Definitions: a simple database for typesetting documentation

John Alan McDonald *
Dept. of Statistics, University of Washington

October 1991

Abstract

This report describes the Definition module, which provides the beginnings of a database for Common Lisp source code objects. The major use of this database at present is in automatically typsetting reference manuals, an example of which is section 4.

*This work was supported in part the Dept. of Energy under contract FG0685-ER25006 and by NIH grant LM04174 from the National Library of Medicine.

Contents

List of Figures

1 Introduction

This report describes the Definitions module, which provides the beginnings of a database for Common Lisp source code objects. The major use of this database at present is in automatically typesetting reference manuals (using Latex [5]), an example of which is section 4. It is inspired in part by the Definition Groups of Bobrow et. al [1] and the USER-MANUAL of Kantrowitz [3].

The Definitions module is a component of a system called Arizona, now under development at the U. of Washington. Arizona is intended to be a portable, public-domain collection of tools supporting scientific computing, quantitative graphics, and data analysis, implemented in Common Lisp and CLOS (the Common Lisp Object System) [9]. This document assumes the reader is familiar with Common Lisp and CLOS. An overview of Arizona is given in [6] and an introduction to the current release is in [8].

The primary purpose of Definitions is to make possible convenient runtime access to information available in Common Lisp source code, that is lost in the normal process of reading, evaluating, and/or compiling.

Evaluating some Lisp definitions, such as defclass, results in a first class Lisp object with a reasonable and reasonably portable protocol for extracting useful information, such as the direct sub- and super-classes. (At least, this should be true once the MOP is stable [2].) However, most Lisp definitions, such as defun, while producing identifiable objects, have limited facilities for extracting useful information; usually the documentation string is all that is available. And other definitions, such as defstruct, do not even produce an identifiable object.

The Definitions module provides functions to read source files and create a Definition object for each lisp definition in those files, retaining the complete original defining lisp form. Note that the "defining lisp form" is not quite the same thing as the original source code. The "defining lisp form" is a value returned by read, which means, for example, that all comments are removed. The reason for simply using read is that it is much easier than parsing the source myself, deciding where to put top level comments, etc. In the future, I may replace the read functions by ones that captured all the free text source, comments, white space, and all, as well as applying read to this source text to get a readily analyzable lisp object.

1.1 Creating Reference Manuals

The present limited user interface to creating a reference manual from Definitions consists of the function read-definitions-from-files, which returns a list of Definitions that can be filtered with exported-definition? and sorted with definition-alpha< before being printed in a fixed Latex format with print-definitions.

Note that the definitions must be loaded in the current environment before read-definitions-from-files is called to "re-read" the definitions.

The simple user interface is documented more formally in section 4, which was produced by evaluating:

```
(defparameter *def-files* '("package.lisp"
                            "exports.lisp"
                            "defs.lisp"
                            "definition.lisp"
                            "pack.lisp"
                            "global.lisp"
                            "fun.lisp"
                            "type.lisp"
                            "build.lisp"
                            "read.lisp"
                            "filter.lisp"
                            "sort.lisp"
                            "print.lisp"))

(defparameter *defs* (df:read-definitions-from-files *def-files*))

(df:print-definitions
 (sort (remove-if-not #'df:exported-definition? *defs*)
       #'df:definition-alpha<)
 "doc/reference-manual.tex")
```

1.2 Coding conventions

Using the above functions on typical lisp code will produce reference manual entries that are not very informative. The manual entries will be more worthy of the paper they consume if the programmer follows certain coding conventions.

All important definitions should have a substantial documentation string, which should emphasize describing the definition at a conceptual level and should not contain information that can be extracted from the definition form (such as the lambda list of a function).

Types for all (non-specialized) arguments should be specified in a declaration. Generally speaking, any documentation that can be reasonably represented in a declaration should, even if it means defining new declaration types. Although, at present, only type and :returns declarations are relevant to the Definitions module, other declarations may be analyzed in future extensions.

1.2.1 Declaring returned values

Information about returned values should be documented by including a :returns declaration, which is a declaration type that has a special meaning to the Definitions module. A :returns declaration can be used to specify the type(s) of returned value(s), and/or the names of the variable(s) whose value(s) that are returned, or the actual returned value(s), when it is constant. Some examples of the use :returns declarations are:

```
(:returns nil)
(:returns result)
(:returns (type List))
(:returns (type List result))
(:returns (values x y z))
(:returns (values (type Boolean) (type Boolean)))
(:returns (values (type Fixnum left)
                  (type Fixnum top)
                  (type Positive-Fixnum left)
                  (type Positive-Fixnum width)))
```

In order to escape compiler warnings, the programmer should place

```
(declaim (declaration :returns))
```

somewhere that will cause it to be evaluated before any code that contains a :returns declaration.

1.2.2 Type checking forms

Sometimes one would prefer to have type checking forms at the top of a function, rather than declarations. The Arizona-Tools package [8] provides a macro called az:declare-check which has the same syntax as declare but expands into check-type expressions. (Unfortunately it is not easy to also have it expand into the equivalent declarations as well.) The Definitions treats any form whose first item is a symbol whose print name is string-equal to "DECLARE-CHECK" the same as it treats declare forms. Using the symbol's print name makes it possible for programmers who do not have access to the Arizona-Tools package to define their own declare-check macros and have them treated appropriately.

2 Definition Protocol

In this section we discuss the builtin Definition classes and their protocol. The purpose is to allow relatively sophisticated Lisp programmers to customize and extend The Definition Objects module.

The major way to extend the Definition Objects module is to define new subclasses of df:Definition or one of the subclasses listed in section 2.1 that correspond to new definition macros (eg. defannouncement [7]).

The protocol for instances of a Definition class, described in more detail in section 2, consists of a few functions for *reading* files and *building* or instantiating Definitions, a large number of functions for *parsing* and extracting useful information from the defining forms, a few functions for *filtering* and *sorting* sequences of Definitions, and a few functions implementing a standard format for *printing* Definitions to a Latex file.

