Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo No. 801 September, 1984

The Description of Large Systems

Kent Pitman

Abstract

In this paper, we discuss the problems associated with the description and manipulation of large systems when their sources are not maintained as single files. We show why and how tools that address these issues, such as Unix MAKE and Lisp Machine DEFSYSTEM, have evolved.

Existing formalisms suffer from the problem that their **syntax** is not easily separable from their **functionality**. In programming languages, standard "calling conventions" exist to insulate the caller of a function from the syntactic details of how that function was defined, but until now no such conventions have existed to hide consumers of program systems from the details of how those systems were specified.

We propose a low-level data abstraction which can support notations such as those used by MAKE and DEFSYSTEM without requiring that the introduction of a new notation be accompanied by a completely different set of tools for instantiating or otherwise manipulating the resulting system.

Lisp is used for presentation, but the issues are not idiosyncratic to Lisp.

Keywords: Compilation, Large Systems, Lisp, System Maintenance.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grants MCS-7912179 and MCS-8117633, and in part by the IBM Corporation.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, expressed or implied, of the Department of Defense, of the National Science Foundation, or of the IBM Corporation.

I. Introduction

For reasons of modularity and editing convenience, the source code for large program systems is rarely maintained as a single file. Instead, it is typically broken into a number of smaller files which together make up the system.

Since system tools such as editors, compilers, loaders and printers tend to be designed to deal with files rather than systems, some extra mechanism is generally required in order to allow their users to deal with systems that span multiple files. In the next section, we trace the evolution of tools for system building, identifying the important issues that early tools sought to address.

We then present an overview of two system specification languages, Unix[1] `MAKE` and Lisp Machine `DEFSYSTEM`, which illustrate the level of technology currently available to programmers for dealing with these issues. Most criticisms which can be made about existing tools are at the syntactic level. The tools address the right issues, but their syntax can be a stumbling block, inhibiting the expression of certain kinds of relations and the ability to make extensions to the tools' original functionality. It is not possible to vary the syntax without rewriting most or all of the underlying support.

We conclude by proposing an organizational strategy which decouples **syntax** and **functionality**. This makes it possible for a programmer to develop alternate system maintenance tools without having to reimplement every aspect of the original tools.

To make the discussion more concrete, we will give several examples of specification languages that could be built under the proposed framework. However, it is important to understand that the purpose of this paper is not to argue in favor of any particular notation. Rather, we wish to illustrate that our proposed organizational strategy establishes an appropriate framework for developing alternate notations such as these. As an appendix, we offer a sample implementation of these specification languages in `Lisp Machine Lisp` in order to further clarify any issues left vague by the examples used in the body of the paper.

II. Background

There are a number of standard maintenance operations performed on systems. These operations include (but are not limited to) the creation of hardcopy listings, copying or renaming the files which make up the source of the system, loading uncompiled source files (*e.g.*, into a lisp interpreter), and compiling changed source files.

For a system maintained as a single monolithic file, it is reasonably obvious how most of these maintenance operations can be performed. Getting hardcopy of the system source is as simple as getting hardcopy of any other file; compiling the system is as simple as compiling any single file.

For a more complex system, where the source spans more than one file, performing these operations may be considerably more complicated. In that case, to get a listing, one must ask that each of the source files be printed. To compile the system, one must ask that each of the source files be compiled and, in some cases, that the compilation occur in a certain order. To load the system may be similarly complex due to another (not necessarily identical) set of ordering constraints.

A Need for Abstraction

It used to be that breaking a system into multiple files meant that one had to remember all the names of the files and manipulate each as a separate object. Programming environments had some primitive understanding of files and operations to be performed upon files, but had no explicit understanding that groups of files could work together as a single unit.

[1] **Unix** is a trademark of Bell Labs.

It was common practice then (and still is today) to create **batch files** holding the commands necessary to accomplish a particular manipulation. For example, the programmer might have kept a file containing code for compiling and loading a particular system and another file containing code for hardcopying its sources:

```
;; Batch sequence to compile/load system.
(LOAD (COMPILE-FILE "MACROS.LISP"))
(LOAD (COMPILE-FILE "UTILITY.LISP"))
(LOAD (COMPILE-FILE "MAIN.LISP"))
```

COMPILE-AND-LOAD-MYSYS.LISP

```
;; Batch sequence to hardcopy system sources.
(HARDCOPY-FILE "MACROS.LISP")
(HARDCOPY-FILE "UTILITY.LISP")
(HARDCOPY-FILE "MAIN.LISP").
```

HARDCOPY-MYSYS.LISP

Common though it may have been to manipulate systems indirectly through such batch files, it was quite clumsy. It meant that any time a change to the system was made, all the relevant batch files had to be updated. If not updated carefully and completely, the batch files could easily become inconsistent, leading to confusing effects. Also, because information about the structure of a system was **procedurally embedded**, the only way a utility could be written to apply a new operation to the system was by creating yet another separately-maintained batch file.

Not surprisingly, programmers have moved away from this batch method of maintaining their systems and toward the notion that a system should be defined abstractly in some central place and then manipulated as a unit rather than as a set of unrelated individual components (files).

A First Approximation

Here is a simple example of the sort of code needed to implement the kind of tool we are discussing:

```
(DEFVAR *SOURCE-INFO* (MAKE-PLIST))

(DEFUN SYSTEM-SOURCES (SYSTEM) (GET *SOURCE-INFO* SYSTEM))

(DEFUN DEFINE-SYSTEM-SOURCES (SYSTEM &REST SOURCES)
  (PUTPROP *SOURCE-INFO* (COPYLIST SOURCES) SYSTEM))

(DEFUN HARDCOPY-SYSTEM (SYSTEM)
  (MAPC #'HARDCOPY-FILE (SYSTEM-SOURCES SYSTEM)))

(DEFUN COMPILE-AND-LOAD-SYSTEM (SYSTEM)
  (MAPC #'(LAMBDA (FILE) (LOAD (COMPILE-FILE FILE)))
        (SYSTEM-SOURCES SYSTEM)))
```

Given this level of support, a system would be "defined" by writing simply:

```
(DEFINE-SYSTEM-SOURCES 'MYSYS
  "MACROS.LISP" "UTILITIES.LISP" "MAIN.LISP").
```

Once defined, such a system could be manipulated by requests such as these:

```
(HARDCOPY-SYSTEM 'MYSYS)
(COMPILE-AND-LOAD-SYSTEM 'MYSYS)
```

Partial Ordering of Dependencies

One problem with this formulation is that the relationship between the modules of a system may be quite complex. For example, some of the files in a system may not depend on other files and the whole system may not need to be recompiled just because one file has changed. To try to account for this, the definition of COMPILE-AND-LOAD-SYSTEM might be changed to read:

```
(DEFUN COMPILE-AND-LOAD-SYSTEM (SYSTEM)
  (LET ((COMPILE-FLAG NIL))
    (DOLIST (FILE (SYSTEM-SOURCES SYSTEM))
      (SETQ COMPILE-FLAG
            (OR COMPILE-FLAG (NEEDS-COMPILATION? FILE)))
      (IF COMPILE-FLAG (COMPILE-FILE FILE))
      (LOAD (BIN-FILE FILE)))))).
```

With this revised definition, files in a system's source file list would have a left-to-right dependency relation. Consider again the system defined by:

```
(DEFINE-SYSTEM-SOURCES 'MYSYS
  "MACROS.LISP" "UTILITIES.LISP" "MAIN.LISP").
```

If MACROS.LISP is changed, UTILITIES.LISP and MAIN.LISP will have to be recompiled. But if MAIN.LISP is changed, it is the only file which will get recompiled.

Simple left-to-right dependency is useful for some applications, but may cause a lot of unneeded work in others. The reason is that the actual ordering may only be partial, but a full ordering is forced by this notation.[2]

For example, it is easy to imagine our system being constructed so that if the file MACROS.LISP changed, both UTILITIES.LISP and MAIN.LISP would need to be recompiled, but if just one of UTILITIES.LISP or MAIN.LISP changed, only that one file would require recompilation.

Because this simple notation provides no way to adequately express such complex relations, it must be judged inadequate to handle the "general case." Nevertheless, there are cases where it would be adequate, and it would be nice to use it (or something equally simple) for those cases.

Orthogonal Dependency Types

Another problem with our original formulation concerns multiple, independent dependency chains. Dependency information for compilation might not be the same as dependency information for producing a runtime environment.

