11. Functions

Functions are the basic building blocks of Lisp programs. This chapter describes the functions in Zetalisp that are used to manipulate functions. It also explains how to manipulate special forms and macros.

This chapter contains internal details intended for those writing programs to manipulate programs as well as material suitable for the beginner. Feel free to skip sections that look complicated or uninteresting when reading this for the first time.

11.1 What Is a Function?

There are many different kinds of functions in Zetalisp. Here are the printed representations of examples of some of them:

```
foo
(lambda (x) (car (last x)))
(named-lambda foo (x) (car (last (x))))
(subst (x) (car (last x)))
#<dtp-fef-pointer append 1424771>
#<dtp-u-entry last 270>
#<dtp-closure 1477464>
```

We will examine these and other types of functions in detail later in this chapter. There is one thing they all have in common: a function is a Lisp object that can be applied to arguments. All of the above objects may be applied to some arguments and will return a value. Functions are Lisp objects and so can be manipulated in all the usual ways; you can pass them as arguments, return them as values, and make other Lisp objects refer to them.

11.2 Function Specs

The name of a function does not have to be a symbol. Various kinds of lists describe other places where a function can be found. A Lisp object that describes a place to find a function is called a *function spec*. ('Spec' is short for 'specification'.) Here are the printed representations of some typical function specs:

```
foo
(:property foo bar)
(:method tv:graphics-mixin :draw-line)
(:internal foo 1)
(:within foo bar)
(:location #<dtp-locative 7435216>)
```

Function specs have two purposes: they specify a place to *remember* a function, and they serve to *name* functions. The most common kind of function spec is a symbol, which specifies that the function cell of the symbol is the place to remember the function. We will see all the different types of function spec, and what they mean, shortly. Function specs are not the same thing as functions. You cannot, in general, apply a function spec to arguments. The time to use a function spec is when you want to *do* something to the function, such as define it, look at its

definition, or compile it.

Some kinds of functions remember their own names, and some don't. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. In the examples of functions in the previous section, the one starting with the symbol named-lambda, the one whose printed representation included dtp-fef-pointer, and the dtp-u-entry remembered names (the function specs foo, append, and last respectively). The others didn't remember their names.

To *define a function spec* means to make that function spec remember a given function. Programs do this by calling fdefine; you give fdefine a function spec and a function, and fdefine remembers the function in the place specified by the function spec. The function associated with a function spec is called the *definition* of the function spec. A single function can be the definition of more than one function spec at the same time, or of no function specs.

The definition of a function spec can be obtained with fdefinition. (function *function-spec*) does so too, but here *function-spec* is not evaluated. For example, (function foo) evaluates to the function definition of foo. fdefinition is used by programs whose purpose is to examine function definitions, whereas function is used in this way by programs of all sorts to obtain a specific definition and use it. See page 48.

To *define a function* means to create a new function and define a given function spec as that new function. This is what the defun special form does. Several other special forms, such as defmethod (page 415) and defselect (page 236), do this too.

These special forms that define functions usually take a function spec, create a function whose name is that function spec, and then define that function spec to be the newly-created function. Most function definitions are done this way, and so usually if you go to a function spec and see what function is there, the function's name is the same as the function spec. However, if you define a function named foo with defun, and then define the symbol bar to be this same function, the name of the function is unaffected; both foo and bar are defined to be the same function, and the name of that function is foo, not bar.

A function spec's definition in general consists of a *basic definition* surrounded by *encapsulations*. Both the basic definition and the encapsulations are functions, but of recognizably different kinds. What defun creates is a basic definition, and usually that is all there is. Encapsulations are made by function-altering functions such as trace, breakon and advise. When the function is called, the entire definition, which includes the tracing and advice, is used. If the function is redefined with defun, only the basic definition is changed; the encapsulations are left in place. See the section on encapsulations, section 11.9, page 244.

A function spec is a Lisp object of one of the following types:

a symbol
: The function is remembered in the function cell of the symbol. See page 130 for an explanation of function cells and the primitive functions to manipulate them.

(:property *symbol property*)
: The function is remembered on the property list of the symbol; doing (get *symbol property*) returns the function. Storing functions on property lists is a frequently-used

technique for dispatching (that is, deciding at run-time which function to call, on the basis of input data).

(:method *flavor-name operation*)
(:method *flavor-name method-type operation*)
(:method *flavor-name method-type operation suboperation*)
The function is remembered inside internal data structures of the flavor system and is called automatically as part of handling the operation *operation* on instances of *flavor-name*. See the chapter on flavors (chapter 21, page 401) for details.

(:handler *flavor-name operation*)
This is a name for the function actually called when an *operation* message is sent to an instance of the flavor *flavor-name*. The difference between :handler and :method is that the handler may be a method inherited from some other flavor or a *combined method* automatically written by the flavor system. Methods are what you define in source files; handlers are not. Note that redefining or encapsulating a handler affects only the named flavor, not any other flavors built out of it. Thus :handler function specs are often used with trace (see page 738), breakon (page 741), and advise (page 742).

(:select-method *function-spec operation*)
This function spec assumes that the definition of *function-spec* is a select-method object (see page 232) containing an alist of operation names and functions to handle them, and refers to one particular element of that alist: the one for operation *operation*.

The function is remembered in that alist element and is called when *function-spec*'s definition is called with first argument *operation*.

:select-method function specs are most often used implicitly through defselect. One of the things done by

```
(defselect foo
  (:win (x) (cons 'win x))
  ...)
```

is to define the function spec (:select-method foo :win).

:select-method function specs are explicitly given function definitions when you use defselect-incremental instead of defselect, as in

```
(defselect-incremental foo)
(defun (:select-method foo :win) (ignore x)
  (cons 'win x))
```

(:lambda-macro *name*)
This is a name for the function that expands the lambda macro *name*.

(:location *pointer*)
The function is stored in the cdr of *pointer*, which may be a locative or a list. This is for pointing at an arbitrary place that there is no other way to describe. This form of function spec isn't useful in defun (and related special forms) because the reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions (see section 11.7, page 239).

(:within *within-function function-to-affect*)
This refers to the meaning of the symbol *function-to-affect*, but only where it occurs in

the text of the definition of *within-function*. If you define this function spec as anything but the symbol *function-to-affect* itself, then that symbol is replaced throughout the definition of *within-function* by a new symbol, which is then defined as you specify. See the section on si:rename-within encapsulations (section 11.9.1, page 249) for more information.

It is rarely useful to define a :within function spec by hand, but often useful to trace or advise one. For example,

```
(breakon '(:within myfunction eval))
```

allows you to break when eval is called from myfunction. Simply doing (breakon 'eval) will probably blow away your machine.

(:internal *function-spec number*)

Some Lisp functions contain internal functions, created by (function (lambda ...)) forms. These internal functions need names when compiled, but they do not have symbols as names; instead they are named by :internal function-specs. *function-spec* is the name of the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function is numbered 0, the next 1, etc. Internal functions are remembered inside the compiled function object of their containing function.