The major way to customize the behavior of Definitions is to add new generic functions to the protocol that, for example, print definitions in a different format.

2.1 Builtin Classes

The Definitions module provides classes corresponding to the following Common Lisp definition macros (see figure 1: defclass, defconstant, defgeneric, defmacro, defmethod, defpackage, defparameter,

Figure 1: Hierarchy of built in definition classes.

defsetf, defstruct, deftype, and defun. At present, it is missing classes for the following definition macros: define-compiler-macro, define-condition, define-declaration, define-method-combination, define-modify-macro, and define-setf-method.

All the built in classes provide the same state: a single, read-only attribute definition-form. The extensive protocol of generic functions for extracting useful information from the definition-form is discussed in section 2.4.

2.2 Reading

The "protocol" for reading definitions consists of read-definitions-from-file, which reads the definitions in a single file and returns a list of the resulting Definitions, and read-definitions-from-files, which merely concatenates together the results of calling read-definitions-from-file on each file in a list.

Note that definitions are assumed to have been loaded into the environment before they are "re-read" with read-definitions-from-file.

2.3 Building

The basic function for making Definition objects is make-definitions. It is responsible for building a Definition object for every "defined" Lisp object that would result from the evaluation of its form argument. Evaluating a defun form results in a single defined Lisp object — the function, but evaluating other forms, such as a defstruct, may result in the creation of a number of Lisp objects (constructor, predicate, accessor functions, etc.) that should be recorded in a definition database.

For progn-like top level forms, make-definitions simply recursive applies itself to each subform.

For definition forms (ones that begin a symbol defsomething), make-definitions simply makes an instance of the definer-class of the definition-definer (the first element) of the form.

definer-class is a function that maps definition symbols (definers) to Definition classes. The mapping can be extended with setf to associate new Definition classes with new definers. For example: `(setf (df:definer-class 'az:defannouncement) 'az:Announcment-Definition)` would cause an az:announcement-Definition object to be instantiated for every az:defannouncement form that is read.

The new Definition is instantiated with two arguments whose values are retained: the :definition-form, an object returned by read, and the :definition-path, the pathname of the file from which the definition was read.

After instantiating the appropriate Definition class, make-definitions applys the generic function make-subdefinitions to the new Definition to get Definitions corresponding to implied constructor, predicate, accessor functions, etc. For Definitions that have slots — by default only Structure-Definitions and Class-Definitions — make-subdefinitions calls the generic function make-slot-accessor-definitions.

2.4 Parsing

The bulk of the protocol for Definition classes is a set of generic functions that extract useful information from the original definition-form.

Definition-definee attempts to return the lisp object that resulted from evaluating the definition, if possible.

There are a variety of possibilities for the "name" of a definition. Definition-name returns a symbol for most definitions, a list of the form (:method foo Number Array) for method definitions, or a list of the form (setf foo) for definitions of setf functions and methods. Definition-name->string returns a representation of the definition-name as a string, which makes it possible to produce consistent capitalization. Definition-symbol returns the obvious choice of symbol associated with the definition-name, eg., foo in (setf foo). Definition-symbol-name is just shorthand for `(symbol-name (definition-symbol def))`. Definition-class-nice-name returns a short string characterizing ther type of the Definition, eg., "Class" for Class-Definition.

Definition-initial-value returns the initial value specified for definitions like defparameter, or nil if there is no initial value.

Definition-method-qualifier returns the method qualifier (eg. around or after) for Method-Definitions.

Definition-lambda-list returns the raw lambda list associated with the definition (or null for definitions that no lambda list, such as defclass). For convenience, the Definition module provides functions for parsing raw lambda lists: lambda-list-arg-names, lambda-list-whole-arg, lambda-list-required-args, lambda-list-required-arg-names, lambda-list-specializers, lambda-list-optional-args, lambda-list-optional-arg-names, lambda-list-rest-arg, lambda-list-keyword-args, and lambda-list-keyword-arg-names.

Definition-documentation extracts the documentation string associated with the definition, if any.

Definition-declarations returns the "top level" declarations associated with the definition. Definition-type-declarations returns just the type declarations. Definition-returns extracts the declarations that begin with :returns, as discussed in section 1.2.

Definition-arg-types looks at the specializers in the lambda list and the type declarations to come up with a list of expected types for the arguments to the definition.

Definition-parents returns a list of the names of the parents (for a class or structure definition). For new Definition classes, it may return anything that is reasonably thought of as a name of a "parent" of the definition.

Definition-children returns a list of subclasses for a class definition. Unfortunately, it's not easy to find the children of a structure definition. For new Definition classes, it may return anything that is reasonably thought of as a name of a "child" of the definition.

definition-slots returns a list of slot specs, which need to be interpreted in a Definition class specific manner (structure slot specs are different from class slot specs).

2.5 Filtering

The only support for filtering a Definition list at present is exported-definition?, which simply tests to see if the definition-symbol is exported.

2.6 Sorting

The support for ordering definitions consists of the function definition-alpha<, which orders definitions alphabetically by definition-symbol-name, and calls definition< to resolve ties. Definition< sorts definitions with identical definition-symbol-names by type, putting type definitions before all others, generic functions before their methods, and methods roughly in order of invocation — first around, then before, then primary, and finally after methods — with around and before methods ordered most specific first and with primary and after methods ordered most specific last.

2.7 Printing

Print-definition produces Latex for a reference manual entry for the definition. Print-definitions produces reference manual entries for all the definitions in a list.

The print functions are designed to produce Latex output. They expect the code document style, originally due to Olin Shivers of CMU, in ~jam/az/definition/doc/code.sty. Fix-latex-string is used ubiquitously to insert escape characters so that characters that are special to Latex (eg. \) print more-or-less literally.

Print-definition-headline produces a boxed, eye-catching headline intended to highlight the start of an entry in a reference manual.