If the only goal is to compile a system, there might be many files which do not need to be loaded because they contain utilities used only at runtime. Alternatively, if the goal is to load an already-compiled system, some files (for example, those containing only macros used at compile time) might not be necessary. If both compilation and loading are to be interleaved, a third ordering might arise.

An adequate notation for describing systems needs to offer a notation for stating different kinds of dependency relations, and should probably be extensible (allowing the addition of new kinds of dependencies).

---

[2] In general, notations must be chosen with extreme care; an ill-chosen notation can have a very adverse effect on specifications that use it, making them seem to imply things which are in fact false. For a more complete discussion of such issues, see [Mackinlay 84].

III. Existing Tools

The Unix MAKE Facility

The Unix MAKE facility [Feldman 78] is frequently pointed to as a model of "the right way to define a system." It is syntactically simple and provides a reasonable amount of power.

A `makefile` contains Unix shell commands augmented with information about which code modules depend on which others. When the `make` command is invoked, it is as if all the shell commands were executed except that some shell commands may be "optimized out" if the dependency information specifies that they are not necessary to preserve correctness.

To make things concrete, here is a sample of how a system might be specified in a `makefile`:

```
mysys: a.o b.o
        cc a.o b.o -lm -o pgm
  a.o: incl a.c
        cc -c a.c
  b.o: incl b.c
        cc -c b.c
```

The first line defines that there is some module called `mysys` which needs to be updated if `a.o` or `b.o` are ever out of date. The indented line following that line specifies how to do the update; specifically, it links the compiled files `a.o`, `b.o`, and standard library `m`. The third and fifth lines define modules `a.o` and `b.o`, saying that they depend on `.c` source files and also on some file called `incl` (which they presumably reference internally via `#include`).

To make comparison easier, we could pretend that MAKE used Lisp expressions rather than requiring a special parser. In such case, a `makefile` might contain an expression like:

```
(DEFINE-FOR-MAKE MYSYS
    (MYSYS ("a.o" "b.o")
           (LOAD-IF-NOT-LOADED "a")
           (LOAD-IF-NOT-LOADED "b")
           (LOAD-IF-NOT-LOADED "m"))
    ("a.o" ("incl" "a.c")
           (COMPILE-FILE "a.c"))
    ("b.o" ("incl" "b.c")
           (COMPILE-FILE "b.c")))
```

One problem with MAKE is that not all the information in a `makefile` is explicit. For example, the fact that the system sources are `incl`, `a.c`, `b.c` is nowhere explicit. There is nothing (other than hoping that `.o` files are not source files and anything else is) which identifies them as the source files. A programmer wanting to write a utility for producing hardcopy of a system's sources would not be able to drive the utility off of the information contained in the `makefile`.

MAKE offers no theory of how to include or process additional information. For example, having the tail of each module clause (the part after the colon in the original Unix syntax) say how to build the module is fine for a compiled-only language, but is not reasonable in a language like lisp which embraces the notion of an interpreter and a compiler that can share the load. There is no obvious way to extend MAKE in an upward-compatible fashion in order to allow specification of commands for compiling modules as well as commands for loading them. There is also no provision for adding other kinds of information, such as an alternate notion of what it means for modules to be out of date or how to handle circular dependencies.

In fact, MAKE amounts to little more than a batch facility with a simple but inflexible provision for ignoring unnecessary commands. Just as with normal batch files, to do two operations (*e.g.*, compile and load) requires two batch files. While it might be syntactically convenient for some common applications in compiled-only languages under Unix, MAKE does not represent a theory of how to maintain systems.

The DEFSYSTEM Facility

The Lisp Machine DEFSYSTEM facility [Weinreb 81] bills itself as a "general and extensible" tool for maintaining systems broken into several files. It provides a means of noting what files belong to what modules, what modules depend on what other modules (both for compilation and for loading), and is extensible to allow the addition of new system-building "transformations" (such as calls to alternate compilers and loaders).

Information about the structure of a system is defined with the DEFSYSTEM special form. Later, MAKE-SYSTEM can be called to perform a pre-defined set of operations upon the system.

Here is a sample system description written in DEFSYSTEM notation:

```
(DEFSYSTEM MYSYS
  (:MODULE MACROS ("incl"))
  (:MODULE A ("a"))
  (:MODULE B ("b"))
  (:MODULE MLIB ("m"))
  (:FASLOAD MLIB)
  (:COMPILE-LOAD MACROS)
  (:COMPILE-LOAD-INIT A (MACROS)
                        (:FASLOAD MACROS)
                        (:FASLOAD MLIB))
  (:COMPILE-LOAD-INIT B (MACROS)
                        (:FASLOAD MACROS)
                        (:FASLOAD MLIB A)))
```

The :MODULE[3] clauses specify which files belong to each of the modules. For example, the INCL module refers to the file "incl.lisp", the A module refers to "a.lisp", *etc.*

The :FASLOAD clause for MLIB says that "m.lisp" is a standard library which must be loaded. This definition doesn't specify when it might be recompiled; presumably that is handled by some other agency.

The :COMPILE-LOAD clause for MACROS says that "incl.lisp" is part of the system and must be recompiled if changed, but it has no particular preconditions to be satisfied prior to compilation or loading.

The :COMPILE-LOAD-INIT clauses for A and B specify various kinds of dependency information. Both must be recompiled not only if they themselves change, but also if MACROS changes. Before compiling A, one must first load MACROS; before loading A, one must first load MLIB. Before compiling B, one must first load MACROS; before loading B, one must first load MLIB and A.

Although names like :COMPILE-LOAD may look like function names, they are not. They simply declare that compilation and/or loading may need to occur under certain circumstances. DEFSYSTEM notation, unlike that of MAKE, is declarative rather than procedural. Because of this, MAKE-SYSTEM can perform more than one kind of operation (for each DEFSYSTEM form), where MAKE could only perform one (for each makefile). For example, given certain arguments, MAKE-SYSTEM will compile a system. Given different arguments, it will simply load an already-compiled version of the system.

On the other hand, the set of operations that MAKE-SYSTEM will perform is pre-defined and not extensible. This means that if someone wanted to add a new utility (*e.g.*, for hardcopying sources) there would be no way to do it because there is no user-advertised mechanism for asking for a list of a system's source files. The information is present and is used internally by various system utilities, but is not advertised to users as part of the standard abstraction.

---

[3] In Lisp Machine Lisp, symbols preceded by a colon are "keywords" interned in a canonical "keyword package." By special decree, keywords are self-quoting (*i.e.*, bound to themselves). Hence, ':FOO and :FOO evaluate to the same thing, :FOO.

Another gripe commonly heard about DEFSYSTEM is that it was designed to handle hard cases, but that for simpler tasks it is generally cumbersome and unpleasant to use.

Still another problem, which contributes to the overall feeling of clumsiness, is that there is no transitivity of dependency information between DEFSYSTEM modules. Hence, if DEFSYSTEM is told that C.LISP depends on B.LISP and that B.LISP depends on A.LISP, DEFSYSTEM will not infer that C.LISP depends on A.LISP. Technically, it would be incorrect to do otherwise because it may be the case that the part of B.LISP which depends on A.LISP is not used by C.LISP. However, the result of this decision on the part of the DEFSYSTEM designers is that large DEFSYSTEM forms tend to take on a pyramidal shape as later compile-load specifications are forced to specify an enormous number of dependencies explicitly:[4]

```
(defsystem KBE
  (:name "Knowledge-Based-Editor")
  (:short-name "KBE")
    :
  (:module LET "KBE: LISPM2; LET" :package KBE)
  (:module LETS "KBE: LMLIB; LETS")
  (:module PP "KBE: LMLIB; GPRINT")
  (:module BASE ("BASIC" "NOISE" "SETS" "CONDITIONS" "ENGLISH"
                 "SYSTEM-MAGIC" "FLAVOR-MAGIC"))
  (:module MACRO ("MACRO"))
  (:module KVARS "KVARS")
    :
  (:fasload PP)
  (:fasload LET)
  (:fasload LETS)
  (:compile-load-init BASE (PP)
                           (:fasload PP LET LETS)
                           (:fasload PP))
  (:compile-load-init MACRO (PP)
                            (:fasload PP LET LETS BASE)
                            (:fasload BASE))
  (:compile-load-init KVARS (PP MACRO)
                            (:fasload PP LET LETS BASE MACRO)
                            (:fasload BASE))
  (:compile-load-init KMAC (PP MACRO)
                           (:fasload PP LET LETS BASE MACRO KVARS)
                           (:fasload BASE KVARS))
  (:compile-load-init IMAC (PP MACRO KMAC)
                           (:fasload PP LET LETS BASE MACRO KVARS KMAC)
                           (:fasload BASE KVARS KMAC))
  (:compile-load-init MBOX (PP MACRO)
                           (:fasload PP LET LETS BASE MACRO KVARS IMAC)
                           (:fasload BASE KVARS IMAC))
  (:compile-load-init BEHAVE (PP MACRO IMAC)
                             (:fasload PP LET LETS BASE MACRO KVARS IMAC)
                             (:fasload BASE KVARS IMAC MBOX))
  (:compile-load-init KUTIL (PP MACRO IMAC BEHAVE)
                            (:fasload PP LET LETS BASE MACRO KVARS IMAC BEHAVE)
                            (:fasload BASE KVARS IMAC MBOX BEHAVE))
    :
  (:compile-load-init MAGIC (PP MACRO IMAC BEHAVE SIMPLE ED1 SOCIETY)
                            (:fasload PP LET LETS BASE MACRO KVARS IMAC
                                      BEHAVE KUTIL SIMPLE ED1 SOCIETY STNDRD ZMAGIC)
                            (:fasload BASE KVARS IMAC BEHAVE KUTIL ED1
                                      SOCIETY STNDRD BRIDGE ZMAGIC))
  ...)
```