(:internal *function-spec symbol*)

If a Lisp function uses flet to name an internal function, you can use the local name defined with flet in the :internal function spec instead of a number. Here is an example of such a function:

```
(defun foo (a)
  (flet ((square (x) (* x x)))
    (+ a (square a))))
```

After compiling foo, you could use the function spec (:internal foo square) to refer to the internal function locally named square. You could also use (:internal foo 0). If there are multiple flet's defining local functions with the same name, only the first can be referred to by name this way.

Here is an example defining a function whose name is not a symbol:

```
(defun (:property foo bar-maker) (thing &optional kind)
  (set-the 'bar thing (make-bar 'foo thing kind)))
```

This puts a function on foo's bar-maker property. Now you can say

```
(funcall (get 'foo 'bar-maker) 'baz)
```

or

```
(funcall #'(:property foo bar-maker) 'bax)
```

Unlike the other kinds of function spec, a symbol *can* be used as a function. If you apply a symbol to arguments, the symbol's function definition is used instead. If the definition of the first symbol is another symbol, the definition of the second symbol is used, and so on, any number of times. But this is an exception; in general, you can't apply function specs to arguments.

A keyword symbol that identifies function specs (i.e., that may appear in the car of a list which is a function spec) is identified by a sys:function-spec-handler property whose value is a function that implements the various manipulations on function specs of that type. The interface to this function is internal and not documented in this manual.

For compatibility with Maclisp, the function-defining special forms defun, macro, and defselect (and other defining forms built out of them, such as defmacro) also accept a list

(*symbol property*)

as a function name. This is translated into

(:property *symbol property*)

symbol must not be one of the keyword symbols that identify a function spec, since that would be ambiguous.

11.3 Simple Function Definitions

defun *Special form*

The usual way of defining a function that is part of a program. A defun form looks like:

```
(defun name lambda-list
  body...)
```

name is the function spec you wish to define as a function. The *lambda-list* is a list of the names to give to the arguments of the function. Actually, it is a little more general than that; it can contain *lambda-list keywords* such as &optional and &rest. (These keywords are explained in section 3.3, page 38 and other keywords are explained in section 3.3.1, page 43.) See page 234 for some additional syntactic features of defun.

defun creates a list that looks like

(named-lambda *name lambda-list body...*)

and puts it in the function cell of *name*. *name* is now defined as a function and can be called by other forms.

Examples:

```
(defun addone (x)
  (1+ x))

(defun foo (a &optional (b 5) c &rest e &aux j)
  (setq j (+ (addone a) b))
  (cond ((not (null c))
         (cons j e))
        (t j)))
```

addone is a function which expects a number as an argument, and returns a number one larger. foo is a complicated function that takes one required argument, two optional arguments, and any number of additional arguments that are given to the function as a list named e.

A declaration (a list starting with declare) can appear as the first element of the body. It applies to the entire function definition; if it is a special declaration, it applies to bindings made in the lambda list and to free references anywhere in the function. For example,

```
(defun foo (x)
  (declare (special x))
  (bar))                ;bar uses x free.
```

causes the binding of x to be a dynamic binding, and

```
(defun foo (&rest args)
  (declare (arglist a b c))
  (apply 'bar args))
```

causes (arglist 'foo) to return (a b c) rather than (&rest args), presumably because the former is more informative in the particular application.

A documentation string can also appear at the beginning of the body; it may precede or follow a declaration. This documentation string becomes part of the function's debugging info and can be obtained with the function documentation (see page 784). The first line of the string should be a complete sentence that makes sense read by itself, since there are two editor commands to get at the documentation, one of which is "brief" and prints only the first line. Example:

```
(defun my-append (&rest lists)
   "Like append but copies all the lists.
This is like the Lisp function append, except that
append copies all lists except the last, whereas
this function copies all of its arguments
including the last one."
   ...)
```

A documentation string may not be the last element of the body; a string in that position is interpreted as a form to evaluate and return and is not considered to be a documentation string.

For more information on defining functions, and other ways of doing so, see section 11.6, page 234.

11.4 User Operations on Functions

Here is a list of the various things a user (as opposed to a program) is likely to want to do to a function. In all cases, you specify a function spec to say where to find the function.

To print out the definition of the function spec with indentation to make it legible, use grindef (see page 528). This works only for interpreted functions. If the definition is a compiled function, it can't be printed out as Lisp code, but its compiled code can be printed by the disassemble function (see page 792).

To find out about how to call the function, you can ask to see its documentation or its argument names. (The argument names are usually chosen to have mnemonic significance for the caller). Use arglist (page 242) to see the argument names and documentation (page 784) to see the documentation string. There are also editor commands for doing these things: the Control-Shift-D and Meta-Shift-D commands are for looking at a function's documentation, and Control-Shift-A is for looking at an argument list.

Control-Shift-A and Control-Shift-D do not ask for the function name; they act on the function that is called by the innermost expression which the cursor is inside. Usually this is the function that will be called by the form you are in the process of writing. They are available in the rubout handler as well.

You can see the function's debugging info alist by means of the function debugging-info (see page 242).

When you are debugging, you can use trace (see page 738) to obtain a printout or a break loop whenever the function is called. You can use breakon (see page 741) to cause the error handler to be entered whenever the function is called; from there, you can step through further function calls and returns. You can customize the definition of the function, either temporarily or permanently, using advise (see page 742).

11.5 Kinds of Functions

There are many kinds of functions in Zetalisp. This section briefly describes each kind of function. Note that a function is also a piece of data and can be passed as an argument, returned, put in a list, and so forth.

There are four kinds of functions, classified by how they work.

First, there are *interpreted* functions: you define them with defun, they are represented as list structure, and they are interpreted by the Lisp evaluator.

Secondly, there are *compiled* functions: they are defined by compile or by loading a QFASL file, they are represented by a special Lisp data type, and they are executed directly by the microcode. Similar to compiled functions are microcode functions, which are written in microcode (either by hand or by the micro-compiler) and executed directly by the hardware.

Thirdly, there are various types of Lisp object that can be applied to arguments, but when they are applied they dig up another function somewhere and apply it instead. These include select-methods, closures, instances, and entities.

Finally, there are various types of Lisp object that, when called as functions, do something special related to the specific data type. These include arrays and stack-groups.

11.5.1 Interpreted Functions

An interpreted function is a piece of list structure that represents a program according to the rules of the Lisp interpreter. Unlike other kinds of functions, interpreted functions can be printed out and read back in (they have a printed representation that the reader understands), can be pretty-printed (see page 528), and can be examined with the usual functions for list-structure manipulation.

There are four kinds of interpreted functions: lambdas, named-lambdas, substs, and named-substs. A lambda function is the simplest kind. It is a list that looks like this:

(lambda *lambda-list form1 form2...*)
The symbol lambda identifies this list as a lambda function. *lambda-list* is a description of what arguments the function takes; see section 3.3, page 38 for details. The *forms* make up the body of the function. When the function is called, the argument variables are bound to the values of the arguments as described by *lambda-list*, and then the forms in the body are evaluated, one by one. The values of the function are the values of its last form.

