing the user to eat dinner during the compilation without worrying whether the compilation hung up on a query a few seconds after the user left.

Some Lisps, however, have broken definitions of `read-char-no-hang` and `clear-input`, which can result in DEFSYSTEM's ignoring user input to the queries. Also, `get-internal-run-time` conses considerably in some Lisps,[11] with the result that `y-or-n-p-wait` conses several megabytes per minute. The variable `*use-timeouts*` has been provided to allow the user to turn off the timeout behavior of `y-or-n-p-wait`, in which case it works just like `y-or-n-p`.

`*use-timeouts*` t [Variable]

> If `t`, timeouts in `y-or-n-p-wait` are enabled. Otherwise it behaves like `y-or-n-p`. This is provided for users whose Lisps don't handle `read-char-no-hang` properly.

`*clear-input-before-query*` t [Variable]

> If `t`, `y-or-n-p-wait` will clear the input before printing the prompt and asking the user for input.

`y-or-n-p-wait` (&optional (default #\y) (timeout 20) format-string &rest args) [Function]

> `y-or-n-p-wait` prints the message, if any, and reads characters from `*query-io*` until the user enters `y`, `Y` or a space as an affirmative, or either `n` or `N` as a negative answer, or the timeout occurs. It asks again if you enter any other characters.

4.5.7. Debugging

The functions `files-which-need-compilation` and `files-in-system` may be useful for debugging an incorrect system definition.

`files-which-need-compilation` (system) [Function]

> Returns a list of files in *system* which currently need to be compiled to be brought up to date.

`files-in-system` (name &optional (force :all) (type :source) version &aux system) [Function]

> Returns a list of all files in the system named *name* in load order.

[11] 500 bytes per call is not unusual.

5. LOGICAL-PATHNAMES: Portable Pathnames

The LOGICAL-PATHNAMES system is a portable implementation of the X3J13 June 1989 specification for logical pathnames, as documented in [7, section 23.1.5]. LOGICAL-PATHNAMES lets programs refer to pathnames and files in a portable manner. The logical pathnames are mapped to physical pathnames by a set of implementation-dependent and site-dependent rules.

5.1. Overview

Logical pathnames allow large programs to be moved between sites by separating pathname reference from actual file location. The program will refer to files using logical pathnames. At each site, a user will specify a set of *translations* which map from the logical pathnames to the physical pathnames used on the device.

Logical pathnames provide a uniform convention for filesystem access, with the following properties:

1. *Pathname Portability.* The program specifies a pathname in a conventional format which may be mapped in a reasonably literal manner onto a variety of filesystems.
2. *Pathname Aliasing.* Logical pathnames introduce a level of indirection in pathname reference, so that the files may exist in different locations in the different filesystems. For example, the root directory might change. The translations make such a change easy to implement.
3. *Cross-host Access.* The files need not all exist on the same physical host, but may still be refered to as one logical unit.

This implementation of logical pathnames provides support for parsing and generating physical pathnames for UNIX, VMS/VAX, Symbolics Lisp Machines and TI Explorers, and is easily extended to handle additional platforms.

The LOGICAL-PATHNAMES system may be used with the DEFSYSTEM program.

5.1.1. Logical Pathname Syntax

Logical pathnames employ the following syntax:

```
[host:] [;] {directory ;}* [name] [. type [. version]]
```

where

```
host               ::= word
directory          ::= word | wildcard-word | wildcard-inferiors
name               ::= word | wildcard-word
type               ::= word | wildcard-word
version            ::= word | wildcard-word
word               ::= {letter | digit | -}*
wildcard-word      ::= [word] * {word *}* [word]
wildcard-inferiors ::= **
```

A wildcard-word of `*` parses as `:wild`; all others as strings. These definitions may be extended (e.g., `"newest"` parsing as `:newest`) by defining new canonical types.

5.1.2. Incompatibilities with the X3J13 Specification

The `logical-pathname` structure is not defined as a subclass of `pathname` since we have no guarantee about the format of `pathname` (i.e., whether it is a defstruct or class definition, what are the types of its slots, etc.).[12] Many Lisps will be able to replace the definition of `physical-pathname` with their definition of `pathname` by substituting the string "pathname" for "physical-pathname" and deleting and/or renaming some of the definitions in LOGICAL-PATHNAMES.

The X3J13 specification does not set a standard for the manner in which wildcards are translated. We use reversible wildcard pathname translation, similar to that used in the Symbolics logical pathnames.

5.2. Installing LOGICAL-PATHNAMES

Before loading LOGICAL-PATHNAMES, you may wish to perform the following implementation-dependent changes:

- Set `local-host-table` to the pathname of the local host table if you're using a host table to determine physical host types. Otherwise, you may wish to redefine the function `physical-host-type` to return the physical host types in an implementation-dependent manner. You may also wish to change the default physical host type.
- Set the value of `directory-structure-type` to match the type of the directory slot of pathname in your Lisp. This should only be necessary if you're porting it to a new Lisp.
- Set `*logical-translations-directory*` to be the pathname of the directory where translation files are kept.
- Define any additional canonical types and translation rules you wish.

After loading LOGICAL-PATHNAMES, load the physical host table using `load-physical-hostab` and any desired translations using `load-logical-pathname-translations`.

If you intend to use LOGICAL-PATHNAMES with DEFSYSTEM, you must load it before compiling or loading DEFSYSTEM.

---

[12] The latest version of Common Lisp tightened up the structure of pathnames, but we want to be compatible with current Lisps.

5.3. Using LOGICAL-PATHNAMES

This section discusses the basics of using LOGICAL-PATHNAMES, with an emphasis on differences between this implementation and the X3J13 specification. See [7, section 23.1.5] for detailed documentation on logical pathnames.

Most of the differences between this implementation and the X3J13 specification are either enhancements or due to the problems of trying to ensure compatability with current Lisps.

For example, nearly every Lisp has a different representation for pathnames, since this was rather loosely specified in [6]. In some Lisps pathnames are classes and in some they are structures, and the slots of a pathname may have arbitrary types, especially the directory slot. Depending on the Lisp, the directory slot may be a list, vector, simple-vector, string, keyword, and/or `nil`. If a list or a vector, the items in the list may be strings, keywords (for canonical types), or `nil`. The first item in the list may or may not be a special keyword (e.g., `:relative` or `:absolute`), with different keywords in different Lisps (e.g., some substitute `:root` for `:absolute`).[13]

5.3.1. Physical Host Types

Since the syntax of a pathname depends on the type of physical host, and such pathnames may be used in the translations,[14] LOGICAL-PATHNAMES needs to be able to determine the type of the physical host in order to translate a logical pathname. The function `physical-host-type` provides a mechanism for determining the host type of a physical host.