Print-definition-documentation prints the documentation string subentry. Because Latex special characters are escaped, Latex formatting instructions cannot be included in documentation strings. This may change in the future, most likely by the addition of a keyword to indicate whether Latex special characters should be escaped or passed through raw.

Definition-usage returns a string that should be the printed representation of an example of how to "call" the definition. Methods for definition-usage should be careful to use the actual argument names, to be consistent with the arg types and returns subentries. Print-definition-usage prints the usage subentry.

Print-definition-arg-types prints an itemized list associating argument names with their expected types. Print-definition-returns prints an itemized list of return value names and/or types.

Print-definition-parents and print-definition-children print the names of the definition's parents and children, respectively.

print-definition-source-path prints the pathname of the file from which the definition was read.

3 Extending the basic system

There are two basic ways to extend the ways in which the Definitions module can generate manual entries: (1) new functions can be added to the Definition protocol (2) the existing protocol can be

implemented for a new Definition class (corresponding to a user defined definition macro).

3.1 Additions to the protocol

3.1.1 Expected Additions

It is expected that users will define new protocol functions to customize filtering, sorting, and formatting behavior, using the existing parsing functions. It is expected that users will occasionally define new parsing functions. It is not expected that users will define new reading and building functions.

3.1.2 Censoring

A good system for producing manuals would aid encapsulation by censoring output to remove private information.

For example, a standard style in Lisp programming is to offer a functional interface to an abstract type and suppress all details of how the abstract type is implemented (see for example, Keene [4] and McDonald [8]). In this situation, one would like a documentation formatting system to automatically hide the existence of methods, the existence of slots, whether a given function is generic or not, whether a given type is implemented as a class or a structure or some other Lisp type, and so on.

The present Definitions module does very little censoring of this type. It does not report slots, but otherwise displays too much information about classes vs. structures, class inheritance, generic vs. ordinary functions, method combination, etc., to be very useful for documenting a functional interface.

I expect that one of the first improvements to the Definitions module will be the addition of protocol for censoring.

3.1.3 Grouping related definitions

The current Definitions module treats each Definition object as an independent entity. This means, for example, that a reference manual will contain separate, and somewhat redundant, entries for a generic function and all its methods. This is also the reason that the actual definitions must be loaded before the Definition objects can be created and printed; in order to compute some relationships between Definition objects it is necessary to refer to actual defined objects. For example, we cannot determine a class's direct subclasses by examining the defclass form, but must query the class object itself.

I expect that the Definitions module will be extended with functions that operate on collections of Definition objects to extract information about relationship between Definition objects and that Definition objects will be extended with state that allows them to keep track of other, related definitions of various kinds.

3.2 New definitions

Adding support for a new definition macro requires defining a new Definition class and then making sure that the Definition protocol is implemented for that class, either by inheritance or by writing new methods where appropriate.

3.2.1 Conventions

To make it easy to extend the Definitions module, new definition macros should follow certain conventions. Basically, whenever possible, a new definition macro should model its syntax on the most similar existing definition, and, where there is more than one possible model, it's best to use the more modern choice (eg. use defclass as a model rather than defstruct).

The name of a definition macro should begin with def or define-, as in, for example, defclass.

Evaluating the definition should create an identifiable object. For example, evaluating a defclass creates an class object.

The defined object should have a name which preferably is a symbol, but in special circumstances might be a list or some other lisp object. It's often a good idea to restrict the names to be keywords, unless it will be natural to associate the defined objects with different packages. For example, it's natural to organize class names by putting them in different packages, but the package names themselves are, for all practical purposes, keywords.

There should be a function, like find-class which returns the defined object, given the name. Generally speaking, it's best if the mapping from names to defined objects is represented through some global table. An acceptable alternative is to keep the defined object on the property list of the name, if the name is restricted to be a symbol. The principal advantage of one global table is that it is easier to find and operate on all the objects created by a given definition macro. A common alternative — to be avoided — is to represent the mapping by making the defined object the symbol-value of the name. This is non-robust, because the global binding of a symbol is easy to change accidentally and even easier to inadvertently shadow by local bindings.

The definition macro should permit a documentation string to be supplied.

Definitions that include variables, either as function arguments or as slots or instance variables, should allow declarations for the types of those variables, and, if possible, allow documentation strings to be associated with each variable.

4 Reference Manual

Class-Definition	Class

Documentation: A definition class for <defclass>.

Usage: (typep x 'Class-Definition)

Parents: User-Type-Definition

Constant-Definition	Class

Documentation: A definition class for <defconstant>.

Usage: (typep x 'Constant-Definition)

Parents: Global-Variable-Definition

declare-check	Macro

Documentation: This macro generates type checking forms and has a syntax like <declare>. Unfortunately, we can't easily have it also generate the declarations.

Usage: (declare-check &rest decls)

definer-class	Function

Documentation: Returns the name of the definition class for the <definer> symbol.
The predefined definer classes are: Class-Definition for defclass, Constant-Definition for defconstant, Function-Definition for defun, Generic-Function-Definition for defgeneric, Macro-Definition for defmacro, Method-Definition for defmethod, Package-Definition for defpackage, Parameter-Definition for defparameter, Setf-Definition for defsetf, Structure-Definition for defstruct, Type-Definition for deftype, and Variable-Definition for defvar.

Usage: (definer-class definer)

Arguments: definer — Symbol

Returns: class-name — Symbol

(setf definer-class)	Setf

Documentation: Assign a definer class name to a definer symbol.

Usage: (setf (definer-class definer) class-name)

Definition	Class

Documentation: Abstract root class for definition objects.

Usage: (typep x 'Definition)

Parents: Standard-Object

Children: User-Type-Definition Lambda-List-Definition Global-Variable-Definition Package-Definition

definition-alpha<	Function

Documentation: Order definitions alphabetically by name. If the names are the same, call <definition<> to resolve the ambiguity.

Usage: (definition-alpha< def0 def1)

Arguments:
def0 — Definition
def1 — Definition

Returns: (Member T Nil)

definition-arg-types	Generic Function

Documentation: Returns a list of arg — type pairs for the definition.