A named-lambda is like a lambda but contains an extra element in which the system remembers the function's name, documentation, and other information. Having the function's name there allows the error handler and other tools to give the user more information. You would not normally write a named-lambda yourself; named-lambda exists so that defun can use it. A named-lambda function looks like this:

(named-lambda *name lambda-list bodyforms...*)

If the *name* slot contains a symbol, it is the function's name. Otherwise it is a list whose car is the name and whose cdr is the function's debugging information alist. (See debugging-info, page 242.) Note that the name need not be a symbol; it can be any function spec. For example,

```
(defun (foo bar) (x)
  (car (reverse x)))
```

gives foo a bar property whose value is

```
(named-lambda ((:property foo bar)) (x) (car (reverse x)))
```

A subst is a function which is open-coded by the compiler. A subst is just like a lambda as far as the interpreter is concerned. It is a list that looks like this:

(subst *lambda-list form1 form2...*)

The difference between a subst and a lambda is the way they are handled by the compiler. A call to a normal function is compiled as a *closed subroutine*; the compiler generates code to compute the values of the arguments and then apply the function to those values. A call to a subst is compiled as an *open subroutine*; the compiler incorporates the body forms of the subst into the function being compiled, substituting the argument forms for references to the variables in the subst's *lambda-list*. subst's are described more fully on page 329, with the explanation of defsubst.

A named-subst is the same as a subst except that it has a name just as a named-lambda does. It looks like

(named-subst *name lambda-list form1 form2 ...*)

where *name* is interpreted the same way as in a named-lambda.

11.5.2 Lambda Macros

Lambda macros may appear in functions where lambda would have previously appeared. When the compiler or interpreter detects a function whose car is a lambda macro, they expand the macro in much the same way that ordinary Lisp macros are expanded—the lambda macro is called with the function as its argument and is expected to return another function as its value. The definition of a lambda macro (that is, the function which expands it) may be accessed with the (:lambda-macro *name*) function spec.

The value returned by the lambda macro expander function may be any valid function. Usually it is a list starting with lambda, subst, named-lambda or named-subst, but it could also be another use of a lambda macro, or even a compiled function.

lambda-macro *name lambda-list* &body *body* *Macro*

By analogy with macro, defines a lambda macro to be called *name*. *lambda-list* should consist of one variable, which is bound to the function that caused the lambda macro to be called. The lambda macro must return a function. For example:

```
(lambda-macro ilisp (x)
  '(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

defines a lambda macro called ilisp which can be used to define functions that accept arguments like a standard Interlisp function: all arguments are optional and extra arguments are ignored. A typical use would be:

```
(fun-with-functional-arg #'(ilisp (x y z) (list x y z)))
```

This passes to fun-with-functional-arg a function which will ignore extra arguments beyond the third, and will default x, y and z to nil.

deflambda-macro *Macro*

deflambda-macro is like defmacro, but defines a lambda macro instead of a normal macro. Here is how ilisp could be defined using deflambda-macro:

```
(deflambda-macro ilisp (argument-list &body body)
  '(lambda (&optional ,@argument-list &rest ignore) . ,body))
```

deffunction *function-spec lambda-macro-name lambda-list* &body *body* *Macro*

Defines a function with a definition that uses an arbitrary lambda macro instead of lambda. It takes arguments like defun, expect that the argument immediatly following the function specifier is the name of the lambda macro to be used. deffunction expands the lambda macro immediatly, so the lambda macro must have been previously defined.

Example:

```
(deffunction some-interlisp-like-function ilisp (x y z)
  (list x y z))
```

would define a function called some-interlisp-like-function with the definition (ilisp (x y z) (list x y z)).

(defun foo ...) could be considered an abbreviation for (deffunction foo lambda ...)

11.5.3 Compiled Functions

There are two kinds of compiled functions: *macrocoded* functions and *microcoded* functions. The Lisp compiler converts lambda and named-lambda functions into macrocoded functions. A macrocoded function's printed representation looks like:

```
#<dtp-fef-pointer append 1424771>
```

This type of Lisp object is also called a 'Function Entry Frame', or 'FEF' for short. Like 'car' and 'cdr', the name is historical in origin and doesn't really mean anything. The object contains Lisp Machine machine code that does the computation expressed by the function; it also contains a description of the arguments accepted, any constants required, the name, documentation, and other things. Unlike Maclisp "subr-objects", macrocoded functions are full-fledged objects and can be passed as arguments, stored in data structure, and applied to arguments.

The printed representation of a microcoded function looks like:

```
#<dtp-u-entry last 270>
```

Most microcompiled functions are basic Lisp primitives or subprimitives written in Lisp Machine microcode. You can also convert your own macrocode functions into microcode functions in some circumstances, using the micro-compiler.

11.5.4 Other Kinds of Functions

A closure is a kind of function that contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed. Closures are made with the function closure. See chapter 12, page 250 for more information. Entities are slightly different from closures; see section 12.4, page 255.

A select-method (internal type code dtp-select-method) contains an alist of symbols and functions. When one is called, the first argument is looked up in the alist to find the particular function to be called. This function is applied to the rest of the arguments. The alist may have a list of symbols in place of a symbol, in which case the associated function is called if the first argument is any of the symbols on the list. If cdr of last of the alist is non-nil, it is a *default handler* function, which gets called if the message key is not found in the alist. Select-methods can be created with the defselect special form (see page 236). If the select-method is the definition of a function-spec, the individual functions in the alist can be referred to or defined using :select-method function specs (see page 225).

An instance is a message-receiving object that has both a state and a table of message-handling functions (called *methods*). Refer to the chapter on flavors (chapter 21, page 401) for further information.

An array can be used as a function. The arguments to the array are the indices and the value is the contents of the element of the array. This is for Maclisp compatibility and is not recommended usage. Use aref (page 170) instead.

A stack group can be called as a function. This is one way to pass control to another stack group. See chapter 13, page 256.

11.5.5 Special Forms and Functions

The special forms of Zetalisp, such as **quote**, **let** and **cond**, are actually implemented with an unusual sort of function.

First, let's restate the outline of how the evaluator works. When the evaluator is given a list whose first element is a symbol, the form may be a function form, a special form, or a macro form (see page 24). If the definition of the symbol is a function, then the function is just applied to the result of evaluating the rest of the subforms. If the definition is a cons whose car is **macro**, then it is a macro form; these are explained in chapter 18, page 320. What about special forms?

A special form is implemented by a function that is flagged to tell the evaluator to refrain from evaluating some or all of the arguments to the function. Such functions make use of the lambda-list keyword **&quote**.

The evaluator, on seeing the **&quote** in the lambda list of an interpreted function (or something equivalent in a compiled function) skips the evaluation of the arguments to which the **&quote** applies. Aside from that, it calls the function normally.

For example, **quote** could be defined as

