Defsystem: A Portable Make Facility for Common Lisp

by Mark Kantrowitz

School of Computer Science

Carnegie Mellon University

January 1990

Introduction

This document describes the defsystem system definition macro, a portable "Make" facility for Common Lisp.

Support is provided for organizing systems into hierarchical layers of modules, with matching directory structure. Moreover, the components of the system may be listed in any order the user desires, because the defsystem command 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 dependencies are not supported (the graph must be a DAG).

Only two operations, compile and load, are currently defined. The interface for defining new operations, however, is simple and extendible.

Though home-grown, the syntax was inspired by fond memories of the defsystem facility on Symbolics 3600 series lisp machines. The exhaustive lists of filename extensions for various lisps and the idea to have one *operate-on-system* function instead of separate *compile-system* and *load-system* functions were taken from Xerox Corporation's PCL (Portable Common Loops) system.

The code for the defsystem facility and this documentation may be found in the files
/afs/cs.cmu.edu/user/mkant/Defsystem/defsystem.{text,lisp}

Please send bug reports, comments and suggestions to mkant@cs.cmu.edu.

Using the System

To use this system,

1. If you want to have a central directory where system definition files will be kept, modify the value of *central-registry* in defsystem.lisp to be the pathname of that directory.
2. Save and load the file defsystem.lisp
3. Load the file containing the defsystem form defining your system. If the name of your system is foo, the file will be named "foo.system". [If you are going to load the system and not compile it, you can use (require "foo") to load it if the definition file is in either the current directory or the central registry.]
4. Use the function *operate-on-system* to do things to your system. For example (operate-on-system "foo" 'load) will load the system, while (operate-on-system "foo" 'compile) will compile it.

External Interface

The external interface to the defsystem facility are 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 definition of require has been modified to mesh well with systems defined using defsystem, and is fully backward-compatible.

In addition, the function afs-binary-directory has been provided for immitating 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.). The afs-binary-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.

Defining Systems with Defsystem

A system is a set of components with associated properties. These properties include the type, name, source and binary pathnames, package, component-dependencies, initializations, and a set of components.

Components may be of three types: :system, :module, or :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 :module have pathnames that are relative to their containing system or module, and may contain a set of files and modules. This enables one to define subsystems, subsubsystems, submodules, and so on.

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. [In some future version of the defsystem facility there will be new component types corresponding to each possible operation.] This is useful if it isn’t possible to convert these systems to the defsystem format all at once.

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 under the system properties of the symbol corresponding to its uppercase 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). It is usually best to use the string version of a system’s name when defining or referring to it. 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".

The absolute (for components of type :system) and relative (for all other components) pathnames of the binary and source files may be specified using the :source-pathname and :binary-pathname keywords in the component definition. 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.

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

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

One may also specify the package to be used and any initializations and finalizations. 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.

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 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 of the hierarchy as the referring component. This enforces the modularity of the defined system. If module A depends on a file contained within module B, then module A depends on module B and should be specified as such. Any other use would make a mockery of the concept of modularity. This requirement is not enforced in the software, but any use contrary to it will produce unpredictable results.

Thus the only requirement in how the files are 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 the 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. Currently, however, it loads them in serial order. In a future version of defsystem this will probably become a supported feature. [In other words, this feature hasn't been tested to make sure that they files are not accidentally loaded in the opposite order in some cases. It all depends on whether the definition of topological-sort used is a stable sort or not.]

The basic algorithm used is to topologically sort the DAG at each level of abstraction (system level, module level, submodule level, etc.) to insure 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, since it probably saves the user some time.

BNF for Components

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>*)]
                                [:compile-form <form>]
                                [:load-form <form>])
<type> ::= :system | :module | :file
```

The toplevel defsystem form substitutes defsystem for :system.

Using Systems with Operate-on-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 added easily.

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

OPERATE-ON-SYSTEM *system-name operation* &key *force test verbose dribble load-source-instead-of-binary load-source-if-no-binary bother-user-if-no-binary*

- **system-name** is the name of the system and may be a symbol or string.
- **operation** is ’compile (or :compile) or ’load (or :load) or any new operation defined by the user.
- **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-sources-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)
- **dribble** should be the pathname of a dribble file if you want to keep a record of the compilation. (default 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.

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.

Files which must not be compiled should be loaded in the initializations or finalizations of a module by means of an explicit load form.

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.

The only 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.)

Defining New Operations

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’s the definition of the ’compile and :compile operations:

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

Eventually this system will include portable definitions of ’hardcopy and ’edit.

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.

To facilitate this, three new optional arguments have been added to require. Thus the new syntax of require is as follows:

REQUIRE *system-name* &optional *pathname definition-pname default-action version*

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 to be 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.

A Sample System Definition and Its Use

Here's a system definition 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
```

```
(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)

<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
```

Miscellaneous Notes

Macintosh pathnames are not fully supported at this time because of irregularities in the Allegro Common Lisp definition of the pathname functions. Thus system definitions will not be portable to the Macintosh. To convert them, include the device in the toplevel pathname and include trailing colons in the pathnames of each module.

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. And if someone has such a large system that this matters, they've got more important problems.