---

[13] This will be remedied somewhat by X3J13's June 1989 specification of the pathname component format for structured directories [7, section 23.1.3]. However, current Lisps do not yet comply with this vote.

[14] The X3J13 specification states that the to-wildnames used in the translations can be anything coercible to a pathname by application of the function `pathname`. However, this really leaves open the question of whether the to-wildnames must be written only in the syntax of the Lisp implementation's underlying operating system, or whether the to-wildnames may be in the syntax of the target physical host. For example, if the following translations are acceptable,

```
(setf (lp:physical-host-type "U") :unix)
(setf (lp:physical-host-type "MY-LISPM") :symbolics)
(setf (lp:logical-pathname-translations "prog")
      '(("RELEASED;*.*.*"     "U:/sys/bin/my-prog/*.*.*")
        ("EXPERIMENTAL;*.*.*" "MY-LISPM:>my-prog>*.*.*")))
```

then the Lisp implementation must be able to parse both UNIX and Symbolics pathnames. The second example in Section 5.4 which is taken from [7] seems to indicate that this is the case. On the other hand, Steele's example of a UNIX system that doesn't support `:wild-inferiors` would imply that the implementation of logical pathnames is relying on the underlying operating system to handle the translation of wildcards, and therefore the to-wildname must be acceptable to the underlying operating system.

In any event, since the intent is for LOGICAL-PATHNAMES to be portable, we parse several common pathname syntax formats and rely on the underlying operating system as little as possible. As a result, we need a mechanism for determining the host type of a physical host.

`physical-host-type` (host) [Function]

Returns a keyword that represents the host type of the physical host *host*.

`(setf physical-host-type)` (type) [Setf Mapping]

Sets the host type of the physical host *host* to *type*.

The function `load-physical-hostab` may be used to set the host types for a collection of physical hosts from a namespace table. The physical host namespace table is compatible with both VMS and Symbolics host tables. The host table consists of a series of lines, one per host, in the following format:

```
HOST NAME,CHAOS-#,STATUS,SYSTEM-TYPE,MACHINE-TYPE,NICKNAMES
```

Lines that don't begin with "`HOST`" are ignored. `NAME` and `SYSTEM-TYPE` are required; all others are optional (but delimiting commas are still required). `SYSTEM-TYPE` specifies the operating system run on the host. Common values are: `LISP`, `LISPM`, `UNIX`, `MACH`, `VMS`, and `EXPLORER`.

`local-host-table` "nethosts.txt" [Constant]

Default name of the local physical host namespace.

`load-physical-hostab` (&optional (local-hostab local-host-table)) [Function]

Loads the physical host namespace table. Can parse VMS and Symbolics host table formats.

If the Lisp implementation has a different mechanism for determining the host type of a physical host, the user should substitute a different definition for `physical-host-type`.

5.3.2. Logical Pathname Translations

The translations for a logical host are the main mechanism for transforming a logical pathname into a physical pathname.

A translation is a list consisting of a from-wildname and a to-wildname. The former is a logical pathname whose host is understood to be the logical host of the translation (i.e., the host of the from-pathname need not be explicitly specified in the translation). The latter is any pathname. If the to-wildname is a logical pathname, `translate-logical-pathname` will retranslate the result, repeatedly if necessary.

The translations are stored in a list according to host, and may be retrieved using the function `logical-pathname-translations` and set using `(setf logical-pathname-translations)`. Since translations are searched in the order listed, more specific from-wildnames must precede more general ones.

`logical-pathname-translations` (host) [Function]

If *host* has been defined as a logical pathname host name by `setf` of `logical-pathname-translations`, this function returns the list of translations for the specified host. Otherwise it signals an error.

`(setf logical-pathname-translations)` (translations) [Setf Mapping]

> `(setf (logical-pathname-translations host)` *translations*`)` sets the list of translations for the logical pathname host to *translations*. If host is a string that has not previously been used as a logical pathname host, a new logical pathname host is defined; otherwise an existing host's translations are replaced. Logical pathname host names are compared with string-equal.

5.3.3. Loading Logical Pathname Translations

Translations for a logical host may be loaded using the function `load-logical-pathname-translations`. If `*logical-translations-directory*` is defined, `load-logical-pathname-translations` will check for an appropriately named translations file in that directory.

`*logical-translations-directory*` () [Variable]

> Directory where logical pathname translations are stored.

`load-logical-pathname-translations` (host) [Function]

> Loads the logical pathname translations for host named *host* if the logical pathname translations are not already defined. First checks for a file with the same name as the host (lowercase) and type `"translations"` in the current directory, then the translations directory. If it finds such a file it loads it and returns `t`, otherwise it signals an error.

5.3.4. Additional Transformations

The function `translate-logical-pathname` may need to perform additional transformations on the pathnames, besides those specified by the translations. For example, the file system may require that pathnames include only uppercase letters, that hyphens not be used, or that filenames be of limited length. In addition, the user may want file types to be translated to local naming conventions. These additional transformations are implemented by translation rules and canonical types.

5.3.4.1. Translation Rules

Translation rules are used to change the case of a pathname, to substitute one character for another, and to replace particular directory components and file names. The macro `define-translation-rule` is used to define translation rules for a particular host.

`define-translation-rule` (host-type &key case char-mappings component-mappings version-case type-case name-case component-case) [Macro]

> Defines translation rules for hosts of type *host-type*. *case* may be `:unchanged` (unchanged), `nil` (use default case), `:upper`, `:lower`, or `:capitalize`.
> *char-mappings* is a list of character substitutions which occur in parallel.
> *component-mappings* is a list of string substitutions.

For example, the following rule changes VMS pathnames into uppercase and substitutes underscores for hyphens.

```
(define-translation-rule :vms
  :case :upper
  :char-mappings ((#\- #\_)))
```

5.3.4.2. Canonical Types

Canonical types are used to translate surface forms according to local naming conventions. For example, the filename extensions "`lsp`", "`lisp`" and "`l`" denote Lisp source files in different Lisps. The canonical type `:lisp` expresses the commonality among these surface forms.

The `define-canonical` macro may be used to define new canonical types. The functions `canonicalize` and `surface-form` may be used to convert to and from canonical types. For example, we may define `:wild` as the canonical type for "`*`" by evaluating

```
(define-canonical name :wild "*")
(define-canonical type :wild "*")
(define-canonical version :wild "*")
```

Then `(canonicalize "*" :unix 'type)` returns `:wild`. Note that we must define it once for each component of a pathname, whether pathname-type, pathname-version, pathname-name, or component of a directory.