```
(defun quote (&quote arg) arg)
```

Evaluation of **(quote x)** would see the **&quote** in the definition, implying that the argument *arg* should not be evaluated. Therefore, the argument passed to the definition of **quote** would be the symbol **x** rather than the value of **x**. From then on, the definition of **quote** would execute in the normal fashion, so **x** would be the value of **arg** and **x** would be returned.

&quote applies to all the following arguments, but it can be cancelled with **&eval**. A simple **setq** that accepted only one variable and one value could be defined as follows:

```
(defun setq (&quote variable &eval value)
  (set variable value))
```

The actual definition of **setq** is more complicated and uses a lambda list **(&quote &rest variables-and-values)**. Then it must go through the rest-argument, evaluating every other element.

The definitions of special forms are designed with the assumption that they will be called by **eval**. It does not usually make much sense to call one with **funcall** or **apply**. **funcall** and **apply** do not evaluate any arguments; they receive *values* of arguments, rather than expressions for them, and pass these values directly to the function to be called. There is no evaluation for **funcall** or **apply** to refrain from performing. Most special forms explicitly call **eval** on some of their arguments, or parts of them, and if called with **apply** or **funcall** they will *still* do so. This behavior is rarely useful, so calling special forms with **apply** or **funcall** should be avoided. Encapsulations can do this successfully, because they can arrange that quoted arguments are quoted also on entry to the encapsulation.

It is possible to define your own special form using **&quote**. Macros can also be used to accomplish the same thing. It is preferable to implement language extensions as macros rather than special forms, because macros directly define a Lisp-to-Lisp translation and therefore can be understood by both the interpreter and the compiler. Special forms, on the other hand, only

extend the interpreter. The compiler has to be modified in an *ad hoc* way to understand each new special form so that code using it can be compiled. For example, it would not work for a compiled function to call the interpreted definition of setq; the set in that definition would not be able to act on local variables of the compiled function.

Since all real programs are eventually compiled, writing your own special functions is strongly discouraged. The purpose of &quote is to be used in the system's own standard special forms.

New Lisp constructs in the system are also implemented as macros most of the time; macros are less work for us, too.

11.6 Function-Defining Special Forms

defun is a special form that is put in a program to define a function; defsubst and macro are others. This section explains how these special forms work, how they relate to the different kinds of functions, and how they interface to the rest of the function-manipulation system.

Function-defining special forms typically take as arguments a function spec and a description of the function to be made, usually in the form of a list of argument names and some forms that constitute the body of the function. They construct a function, give it the function spec as its name, and define the function spec to be the new function. Different special forms make different kinds of functions. defun makes a named-lambda function, and defsubst makes a named-subst function. macro makes a macro; though the macro definition is not really a function, it is like a function as far as definition handling is concerned.

These special forms are used in writing programs because the function names and bodies are constants. Programs that define functions usually want to compute the functions and their names, so they use fdefine. See page 239.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See section 11.9, page 244.

The special forms only create interpreted functions. There is no special way of defining a compiled function. Compiled functions are made by compiling interpreted ones. The same special form that defines the interpreted function, when processed by the compiler, yields the compiled function. See chapter 17, page 301 for details.

Note that the editor understands these and other "defining" special forms (e.g. defmethod, defvar, defmacro, defstruct, etc.) to some extent, so that when you ask for the definition of something, the editor can find it in its source file and show it to you. The general convention is that anything that is used at top level (not inside a function) and starts with def should be a special form for defining things and should be understood by the editor. defprop is an exception.

The defun special form (and the defunp macro which expands into a defun) are used for creating ordinary interpreted functions (see page 227).

For Maclisp compatibility, a *type* symbol may be inserted between *name* and *lambda-list* in the defun form. The following types are understood:

expr The same as no type.

fexpr **&quote** and **&rest** are prefixed to the lambda list.

macro A macro is defined instead of a normal function.

If *lambda-list* is a non-nil symbol instead of a list, the function is recognized as a Maclisp *lexpr* and it is converted in such a way that the **arg**, **setarg**, and **listify** functions can be used to access its arguments (see page 238).

The **defsubst** special form is used to create substitutible functions. It is used just like **defun** but produces a list starting with **named-subst** instead of one starting with **named-lambda**. The **named-subst** function acts just like the corresponding **named-lambda** function when applied, but it can also be open-coded (incorporated into its callers) by the compiler. See page 329 for full information.

The **macro** special form is the primitive means of creating a macro. It gives a function spec a definition that is a macro definition rather than a actual function. A macro is not a function because it cannot be applied, but it *can* appear as the car of a form to be evaluated. Most macros are created with the more powerful **defmacro** special form. See chapter 18, page 320.

The **defselect** special form defines a select-method function. See page 236.

Unlike the above special forms, the next two (**deff** and **def**) do not create new functions. They simply serve as hints to the editor that a function is being stored into a function spec here, and therefore if someone asks for the source code of the definition of that function spec, this is the place to look for it.

def *Special form*

If a function is created in some strange way, wrapping a **def** special form around the code that creates it informs the editor of the connection. The form

```
(def function-spec
  form1 form2...)
```

simply evaluates the forms *form1*, *form2*, etc. It is assumed that these forms will create or obtain a function somehow, and make it the definition of *function-spec*.

Alternatively, you could put (**def** *function-spec*) in front of or anywhere near the forms which define the function. The editor only uses it to tell which line to put the cursor on.

deff *function-spec definition-creator* *Special form*

deff is a simplified version of **def**. It evaluates the form *definition-creator*, which should produce a function, and makes that function the definition of *function-spec*, which is not evaluated. **deff** is used for giving a function spec a definition which is not obtainable with the specific defining forms such as **defun**. For example,

```
(deff foo 'bar)
```

makes **foo** equivalent to **bar**, with an indirection so that if **bar** changes **foo** will likewise change; conversely,

```
(deff foo (function bar))
```

copies the definition of **bar** into **foo** with no indirection, so that further changes to **bar** will have no effect on **foo**.

deff-macro *function-spec definition-creator* *Special form*

Is like deff (see page 235) but for defining macros. *definition-creator* is evaluated to produce a suitable definition-as-a-macro and then *function-spec* is defined that way. The definition-as-a-macro should be a cons whose car is macro and whose cdr is an expander function. Alternatively, a definition as a subst function can be used; either a list starting with subst or named-subst or a FEF which records it was compiled from such a list.

The difference between deff and deff-macro is that compile-file assumes that deff-macro is defining something which should be expanded during compilation. For the rest of the file, the macro defined here is available for expansion. When the file is ultimately loaded, or if compilation is done in-core, deff and deff-macro are equivalent.

@define *Macro*

This macro turns into nil, doing nothing. It exists for the sake of the @ listing generation program, which uses it to declare names of special forms that define objects (such as functions) that @ should cross-reference.

si:defun-compatibility *x*

This function is used by defun and the compiler to convert Maclisp-style lexpr, fexpr, and macro defuns to Zetalisp definitions. *x* should be the cdr of a (defun ...) form. defun-compatibility returns a corresponding (defun ...) or (macro ...) form, in the usual Zetalisp format. You shouldn't ever need to call this yourself.

defselect *Special form*

defselect defines a function which is a select-method. This function contains a table of subfunctions; when it is called, the first argument, a symbol on the keyword package called the *operation*, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments and have a different pattern of arguments. defselect is useful for a variety of "dispatching" jobs. By analogy with the more general message-passing facilities described in chapter 21, page 401, the subfunctions are called *methods* and the list of arguments is sometimes called a *message*.

The special form looks like