[4]This example is taken directly from a real program system developed by the MIT Programmer's Apprentice group. The full definition is much longer, describing the relations between about 90 files, grouping them into about 50 logical modules.

DEFSYSTEM is extensible, but in addition to its other problems, much of the data available to extension writers is in the form of special variables which are available while its "transformations" are happening. To illustrate this, we include an excerpt from the *Lisp Machine Manual* section describing state variables which can be used in writing DEFSYSTEM extensions:

SI:*SYSTEM-BEING-MADE* *Variable*

The internal data structure which represents the system being made.

SI:*MAKE-SYSTEM-FORMS-TO-BE-EVALED-BEFORE* *Variable*

A list of forms which are evaluated before the transformations are performed.

SI:*MAKE-SYSTEM-FORMS-TO-BE-EVALED-AFTER* *Variable*

A list of forms which are evaluated after the transformations are performed.

SI:*MAKE-SYSTEM-FORMS-TO-BE-EVALED-FINALLY* *Variable*

A list of forms which are evaluted after the body of MAKE-SYSTEM has completed. This differs from SI:*MAKE-SYSTEM-FORMS-TO-BE-EVALED-AFTER* in that these forms are evaluated outside the "compiler context," which sometimes makes a difference.

SI:*QUERY-TYPE* *Variable*

Controls how questions are asked. Its normal value is :NORMAL. :NOCONFIRM means no questions will be asked and :SELECTIVE asks a question for each individual file.

⋮

(SI:DEFINE-MAKE-SYSTEM-SPECIAL-VARIABLE *var val [defvar-p]*) *Special Form*

Causes *var* to be bound to *val*, which is evaluated at MAKE-SYSTEM time, during the body of the call to MAKE-SYSTEM. This allows you to define new variables similar to those listed above. If *defvar-p* is specified as (or defaulted to) T, *var* is defined with DEFVAR. It is not given an initial value. If *defvar-p* is specified as NIL, *var* belongs to some other program and is not DEFVAR'ed here.

Because information is procedurally embedded in this way, it may not be explicit at arbitrary times. Also, it imposes a lot of pre-defined mechanism for talking about systems which might not be convenient or even relevant in certain situations.

These problems with DEFSYSTEM's design make it difficult to develop tools which interface to those provided with DEFSYSTEM. The net result is that the DEFSYSTEM formalism is far less flexible than we might want it to be.

IV. Proposal

Both DEFSYSTEM and MAKE offer interesting functionality, but that functionality is tightly bonded to the accompanying syntax. Neither seeks to provide a theory of how to allow specification of a system without binding the user to a particular syntax.

No matter what the nature of the specification language, the kinds of high level operations to be performed upon systems is not going to vary. We will still want to edit, compile, load, and hardcopy systems. Hence, rather than propose yet another system description language with a *new, improved* notation, we will propose a framework in which syntax and functionality are permitted to vary independently.

In this section, we review the details of the proposal, but its essence is the suggestion that associated with every system there should be some object which responds appropriately to a pre-defined set of operations which support appropriate maintenance of the system. Put another way, we assert that a key problem in previous system maintenance utilities is that they either were not object oriented or did not use their object oriented nature to their best advantage.

Any sort of generic operations facility (*e.g.*, that provided in T [Rees 84] or Act-II [Theriault 83]) would suffice to implement this proposal. Wherever possible, we will use an abstract function-calling syntax for presentation. In places which call for examples from a particular language, we will use Lisp Machine Lisp for presentation purposes.

Basic Protocol

By studying operations that are typically performed upon files, we can make a list of the common operations we might expect to be performed upon systems. One obvious choice is the **update** operation, which includes compilation or translation from one language to another, parser generation, *etc.* Another is **instantiation**, which includes the loading of files or the execution of some kinds of setup code. Other useful operations might be **editing**, **hardcopying**, and **archiving**.

To support these operations, we define the following functions:

(SYSTEM:GENERATE-PLAN *system operation*) → *actions*

Given the name of an abstract *operation* (such as :UPDATE, :INSTANTIATE, or :HARDCOPY), returns a list of *actions* (abstract plan steps) which will accomplish the *operation*.

(SYSTEM:EXECUTE-PLAN *system actions*)

Executes a list of *actions* (or plan steps), such as those returned from a SYSTEM:GENERATE-PLAN request.

(SYSTEM:EXPLAIN-PLAN *system actions*)

Types out an explanation of what would be the effect of executing *actions*.

(SYSTEM:SOURCE-FILES *system*) → *files*

Returns a list of the source *files* for the system.

To this basic set, we will add the following two functions. It might be argued that they are superfluous in the presence of the above functions, but in practice having them will greatly enhance the clarity of some code. By adding them to the standard protocol, we encourage a clearer programming style and standardize on terms:

(SYSTEM:EXECUTE-ACTION *system action*)

Executes a given *action* (plan step).

(SYSTEM:EXPLAIN-ACTION *system action*)

Types out an explanation of what would be the effect of executing *action*.

For convenience, we'll also define that if the argument given to EXPLAIN or EXECUTE is the name of an operation rather than a set of steps, then the plan will be generated automatically. This allows us to say:

```
    (SYSTEM:EXECUTE-PLAN system :UPDATE)
and (SYSTEM:EXPLAIN-PLAN system :UPDATE)
```

where formerly we would have had to say:

```
    (SYSTEM:EXPLAIN-PLAN system (SYSTEM:GENERATE-PLAN system :UPDATE))
and (SYSTEM:EXECUTE-PLAN system (SYSTEM:GENERATE-PLAN system :UPDATE)).
```

Defining systems

Under our proposal, systems are described using the DEFINE-SYSTEM special form. It creates a system object and stores it globally for use at a later time. It has the syntax:

(DEFINE-SYSTEM *name type . options*).

The exact nature of the *options* will vary depending on the *type* of the system. For some systems, it may just be a list of files. For others, it might be a more complex data structure specifying specific dependency information. This proposal is designed explicitly to avoid taking a stand on what goes in this portion of the system specification.

To support this kind of type-specific option processing, we need functions to digest a type-specific *options* list:

`(SYSTEM:PROCESS-OPTIONS` *`system options-list`*`)`

Processes an *options-list*, such as the body of a `DEFINE-SYSTEM` form. This might, but need not necessarily, be done by mapping `SYSTEM:PROCESS-OPTION` down the *options-list*.

`(SYSTEM:PROCESS-OPTION` *`system name . data`*`)`

Processes a single option with given *name* and *data*. The option *name* `:NAME` must be handled. Handling of any other option is at the discretion of the particular system type.

Creating System Objects

The `DEFINE-SYSTEM` special form is supported by a normal function, called `CREATE-SYSTEM`, which has the syntax:

`(CREATE-SYSTEM` *`name type options`*`).`

`CREATE-SYSTEM` returns an object representing the system, but does not store it in any global place. Such an object is called an *anonymous system*. `CREATE-SYSTEM` (and hence `DEFINE-SYSTEM`) works by creating an object of the designated system flavor and then calling appropriate functions to set its name and process its options.