Usage: (definition-arg-types def)

Arguments: def — Definition

Returns: List

definition-arg-types Definition	Primary Method

Documentation: The default method returns ().

Usage: (definition-arg-types def)

Arguments: def — Definition

Returns: nil

definition-arg-types Lambda-List-Definition	Primary Method

Documentation: Extract a list of lists of length 2, where each sublist is a name — type pair. The types are gotten first from the arg specializers, if there are any, and are overridden by any top level type declarations.

Usage: (definition-arg-types def)

Arguments: def — Lambda-List-Definition

Returns: List

definition-arg-types Macro-Definition	Primary Method

Documentation: I haven't figured out a good way to get at the equivalent of arg type declarations for macros, so this just returns nil.

Usage: (definition-arg-types def)

Arguments: def — Macro-Definition

Returns: String

definition-arg-types Setf-Definition	Primary Method

Documentation: I haven't figured out a good way to get at the equivalent of arg type declarations for defsetf, so this just returns nil.

Usage: (definition-arg-types def)

Arguments: def — Setf-Definition

Returns: nil

definition-children	Generic Function

Documentation: Returns a list of names of children (eg. direct subclasses) of the definition. For new Definition classes, it may return anything that is reasonably thought of as a name of a "child" of the definition.

Usage: (definition-children def)

Arguments: def — Definition

Returns: List

definition-children Definition	Primary Method

Documentation: The default method returns ().

Usage: (definition-children def)

Arguments: def — Definition

Returns: nil

definition-children Class-Definition	Primary Method

Documentation: Returns the names of the direct subclasses. This method requires the class definition to be loaded and returns the names of all direct subclasses, not just those that have corresponding definition objects.

Usage: (definition-children def)

Arguments: def — Class-Definition

Returns: List

definition-class-nice-name	Generic Function

Documentation: Returns a short string for the type of the definition, eg. "Class" for <Class-Definition>.

Usage: (definition-class-nice-name def)

Arguments: def — Definition

Returns: String

definition-class-nice-name Class-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Class-Definition

Returns: Class

definition-class-nice-name Constant-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Constant-Definition

Returns: Constant

definition-class-nice-name Function-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Function-Definition

Returns: Function

definition-class-nice-name Generic-Function-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Generic-Function-Definition

Returns: nil

definition-class-nice-name Macro-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Macro-Definition

Returns: Macro

definition-class-nice-name Method-Definition	Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Method-Definition

Returns: String

definition-class-nice-name Package-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Package-Definition

Returns: Package

definition-class-nice-name Parameter-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Parameter-Definition

Returns: Parameter

definition-class-nice-name Setf-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Setf-Definition

Returns: Setf

definition-class-nice-name Structure-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Structure-Definition

Returns: Structure

definition-class-nice-name Type-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Type-Definition

Returns: Type

definition-class-nice-name Variable-Definition — Primary Method

Usage: (definition-class-nice-name def)

Arguments: def — Variable-Definition

Returns: Variable

definition-declarations	Generic Function

Documentation: Returns all the decl-specs associated with the entire definition. A decl-spec is a list like (type Fixnum x). By associated with the entire definition, we mean, for example, the declarations with scope over an entire function body, excluding those local to a particular <let>. At present, global declarations (from <proclaim> or <declaim>) are ignored.

Usage: (definition-declarations def)

Arguments: def — Definition

Returns: List

definition-declarations Definition	Primary Method

Documentation: The default method returns ().

Usage: (definition-declarations def)

Arguments: def — Definition

Returns: nil

definition-declarations Lambda-List-Definition	Primary Method

Documentation: Returns all the decl-specs from <declare> forms that come before the first non-string non-<declare> form in the function body. Forms that begin with <declare-check> are also treated as declarations.

Usage: (definition-declarations def)

Arguments: def — Lambda-List-Definition

Returns: List

definition-declarations Method-Definition	Primary Method

Documentation: Returns all the decl-specs from <declare> forms that come before the first non-string non-<declare> form in the function body. Forms that begin with <declare-check> are also treated as declarations.

Usage: (definition-declarations def)

Arguments: def — Method-Definition

Returns: List

definition-declarations Setf-Definition	Primary Method

Documentation: Declarations are only allowed in the long version of defsetf.

Usage: (definition-declarations def)

Arguments: def — Setf-Definition

Returns: List

definition-definee — Generic Function

Documentation: Returns the lisp object that was created when the definition was loaded (which is assumed to have happen before the definition object was created), or nil if it is not possible to retrieve a lisp object corresponding to the definition. For example, one can get the appropriate class object by calling find-class on the <definition-symbol>, but, there is in general no portable way a lisp object associated with the result of evaluating a defstruct.

Usage: (definition-definee def)

Arguments: def — Definition

Returns: T

definition-definee Definition — Primary Method

Documentation: The default method returns nil.

Usage: (definition-definee def)

Arguments: def — Definition

Returns: nil

definition-definee Class-Definition — Primary Method

Documentation: Get the corresponding class object.

Usage: (definition-definee def)

Arguments: def — Class-Definition

Returns: Class

definition-definer — Function

Documentation: Returns the definer symbol, eg. <defun> or <defclass>.

Usage: (definition-definer def)

Arguments: def — Definition

Returns: Symbol

definition-documentation — Generic Function

Documentation: Return the documentation string associated with <def>. Return a string of length zero if there is no documentation string.

Usage: (definition-documentation def)

Arguments: def — Definition

Returns: String

definition-documentation Definition	Primary Method

Documentation: The default is the 4th item in the definition form, if it's a string, otherwise we return the empty string.

Usage: (definition-documentation def)

Arguments: def — Definition

Returns: String

definition-documentation Class-Definition	Primary Method

Documentation: Return the class documentation string or an empty string.