`define-canonical` (level canonical default &body specs) [Macro]

> Defines a new canonical type. *level* specifies whether it is a canonical `type`, `version`, `name`, or `component`. *default* is a string containing the default surface type for any kind of host not mentioned explicitly. The body contains a list of specs that define the surface types that represent the new canonical type on each host. For systems with more than one possible default surface form, the form that appears first becomes the preferred form for the type.

`surface-form` (canonical host-type &optional (level (quote type))) [Function]

> Given the canonical form of some canonical type, replaces it with the appropriate surface form.

`canonicalize` (surface-form host-type &optional (level (quote type))) [Function]

> Given the surface form of some canonical type, replaces it with the appropriate canonical type.

5.3.5. Using Logical Pathnames

The LOGICAL-PATHNAMES system redefines several functions that use pathnames to first check if the host is a logical host, and if so, apply the translations for the host using `translate-logical-pathname`. The original function is then called on the translated pathname. Accordingly, the user rarely has to manually translate a logical pathname to the corresponding physical pathname, but may do so by calling `translate-logical-pathname` directly.

`translate-logical-pathname` (logical-pathname &optional (output-format *translation-output*)) [Function]

Translates a logical pathname to the corresponding physical pathname. The pathname argument is first coerced to a logical pathname, if possible. If the coerced argument is a logical pathname, the first matching translation (according to `logical-pathname-match-p`) of the logical pathname host is applied. If the result is a physical pathname it is returned, otherwise this process is repeated until the result is finally a physical pathname. If no translation matches a logical pathname, or the resolution process loops, an error is signaled. `translate-logical-pathname` may perform additional transformations, as specified by the translation rules and canonical types.

`logical-pathname-match-p` (logical-pathname from-pathname) [Function]

Returns `t` if the logical pathname matches the test pathname.

5.3.6. LOGICAL-PATHNAMES Variables

The variables in this section control the operation of LOGICAL-PATHNAMES.

`*translation-output*` :namestring [Variable]

Specifies whether the output of translate-logical-pathname should be a namestring (`:namestring`), a pathname made with `lisp:make-pathname` (`:pathname`), or as is (`:as-is`).

`*warn-about-host-type-collisions*` t [Variable]

Warn user when a logical host type definition collides with a physical host type definition.

5.4. Examples of Using LOGICAL-PATHNAMES

This section gives several examples of the use of logical pathnames. They are taken from [7, section 23.1.5.4].

The first example shows how to specify the root of the physical directory tree that corresponds to the logical pathnames. Note that we have to declare the type of the physical host "MY-LISPM".

```
(setf (lp:physical-host-type "MY-LISPM") :symbolics)
(setf (lp:logical-pathname-translations "foo")
      '(("**;*.*.*" "MY-LISPM:>library>foo>**>")))
```

When using a logical pathname, we can translate it with `translate-logical-pathname`.

```
<cl> (lp:translate-logical-pathname "foo:bar;baz;mum.quux.3" :namestring)
"MY-LISPM:>library>foo>bar>baz>mum.quux.3"
```

Many of the functions that use pathnames, such as `load` or `delete-file`, have been redefined to use `translate-logical-pathname` if the host of the pathname is a logical host. Note how `translate-logical-pathname` takes an additional argument (`:namestring` or

`:pathname`) to specify whether a namestring or actual pathname is returned.[15]

The next example illustrates splitting a logical host across two physical hosts and translating the type `.MAIL` to `.MBX`.[16] Since this UNIX file system doesn't support `:wild-inferiors` in the pathname directory, each directory level must be translated individually.[17]

```
(setf (lp:physical-host-type "U") :unix)
(setf (lp:physical-host-type "V") :vms)
(setf (lp:logical-pathname-translations "prog")
      '(("RELEASED;*.*.*"    "U:/sys/bin/my-prog/")
        ("RELEASED;*;*.*.*"  "U:/sys/bin/my-prog/*/")
        ("EXPERIMENTAL;*.*.*" "U:/usr/Joe/development/prog/")
        ("EXPERIMENTAL;DOCUMENTATION;*.*.*" "V:SYS$DISK:[JOE.DOC]")
        ("EXPERIMENTAL;*;*.*.*" "U:/usr/Joe/development/prog/*/")
        ("MAIL;**;*.MAIL"       "V:SYS$DISK:[JOE.MAIL.PROG...]*.MBX")))
```

Using these translations, we can obtain pathnames for either the UNIX or VMS physical hosts.

```
<cl> (lp:translate-logical-pathname "prog:mail;save;ideas.mail.3"
      :namestring)
"V:SYS$DISK:[JOE.MAIL.PROG.SAVE]IDEAS.MBX.3"
<cl> (lp:translate-logical-pathname "prog:experimental;spreadsheet.c"
      :namestring)
"U:/usr/Joe/development/prog/spreadsheet.c"
```

The last three examples demonstrate how logical pathnames may be used to shorten file names to conform with a file system with limited-length file names.

```
(setf (lp:logical-pathname-translations "prog")
      '(("CODE;*.*.*"    "/lib/prog/")))
<cl> (lp:translate-logical-pathname "prog:code;documentation.lisp"
      :namestring)
"/lib/prog/documentation.lisp"

(setf (lp:logical-pathname-translations "prog")
      '(("CODE;DOCUMENTATION.*.*"    "/lib/prog/docum.*")
        ("CODE;*.*.*"    "/lib/prog/")))
<cl> (lp:translate-logical-pathname "prog:code;documentation.lisp"
      :namestring)
"/lib/prog/docum.lisp"

(setf (lp:logical-pathname-translations "prog")
      `(("**;*.LISP.*"  ,(lp:logical-pathname "PROG:**;*.L.*"))
        ("**;*.FASL.*"  ,(lp:logical-pathname "PROG:**;*.B.*"))
```

[15]This is an extension to the X3J13 specification. When redefining functions that use pathnames, it was felt that providing a translated namestring would be safer than providing an actual pathname.

[16]The type translations could also be accomplished by defining `:mail` as a canonical type, `(define-canonical type :mail "MAIL" (:vms "MBX"))`. This is an extension to the X3J13 specification.