```
(defselect (function-spec default-handler no-which-operations)
  (operation (args...)
        body...)
  (operation (args...)
        body...)
  ...)
```

function-spec is the name of the function to be defined. *default-handler* is optional; it must be a symbol and is a function which gets called if the select-method is called with an unknown operation. If *default-handler* is unsupplied or nil, then an unknown operation causes an error with condition name sys:unclaimed-message (see page 423).

Normally, methods for the operations :which-operations, :operation-handled-p and :send-if-handles are generated automatically based on the set of existing methods. These operations have the same meaning as they do on flavor instances; see section 21.10, page 432 for their definitions. If *no-which-operations* is non-nil, these methods are not created automatically; however, you can supply them yourself.

If *function-spec* is a symbol, and *default-handler* and *no-which-operations* are not supplied, then the first subform of the defselect may be just *function-spec* by itself, not enclosed in a list.

The remaining subforms in a defselect are the clauses, each defining one method. *operation* is the operation to be handled by this clause or a list of several operations to be handled by the same clause. *args* is a lambda-list; it should not include the first argument, which is the operation. *body* is the body of the function.

A clause can instead look like:

(*operation* . *symbol*)

In this case, *symbol* is the name of a function that is to be called when the *operation* operation is performed. It will be called with the same arguments as the select-method, including the operation symbol itself.

The individual methods of the defselect can be examined, redefined, traced, etc. using :select-method function specs (see page 225).

defselect-incremental *function-spec default-handler* *Special form*

defselect defines a select-method function all at once. By contrast, defselect-incremental defines an empty select-method to which methods can be added with defun.

Specifically, defselect-incremental *function-spec*, with just a default handler and the standard methods :which-operations, :operation-handled-p and :send-if-handles.

Individual methods are defined by using defun on a function spec of the form (:select-method *function-spec operation*). *function-spec* specifies where to find the select-method, and *operation* is the operation for which a method should be defined. The argument list of the defun must include a first argument which receives the operation name.

Example:

```
(defselect-incremental foo ignore)
    ;The function ignore is the default handler
(defun (:select-method foo :lose) (ignore a)
  (1+ a))
```

defines the same function foo as

```
(defselect (foo ignore)
  (:lose (a) (1+ a)))
```

These two examples are not completely equivalent, however. Reevaluating the defselect gets rid of any methods that used to exist but have been deleted from the defselect itself. Reevaluating the defselect-incremental has no such effect, and reevaluating an individual defun redefines only that method. Methods can be removed only with fundefine.

11.6.1 Maclisp Lexprs

Lexprs are the way Maclisp functions can accept variable numbers of arguments. They are supported for compatibility only; &optional and &rest are much preferable. A lexpr definition looks like

```
(defun foo nargs body...)
```

where a symbol (nargs, here) appears in place of a lambda-list. When the function is called, nargs is bound to the number of arguments it was given. The arguments themselves are accessed using the functions arg, setarg, and listify.

arg *x*

(arg nil), when evaluated during the application of a lexpr, gives the number of arguments supplied to that lexpr. This is primarily a debugging aid, since lexprs also receive their number of arguments as the value of their lambda-variable.

(arg *i*), when evaluated during the application of a lexpr, gives the value of the *i*'th argument to the lexpr. *i* must be a fixnum in this case. It is an error if *i* is less than 1 or greater than the number of arguments supplied to the lexpr. Example:

```
(defun foo nargs                ;define a lexpr foo.
   (print (arg 2))              ;print the second argument.
   (+ (arg 1)                   ;return the sum of the first
      (arg (- nargs 1))))       ;and next to last arguments.
```

setarg *i x*

setarg is used only during the application of a lexpr. (setarg *i x*) sets the lexpr's *i*'th argument to *x*. *i* must be greater than zero and not greater than the number of arguments passed to the lexpr. After (setarg *i x*) has been done, (arg *i*) returns *x*.

listify *n*

(listify *n*) manufactures a list of *n* of the arguments of a lexpr. With a positive argument *n*, it returns a list of the first *n* arguments of the lexpr. With a negative argument *n*, it returns a list of the last (abs *n*) arguments of the lexpr. Basically, it works as if defined as follows:

```
(defun listify (n)
     (cond ((minusp n)
            (listify1 (arg nil) (+ (arg nil) n 1)))
           (t
            (listify1 n 1))))