Usage: (definition-documentation def)

Arguments: def — Class-Definition

Returns: String

definition-documentation Lambda-List-Definition	Primary Method

Documentation: The documentation string is the first string that comes before the first non-<declare> form at the top of the function body, unless it's the returned value.

Usage: (definition-documentation def)

Arguments: def — Lambda-List-Definition

Returns: String

definition-documentation Generic-Function-Definition	Primary Method

Documentation: Return the empty string if no :documentation option is present.

Usage: (definition-documentation def)

Arguments: def — Generic-Function-Definition

Returns: String

definition-documentation Method-Definition	Primary Method

Documentation: Finding a method's doc string list requires checking for qualifiers.

Usage: (definition-documentation def)

Arguments: def — Method-Definition

Returns: String

definition-documentation Package-Definition	Primary Method

Documentation: Returns the value of the :documentation option or an empty string.

Usage: (definition-documentation def)

Arguments: def — Package-Definition

Returns: String

definition-documentation Setf-Definition	Primary Method

Documentation: The doc string for defsetf is, in the long version of defsetf, the 5th item in the definition form (if it's a string), and, in the short version, the 4th item (again, if it's a string).

Usage: (definition-documentation def)

Arguments: def — Setf-Definition

Returns: String

definition-documentation Structure-Definition	Primary Method

Documentation: Return the defstruct's doc string, or an empty string.

Usage: (definition-documentation def)

Arguments: def — Structure-Definition

Returns: String

definition-documentation Type-Definition	Primary Method

Usage: (definition-documentation def)

Arguments: def — Type-Definition

Returns: String

definition-form Definition	Primary Method

Documentation: A reader method for the definition-form slot.
The Lisp form that results from reading the definition.

Usage: (definition-form definition)

Arguments: definition — Definition

Returns: List

definition-initial-value	Function

Documentation: The initial value supplied for a global variable (or constant) definition.

Usage: (definition-initial-value def)

Arguments: def — Definition

Returns: T

definition-lambda-list	Generic Function

Documentation: Returns an arglist for the definition, or nil.

Usage: (definition-lambda-list def)

Arguments: def — Definition

Returns: List

definition-lambda-list Definition	Primary Method

Documentation: The default method returns ().

Usage: (definition-lambda-list def)

Arguments: def — Definition

Returns: nil

definition-lambda-list Lambda-List-Definition	Primary Method

Documentation: The lambda list is the third item in most definitions.

Usage: (definition-lambda-list def)

Arguments: def — Lambda-List-Definition

Returns: List

definition-lambda-list Method-Definition	Primary Method

Documentation: Finding a method's lambda list requires checking for qualifiers.

Usage: (definition-lambda-list def)

Arguments: def — Method-Definition

Returns: List

definition-lambda-list Setf-Definition	Primary Method

Documentation: Returns a lamdba list one would have for the equivalent setf method, that is, new value first, followed by the lambda list for the generalized variable.

Usage: (definition-lambda-list def)

Arguments: def — Setf-Definition

Returns: List

definition-method-qualifier	Function

Documentation: The method qualifier, eg., <after>. Returns <:primary> if no qualifier present.

Usage: (definition-method-qualifier def)

Arguments: def — Method-Definition

Returns: Symbol

definition-name	Generic Function

Documentation: Returns the name of a definition object, which is usually either a symbol, eg. <foo> from (defun foo ...), (defclass Foo ...), etc., or a list, eg. (setf foo) from (defmethod (setf foo) ...) or (defsetf foo ...).

Usage: (definition-name def)

Arguments: def — Definition

Returns: (Or Symbol List)

definition-name Definition	Primary Method

Documentation: By default, the <definition-name> is the second item in the definition-form.

Usage: (definition-name def)

Arguments: def — Definition

Returns: (Or Symbol List)

definition-name Method-Definition	Primary Method

Documentation: The <definition-name> of a method is a list whose first item is the symbol :method, whose second item is the function name, and whose remaining items are the specializers for the required arguments.

Usage: (definition-name def)

Arguments: def — Definition

Returns: List

definition-name Setf-Definition	Primary Method

Documentation: The name of a defsetf definition is a list like (setf foo).

Usage: (definition-name def)

Arguments: def — Setf-Definition

Returns: List

definition-name Structure-Definition	Primary Method

Documentation: Getting the name of a defstruct requires a little analysis of the second item in the definition form.

Usage: (definition-name def)

Arguments: def — Structure-Definition

Returns: Symbol

definition-name->string	Generic Function

Documentation: Returns a string containing the name of a definition object, appropriately capitalized.

Usage: (definition-name->string def)

Arguments: def — Definition

Returns: String

definition-name->string Definition	Primary Method

Documentation: The default method simply calls format on the <definition-name>, printing in lower case.

Usage: (definition-name->string def)

Arguments: def — Definition

Returns: String

definition-name->string Method-Definition	Primary Method

Documentation: The name string for methods includes the specializers, so the different methods for a generic function can be distinguished.

Usage: (definition-name->string def)

Arguments: def — Method-Definition

Returns: String

definition-name->string Package-Definition	Primary Method

Documentation: Package name strings should be capitalized.

Usage: (definition-name->string def)

Arguments: def — Package-Definition

Returns: String

definition-name->string Setf-Method-Definition	Primary Method

Documentation: The name string for methods includes the specializers, so the different methods for a generic function can be distinguished.

Usage: (definition-name->string def)

Arguments: def — Setf-Method-Definition

Returns: String

definition-name->string User-Type-Definition — Primary Method

Documentation: Type name strings should be capitalized.

Usage: (definition-name->string def)

Arguments: def — User-Type-Definition

Returns: String

definition-parents — Generic Function

Documentation: Returns a list of the names of the parents (eg. direct superclasses) of the definition. For new Definition classes, it may return anything that is reasonably thought of as a name of a "parent" of the definition.

Usage: (definition-parents def)

Arguments: def — Definition

Returns: List

definition-parents Definition — Primary Method

Documentation: The default method returns ().