[17]This is not strictly true of the LOGICAL-PATHNAMES system. Since LOGICAL-PATHNAMES parses physical pathnames into a canonical format and can print pathnames in the formats of several Lisps, it may translate `:wild-inferiors` itself instead of relying on the filesystem. This is an extension to the X3J13 specification.

```
          ("CODE;DOCUMENTATION.*.*" "/lib/prog/documentatio.*")
          ("CODE;*.*.*"             "/lib/prog/")))
<cl> (lp:translate-logical-pathname "prog:code;documentation.lisp"
      :namestring)
"/lib/prog/documentatio.l"
```

5.5. Implementation Notes

The LOGICAL-PATHNAMES system can be divided into two major pieces. The first is parsing and generating the syntax of various pathname formats, and the second is the translation algorithm itself.

The parsing of the various pathname formats is straightforward. All of the associated operations involve breaking a string into two pieces around a character or string delimiter. The function `parse-generic-namestring` extracts the host from the namestring and uses it to determine the physical host type. Then `do-generic-pathname-parse` decides what parsing function to call based on the host type. This is where one would add new pathname types to LOGICAL-PATHNAMES. Most types of physical host have a similar pathname structure and may be parsed using `parse-generic-pathname`.

The function `physical-namestring` returns the appropriate surface form of a `physical-pathname` (the underlying structure that all pathnames are parsed into by LOGICAL-PATHNAMES) corresponding to its host type.

The function `translate-logical-pathname` calls the function `resolve-logical-pathname` to translate the logical pathname into a physical pathname. `resolve-logical-pathname` calls `map-logical-pathname` to retrieve and apply a single translation pair to the logical pathname. If the result is a physical pathname it is returned. If the result is a logical pathname, `resolve-logical-pathname` calls itself recursively. A table of previously seen logical pathnames, `*circularity-check-table*`, is maintained to prevent infinite loops. `resolve-logical-pathname` calls `check-logical-pathname` to check and update this table, signalling an error if a logical pathname is repeated.

The function `map-logical-pathname` iterates down the list of translation pairs for the logical host, stopping with the first translation pair whose from-wildname matches the logical pathname according to `logical-pathname-match-p` and returning the result of `translate-logical-pathname-aux` being called on the logical pathname and translation pair. `translate-logical-pathname-aux` uses the functions `map-directories` and `map-wildcard-word` to do the translation.

`map-wildcard-word` calls `map-strings` to transform individual strings. `map-strings` translates a string from the source wild-string to the target wild-string. It assumes that wildcards (`"*"`) in the source wild-string will correspond to wildcards in the target wild-string, and replaces wildcards in the target pattern with the string's contents as specified by the corresponding wildcard in the source wild-string. Literal strings are copied as is from source wild-string to target wild-string. When not enough matching wildcards are available due to too few asterisks in the source wild-string, the null string is used as the matching value for any wildcards remaining in the target wild-string. When the source wild-string has too many wildcards, the first extra wildcard and everything following it are ignored. The operation of `map-directories` with respect to the `:wild` and `:wild-inferiors` wildcards is analogous.

The function `append-logical-directories` is provided to tack a subdirectory onto a logical pathname. It is used by the DEFSYSTEM program.

6. SOURCE-COMPARE: A "diff" for Lisp

The SOURCE-COMPARE system is a portable tool for comparing Lisp source files. While it may be used to find the differences between arbitrary text files, it has several features customized for Lisp, such as the ability to ignore Lisp comments. It uses a greedy algorithm for longest common substring that may not necessarily find the longest common substring, but which runs in average case linear time and works well in practice.

6.1. Overview

SOURCE-COMPARE is a portable Common Lisp tool for comparing Lisp source files, similar in functionality to the UNIX program "diff". Like diff it can ignore case, whitespace, and blank lines. In addition, it can also correctly ignore certain classes of Lisp comments. It uses a different algorithm from diff, and runs in average-case $O(m+n)$ time, where m and n are the lengths in lines of the files being compared.

The algorithm is a greedy variation on the usual dynamic programming implementation of the algorithm for finding the longest common substring of two strings. When comparing two files, SOURCE-COMPARE tries to maintain the two files in sync, and when a difference is encountered, uses the closest next match, where distance is minimized according to some metric. Since this is a greedy algorithm, it is possible that it will not find the optimum global match sequence. However, the suboptimal case hardly ever occurs in practice, and when it does occur, it doesn't make much of a difference for comparing different versions of source files.

The metrics should be chosen so that minimizing distance is equivalent to minimizing the edits necessary to bring the two files into agreement. Two such metrics include

- x + y, the total length of additions and deletions from both files
- max(x,y), the length of the largest addition or deletion from either file

where x is a line number from the first file and y is a line number from the second file. Both of these metrics are appropriate to the problem, since the former tries to minimize the total changes and the latter gives a preference to small changes.

While neither metric actually builds the dynamic programming table, they can be considered as exploring the table in successive rectilinear and diagonal layers, respectively. The metrics are illustrated in Figure 6-1. Both metrics have been implemented.

If the two files have no lines in common, we get a worst-case running time of $O(mn)$, where m is the length in lines of the first file and n the length in lines of the second file. In practice, however, the algorithm seems to always run in linear time.[18] We show in Section 6.4 that the algorithm has an average case running time of $O(m+n)$. The diagonal metric seems to run

[18]Presumably because the files one compares tend to have many lines in common.

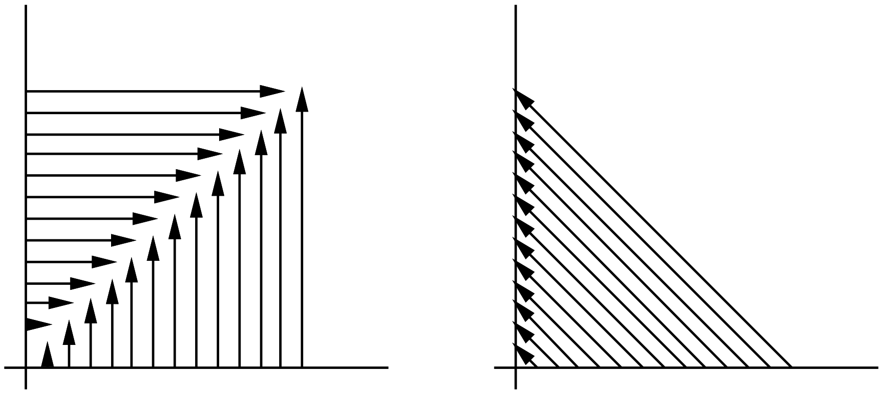

Figure 6-1: Two Greedy Metrics

slightly faster and use less space than the rectilinear metric, so it has been made the default.

6.2. Using SOURCE-COMPARE

This section describes all of the basic SOURCE-COMPARE commands and the variables which control their behavior.

6.2.1. Comparing Files

`source-compare` is the main function for comparing files. The variable `*greedy-metric*` contains the name of the greedy metric used to calculate the closest next match.