(defun listify1 (n m)          ; auxiliary function.
     (do ((i n (1- i))
          (result nil (cons (arg i) result)))
         ((< i m) result)))
```

11.7 How Programs Manipulate Function Specs

fdefine *function-spec definition* &optional (*carefully* nil) (*no-query* nil)

This is the primitive used by defun and everything else in the system to change the definition of a function spec. If *carefully* is non-nil, which it usually should be, then only the basic definition is changed; the previous basic definition is saved if possible (see undefun, page 241), and any encapsulations of the function such as tracing and advice are carried over from the old definition to the new definition. *carefully* also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally. However, this warning is suppressed if either the argument *no-query* is non-nil, or if the global variable inhibit-fdefine-warnings is t.

If fdefine is called while a file is being loaded, it records what file the function definition came from so that the editor can find the source code.

If *function-spec* was already defined as a function, and *carefully* is non-nil, the function-spec's :previous-definition property is used to save the previous definition. This property is used by the undefun function (page 241), which restores the previous definition. The properties for different kinds of function specs are stored in different places; when a function spec is a symbol its properties are stored on the symbol's property list.

defun and the other function-defining special forms all supply t for *carefully* and nil or nothing for *no-query*. Operations that construct encapsulations, such as trace, are the only ones which use nil for *carefully*.

si:record-source-file-name *name* &optional (*type* defun) *no-query*

Records a definition of *name*, of type *type*. *type* should be defun to record a function definition; then *name* is a function spec. *type* can also be defvar, defflavor, defresource, defsignal or anything else you want to use.

The value of sys:fdefine-file-pathname is assumed to be the generic pathname of the file the definition is coming from, or nil if the definition is not from a file. If a definition of the same *name* and *type* has already been seen but not in the same file, and *no-query* is nil, a condition is signaled and then the user is queried.

If si:record-source-file-name returns nil, it means that the user or a condition handler said the redefinition should not be performed.

sys:fdefine-file-pathname *Variable*

While the system is loading a file, this is the generic pathname for the file. The rest of the time it is nil. fdefine uses this to remember what file defines each function.

si:get-source-file-name *function-spec* &optional *type*

Returns the generic pathname for the file in which *function-spec* received a definition of type *type*. If *type* is nil, the most recent definition is used, regardless of its type.

function-spec really is a function spec only if *type* is defun; for example, if *type* is defvar, *function-spec* is a variable name. Other types that are used by the system are defflavor and defstruct.

This function returns the generic pathname of the source file. To obtain the actual source file pathname, use the :source-pathname operation (see page 563).

A second value is returned, which is the type of the definition that was reported.

si:get-all-source-file-names *function-spec*

Returns a list describing the generic pathnames of all the definitions this function-spec has received, of all types. The list is an alist whose elements look like

(*type pathname...*)

sys:redefinition (sys:warning) *Condition Flavor*

This condition, which is not an error, is signaled by si:record-source-file-name when something is redefined by a different file. The handler for this condition can control what is done about the redefinition.

The condition instance provides the operations :name, :definition-type, :old-pathname and :new-pathname. :name and :definition-type return the *name* and *type* arguments to si:record-source-file-name. :old-pathname and :new-pathname return two generic pathnames saying where the old definition was and where this one is. The new pathname may be nil, meaning that the redefinition is being done by the user, not in any file.

Two proceed types are available, :proceed and :inhibit-definition. The first tells si:record-source-file-name to return t, the second tells it to return nil. If the condition is not handled at all, the user is queried or warned according to the value of inhibit-fdefine-warnings.

inhibit-fdefine-warnings *Variable*

This variable is normally nil. Setting it to t prevents si:record-source-file-name from warning you and asking about questionable redefinitions such as a function being redefined by a different file than defined it originally, or a symbol that belongs to one package being defined by a file that belongs to a different package. Setting it to :just-warn allows the warnings to be printed out, but prevents the queries from happening; it assumes that your answer is 'yes', i.e. that it is all right to redefine the function.

fset-carefully *symbol definition* &optional *force-flag*

This function is obsolete. It is equivalent to

(**fdefine** *symbol definition* **t** *force-flag*)

fdefinedp *function-spec*

This returns t if *function-spec* has a definition, nil if it does not.

fdefinition *function-spec*

This returns *function-spec*'s definition. If it has none, an error occurs.

fdefinition-location *function-spec*

Equivalent to (locf (fdefinition *function-spec*)). For some kinds of function specs, though not for symbols, this (whichever way you write it) can cause data structure to be created to hold a definition. For example, if *function-spec* is of the :property kind, then an entry may have to be added to the property list if it isn't already there.

fundefine *function-spec*

Makes *function-spec* undefined; the cell where its definition is stored becomes void. For symbols this is equivalent to **fmakunbound**. If the function is encapsulated, **fundefine** removes both the basic definition and the encapsulations. Some types of function specs (**:location** for example) do not implement **fundefine**. **fundefine** on a **:within** function spec removes the replacement of *function-to-affect*, putting the definition of *within-function* back to its normal state. **fundefine** on a **:method** function spec removes the method completely, so that future messages will be handled by some other method (see the flavor chapter).

undefun *function-spec*

If *function-spec* has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. If *function-spec* has no saved previous definition, **undefun** asks the user whether to make it undefined.

This undoes the effect of redefining a function. See also **uncompile** (page 301).

si:function-spec-get *function-spec indicator*

Returns the value of the *indicator* property of *function-spec*, or **nil** if it doesn't have such a property.

si:function-spec-putprop *function-spec value indicator*

Gives *function-spec* an *indicator*-property whose value is *value*.

si:function-spec-lessp *function-spec1 function-spec2*

Compares the two function specs with an ordering that is useful in sorting lists of function specs for presentation to the user.

si:function-parent *function-spec*

If *function-spec* does not have its own definition, textually speaking, but is defined as part of the definition of something else, this function returns the function spec for that something else. For example, if *function-spec* is an accessor function for a **defstruct**, the value returned is the name of the **defstruct**.

The intent is that if the caller has not been able to find the definition of *function-spec* in a more direct fashion, it can try looking for the definition of the *function-parent* of *function-spec*. This is used by the editor's **Meta-.** command.

sys:invalid-function-spec (error) *Condition*

This condition name belongs to the error signaled when you refer to a function spec that is syntactically invalid; such as, if it is a list whose car is not a recognized type of function spec.

The condition object supports the operation **:function-spec**, which returns the function spec which was invalid.

Note that in a few cases the condition **:wrong-type-argument** is signaled instead. These are the cases in which the error is correctable.

11.8 How Programs Examine Functions

These functions take a function as argument and return information about that function. Some also accept a function spec and operate on its definition. The others do not accept function specs in general but do accept a symbol as standing for its definition. (Note that a symbol is a function as well as a function spec).

The function documentation can be used to examine a function's documentation string. See page 784.

debugging-info *function*

This returns the debugging info alist of *function*, or nil if it has none.

arglist *function* &optional *real-flag*

arglist is given a function or a function spec, and returns its best guess at the nature of the function's lambda-list. It can also return a second value which is a list of descriptive names for the values returned by the function.

If *function* is a symbol, arglist of its function definition is used.

If the *function* is an actual lambda-expression, its cadr, the lambda-list, is returned. But if *function* is compiled, arglist attempts to reconstruct the lambda-list of the original definition, using whatever debugging information was saved by the compiler.

Some functions' real argument lists are not what would be most descriptive to a user. A function may take a rest argument for technical reasons even though there are standard meanings for the first elements of that argument. For such cases, the definition of the function can specify, with a local declaration, a value to be returned when the user asks about the argument list. Example:

```
(defun foo (&rest rest-arg)
  (declare (arglist x y &rest z))
  .....)
```

real-flag has one of three values:

nil	Return the arglist declared by the user in preference to the actual one.
t	Return the actual arglist as computed from the function definition's handling of arguments, ignoring any arglist declaration. For a compiled function, this omits all keyword arguments (replacing them with a rest argument) and may replace initial values of optional arguments with si:*hairy* if the actual expressions are too complicated.
compile	Like nil, but in the case of a compiled function it returns the actual arglist of the lambda-expression that was originally compiled. The compiler uses this as a basis for checking for incorrect calls to the function.

Programs interested in how many and what kind (evaluated or quoted) of arguments to pass should use args-info instead.

When a function returns multiple values, it is useful to give the values names so that the caller can be reminded which value is which. By means of a return-list declaration in the function's definition, entirely analogous to the arglist declaration above, you can specify a list of mnemonic names for the returned values. This list is then returned by arglist as the second value.