Extensibility

To be appropriately extensible, each implementation would have to define how these functions related to the generic operations facility provided by that language. For example, on the Lisp Machine, the interface to flavors might look like:

```
(DEFUN SYSTEM:PROCESS-OPTIONS (SYSTEM OPTIONS-ALIST)
  (SEND SYSTEM :PROCESS-OPTIONS OPTIONS-ALIST))

(DEFUN SYSTEM:PROCESS-OPTION (SYSTEM NAME &REST DATA)
  (LEXPR-SEND SYSTEM :PROCESS-OPTION NAME DATA))

  ⋮
```

Exploiting Inheritance

Languages which provide generic operations and facilities for type inheritance would probably offer at least two pre-defined types.

A type called `SYSTEM` should be at the base. It should have no properties other than identifying the object as a system. Designers who wish to start over "from scratch" in designing new types of systems adhering to the protocol we propose would start with the `SYSTEM` type and work from there.

Another type, which we shall call `VANILLA-SYSTEM`, might offer some very general functionality which might be of use to many kinds of systems. Using whatever inheritance mechanism was appropriate to the language, implementors of many new system types might be able to inherit from this slightly less general type rather than starting from scratch and building their system type from type `SYSTEM`. Naturally, `VANILLA-SYSTEM` would inherit from `SYSTEM`.

In the Lisp Machine, for example, customizing can frequently be done by mixing `VANILLA-SYSTEM` into the new flavor and adding or changing a few methods. Nothing prevents the designer from starting from scratch and implementing all the methods from scratch; but this will typically involve more work than is necessary.

A typical system flavor might look like:

```
(DEFFLAVOR system-type (...instance variables...)
          (VANILLA-SYSTEM))
```

followed by definitions of new or customized methods.

For example, on the Lisp Machine, the :PROCESS-OPTIONS method might be expected to be defined by:

```
(DEFMETHOD (VANILLA-SYSTEM :PROCESS-OPTIONS) (OPTIONS)
  (DOLIST (OPTION OPTIONS)
    (SEND SELF :PROCESS-OPTION OPTION)))
```

This is defined as part of VANILLA-SYSTEM to save everyone the trouble of writing that same method. In fact, VANILLA-SYSTEM might even define :PROCESS-OPTION to use :CASE method dispatch (so that handling each kind of option may be defined by a separate DEFMETHOD form). If so, we might expect to also find definitions such as these in VANILLA-FLAVOR:

```
(DEFMETHOD (VANILLA-SYSTEM :CASE :PROCESS-OPTION :FULL-NAME) (NAME)
  (SETQ FULL-NAME (STRING NAME)))

(DEFMETHOD (VANILLA-SYSTEM :CASE :PROCESS-OPTION :SHORT-NAME) (NAME)
  (SETQ SHORT-NAME (STRING NAME)))

(DEFMETHOD (VANILLA-SYSTEM :CASE :PROCESS-OPTION :NAME) (NAME)
  (IF (NOT SHORT-NAME) (SETQ SHORT-NAME (STRING NAME)))
  (IF (NOT FULL-NAME) (SETQ FULL-NAME (STRING NAME))))

  :

(DEFMETHOD (VANILLA-SYSTEM :OTHERWISE :PROCESS-OPTION) (NAME &REST DATA)
  (FERROR "Bad option: ~S~%Data: ~S" NAME (COPYLIST DATA)))
```

Presumably, systems inheriting from VANILLA-SYSTEM would define additional :PROCESS-OPTION methods for any specifications appropriate to them.

Modular Extensions

Given these basic facilities, it is easy to make modular extensions. The function HARDCOPY-SYSTEM discussed earlier could be written simply as:

```
(DEFUN HARDCOPY-SYSTEM (SYSTEM)
  (MAPC #'HARDCOPY-FILE (SYSTEM:SOURCE-FILES SYSTEM)))
```

Likewise, a facility for saving a snapshot of a system's source files to another directory might be written:

```
(DEFUN ARCHIVE-SYSTEM (SYSTEM ARCHIVE-DIRECTORY)
  (DOLIST (FILE (SYSTEM:SOURCE-FILES SYSTEM))
    (COPY-FILE FILE ARCHIVE-DIRECTORY)))
```

There's no reason the user should necessarily have to write things like this himself. In general, it's nice to have system libraries that have this sort of thing pre-defined. The important thing is that if they were not primitively provided, they would be no problem to write as extensions because a basic set of operations has been chosen which lends itself to modular extension.

Even the fancier options to DEFSYSTEM, such as the :SELECTIVE option, fall directly out of this modularization. For example, the essence of the :SELECTIVE option is captured by:

```
(LET ((PLAN (SYSTEM:GENERATE-PLAN system :UPDATE)))
  (FORMAT T "~&To update ~A:" system)
  (DOLIST (STEP PLAN)
    (SYSTEM:EXPLAIN-ACTION system STEP))
  (UNLESS (NOT (Y-OR-N-P "Ready to go ahead?  "))
    (SYSTEM:EXECUTE-PLAN system PLAN)))
```

Variations are also simple. For example, per-step querying could be achieved by:

```
(DOLIST (STEP (SYSTEM:GENERATE-PLAN system :UPDATE))
  (SYSTEM:EXPLAIN-ACTION system STEP)
  (UNLESS (NOT (Y-OR-N-P "OK? "))
    (SYSTEM:EXECUTE-ACTION system STEP)))
```

Systems with Simple Dependencies

Until now, not much has been said about what kind of information should go into the options portion of a DEFINE-SYSTEM form.

In the simplest case, all we might want to specify is the set of files involved. So, for example, we might imagine a kind of system called SIMPLE-SYSTEM in which the options section was just a list of files, so that the system we earlier specified by:

```
(DEFINE-SYSTEM-SOURCES 'MYSYS
  "MACROS.LISP" "UTILITIES.LISP" "MAIN.LISP")
```

would now be specified by:

```
(DEFINE-SYSTEM MYSYS SIMPLE-SYSTEM
  "MACROS.LISP" "UTILITIES.LISP" "MAIN.LISP").
```

Contrast the simplicity of this approach with the MAKE specification of:

```
(DEFINE-FOR-MAKE MYSYS
  ("main.bin" ("utilities.bin" "macros.bin" "main.lisp")
              (LOAD-IF-NOT-LOADED "macros.bin")
              (LOAD-IF-NOT-LOADED "utilities.bin")
              (LOAD (COMPILE-FILE "main.lisp")))
  ("utilities.bin" ("macros.bin" "utilities.lisp")
                   (LOAD-IF-NOT-LOADED "macros.bin")
                   (LOAD (COMPILE-FILE "utilities.lisp")))
  ("macros.bin" ("macros.lisp")
                (LOAD (COMPILE-FILE "macros.lisp"))))
```

or the DEFSYSTEM form:

```
(DEFSYSTEM MYSYS
  (:MODULE MACROS ("macros"))
  (:MODULE UTIL ("utilities"))
  (:MODULE MAIN ("main"))
  (:COMPILE-LOAD MACROS)
  (:COMPILE-LOAD-INIT UTIL (MACROS)
                      (:FASLOAD MACROS)
                      (:FASLOAD MACROS))
  (:COMPILE-LOAD-INIT MAIN (MACROS UTIL)
                      (:FASLOAD MACROS UTIL)
                      (:FASLOAD MACROS UTIL))).
```

The system defined by this DEFINE-SYSTEM form is a first-class object which can be inspected and manipulated by the abstraction functions proposed in the last section.

Systems with Complex Dependencies

Consider now a system with a set of macros (in MACROS.LISP) that expand into calls to functions in some utility package (in MACRO-SUPPORT.LISP). It should be an abstraction violation for the consumers of the macro package to have to know what support is necessary for the package to run. Yet with DEFSYSTEM, the specification must be written:

```
(DEFSYSTEM MYSYS
  (:MODULE MACROS ("macros"))
  (:MODULE MACRO-SUPPORT ("macro-support"))
  (:MODULE FOO ("foo"))
    ⋮
  (:COMPILE-LOAD MACRO-SUPPORT)
  (:COMPILE-LOAD MACROS)
  (:COMPILE-LOAD-INIT FOO (MACROS)
                          (:FASLOAD MACROS)
                          (:FASLOAD MACRO-SUPPORT))
  ...)
```

A user interested in abstraction might object to having to specify FOO's dependency upon MACRO-SUPPORT explicitly. Certainly users of DEFSYSTEM have complained that this lack of abstraction makes very large systems very hard to specify and maintain using DEFSYSTEM.