`source-compare` (filename-1 filename-2 &key [Function]
(output-stream *standard-output*)
(ignore-case *ignore-case*)
(ignore-whitespace *ignore-whitespace*)
(ignore-comments *ignore-comments*)
(ignore-blank-lines *ignore-blank-lines*)
(print-context *print-context*)
(print-fancy-header *print-fancy-header*))

Compares the contents of the two files, outputting a report of what lines must be changed to bring the files into agreement. The report is similar to that generated by 'diff': Lines of the forms

```
n1 a n3,n4
n1,n2 d n3
n1,n2 c n3,n4
```

(where a is for *add*, d is for *delete*, and c is for *change*) are followed by the lines affected in the first (left) file flagged by '<' then all the lines affected in the second (right) file flagged by '>'. If *print-context* is `t`, will print out some additional contextual information, such as additional lines before and after the affected text and the definition most likely to be affected by the changes. If *print-fancy-header* is `t`, prints the `file-author` and `file-write-date` in the header. The report is output to *output-stream*. Returns `t` if the files were "identical", `nil` otherwise. If *ignore-case* is `t`, uses a case insensitive comparison. If *ignore-whitespace* is `t`, ignores spaces and tabs that occur at the beginning of the line. If *ignore-comments* is `t`, tries to ignore comments at the end of the line. If `*dont-ignore-major-comments*` is `t`, will also ignore major comments (comments with a semicolon as the first character of the line). If *ignore-blank-lines* is `t`, will ignore blank lines in both files, including lines that are effectively blank because of ignored comments.

`*greedy-metric*` (quote find-next-diagonal-match) [Variable]

Variable containing the name of the greedy matching function used to minimize distance to the next match:

- `find-next-rectilinear-match` minimizes `max(x,y)`
- `find-next-diagonal-match` minimizes `x+y`

where x is a line number from the first file and y is a line number from the second file.

`find-next-diagonal-match` (file-1 start-1 file-2 start-2) [Function]

First difference detected, look ahead for a match [x+y version].

`find-next-rectilinear-match` (file-1 start-1 file-2 start-2) [Function]

First difference detected, look ahead for a match [max(x,y) version].

6.2.2. SOURCE-COMPARE Variables

The following four variables control the appearance of the report on the differences between the files.

`*print-context*` t [Variable]

If `t`, we print the context marking lines that occur before the difference.

`*print-fancy-header*` () [Variable]

If `t`, prints a fancy header instead of the simple one.

`*context-lines-before-difference*` 0 [Variable]

Number of lines of context to print before a difference.

`*context-lines-after-difference*` 1 [Variable]

Number of lines of context to print after a difference.

The next variable controls whether small changes close together are merged into a larger group.

`*minimum-match-length*` 2 [Variable]

The minimum number of lines that must match for it to be considered a match. This has the effect of collecting lots of adjacent small differences together into one larger difference.

The next five variables control sensitivity to whitespace, case, blank lines, and comments.

`*ignore-whitespace*` t [Variable]

If `t`, will ignore spaces and tabs that occur at the beginning of the line before other text appears and at the end of the line after the last text has appeared.

`*ignore-case*` t [Variable]

If `t`, uses a case insensitive comparison. Otherwise uses a case sensitive comparison.

`*ignore-blank-lines*` t [Variable]

If `t`, will ignore blank lines when doing the comparison.

`*ignore-comments*` t [Variable]

If `t`, will try to ignore comments of the semicolon variety when comparing lines. Tries to be rather intelligent about the context to avoid ignoring something that really isn't a comment. For example, semicolons appearing within strings, even multi-line strings, are not considered comment characters. Uses the following heuristics to decide if a semicolon is a comment character or not:

- Slashification (\) works inside strings ("foo\"bar") and symbol names (|foo\|bar|), but not balanced comments (#|foobar\|#).
- Balanced comments do not work inside strings ("#|") or symbol names.
- Strings do not work inside balanced comments (#|"|#)
- Regular semicolon comments do not work inside strings, symbol names, or balanced comments (#|foo;bar|#).

All this is necessary for it to correctly identify when a semicolon indicates the beginning of a comment. Conceivably we should consider a semicolon as a comment when it is inside a balanced comment which isn't terminated from the semicolon to the end of the line. However, besides being complicated and time-consuming to implement, the Lisp interpreter doesn't treat it this way, and we like to err on the side of caution. Anyway, changes in the comments within commented out regions of code is worth knowing about.

`*dont-ignore-major-comments*` () [Variable]

If `t`, ignoring comments does not ignore comments with a semicolon as the first character of the line.

6.3. Example of Using SOURCE-COMPARE

SOURCE-COMPARE is loaded into the "SOURCE-COMPARE" package, so we prefix the functions and variables with a "SOURCE-COMPARE:" or the nickname "SC:".

The following example shows what the output of the source comparison program looks like.

```
<cl> (SC:source-compare "~/old/glinda.lisp" "glinda.lisp" :ignore-comments t)

============================================================================
Source compare of
     ~/old/glinda.lisp
     (written by mkant, FRI 20-JUL-90 11:59:05)
  with
     glinda.lisp
     (written by mkant, THU 15-NOV-90 15:53:44)
============================================================================
46c46
**** File ~/old/glinda.lisp, After "(defvar *glinda-version* nil)"
< (setq *glinda-version*  "6/19/90")
```

```
< (format t "~%Using Glinda Generation, Generator Version ~A." *glinda-version*)
---
**** File glinda.lisp, After "(defvar *glinda-version* nil)"
> (setq *glinda-version*  "11/13/90")
> (format t "~%Using Glinda Generation, Generator Version ~A." *glinda-version*)
===========================================================================
550c550
**** File ~/old/glinda.lisp, After "(defun constraint-match (cvalue gvalue)"
<        ((or (symbolp cvalue) (numberp cvalue))
<         (ontological-supertypep cvalue gvalue))
---
**** File glinda.lisp, After "(defun constraint-match (cvalue gvalue)"
>        ((or (symbolp cvalue) (stringp cvalue)(numberp cvalue))
>         (ontological-supertypep cvalue gvalue))
===========================================================================
562a563,567
**** File ~/old/glinda.lisp, After "(defun constraint-match (cvalue gvalue)"
< (defun find-organization (head type features &optional group)
---
**** File glinda.lisp, After "(defun constraint-match (cvalue gvalue)"
> (defvar *which-rule-to-choose* :random ; &new11/13/90
>   "If find-rule returns more than one rule, specifies which rule we use.
>    :first   -- just take the first rule.
>    :random  -- pick a rule at random.")
>
> (defun find-organization (head type features &optional group) ; &mod11/13/90
===========================================================================
565c570,580
**** File ~/old/glinda.lisp, After "(defun find-organization (head type features &op
<      (car (find-rule (lexical-organization category type) features group))))
<
---
**** File glinda.lisp, After "(defun find-organization (head type features &optional
>      (let ((rules (find-rule (lexical-organization category type)
>                              features group)))
>        (case *which-rule-to-choose*
>        (:first  (car rules))
>        (:random (choose-random rules))))))
>
> (defun choose-random (list) ; &new11/13/90
>   "Chooses a random element of the list."
>   (if (null (cdr list))
>       (car list)
>     (nth (random (length list)) list)))
>
===========================================================================
Done.
```

6.4. Proof of Average Case Linear Running Time

We prove that SOURCE-COMPARE runs in average case linear time.