```
(arglist 'arglist)
  => (function &optional real-flag) and (arglist return-list)
```

function-name *function* &optional *try-flavor-name*

Returns the name of the function *function*, if that can be determined. If *function* does not describe what its name is, *function* itself is returned.

If *try-flavor-name* is non-nil, then if *function* is a flavor instance (which can, after all, be used as a function), then the flavor name is returned. If the optional argument is nil, flavor instances are treated as anonymous.

eh:arg-name *function arg-number*

Returns the name of argument number *arg-number* in function *function*. Returns nil if the function doesn't have such an argument, or if the name is not recorded. &rest arguments are not obtained with arg-number; use rest-arg-name to obtain the name of *function*'s &rest argument, if any.

eh:rest-arg-name *function*

Returns the name of the rest argument of function *function*, or nil if *function* does not have one.

eh:local-name *function local-number*

Returns the name of local variable number *local-number* in function *function*. If *local-number* is zero, this gets the name of the rest arg in any function that accepts a rest arg. Returns nil if the function doesn't have such a local.

args-info *function*

Returns a fixnum called the "numeric argument descriptor" of the *function*, which describes the way the function takes arguments. This descriptor is used internally by the microcode, the evaluator, and the compiler. *function* can be a function or a function spec.

The information is stored in various bits and byte fields in the fixnum, which are referenced by the symbolic names shown below. By the usual Lisp Machine convention, those starting with a single '%' are bit-masks (meant to be logand'ed or bit-test'ed with the number), and those starting with '%%' are byte specifiers, meant to be used with ldb or ldb-test.

Here are the fields:

%%arg-desc-min-args

This is the minimum number of arguments that may be passed to this function, i.e. the number of required parameters.

%%arg-desc-max-args
This is the maximum number of arguments that may be passed to this function, i.e. the sum of the number of required parameters and the number of optional parameters. If there is a rest argument, this is not really the maximum number of arguments that may be passed; an arbitrarily-large number of arguments is permitted, subject to limitations on the maximum size of a stack frame (about 200 words).

%arg-desc-evaled-rest
If this bit is set, the function has a rest argument, and it is not quoted.

%arg-desc-quoted-rest
If this bit is set, the function has a rest argument, and it is quoted. Most special forms have this bit.

%arg-desc-fef-quote-hair
If this bit is set, there are some quoted arguments other than the rest argument (if any), and the pattern of quoting is too complicated to describe here. The ADL (Argument Description List) in the FEF should be consulted. This is only for special forms.

%arg-desc-interpreted
This function is not a compiled-code object, and a numeric argument descriptor cannot be computed. Usually **args-info** does not return this bit, although **%args-info** does.

%arg-desc-fef-bind-hair
There is argument initialization, or something else too complicated to describe here. The ADL (Argument Description List) in the FEF should be consulted.

Note that **%arg-desc-quoted-rest** and **%arg-desc-evaled-rest** cannot both be set.

%args-info *function*
This is an internal function; it is like **args-info** but does not work for interpreted functions. Also, *function* must be a function, not a function spec. It exists because it has to be in the microcode anyway, for **apply** and the basic function-calling mechanism.

11.9 Encapsulations

The definition of a function spec actually has two parts: the *basic definition*, and *encapsulations*. The basic definition is what is created by functions like **defun**, and encapsulations are additions made by **trace** or **advise** to the basic definition. The purpose of making the encapsulation a separate object is to keep track of what was made by **defun** and what was made by **trace**. If **defun** is done a second time, it replaces the old basic definition with a new one while leaving the encapsulations alone.

Only advanced users should ever need to use encapsulations directly via the primitives explained in this section. The most common things to do with encapsulations are provided as higher-level, easier-to-use features: **trace** (see page 738), **breakon** (see page 741) and **advise** (see page 742).

The actual definition of the function spec is the outermost encapsulation; this contains the next encapsulation, and so on. The innermost encapsulation contains the basic definition. The way this containing is done is as follows. An encapsulation is actually a function whose debugging info alist contains an element of the form

(si:encapsulated-definition *uninterned-symbol encapsulation-type*)

The presence of such an element in the debugging info alist is how you recognize a function to be an encapsulation. An encapsulation is usually an interpreted function (a list starting with named-lambda) but it can be a compiled function also, if the application which created it wants to compile it.

uninterned-symbol's function definition is the thing that the encapsulation contains, usually the basic definition of the function spec. Or it can be another encapsulation, which has in it another debugging info item containing another uninterned symbol. Eventually you get to a function which is not an encapsulation; it does not have the sort of debugging info item which encapsulations all have. That function is the basic definition of the function spec.

Literally speaking, the definition of the function spec is the outermost encapsulation, period. The basic definition is not the definition. If you are asking for the definition of the function spec because you want to apply it, the outermost encapsulation is exactly what you want. But the basic definition can be found mechanically from the definition, by following the debugging info alists. So it makes sense to think of it as a part of the definition. In regard to the function-defining special forms such as defun, it is convenient to think of the encapsulations as connecting between the function spec and its basic definition.

An encapsulation is created with the macro si:encapsulate.

si:encapsulate *Macro*

A call to si:encapsulate looks like

(si:encapsulate *function-spec outer-function type*
body-form
extra-debugging-info)

All the subforms of this macro are evaluated. In fact, the macro could almost be replaced with an ordinary function, except for the way *body-form* is handled.

function-spec evaluates to the function spec whose definition the new encapsulation should become. *outer-function* is another function spec, which should often be the same one. Its only purpose is to be used in any error messages from si:encapsulate.

type evaluates to a symbol which identifies the purpose of the encapsulation and says what the application is. For example, that could be advise or trace. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type (see si:encapsulation-standard-order, page 247). *type* should have an si:encapsulation-grind-function property which tells grindef what to do with an encapsulation of this type.

body-form evaluates to the body of the encapsulation-definition, the code to be executed when it is called. Backquote is typically used for this expression; see section 18.2.2, page 325. si:encapsulate is a macro because, while *body* is being evaluated, the variable si:encapsulated-function is bound to a list of the form (function *uninterned-symbol*),

referring to the uninterned symbol used to hold the prior definition of *function-spec*. If si:encapsulate were a function, *body-form* would just get evaluated normally by the evaluator before si:encapsulate ever got invoked, and so there would be no opportunity to bind si:encapsulated-function. The form *body-form* should contain '(apply ,si:encapsulated-function arglist) somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable arglist is bound by some of the code which the si:encapsulate macro produces automatically. When the body of the encapsulation is run arglist's value will be the list of the arguments which the encapsulation received.)

extra-debugging-info evaluates to a list of extra items to put into the debugging info alist of the encapsulation function (besides the one starting with si:encapsulated-definition, which every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

If compile-encapsulations-flag is non-nil, the encapsulation is compiled before it is installed. The encapsulations on a particular function spec can be compiled by calling compile-encapsulations. See page 302. Compiled encapsulations can still be unencapsulated since the information needed to do so is stored in the debugging info alist, which is preserved by compilation. However, applications which wish to modify the code of the encapsulations they previously created must check for encapsulations that have been compiled and uncompile them. This can be done by finding the sys:interpreted-definition entry in the debugging info alist, which is present in all compiled functions except those made by file-to-file compilation.

When a special function is encapsulated, the encapsulation is itself a special function with the same argument quoting pattern. Therefore, when the outermost encapsulation is started, each argument has been evaluated or not as appropriate. Because each encapsulation calls the prior definition with apply, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not as appropriate. The basic definition may call eval on some of these arguments or parts of them; the encapsulations should not.

Macros cannot be encapsulated, but their expander functions can be; if the definition of *function-spec* is a macro, then si:encapsulate automatically encapsulates the expander function instead. In this case, the definition of the uninterned symbol is the original macro definition, not just the original expander function. It would not work for the encapsulation to apply the macro definition. So during the evaluation of *body-form*, si:encapsulated-function is bound to the form (cdr (function *uninterned-symbol*)), which extracts the expander function from the prior definition of the macro.

Because only the expander function is actually encapsulated, the encapsulation does not see the evaluation or execution of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

A program which creates encapsulations often needs to examine an encapsulation it created and find the body. For example, adding a second piece of advice to one function requires doing this. The proper way to do it is to use si:encapsulation-body.

si:encapsulation-body *encapsulation*

Returns a list whose car is the body-form of *encapsulation*. It is the form that was the fourth argument of si:encapsulate when *encapsulation* was created. To illustrate this relationship,