Armed with our new DEFINE-SYSTEM proposal, a new notation could be developed to handle the situation. We might, for example, propose a notation where rather than say "FOO depends on MACRO-SUPPORT" explicitly (as happens in DEFSYSTEM), we could say "Using MACROS causes a need for MACRO-SUPPORT." Such a notation might look like:

```
(DEFINE-SYSTEM MYSYS MODULAR-SYSTEM
  (:MODULE MACRO-SUPPORT ("macro-support"))
  (:MODULE MACROS ("macros")
   (:CAUSES
    (:NEEDS
     (:INSTANTIATE MACRO-SUPPORT))))
  (:MODULE FOO ("foo")
   (:NEEDS
    (:UPDATE MACROS)))
  ...)
```

To argue for or against some particular new notation is not the point of this paper. The real point is that the proposed framework provides a means of introducing alternate notations in a way that does not interfere with existing notations and tools. Existing tools can operate correctly upon systems created with new notations such as this because it is the functional behavior of systems which has been standardized, not the notation.

In [Robbins 84], still another notation (to accompany a tool called BUILD) is proposed for specifying module dependency information. Although the data abstractions proposed in this paper were not designed with BUILD in mind, they seem appropriate to support it anyway. Had the proposed framework already been in effect, it would probably have been considerably simpler for Robbins to experiment with his new notation.

Systems with "Idiosyncratic" Dependencies

Some systems may have very complicated file dependencies. In some cases, for example, code may have evolved in a way which demands that an earlier copy of itself be loaded in order to support its compilation or instantiation. The description of such systems and how they are to be constructed may require a complex notation. This is acceptable only if it does not affect the simplicity of notation used to describe simple systems.

Some users have suggested that the complexity of DEFSYSTEM stems from the fact that the same notational devices must be used for all kinds of systems, whether simple or complex. In this new proposal, systems can be classified into different kinds, each with their own notation. We have illustrated that simple systems might require only the specification of the files involved and nothing else, while some more complex systems might be specified in terms of inter-file dependencies.

For the completely general case, however, special "one-shot" notations can still be developed to handle the specific needs of situations which are not yet sufficiently well-understood to be handled by a more standard notation. For example,

```
(DEFFLAVOR HAIRY-SYSTEM () (VANILLA-SYSTEM))

(DEFMETHOD (HAIRY-SYSTEM :CASE :GENERATE-PLAN :UPDATE) ()
  '((:LOAD "Foo")
    (:COMPILE "Foo") ;Needs self to compile
    (:LOAD "Foo")
    (:LOAD "Bar")))

(DEFINE-SYSTEM MY-HAIRY-SYSTEM HAIRY-SYSTEM) ;needs no options

(SYSTEM:EXECUTE-PLAN (SYSTEM 'MY-HAIRY-SYSTEM) :UPDATE)
```

A slight generalization of this idea leads to another kind of system, which allows the system maintainer to specify explicitly how to handle each kind of :GENERATE-PLAN request:

```
(DEFFLAVOR PREPLANNED-SYSTEM ((FILES '())
                              (PLANS '()))
           (VANILLA-SYSTEM))

(DEFMETHOD (PREPLANNED-SYSTEM :PROCESS-OPTIONS) (OPTIONS)
  (SETQ FILES (CAR OPTIONS))
  (SETQ PLANS (CDR OPTIONS)))

(DEFMETHOD (PREPLANNED-SYSTEM :GENERATE-PLAN) (OPERATION)
  (LET ((P (ASSQ OPERATION PLANS)))
    (COND (P (CDR P))
          (T (FERROR "No plan for operation ~S" OPERATION)))))).
```

Having done this, the same system could be re-written:

```
(DEFINE-SYSTEM MY-HAIRY-SYSTEM PREPLANNED-SYSTEM
               ("Foo" "Bar")
  (:UPDATE
    (:LOAD "Foo")
    (:COMPILE "Foo") ;Needs self to compile
    (:LOAD "Foo")
    (:LOAD "Bar"))
  ...).
```

This notation has some of the character of the batch files discussed earlier, but is much better integrated with existing tools. Users of a system could load or compile it without knowing how it was defined. Later, if system dependencies changed or if a new notation became available, the system's specification could be changed without notifying the users.

Systems with Dependencies to be Inferred by the Compiler

Some people have objected to the idea that a system description language should be needed at all. Their claim is that the compiler should somehow be able to infer compilation dependencies by recognizing and recording assumptions made during compilation.

This sort of inference is not completely reliable because compilers cannot always accurately detect compilation dependencies induced by changes to the state of the global environment during compilation. The problem stems from the presence in Lisp of powerful state-affecting primitives such as EVAL-WHEN and the general ability of macros to read or alter global state during compilation.

To see the problem, consider a data-driven macro facility such as the following, which maintains its state both in the compiler and in the runtime environment:

```
  ⋮
(DEFUN EXPAND-DEFINITION (OP NAME BODY)
  (SELECTQ OP
    ((DEFINE) ···)
      ⋮
    (OTHERWISE ···)))
  ⋮
```

A.LISP

```
  ⋮
(DEFMACRO DEFINE (NAME . BODY)
  (EXPAND-DEFINITION 'DEFINE NAME BODY))
  ⋮
```

B.LISP

```
  ⋮
(DEFINE FOO ···)
  ⋮
```

C.LISP

If the definition of EXPAND-DEFINITION had changed, it would obviously have to be recompiled. On the other hand, the definition of DEFINE, while it directly refers to EXPAND-DEFINITION, is not affected by the change to EXPAND-DEFINITION. So A.LISP and C.LISP would have to be recompiled, but B.LISP would not.

To complicate matters, however, suppose that instead of the above definition, EXPAND-DEFINITION had been defined by:

```
(DEFUN EXPAND-DEFINITION (OP NAME BODY)
  (FUNCALL (GET OP 'EXPANDER) NAME BODY))
```

In that case, a seemingly unrelated change such as a change to a definition elsewhere such as:

```
(DEFUN (:PROPERTY DEFINE EXPANDER) (NAME BODY) ···)
```

could affect the expansion of FOO, causing C.LISP to need recompilation.

There might be a temptation to suggest that the compiler notice that the second argument to GET in EXPAND-DEFINITION is the constant symbol EXPANDER and that a function stored on an EXPANDER property was changed. Some special cases might be handled this way, but in general the problem can become arbitrarily complex and a correct analysis may be uncomputable. Consider the difficulty required in understanding the implications even of:

```
(EVAL-WHEN (EVAL COMPILE)
  (DEFUN DEFINE-EXPANDER (NAME BODY) ···)
  (DEFUN FOO-EXPANDER ···)
    ⋮
  (MAPC #'(LAMBDA (X Y) (PUTPROP X Y 'EXPANDER))
        '(DEFINE FOO ···)
        '(DEFINE-EXPANDER FOO-EXPANDER ···))).
```

This is the sort of thing that we might imagine a sufficiently advanced compiler being able to do, but we might not be willing to pay for the overhead needed to deduce the information. Performance issues are especially important in systems which must perform interactively and allow for user intervention, incremental development, and runtime redefinition. The cost to the programmer of having to specify module dependencies may be very cheap in comparison with the cost of his having to sit idle waiting for the machine to deduce them.

The detection of loadtime dependencies is made similarly difficult by the ability of the programmer to include unconstrained toplevel forms in a file, to be executed at load time.

Of course, the real problem is probably that unconstrained changes to global state are not well understood and may even be a bad idea. There are many active language design efforts which seek to show that languages which employ global state should just be thrown out the window. Until such efforts succeed, however, the problem of how to specify and manipulate large systems will remain an important one.

And if it ever does happen that languages become sufficiently constrained that all the dependencies can be inferred mechanically, our data abstractions will still provide a needed interface between the new dependency-inference technology and the standard maintenance utilities for loading, compiling, hardcopying, *etc.* The only thing we might expect to change is that complicated notations for describing systems might give way to simpler ones. So even if the dependency information were complex, one might eventually only need to write:

```
(DEFINE-SYSTEM MYSYS ULTIMATE-SYSTEM
  "MACROS.LISP" "UTILITIES.LISP" "MAIN.LISP")
```

and the rest of the information could be inferred mechanically.

Such a syntax might be sufficiently attractive that people would want to rewrite all their systems using it. But in the interim, while people converted the definitions of their systems to use this simpler notation, systems defined using older notations could continue to work correctly and compatibly under protective cover of our abstraction mechanisms.

In preceding examples, we have shown how allowing multiple notations to coexist compatibly in the same environment can be used to allow system maintainers flexibility in choosing a notation which is appropriate for a particular application. Here we see a second reason for allowing multiple notations: to ease the transition from each generation of system description languages to the next.

V. Summary

We have motivated the need for system-definition tools, specified some criteria which such tools should satisfy, and proposed a set of tools which satisfy those criteria.