Let a_i and b_i be the ith distances between matches in files A and B, respectively. Let k, $1 \le k \le n$, be the number of matches. Then $\sum_{i=1}^{k} a_i = m$ and $\sum_{i=1}^{k} b_i = n$, where m is the length in lines of file A and n is the corresponding length for file B. The running time of the algorithm is proportional to $\sum_{i=1}^{k} a_i b_i$.

Since a_i and b_i are positive integers, it follows that

$$\sum_{i=1}^{k} a_i b_i \le \sum_{i=1}^{k} a_i \sum_{i=1}^{k} b_i = m\,n$$

and hence the worst-case running time is $O(mn)$. But the worst-case running time is atypical of the average-case behavior. As we shall show, the average-case running time is $O(m+n)$.

Combining the Cauchy-Schwartz inequality[19]

$$\sum_i a_i b_i \le \sqrt{\sum_i (a_i)^2}\ \sqrt{\sum_i (b_i)^2}$$

with the arithmetic-mean geometric-mean inequality

$$\sqrt{x \cdot y} \le \frac{x+y}{2}$$

yields

$$\sum_i a_i b_i \le \frac{\sum_i (a_i)^2 + \sum_i (b_i)^2}{2}$$

So it suffices to consider the average value of $\sum_{i=1}^{k} (r_i)^2$ over all possible ordered sequences r_i of positive integers for $k=1$ to n such that $\sum_{i=1}^{k} r_i = n$. Such a sequence is called a composition of n into k distinct parts.[20]

To compute this average we sum the squares of the parts of the compositions of n, and divide by the total number of such compositions. We shall show that the former is equal to $(3n-4)2^{n-1} + 2$ and the latter to 2^{n-1}, and hence that the average is equal to $3n-4+2^{-(n-2)}$.

The number of occurrences of part i in the k-compositions of n is the same as the number of $(k-1)$-compositions of $n-i$ multiplied by k, the number of positions in which i could be inserted to form a k-composition of n. To see that the former is $\binom{n-i-1}{k-2}$, consider $n-i$ dots separated by $(n-i)-1$ spaces, and choose $(k-1)-1$ of them to form $k-1$ integers. This gives us $k\binom{n-i-1}{k-2}$ occurrences of i in the k-compositions of n.

Thus $f(n,k)=\sum_{i=1}^{n} \sum_{j=1}^{n} i^k j\binom{n-i-1}{j-2}$. Substituting $j\binom{n-i-1}{j-2} = (n-i-1)\binom{n-i-2}{j-3} + 2\binom{n-i-1}{j-2}$ yields

---

[19]One sentence proof: Given vectors $\vec{a}$ and $\vec{b}$, $\vec{a}\cdot\vec{b} = \|\vec{a}\|\,\|\vec{b}\|\cos\theta \le \|\vec{a}\|\,\|\vec{b}\|$, with equality when $\vec{a}$ and $\vec{b}$ are parallel ($\cos\theta = 1$).

[20]The word distinct here signifies that permutations of a sequence are not considered identical -- the cells are distinct. A composition of n is an *ordered* sequence of positive integers whose sum is equal to n. The elements of the sequence are called parts. A composition with exactly k parts is called a k-composition. For example, there are sixteen compositions of 5:

(5)
(1,4) (4,1) (3,2) (2,3)
(1,1,3) (1,3,1) (3,1,1) (1,2,2) (2,1,2) (2,2,1)
(1,1,1,2) (1,1,2,1) (1,2,1,1) (2,1,1,1)
(1,1,1,1,1)

$$f(n,k) = \sum_{i=1}^{n-2} i^k (n-i-1)\, 2^{n-i-2} + \sum_{i=1}^{n} i^k\, 2^{n-i}$$

.

For k=2, substituting using $j=n+1-i$ and using the identities

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$$

$$\sum_{i=1}^{n} i\, 2^i = (n-1)\, 2^{n+1} + 2$$

$$\sum_{i=1}^{n} i^2\, 2^i = (n^2-2n+3)\, 2^{n+1} - 6$$

$$\sum_{i=1}^{n} i^3\, 2^i = (n^3-3n^2+9n-13)\, 2^{n+1} + 26$$

yields $(3n-4)2^{n-1} + 2$ as desired.

Note that since there are $\binom{n-1}{k-1}$ k-compositions of n, the total number of compositions is $\sum_{k=1}^{n} \binom{n-1}{k-1} = 2^{n-1}$.

Thus the average value of the sum of the squares of the parts of the compositions of n is equal to $3n-4+2^{-(n-2)}$. Thus

$$\sum_i a_i b_i \le \frac{\sum_i (a_i)^2 + \sum_i (b_i)^2}{2} = \frac{3\,(n+m)}{2} - 4 + 2^{-(n-1)} + 2^{-(m-1)}$$

on average, and hence the average case running time is $O(m+n)$.

6.5. Implementation Notes

SOURCE-COMPARE has four major pieces: line comparison, the outer iteration loop that keeps the files in sync, the inner iteration loops that find the next match, and the report generator.

6.5.0.1. Line Comparison

The function `compare-lines` is used to compare a line from each file. It uses `line-start` to find the positions in each line where it should begin comparing them, and `line-end` to find the positions where it should stop comparing them. `line-start` and `line-end` use `first-non-whitespace-char` to find the position in the line where the whitespace ends and begins, respectively. `line-end` calls `get-comment-position` to find the comment position for the current line, if any, given the cached position information for the previous line. `get-comment-position` calls `find-comment-position` to actually determine where in the line the comment begins, if at all.

`compare-lines` (file-1 line-no-1 file-2 line-no-2) [Function]

> Intelligently compare two lines. If `*ignore-case*` is `t`, uses case-insensitive comparison. If `*ignore-whitespace*` is `t`, ignores spaces and tabs at the beginning of the line. If `*ignore-comments*` is `t`, tries to ignore comments at the end of the line.