Usage: (definition-parents def)

Arguments: def — Definition

Returns: nil

definition-parents Class-Definition — Primary Method

Documentation: Returns the names of the direct superclasses.

Usage: (definition-parents def)

Arguments: def — Class-Definition

Returns: List

definition-returns — Function

Documentation: Returns the :returns decl-spec or nil if there isn't one.

Usage: (definition-returns def)

Arguments: def — Definition

Returns: List

definition-slots — Generic Function

Documentation: Returns a list of slot specs, which need to be interpreted in a Definition class specific manner (structure slot specs are different from class slot specs).

Usage: (definition-slots def)

Arguments: def — Definition

Returns: List

definition-slots Definition — Primary Method

Documentation: The default method returns ().

Usage: (definition-slots def)

Arguments: def — Definition

Returns: nil

definition-slots Class-Definition — Primary Method

Documentation: Return the forms defining the slots of this class.

Usage: (definition-slots def)

Arguments: def — Class-Definition

Returns: List

definition-slots Structure-Definition — Primary Method

Documentation: Return the forms defining the slots of this structure.

Usage: (definition-slots def)

Arguments: def — Structure-Definition

Returns: List

definition-symbol — Generic Function

Documentation: Returns a symbol naming the definition. For definitions whose <definition-name> is a symbol, <definition-symbol> is the same. For definitions whose <definition-name> is a list like (setf foo), <definition-symbol> is <foo>.

Usage: (definition-symbol def)

Arguments: def — Definition

Returns: Symbol

definition-symbol Definition	Primary Method

Documentation: Returns a symbol naming the definition. For definitions whose <definition-name> is a symbol, <definition-symbol> is the same. For definitions whose <definition-name> is a list like (setf foo), <definition-symbol> is <foo>.

Usage: (definition-symbol def)

Arguments: def — Definition

Returns: Symbol

definition-symbol-name	Function

Documentation: Returns the symbol-name of the <definition-symbol>.

Usage: (definition-symbol-name def)

Arguments: def — Definition

Returns: String

definition-type-declarations	Function

Documentation: Returns a list of the type decl specs. At the moment, a type decl spec must have the symbol <type> as it's first item. In the future this may be extended to cover decl specs whose first entry is, for example, <Fixnum>.

Usage: (definition-type-declarations def)

Arguments: def — Definition

Returns: List

definition-usage	Generic Function

Documentation: Returns a string showing how to "call" the definition.

Usage: (definition-usage def)

Arguments: def — Definition

Returns: String

definition-usage Definition	Primary Method

Documentation: The default for usage is just the <definition-name-string>.

Usage: (definition-usage def)

Arguments: def — Definition

Returns: String

definition-usage Lambda-List-Definition	Primary Method

Documentation: Construct a string reflecting a typical function call.

Usage: (definition-usage def)

Arguments: def — Lambda-List-Definition

Returns: String

definition-usage Macro-Definition	Primary Method

Documentation: Construct a string for a typical call to the macro.

Usage: (definition-usage def)

Arguments: def — Macro-Definition

Returns: String

definition-usage Package-Definition	Primary Method

Documentation: The example of package use is a call to <in-package>.

Usage: (definition-usage def)

Arguments: def — Package-Definition

Returns: String

definition-usage User-Type-Definition	Primary Method

Documentation: The example of use of a type definition is a call to <typep>.

Usage: (definition-usage def)

Arguments: def — User-Type-Definition

Returns: String

definition<	Generic Function

Documentation: Resolve the ambiguity in alphabetic ordering for multiple definitions with the same name.

Usage: (definition< def0 def1)

Arguments:

def0 — Definition
def1 — Definition

Returns: (Member T Nil)

definition< Definition Definition — Primary Method

Documentation: The default method returns nil (not comparable).

Usage: (definition< def0 def1)

Arguments:
def0 — Definition
def1 — Definition

Returns: nil

definition< Lambda-List-Definition Lambda-List-Definition — Primary Method

Documentation: Normal Functions come before setf functions.

Usage: (definition< def0 def1)

Arguments:
def0 — Lambda-List-Definition
def1 — Lambda-List-Definition

Returns: (Member T Nil)

definition< Generic-Function-Definition Method-Definition — Primary Method

Documentation: Generic Functions come before Methods.

Usage: (definition< def0 def1)

Arguments:
def0 — Generic-Function-Definition
def1 — Method-Definition

Returns: t

definition< Method-Definition Method-Definition — Primary Method

Documentation: Method ordering attempts to mimic calling order for combinable methods and be alphabetic otherwise.

Usage: (definition< def0 def1)

Arguments:
def0 — Method-Definition
def1 — Method-Definition

Returns: (Member T Nil)

definition< User-Type-Definition Definition — Primary Method

Documentation: User defined types (deftype, defstruct, defclass) come before others.

Usage: (definition< def0 def1)

Arguments:
def0 — User-Type-Definition
def1 — Definition

Returns: t

:Definitions — Package

Usage: (in-package :Definitions)

exported-definition? — Function

Documentation: Has the <definition-symbol> of <def> been exported?

Usage: (exported-definition? def &key package)

Arguments:
def — Definition
package — T

Returns: (Member T Nil)

fix-latex-string — Function

Documentation: Massage a string so that TeX special characters will come out as something reasonable. See cite{Lamp86} pp. 15, 65.

Usage: (fix-latex-string s0)

Arguments: s0 — String

Returns: String

Function-Definition — Class

Documentation: A definition class for <defun>.

Usage: (typep x 'Function-Definition)

Parents: Lambda-List-Definition

Generic-Function-Definition — Class

Documentation: A definition class for <defgeneric>.

Usage: (typep x 'Generic-Function-Definition)

Parents: Lambda-List-Definition

Global-Variable-Definition	Class

Documentation: An abstract super class for global variable definitions.

Usage: (typep x 'Global-Variable-Definition)

Parents: Definition

Children: Variable-Definition Parameter-Definition Constant-Definition

lambda-list-arg-names	Function

Documentation: A list of all the arg names.