System maintenance tools should be **data driven**, allowing new tools to be written as extensions to the existing tools, driving off the same data.

The tools should be **general purpose**, allowing arbitrary kinds of systems to be built from them. However, the need for generality should not infect the notation, making common cases notationally too complex to specify conveniently.

We have suggested that these ends should be achieved through a **protocol-based approach**. The proposed approach deemphasizes the particular syntax used to specify a system and emphasizes the importance of making systems with a well-defined functional behavior.

The proposal provides for the construction of systems which satisfy a pre-defined functional protocol. This protocol allows system maintenance utilities to access and manipulate the system specification. The proposed functions provide an interface for finding what files make up a system and inquiring about how to perform system maintenance operations such as **editing**, **compilation**, **instantiation**, and **hardcopying**.

Because a system can be asked to produce a **plan** for an operation such as compilation without actually performing the operation, it is possible to write programs which inspect the plan, possibly optimizing it or presenting it for interactive approval, before executing it.

The proposal also provides for the possibility of having **multiple system description languages** available in the same environment at the same time. This capability allows a system maintainer the freedoom to choose the notation which is right for a given application, without requiring those who need to manipulate (compile, load, *etc.*) the system to know which notation was used.

Examples have been given to illustrate how the various features of this proposal work together in a variety of situations to provide usefulness and flexibility.

References

[Bonanni 77] L. E. **Bonanni** and A. L. **Glasser**, "SCCS/PWB User's Manual," Bell Laboratories, Murray Hill, NJ, November, 1977.

[Feldman 78] S. I. **Feldman**, "Make—A Program for Maintaining Computer Programs," Bell Laboratories, Murray Hill, NJ, August, 1978.

[Mackinlay 84] J. **Mackinlay** and M. **Genesereth**, "Expressiveness of Languages," *Proceedings of the National Conference on Artificial Intelligence*, University of Texas, Austin, TX, August, 1984.

[Rees 84] J. **Rees**, N. Adams and J. **Meehan**, *The T Manual*, Computer Science Department, Yale University, New Haven, CT, January, 1984.

[Robbins 84] R. **Robbins**, *BUILD—A System Construction Tool*, Working Paper 261, Artificial Intelligence Laboratory, MIT, Cambridge, MA, 1984.

[Steele 84] G. L. **Steele**, Jr., *Common LISP: The Language*, Digital Press, Burlington, MA, 1984.

[Theriault 83] D. G. **Theriault**, *Issues in the Design and Implementation of Act2*, Technical Report 728, Artificial Intelligence Laboratory, MIT, Cambridge, MA, June, 1983.

[Weinreb 81] D. **Weinreb** and D. **Moon**, *Lisp Machine Manual*, Fourth Edition, MIT Artificial Intelligence Laboratory, July, 1981.

Acknowledgements

Dan Brotsky, Dan Carnese, Henry Lieberman, Chuck Rich, Patrick Sobalvarro, Dick Waters, and Dan Weld read drafts of this paper, providing support and commentary. Comments by Waters and Brotsky, who read multiple drafts, played an especially important role in improving the clarity and organization of my presentation.

Stephen Gildea provided useful background documentation and answered questions about Unix and its MAKE facility.

Appendix A: Examples

These examples are provided to illustrate how the code in *Appendix B* might be used in practice.

Example 1

Consider a system made of three files, "foo", "bar", and "baz". Such a system could be described conveniently by a DEFINE-SYSTEM form using the SIMPLE-SYSTEM notation as follows:

```
(define-system Example1 simple-system
  "oz:ps:<zippy>foo.lisp" "bar.lisp" "baz.lisp")
```

It could also be described using the MODULAR-SYSTEM notation. For example,

```
(define-system Example1 modular-system
  (:module m1 "oz:ps:<zippy>foo.lisp")
  (:module m2 "bar.lisp"
   (:needs (:instantiate m1)
           (:update m1)))
  (:module m3 "baz.lisp"
   (:needs (:instantiate m2)
           (:update m2))))
```

In this case, the SIMPLE-SYSTEM notation is obviously to be preferred since it makes clear the simplicity of the relationships between the files. In other cases, however, SIMPLE-SYSTEM is not going to be powerful enough to express the inter-module relationships, as our next example will illustrate.

Example 2

There is a file called "MACROS" which requires compile time support from "MACRO-HELPERS". The result of the expansions of these macros needs support from "MACRO-SUPPORT-1" and "MACRO-SUPPORT-2" at compile time and "BASIC" at runtime. Users of "MACROS" should not have to know about these support files, so we want the fact that any time a module says it needs "MACROS", all the other dependencies are added implicitly.

There is a file called "META-MACROS". The macros in that file will expand into calls to macros in "MACROS", though its consumer should not have to know this.

There is a file called "UTILITIES" which needs "MACROS" at compile time and "BASE" at runtime. It uses the functions in "BASE" explicitly, so must specify an explicit dependency upon it even though it happens that "MACROS" provides an implicit dependency.

There is a file called "MAIN" which depends upon "UTILITIES" at runtime and "META-MACROS" at compile time.

```
(define-system Example2 modular-system
  (:full-name "The Second Example")
  (:module base "OZ:PS:<FOO>BASIC.LISP")
  (:module macro-helpers "OZ:PS:<FOO>MACRO-HELPERS.LISP")
  (:module macros "OZ:PS:<FOO>MACROS.LISP"
   (:needs
    (:instantiate macro-helpers))
   (:causes
    (:needs
     (:instantiate base)
     (:update macro-support))))
  (:module macro-support ("OZ:PS:<FOO>MACRO-SUPPORT-1.LISP"
                          "OZ:PS:<FOO>MACRO-SUPPORT-2.LISP"))
  (:module meta-macros "OZ:PS:<FOO>META-MACROS.LISP"
   (:causes
    (:needs
     (:update macros))))
  (:module util "OZ:PS:<FOO>UTILITIES.LISP"
   (:needs
    (:update macros)
    (:instantiate base)))
  (:module main "OZ:PS:<FOO>MAIN.LISP"
   (:needs
    (:instantiate util)
    (:update meta-macros))))
```

The equivalent in DEFSYSTEM would be:

```
(defsystem Example2
  (:short-name "EXAMPLE2")
  (:full-name "The Second Example")
  (:module BASE ("OZ:PS:<FOO>BASIC.LISP"))
  (:module MACRO-HELPERS ("OZ:PS:<FOO>MACRO-HELPERS.LISP"))
  (:module MACROS ("OZ:PS:<FOO>MACROS.LISP"))
  (:module MACRO-SUPPORT ("OZ:PS:<FOO>MACRO-SUPPORT-1.LISP"
                          "OZ:PS:<FOO>MACRO-SUPPORT-2.LISP"))
  (:module META-MACROS ("OZ:PS:<FOO>META-MACROS.LISP"))
  (:module UTILITIES ("OZ:PS:<FOO>UTILITIES.LISP"))
  (:module MAIN ("OZ:PS:<FOO>MAIN.LISP"))
  (:compile-load BASE)
  (:compile-load MACRO-HELPERS)
  (:compile-load MACROS NIL (:FASLOAD MACRO-HELPERS))
  (:compile-load MACRO-SUPPORT)
  (:compile-load META-MACROS)
  (:compile-load-init UTIL (MACRO-SUPPORT MACROS)
                           (:FASLOAD MACRO-SUPPORT MACROS)
                           (:FASLOAD BASE))
  (:compile-load-init MAIN (MACRO-SUPPORT MACROS META-MACROS)
                           (:FASLOAD MACRO-SUPPORT MACROS META-MACROS)
                           (:FASLOAD UTIL BASE)))
```

Note, however, that even in this small example, we can see the characteristic pyramidal shape that DEFSYTEM's dependency clauses tend to take on.