`line-end` (line file line-no &optional (start 0) end) [Function]

> Returns the position of where in *line* to end the comparison. If the comparison should end at the end of the line, returns `nil`. *start*, if supplied, is where to start looking for the end.

`line-start` (line &optional (start 0)) [Function]

> Returns the position of where in *line* to start the comparison.

`first-non-whitespace-char` (line &key from-end (start 0) end) [Function]

> Finds the position of the first character of *line* which is neither a space or a tab. Returns `nil` if no character found.

`get-comment-position` (line file line-no &optional (start 0) end) [Function]

> Returns the position of the beginning of the semicolon variety comment on this line.

The function `find-comment-position` has been carefully constructed to return the correct position of the comment character, despite the many interactions of slashification, delimited strings, delimited symbol names, balanced comments, and regular comments. For example, a semicolon appearing inside a documentation string should not be counted as the beginning of a comment. As another example, a slashified semicolon should not count as a comment unless the slash is itself slashified.

`find-comment-position` (line &optional (start 0) end &key inside-string (splat-bar-count 0)) [Function]

> Tries to find the position of the beginning of the comment at the end of *line*, if there is one. *start* and *end* delimit the search. *end* defaults to the end of the line. If *inside-string* is non-nil, it is assumed that we're inside a string before we began (if so, *inside-string* is set to the character which will terminate the string (`#\"` or `#\|`). *splat-bar-count* is the number of unbalanced begin balanced comments (`#|`) that have been seen so far.

6.5.0.2. Outer Iteration Loop

The function `source-compare-internal` maintains indices into both files, always positioned so that they correspond to a match. If the next pair of lines are a mismatch, it calls the current metric (`*greedy-metric*`) to find the indices of the lines where the files match up again. It then generates a report for the mismatch using `print-differences`, and continues from where they match until it reaches the end of the files.

6.5.0.3. Finding the Next Match

The function `find-next-diagonal-match` explores successive diagonals of the dynamic programming table in order of increasing sum x+y. It calls `find-diagonal-match` to explore the diagonal from top to bottom, possibly truncating it at the ends of the table.

The function `find-next-rectilinear-match` explores successive rectangular layers of the dynamic programming table, calling `find-linear-match` alternately to explore horizontal and vertical layers.

Both `find-next-diagonal-match` and `find-next-rectilinear-match` call `found-match` to verify that a match has been found by checking that the next few lines (up to `*minimum-match-length*`) are identical. If `found-match` returns `nil`, this has the effect of clumping together differences separated only by a few matching lines. If a definition changed between the files, it is sometimes preferable to be given the entire definition as a change than a lot of small slices of the definition.

6.5.0.4. Report Generator

The function `print-differences` prints the differences in the two files. It prints a one line summary of the change in a format similar to diff, giving the ranges of lines from each file, and using a single letter (a, d, or c) to indicate additions, deletions and changes, respectively. It then prints out the appropriate section of each file, possibly with a few lines before and after to give context. Also for context, it hunts backwards in the file until it finds the nearest line that begins a definition (left parenthesis on column zero) and prints that line.

7. USER-MANUAL: Extracting Program Documentation

The USER-MANUAL program is a portable tool for extracting documentation from Lisp source code. It helps create user guides and program documentation.[21]

7.1. Overview

USER-MANUAL reads in the source code from a Lisp program, extracts the function name, argument list, and documentation string, and formats it either for use as a Lisp comment or for use in a Scribe document.

USER-MANUAL can format documentation for several types of definition forms, including functions, macros, variable definitions, defstructs, class and method definitions, and defsetf forms. It is easy to add documentation handlers for new types of definition forms.

7.2. Using USER-MANUAL

The function `create-user-manual` is the main routine for generating the documentation for the definitions of a program.

`create-user-manual` (filename &key (output-format (quote text)) (output-stream *standard-output*)) [Function]

> Automatically creates a user manual for the functions in a file by collecting the documentation strings and argument lists of the functions and formatting the output nicely. Returns a list of the definition types of the forms it couldn't handle.
> *output-format* may be either `'text` or `'scribe`.

7.3. An Example of Using USER-MANUAL

The definition entry in Section 7.2 was generated by evaluating

```
(create-user-manual "user-manual.lisp" :output-format 'scribe)
```

The following is the same entry, but in `'text` format:

```
;;;
;;; CREATE-USER-MANUAL (filename &key (output-format (quote text))   [FUNCTION]
;;;                      (output-stream *standard-output*))
;;;    Automatically creates a user manual for the functions in a file by
;;;    collecting the documentation strings and argument lists of the
;;;    functions and formatting the output nicely. Returns a list of the
;;;    definition types of the forms it couldn't handle. Output-format
;;;    may be either 'TEXT or 'SCRIBE.
;;;
```

[21]The documentation in this user guide was created using the USER-MANUAL program.

7.4. Extending USER-MANUAL

The macro `define-doc-handler` is used to define a new documentation handler. For example, the documentation handler for `defvar` was defined as follows:

```
(define-doc-handler defvar (form)
  "variable"
  (values (second form)
          (third form)
          (fourth form)))
```

Definitions with more complex syntax, such as `defmethod` have correspondingly more complex documentation handlers.

`define-doc-handler` (definer arglist description &body body) [Macro]

> Defines a new documentation handler. *definer* is the car of the definition form handled (e.g., defun), *description* is a one-word string equivalent of definer (e.g., "function"), and *arglist* and *body* together define a function that takes the form as input and value-returns the name, argument-list, documentation string, and a list of any qualifiers of the form.

7.5. Implementation Notes

The only complicated aspect of USER-MANUAL is the formatting of the argument lists. If Waters' XP Lisp pretty printer [9] [10] is present in the Lisp environment USER-MANUAL uses it to format the argument lists. If not, USER-MANUAL uses several heuristics for formating the argument lists nicely.

The function `split-string` is used to break up both long argument lists and lines of documentation that are too wide. It calls the functions `lambda-list-keyword-position`, `split-point`, `balanced-parenthesis-position`, and `parse-with-delimiter`. The basic idea is to split the argument list so that it fits on the line, and walk backwards to the first balanced parenthesis on the line, unless it's the first character on the line. Then it checks whether the previous "word" is a lambda-list keyword, and if so splits the argument list just before the keyword, otherwise at the balanced parenthesis position.