Usage: (lambda-list-arg-names lambda-list)

Arguments: lambda-list — List

Returns: List

Lambda-List-Definition	Class

Documentation: An abstract super class for definitions that include lambda lists.

Usage: (typep x 'Lambda-List-Definition)

Parents: Definition

Children: Type-Definition Method-Definition Generic-Function-Definition Setf-Definition Macro-Definition Function-Definition

lambda-list-keyword-arg-names	Function

Documentation: A list of the names only of the &keyword args.

Usage: (lambda-list-keyword-arg-names lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-keyword-args	Function

Documentation: A list of the &keyword args, with default values, etc.

Usage: (lambda-list-keyword-args lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-optional-arg-names Function

Documentation: A list of the names only of the &optional args.

Usage: (lambda-list-optional-arg-names lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-optional-args Function

Documentation: A list of the &optional args, with default values, etc.

Usage: (lambda-list-optional-args lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-required-arg-names Function

Documentation: a list of the name oif the required args (specializers are stripped off.)

Usage: (lambda-list-required-arg-names lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-required-args Function

Documentation: A list of the required args (with specializers when given).

Usage: (lambda-list-required-args lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-rest-arg-name Function

Documentation: The name of the &rest or &body arg.

Usage: (lambda-list-rest-arg-name lambda-list)

Arguments: lambda-list — List

Returns: Symbol

lambda-list-specializers Function

Documentation: A list of the specializers for the required args, with T given for any unspecialized args.

Usage: (lambda-list-specializers lambda-list)

Arguments: lambda-list — List

Returns: List

lambda-list-whole-arg	Function

Documentation: The name of the &whole arg in a (macro's) lambda list.

Usage: (lambda-list-whole-arg lambda-list)

Arguments: lambda-list — List

Returns: Symbol

Macro-Definition	Class

Documentation: A definition class for <defmacro>.

Usage: (typep x 'Macro-Definition)

Parents: Lambda-List-Definition

make-definitions	Function

Documentation: Make and return a list of the definition objects corresponding to the result of evaluating <form> (which was read from the file corresponding to <path>.

Usage: (make-definitions form path)

Arguments:
form — List
path — Pathname

Returns: definitions — List

make-slot-accessor-definitions	Generic Function

Documentation: Make definition objects corresponding to automatically generated accessor functions.

Usage: (make-slot-accessor-definitions def slot-spec path)

Arguments:
def — User-Type-Definition
slot-spec — List
path — Pathname

Returns: definitions — List

make-slot-accessor-definitions Class-Definition T T	Primary Method

Documentation: Make definition objects corresponding to automatically generated accessor functions for a class.

Usage: (make-slot-accessor-definitions def slot-spec path)

Arguments:
- def — Class-Definition
- slot-spec — List
- path — Pathname

Returns: definitions — List

make-slot-accessor-definitions Structure-Definition T T	Primary Method

Documentation: Make definition objects corresponding to automatically generated accessor functions for a structure.

Usage: (make-slot-accessor-definitions def slot-spec path)

Arguments:
- def — Structure-Definition
- slot-spec — List
- path — Pathname

Returns: definitions — List

make-subdefinitions	Generic Function

Documentation: Make definition objects for definitions automatically generated by the evaluation of the form corresponding to <def>. An example of a subdefinition is a slot accessor function automatically generated by a class definition.

Usage: (make-subdefinitions def path)

Arguments:
- def — Definition
- path — Pathname

Returns: definitions — List

make-subdefinitions Definition T	Primary Method

Documentation: The default method for <make-subdefinitions> returns ().

Usage: (make-subdefinitions def path)

Arguments:
- def — Definition
- path — Pathname

Returns: nil

make-subdefinitions Class-Definition T	Primary Method

Documentation: The method for classes returns a list of definition objects for the slot accessor functions.

Usage: (make-subdefinitions def path)

Arguments:
def — Definition
path — Pathname

Returns: definitions — List

make-subdefinitions Structure-Definition T	Primary Method

Documentation: The method for structures returns a list of definition objects for the automatically generated constructor, copier, and predicate functions, if they are generated, and the slot accessor functions.

Usage: (make-subdefinitions def path)

Arguments:
def — Definition
path — Pathname

Returns: definitions — List

Method-Definition	Class

Documentation: A definition class for <defmethod>.

Usage: (typep x 'Method-Definition)

Parents: Lambda-List-Definition

Children: Setf-Method-Definition

Package-Definition	Class

Documentation: A definition class for <defpackage>.

Usage: (typep x 'Package-Definition)

Parents: Definition

Parameter-Definition	Class

Documentation: A definition class for <defparameter>.

Usage: (typep x 'Parameter-Definition)

Parents: Global-Variable-Definition

print-definition — Generic Function

Documentation: Print a definition object in latex.

Usage: (print-definition def &key stream)

Arguments:
def — Definition
stream — Stream

print-definition Definition — Primary Method

Documentation: Prints out the reference manual entry for a definition object in Latex.

Usage: (print-definition def &key stream)

Arguments:
def — Definition
stream — Stream

Returns: def

print-definition-arg-types — Generic Function

Documentation: Print a list of args and their expected types.

Usage: (print-definition-arg-types def &key stream)

Arguments:
def — Definition
stream — Stream

print-definition-arg-types Definition — Primary Method

Documentation: Print a list of args and their expected types.

Usage: (print-definition-arg-types def &key stream)

Arguments:
def — Definition
stream — Stream

Returns: def

print-definition-children — Generic Function

Documentation: Print a list of parent definitions.

Usage: (print-definition-children def &key stream)

Arguments:
def — Definition
stream — Stream

print-definition-children Definition	Primary Method

Documentation: Print a list of parent definitions.

Usage: (print-definition-children def &key stream)

Arguments:
def — Definition
stream — Stream

Returns: def

print-definition-documentation	Generic Function

Documentation: Print the documentation string for the definition. Because Latex special characters are escaped, Latex formatting instructions cannot be included in documentation strings. This may change in the future, most likely by the addition of a keyword to indicate whether Latex special characters should be escaped or passed through raw.