Appendix B: Code

```
;;; -*- Mode:LISP; Package:USER; Base:10; Fonts:MEDFNB -*-
;;;; System
;;; SYSTEM
;;;
;;; Any flavor which claims to satisfy the SYSTEM protocol should
;;; include this flavor in its component flavors list.
(defflavor system () ()
  (:required-methods
     :process-options
     :process-option
     :source-files
     :generate-plan
     :explain-plan
     :execute-plan
     :explain-action
     :execute-action))
```

```
;;;; Vanilla System

;;; VANILLA-SYSTEM
;;;
;;; A vanilla system knows about names and how to process options,
;;; but has no interesting options it is willing to process that would
;;; make it useful as something to instantiate.

(defflavor vanilla-system ((short-name nil) (full-name nil)) (system)
  (:method-combination (:case :base-flavor-last
                              :generate-plan
                              :process-option
                              :explain-action
                              :execute-action))
  :settable-instance-variables)

;;; :NAME (to VANILLA-SYSTEM)
;;;
;;; Returns the name of the system
;;; Long name is preferred over short name where both are available.

(defmethod (vanilla-system :name) () (or full-name short-name))

;;; :PRINT-SELF ...  (to VANILLA-SYSTEM)
;;;
;;; For debugging convenience,
;;; (PRIN1 mysys) types something like: #<SYSTEM "My System" 343324>
;;; (PRINC mysys) types something like: My System

(defmethod (vanilla-system :print-self) (stream level prin1?  &rest ignore)
  level ;ignored
  (let ((my-name (send self :name)))
    (cond ((not prin1?)  (format stream "~A" my-name))
          (t
           (format stream "#<~S ~A ~O>"
                   (typep self) my-name (%pointer self))))))

;;; :DESCRIBE (to VANILLA-SYSTEM)
;;;
;;; Does that part of the explanation relevant to the flavor.
;;; Other flavors mixing this in should use :AFTER or :BEFORE
;;; daemons to modify this method.

(defmethod (vanilla-system :describe) ()
  (format t "~2&~A is a system of type ~S.~%" self (typep self))
  self)
```

```
;;;; Options Facility

;;; :PROCESS-OPTIONS options (to VANILLA-SYSTEM)
;;;
;;; Maps across the given options, digesting them.

(defmethod (vanilla-system :process-options) (options)
  (dolist (data options)
    (lexpr-send self :process-option (car data) (cdr data)))
  self)

;;; :PROCESS-OPTION opt-name .  opt-args (to VANILLA-SYSTEM)
;;;
;;; :NAME Sets defaults for all name types.
;;; :SHORT-NAME Sets the short name (overrides :NAME if given).
;;; :LONG-NAME Sets the long name (overrides :NAME if given).
;;; otherwise Signals an error.

(defmethod (vanilla-system :case :process-option :name) (data)
  (if (not short-name) (setq short-name (string data)))
  (if (not full-name) (setq full-name (string data))))

(defmethod (vanilla-system :case :process-option :short-name) (data)
  (setq short-name (string data)))

(defmethod (vanilla-system :case :process-option :full-name) (data)
  (setq full-name (string data)))

(defmethod (vanilla-system :otherwise :process-option) (key &rest data)
  (ferror "The option ~S is not known to ~S.~%Data: ~S"
          key self (copylist data)))
```

```
;;;; Planning/Executing Actions

;;; :GENERATE-PLAN action (to VANILLA-SYSTEM)
;;;
;;; This method returns abstract information about how to perform
;;; a specified ACTION. The reply is in the form of a list of the
;;; form ((MSG1 . MSG-ARGS1) (MSG2 . MSG-ARGS2) ...), such that
;;; sending each MSG (with the given MSG-ARGS) to the object in
;;; order will accomplish the action in question.
;;;
;;; :UPDATE How to compile (or otherwise update) the system.
;;; :INSTANTIATE How to load (or otherwise instantiate) the system.
;;; otherwise An error results if the action isn't defined.

(defmethod (vanilla-system :otherwise :generate-plan) (key &rest data)
  (ferror "The object ~S does not know how to ~S.~%Data: ~S"
          self key (copylist data)))

;;; :EXECUTE-PLAN plan (to VANILLA-SYSTEM)
;;;
;;; The steps of the PLAN are executed.
;;;
;;; PLAN may be either a plan name (a symbol) or a list of steps
;;; such as that returned by a :GENERATE-PLAN message.

(defmethod (vanilla-system :execute-plan) (plan)
  (cond ((symbolp plan)
         (send self :execute-plan (send self :generate-plan plan)))
        (t
         (dolist (step plan)
           (lexpr-send self :execute-action step))))
  self)

;;; :EXPLAIN-PLAN plan (to VANILLA-SYSTEM)
;;;
;;; The steps of the PLAN are explained.
;;;
;;; PLAN may be either a plan name (a symbol) or a list of steps
;;; such as that returned by a :GENERATE-PLAN message.

(defmethod (vanilla-system :explain-plan) (plan)
  (cond ((symbolp plan)
         (send self :explain-plan (send self :generate-plan plan)))
        (t
         (dolist (step plan)
           (lexpr-send self :explain-action step))))
  self)
```

```
;;; :EXECUTE-ACTION name .  args (to VANILLA-SYSTEM)
;;; :EXPLAIN-ACTION name .  args (to VANILLA-SYSTEM)
;;;
;;; Sending :EXECUTE-ACTION causes a given action to occur.
;;;
;;; Sending :EXPLAIN-ACTION describes what a given action would
;;; do if performed.
;;;
;;; These messages are handled by :CASE method dispatch.
;;;
;;; :LOAD Load a (Lisp) file.
;;; :COMPILE Compile a (Lisp) file.
;;; Otherwise an error.
(defmethod (vanilla-system :case :execute-action :load) (file)
  (load file))

(defmethod (vanilla-system :case :execute-action :compile) (file)
  (compiler:compile-file file))

(defmethod (vanilla-system :otherwise :execute-action) (key &rest data)
  (ferror "The action ~S is not known to ~S.~%Data: ~S"
          key self (copylist data)))

;;; :EXPLAIN-ACTION name .  args (to VANILLA-SYSTEM)
;;;
;;; If the action is valid but a description wasn't available, try
;;; to conjure up a plausible description based on the name of the
;;; action and its arguments.

(defmethod (vanilla-system :otherwise :explain-action) (key &rest data)
  (cond ((not (memq key (send self :execute-action :which-operations)))
         (ferror "The action ~S is not known to ~S, so can't describe it.~
                  ~%Data: ~S"
                 key self (copylist data)))
        (t (format t "~&~A~@[ ~{~A~^, ~}.~]~%"
                   (string-capitalize-words key) data))))
```

```
;;;; Simple System

;;; SIMPLE-SYSTEM
;;;
;;; A simple system is a system which has left-to-right file
;;; dependencies.

(defflavor simple-system ((source-files '())) (vanilla-system)
  :gettable-instance-variables)

;;; :AFTER :DESCRIBE (to SIMPLE-SYSTEM)
;;;
;;; Tacks on some information about the files which make up this system.

(defmethod (simple-system :after :describe) ()
  (format t "~&It has source files~{~<~%~1:; ~S~>~^,~}.~%" source-files))

;;; :PROCESS-OPTIONS (to SIMPLE-SYSTEM)
;;;
;;; The only options allowed to a simple system is a list of file
;;; names with left-to-right ordering dependencies.

(defmethod (simple-system :process-options) (options)
  (when options
    (setq source-files '())
    (let ((default-pathname (fs:merge-pathname-defaults (car options))))
      (dolist (file options)
        (setq default-pathname
              (fs:merge-pathname-defaults file default-pathname))
        (push default-pathname source-files))
      (setq source-files (nreverse source-files)))
    t))

;;; :GENERATE-PLAN :UPDATE (to SIMPLE-SYSTEM)
;;;
;;; To update this kind of system, one must compile and load each
;;; of its files in sequence.

(defmethod (simple-system :case :generate-plan :update) ()
  (mapcan #'(lambda (file)
              (list (list ':compile file)
                    (list ':load (send file :new-pathname
                                       :type :bin))))
          source-files))

;;; :GENERATE-PLAN :INSTANTIATE (to SIMPLE-SYSTEM)
;;;
;;; To instantiate this kind of system, one must simply load each
;;; of its files in sequence.

(defmethod (simple-system :case :generate-plan :instantiate) ()
  (mapcan #'(lambda (file)
              (list (list ':load
                          (send file :new-pathname :type :bin))))
          source-files))
```

```
;;;; Pre-Planned System

;;; PREPLANNED-SYSTEM
;;;
;;; A preplanned system is a system which has its plans for manipulation
;;; specified explicitly rather than inferred.

(defflavor preplanned-system ((files '())
                              (plans '()))
          (vanilla-system))

;;; :PROCESS-OPTIONS (to PREPLANNED-SYSTEM)
;;;
;;; The clauses in the DEFINE-SYSTEM for this kind of system are
;;; just (<plan-name> .  <commands>).

(defmethod (preplanned-system :process-options) (options)
  (setq files (car options))
  (setq plans (cdr options)))

;;; :GENERATE-PLAN (to PREPLANNED-SYSTEM)
;;;
;;; This does simple table-lookup to find the plan.

(defmethod (preplanned-system :generate-plan) (operation)