`split-string` (string width &optional arglistp filled (trim-whitespace t)) [Function]

> Splits a string into a list of strings, each of which is shorter than the specified width. Tries to be intelligent about where to split the string if it is an argument list. If *filled* is `t`, tries to fill out the strings as much as possible. This function is used to break up long argument lists nicely, and to break up wide lines of documentation nicely.

`split-point` (string max-length &optional arglistp filled) [Function]

> Finds an appropriate point to break the string at given a target length. If arglistp is `t`, tries to find an intelligent position to break the string. If filled is `t`, tries to fill out the string as much as possible.

`lambda-list-keyword-position` (string &optional end trailer-only) [Function]

If the previous symbol is a lambda-list keyword, returns its position. Otherwise returns end.

`balanced-parenthesis-position` (string &optional end) [Function]

Finds the position of the left parenthesis which is closest to *end* but leaves the prefix of the string with balanced parentheses or at most one unbalanced left parenthesis.

`parse-with-delimiter` (line &optional (delim #\newline)) [Function]

Breaks *line* into a list of strings, using *delim* as a breaking point.

Appendix I
Test Source File for XREF

The following is a short nonsense program used to test XREF and produce the output in Section 2.4. It may be found in the file `xref-test.lisp`.

```
(defun top-level ()
  "Top level function with null lambda list."
  (let* ((input (read))
         (key (car input)))
    (declare (special key))
    (case key
      (quit
       (return (values (frob (rest input)) ’quit)))
      (otherwise
       (cond ((member key ’(foo bar baz))
              (barf key (rest input)))
             (t
              (frowz (rest input) :key key)))))))

(defun frob (items)
  "Here we test mapcar."
  (mapcar #’frob-item items))

(defun frob-item (item)
  "Here we test apply."
  (apply #’append-frobs item))

(defun barf (key &optional items)
  "Optional args test."
  (cons key (frowz items)))

(defun frowz (items &key key)
  "Keyword args test."
  (dolist (item items)
    (let ((frowz
           (if (eq key ’quit)
               (intern
                (format nil "FOO~A"
                        (round (+ (length (process-keys items))
                                  10))) ’keyword)
               (snarf-item item))))
      (when (string-equal frowz (process-key key))
        (setf (node-position key) 12)
        (return frowz)))))

(defun process-key (key)
  (funcall #’symbol-name-key key))
```

Appendix II
Extensions to Common Lisp

In the course of writing these utilities, often there were implementation-dependent functions which represent functionality that is missing from Common Lisp. This appendix lists some of those functions.

`arglist` (symbol) [Function]

Returns the argument list of *symbol*.

`append-directories` (absolute-pathname relative-pathname) [Function]

Tacks a subdirectory onto a directory. Returns the pathname *absolute-pathname* with the components of the directory of *relative-pathname* appended onto the end of its directory.

`space` () [Function]

Value returns three numbers relating to memory usage. The first is the number of bytes of dynamic storage currently allocated. The second is the amount of space remaining. The third is the total number of bytes consed since time zero (alternately, since the first time `space` was called, with the first time returning zero).

The definition of `room` is inadequate because it is implementation dependent and lacks a convenient interface for programs. Having to call `parse-integer` on the output of `(room nil)` is unacceptable.

The macro `defsetf` currently restricts the setf method to a single store variable. If we modify `defsetf` to allow multiple store variables, with assignment via multiple values (e.g., `(setf (frob x) (values 1 2))`), then `get-setf-method-multiple-values` can be removed from the language.

Some Lisps buffer the input lines at read-eval-print loop prompt. This interferes with the desired operation of `listen` and `read-char-no-hang`, since they should not have to wait until the user hits a carriage return and linefeed to get their input. Perhaps Common Lisp should include a `with-unbuffered-reading` macro. This macro could put the tty in RAW or CBREAK mode to allow unbuffered reading, and back to COOKED mode afterwards.

Common Lisp currently avoids discussing memory management and garbage collection. A set of naming conventions for the basic gc functions for Lisps that involve garbage collection would be helpful.

Common Lisp should specify more of the keywords that should appear in the *features* list. For example, each Lisp implementation should have symbols that distinguish it from other Lisps and distinguish major versions of the implementation. Major subsystems such as CLOS, LOOP, SERIES, etc., should have associated keywords.

Miscellaneous minor functions:

- `firstn` returns the list containing the first n elements of its argument.

- subst:sublis::substitute:? Add a definition `parallel-substitute` for performing many substitutions on a sequence in parallel.
- Equivalents of `last` and `butlast` for sequences.
- `userid` and `username` return the user's id and name, if available.
- `copy-file` to make a copy of a file.
- `create-directory` to create a new directory.

It is unfortunate that a portable DEFSYSTEM facility must be file-based. Nothing in the definition of the Lisp language requires that definitions be stored in files, but there seems to be an implicit assumption that this is so. In Common Lisp one may either compile an entire file or an individual definition, but there is no mechanism for compiling a single definition and saving its compiled code in a file. This imposes artificial constraints on a system like DEFSYSTEM. If instead Lisp definitions and compiled code were stored in a database, one could still edit the definitions using a text editor, but the compiler would be able to ensure that the compiled code in the database is up to date on a package by package (or even defun by defun) basis.

References

[1] Feldman, S. I.
Make - A Program for Maintaining Computer Programs.
Software - Practice and Experience 9(3):255-265, March, 1979.

[2] Masinter, Larry M.
Global Program Analysis in an Interactive Environment.
PhD thesis, Stanford University, 1980.

[3] Moon, David, Stallman, Richard, and Weinreb, Daniel.
Lisp Machine Manual
6th edition, MIT AI Laboratory, Cambridge, Massachusetts, June 1984.

[4] *Program Development Utilities, Volume 4*
Symbolics, Cambridge, MA, August 1986.

[5] Robbins, Richard E.
BUILD: a tool for maintaining consistency in modular systems.
AI Memo 874, MIT AI Laboratory, Cambridge, Massachusetts, 1985.

[6] Steele, Guy L. Jr.
Common LISP: The Language.
Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

[7] Steele, Guy L. Jr.
Common LISP: The Language; 2nd Edition.
Digital Press, 30 North Avenue, Burlington, MA 01803, 1990.

[8] *User's Guide to Symbolics Computers, Volume 1*
Symbolics, Cambridge, MA, July 1986.

[9] Waters, Richard C.
PP: A Lisp Pretty Printing System.
AI Memo 816, MIT AI Laboratory, Cambridge, Massachusetts, 1984.

[10] Waters, Richard C.
XP: A Common Lisp Pretty Printing System.
AI Memo 1102, MIT AI Laboratory, Cambridge, Massachusetts, March, 1989.

Table of Contents

List of Figures