Usage: (print-definition-documentation def &key stream)

Arguments:
def — Definition
stream — Stream

Returns: def — Definition

print-definition-documentation Definition	Primary Method

Documentation: Print the documentation string for the definition.

Usage: (print-definition-documentation def &key stream)

Arguments:
def — Definition
stream — Stream

Returns: def

print-definition-headline	Generic Function

Documentation: Print a Headline for the definition.

Usage: (print-definition-headline def &key stream)

Arguments:
def — Definition
stream — Stream

print-definition-headline Definition	Primary Method

Documentation: Print a Headline for the definition.

Usage: (print-definition-headline def &key stream)

Arguments:
- def — Definition
- stream — Stream

Returns: def

print-definition-parents	Generic Function

Documentation: Print a list of parent definitions.

Usage: (print-definition-parents def &key stream)

Arguments:
- def — Definition
- stream — Stream

print-definition-parents Definition	Primary Method

Documentation: Print a list of parent definitions.

Usage: (print-definition-parents def &key stream)

Arguments:
- def — Definition
- stream — Stream

Returns: def

print-definition-returns	Generic Function

Documentation: Print a list of returned values and/or their types.

Usage: (print-definition-returns def &key stream)

Arguments:
- def — Definition
- stream — Stream

print-definition-returns Definition	Primary Method

Documentation: Print a list of returned values and/or their types.

Usage: (print-definition-returns def &key stream)

Arguments:
- def — Definition
- stream — Stream

Returns: def

print-definition-source-path	Generic Function

Documentation: Print the pathname from which this definition was read.

Usage: (print-definition-source-path def &key stream)

Arguments:
- def — Definition
- stream — Stream

print-definition-source-path Definition	Primary Method

Documentation: Print the pathname from which this definition was read.

Usage: (print-definition-source-path def &key stream)

Arguments:
- def — Definition
- stream — Stream

Returns: def

print-definition-usage	Generic Function

Documentation: Print a description of how to "call" the definition.

Usage: (print-definition-usage def &key stream)

Arguments:
- def — Definition
- stream — Stream

print-definition-usage Definition	Primary Method

Documentation: Print a description of how to "call" the definition.

Usage: (print-definition-usage def &key stream)

Arguments:
- def — Definition
- stream — Stream

Returns: def

print-definitions	Function

Documentation: Print a latex representation of the definition objects in <defs> on the file corresponding to <path>.

Usage: (print-definitions defs path)

Arguments:
- defs — List
- path — (Or String Pathname)

Returns: defs

print-object Definition T — Primary Method

Documentation: A generic method for printing Definition objects.

Usage: (print-object def stream)

Arguments:
def — Definition
stream — Stream

Returns: def

read-definitions-from-file — Function

Documentation: Read all the forms in the file corresponding to <path>, and create and return a list of definition objects corresponding to the result of evaluating those forms (for which <definer-class> does not return <nil>).

Usage: (read-definitions-from-file path)

Arguments: path — (Or String Pathname)

Returns: List

read-definitions-from-files — Function

Documentation: Call <read-definitions-from-file> on each path in <paths>, concatenating together the results.

Usage: (read-definitions-from-files paths)

Arguments: paths — List

Returns: List

Setf-Definition — Class

Documentation: A definition class for <defsetf>.

Usage: (typep x 'Setf-Definition)

Parents: Lambda-List-Definition

Setf-Method-Definition — Class

Documentation: A definition class for <defmethod> of setf generic functions.

Usage: (typep x 'Setf-Method-Definition)

Parents: Method-Definition

Structure-Definition	Class

Documentation: A definition class for <defstruct>.

Usage: (typep x 'Structure-Definition)

Parents: User-Type-Definition

type-check	Macro

Documentation: A <check-type> that takes arguments more like declarations, eg, (declare (type Integer x y))

Usage: (type-check type &rest args)

Type-Definition	Class

Documentation: A definition class for <deftype>.

Usage: (typep x 'Type-Definition)

Parents: User-Type-Definition Lambda-List-Definition

User-Type-Definition	Class

Documentation: An abstract super class for user defined types.

Usage: (typep x 'User-Type-Definition)

Parents: Definition

Children: Structure-Definition Class-Definition Type-Definition

Variable-Definition	Class

Documentation: A definition class for <defvar>.

Usage: (typep x 'Variable-Definition)

Parents: Global-Variable-Definition

References

[1] Daniel Bobrow, David Fogelsong, and Mark Miller. Definition Groups: Making sources into first-class objects. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge MA, 1987.

[2] Daniel G. Bobrow and Gregor Kiczales. The common lisp object system metaobject kernel: A status report. In *Proc. 1988 ACM Symp. on Lisp and Functional Programming*, pages 309–315, 1988.

[3] Mark Kantrowitz. Portable Utilities for Common Lisp, User Guide and Implementation Notes. Technical Report CMU-CS-91-143, School of Computer Science, CMU, 1991.

[4] Sonya E. Keene. *Object-oriented programming in Common Lisp: a programmer's guide to CLOS.* Symbolics Press and Addison-Wesley, Reading, MA, 1988.

[5] Leslie Lamport. *Latex, a Document Preparation System.* Addison-Wesley, Reading, MA, 1985.

[6] John Alan McDonald. An outline of Arizona. In *Computer Science and Statistics: Proc. 20th Symp. on the Interface*, pages 282–291, Washington, D.C., 1988. ASA.

[7] John Alan McDonald and Mark Niehaus. Announcements: an implementation of implicit invocation. Technical report, Dept. of Statistics, U. of Washington, October 1991.

[8] John Alan McDonald and Michael Sannella. Arizona overview and notes for release 1.0. Technical report, Dept. of Statistics, U. of Washington, October 1991.

[9] G.L. Steele. *Common Lisp, The Language.* Digital Press, second edition, 1990.