  (let ((p (assq operation plans)))
    (cond (p (cdr p))
          (t (ferror "No plan for operation ~S" operation)))))
```

```
;;;; Modular System

;;; MODULAR-SYSTEM
;;;
;;; A modular system is a system which allows specification of
;;; inter-module dependencies, both implicit and explicit.

(defflavor modular-system ((modules nil)) (vanilla-system)
  :initable-instance-variables
  :gettable-instance-variables
  :settable-instance-variables)

;;; :AFTER :DESCRIBE (to MODULAR-SYSTEM)
;;;
;;; When a modular system is described, we tack on information
;;; saying how many modules it has and then we ask each module
;;; to describe itself.

(defmethod (modular-system :after :describe) ()
  (let ((m-list (send self :modules)))
    (format t "~&It has ~D module~:P:" (length m-list))
    (dolist (m m-list)
      (send m :describe))))

;;; :GET-MODULE name (to MODULAR-SYSTEM)
;;;
;;; Returns the component module with the given name (or NIL if none).

(defmethod (modular-system :get-module) (name)
  (dolist (m modules)
    (if (eq (send m :name) name) (return m))))

;;; :SOURCE-FILES
;;;
;;; Returns a list of the source files for the system.

(defmethod (modular-system :source-files) ()
  (apply #'append
         (mapcar #'(lambda (module)
                     (send module :source-files))
                 modules)))

;;; :PROCESS-OPTION :MODULE .  spec (to MODULAR-SYSTEM)
;;;
;;; Declares how to handle the :MODULE option.  Creates an object
;;; of type MODULE and lets it process the associated spec.

(defmethod (modular-system :case :process-option :module) (&rest spec)
  (setq modules
        (nconc modules
               (ncons (make-instance 'module
                                     :system self
                                     :spec (copylist spec))))))
```

```
;;;; Module

;;; MODULE
;;;
;;; A module is a collection of files to be used as a building
;;; block for modular systems.

(defflavor module ((name nil)
                   (system nil)
                   (spec nil)
                   (source-files '())
                   (assertions '())
                   (needs '())
                   (causes '()))
          ()
  :initable-instance-variables
  :gettable-instance-variables
  :settable-instance-variables
  (:method-combination (:case :base-flavor-last :process-assertion)))

;;; :AFTER :INIT (to MODULE)
;;;
;;; See to it that if SPEC was given, it gets appropriately processed.

(defmethod (module :after :init) (&rest ignore)
  (send self :process-spec spec))

;;; :PRINT-SELF ...  (to MODULE)
;;;
;;; For debugging convenience,
;;; (PRIN1 mod) types something like: #<Module MYSYS•MOD1 234567>
;;; (PRINC mod) types something like: MOD1

(defmethod (module :print-self) (stream level prin1?  &rest ignore)
  level ;ignored
  (let ((my-name (send self :name)))
    (cond ((not prin1?)  (format stream "~A" my-name))
          (t
           (format stream "#<~S ~@[~A•~]~:[Anonymous~;~:*~A~] ~O>"
                   (typep self)
                   (let ((sys (send self :system)))
                     (if sys (send sys :short-name)))
                   my-name
                   (%pointer self))))))

;;; :DESCRIBE (to MODULE)
;;;
;;; Details the source files and dependency information
;;; for the module.

(defmethod (module :describe) ()
  (format t "~2& ~A~@[  ~{~%~A~↑,~}~]~%" self source-files)
  (do ((n needs (cddr n)))
      ((null n))
    (format t "~& ~S dependenc~@P: ~{~S~↑, ~}.~%"
            (car n) (length (cadr n)) (cadr n)))
  (format t "~&")
  self)
```

```
;;; :PROCESS-SPEC spec (to MODULE)
;;;
;;; Process the given SPEC absorbing relevant info.
;;;
;;; The NAME is only absorbed if name info isn't already set up.
;;; This is because :PROCESS-SPEC may be recursively called on others'
;;; assertion lists if there are included modules with specs of their
;;; own.  In such case, we want to accept their attributes, but not
;;; their names.
;;;
;;; The ASSERTIONS are processed next, because presumably they specify
;;; prerequisites for this module and any files they need loaded should
;;; get set up before we set up the files particular to this module.
;;;
;;; Finally, the FILES associated with this module are processed.
(defmethod (module :process-spec) (s)
  (when s
    (if (not name) (setq name (car s)))
    (send self :process-assertions (cddr s))
    (send self :process-files (cadr s))))
;;; :PROCESS-FILES files-list (to MODULE)
;;;
;;; Adds file info given in FILES-LIST to the module's master FILES list.
(defmethod (module :process-files) (file-list)
  (if (atom file-list) (setq file-list (list file-list)))
  (dolist (file file-list)
    (cond ((typep file 'fs:pathname)
           (setq source-files (nconc source-files (ncons file))))
          ((stringp file)
           (setq source-files (nconc source-files
                                     (ncons (fs:parse-pathname file)))))
          ((symbolp file)
           (send self :process-spec
                 (send (send system :get-module file) :spec)))
          (t
           (ferror "Bad object in file list: ~S - ~S" file self)))))
;;; :PROCESS-ASSERTIONS spec (to MODULE)
;;;
;;; Iterates across assertions, processing each.
(defmethod (module :process-assertions) (assertion-list)
  (dolist (assertion assertion-list)
    (lexpr-send self :process-assertion assertion)))
```

```
;;; :PROCESS-ASSERTION .  data (to MODULE)
;;;
;;; This method is used to process dependency assertions, etc.
;;; for the given module.  It uses case method dispatch:
;;;
;;; :NEEDS Declares need to instantiate modules at certain times.
;;; :CAUSES Declares assertions to be forwarded to the consumer.
;;; Otherwise an error.

(defmethod (module :case :process-assertion :needs) (&rest data)
  (let ((pl (locf needs)))
    (dolist (item data)
      (let ((marker (car item)))
        (dolist (module-name (cdr item))
          (when (not (memq module-name (get pl marker)))
            (let ((m (send (send self :system) :get-module module-name)))
              ;; This may be overly conservative, but will work...
              (send self :process-assertions (send m :causes))
              (setf (get pl marker)
                    (nconc (get pl marker) (ncons module-name))))))))))

(defmethod (module :case :process-assertion :causes) (&rest data)
  ;; Filtering this is technically unnecessary, but it will keep
  ;; redefinition from swamping us.
  (dolist (item data)
    (if (not (mem #'equal item causes))
        (setf causes (nconc causes (ncons item))))))

(defmethod (module :otherwise :process-assertion) (key &rest data)
  (ferror "The ~S assertion is not known to ~S.~%Data: ~S"
          key self (copylist data)))
```

```
;;;; User Interface

;;; (CREATE-SYSTEM name type [options])
;;;
;;; Creates a system object of the given TYPE, initializing it with
;;; with the given NAME and OPTIONS. Returns the created object without
;;; storing it permanently anywhere.

(defun create-system (name type &optional options)
  (let ((system (make-instance type)))
    (send system :process-option :name name)
    (send system :process-options options)
    system))

;;; (SYSTEM name)
;;;
;;; Gets the definition of some globally defined system object.

(defsubst system (name) (get name 'system))

;;; (DEFINE-SYSTEM name type .  options)
;;;
;;; Creates and initializes a system with the given name.
;;; Stores the definition globally for access later.

(defmacro define-system (name type &body data)
  `(setf (system ',name)
         (create-system ',name ',type ',data)))
```

```
;;;; Utility functions

(defun process-options (system options-alist)
  "Tells SYSTEM to process the given OPTIONS-ALIST."
  (send system :process-options options-alist))

(defun process-option (system option-name &rest option-data)
  "Tells SYSTEM to process an individual option, given its NAME and DATA."
  (lexpr-send system :process-option option-name option-data))

(defun source-files (system)
  "Returns a list of the source files for SYSTEM."
  (declare (values files))
  (send system :source-files))

(defun plan (system operation)
  "Returns a list of ACTIONS (plan steps) for doing OPERATION."
  (declare (values actions))
  (send system :generate-plan operation))

(defun execute (system actions)
  "Tells SYSTEM to execute the given ACTIONS (plan steps)."
  (send system :execute-plan actions))

(defun explain (system actions)
  "Tells SYSTEM to explain the given ACTIONS (plan steps)."
  (send system :explain-plan actions))

(defun execute-action (system action)
  "Tells SYSTEM to execute the given ACTION (plan step)."
  (send system :execute-action action))

(defun explain-action (system action)
  "Tells SYSTEM to explain the given ACTION (plan step)."
  (send system :explain-action action))
```