```
(si:encapsulate 'foo 'foo 'trace 'body)

(si:encapsulation-body (fdefinition 'foo))
  => (body)
```

It is possible for one function to have multiple encapsulations, created by different subsystems. In this case, the order of encapsulations is independent of the order in which they were made. It depends instead on their types. All possible encapsulation types have a total order and a new encapsulation is put in the right place among the existing encapsulations according to its type and their types.

si:encapsulation-standard-order *Variable*

The value of this variable is a list of the allowed encapsulation types, in the order in which the encapsulations are supposed to be kept (innermost encapsulations first). If you want to add new kinds of encapsulations, you should add another symbol to this list. Initially its value is

```
(advise breakon trace si:rename-within)
```

advise encapsulations are used to hold advice (see page 742). breakon encapsulations are used for implementing breakon (see page 741). trace encapsulations are used for implementing tracing (see page 738). si:rename-within encapsulations are used to record the fact that function specs of the form (:within *within-function altered-function*) have been defined. The encapsulation goes on *within-function* (see section 11.9.1, page 249 for more information).

Every symbol used as an encapsulation type must be on the list si:encapsulation-standard-order. In addition, it should have an si:encapsulation-grind-function property whose value is a function that grindef will call to process encapsulations of that type. This function need not take care of printing the encapsulated function because grindef will do that itself. But it should print any information about the encapsulation itself which the user ought to see. Refer to the code for the grind function for advise to see how to write one.

To find the right place in the ordering to insert a new encapsulation, it is necessary to parse existing ones. This is done with the function si:unencapsulate-function-spec.

si:unencapsulate-function-spec *function-spec* &optional *encapsulation-types*

This takes one function spec and returns another. If the original function spec is undefined, or has only a basic definition (that is, its definition is not an encapsulation), then the original function spec is returned unchanged.

If the definition of *function-spec* is an encapsulation, then its debugging info is examined to find the uninterned symbol that holds the encapsulated definition and the encapsulation type. If the encapsulation is of a type that is to be skipped over, the uninterned symbol replaces the original function spec and the process repeats.

The value returned is the uninterned symbol from inside the last encapsulation skipped. This uninterned symbol is the first one that does not have a definition that is an encapsulation that should be skipped. Or the value can be *function-spec* if *function-spec*'s definition is not an encapsulation that should be skipped.

The types of encapsulations to be skipped over are specified by *encapsulation-types*. This can be a list of the types to be skipped, or nil meaning skip all encapsulations (this is the default). Skipping all encapsulations means returning the uninterned symbol that holds the basic definition of *function-spec*. That is, the *definition* of the function spec returned is the *basic definition* of the function spec supplied. Thus,

```
(fdefinition (si:unencapsulate-function-spec 'foo))
```

returns the basic definition of foo, and

```
(fdefine (si:unencapsulate-function-spec 'foo) 'bar)
```

sets the basic definition (just like using fdefine with *carefully* supplied as t).

encapsulation-types can also be a symbol, which should be an encapsulation type; then we skip all types that are supposed to come outside of the specified type. For example, if *encapsulation-types* is trace, then we skip all types of encapsulations that come outside of trace encapsulations, but we do not skip trace encapsulations themselves. The result is a function spec that is where the trace encapsulation ought to be, if there is one. Either the definition of this function spec is a trace encapsulation, or there is no trace encapsulation anywhere in the definition of *function-spec*, and this function spec is where it would belong if there were one. For example,

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (and (eq tem (si:unencapsulate-function-spec tem '(trace)))
       (si:encapsulate tem spec 'trace '(...body...))))
```

finds the place where a trace encapsulation ought to go and makes one unless there is already one there.

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (fdefine tem (fdefinition (si:unencapsulate-function-spec
                              tem '(trace)))))
```

eliminates any trace encapsulation by replacing it by whatever it encapsulates. (If there is no trace encapsulation, this code changes nothing.)

These examples show how a subsystem can insert its own type of encapsulation in the proper sequence without knowing the names of any other types of encapsulations. Only the variable si:encapsulation-standard-order, which is used by si:unencapsulate-function-spec, knows the order.

11.9.1 Rename-Within Encapsulations

One special kind of encapsulation is the type si:rename-within. This encapsulation goes around a definition in which renamings of functions have been done.

How is this used?

If you define, advise, or trace (:within foo bar), then bar gets renamed to #:altered-bar-within-foo wherever it is called from foo, and foo gets a si:rename-within encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, grindef (see page 528) notices the encapsulation, and so knows to print bar instead of #:altered-bar-within-foo when grinding the definition of foo.

Also, if you redefine foo, or trace or advise it, the new definition gets the same renaming done (bar replaced by #:altered-bar-within-foo). To make this work, everyone who alters part of a function definition should pass the new part of the definition through the function si:rename-within-new-definition-maybe.

si:rename-within-new-definition-maybe *function-spec new-structure*

Given *new-structure*, which is going to become a part of the definition of *function-spec*, perform on it the replacements described by the si:rename-within encapsulation in the definition of *function-spec*, if there is one. The altered (copied) list structure is returned.

It is not necessary to call this function yourself when you replace the basic definition because fdefine with *carefully* supplied as t does it for you. si:encapsulate does this to the body of the new encapsulation. So you only need to call si:rename-within-new-definition-maybe yourself if you are rplac'ing part of the definition.

For proper results, *function-spec* must be the outer-level function spec. That is, the value returned by si:unencapsulate-function-spec is *not* the right thing to use. It will have had one or more encapsulations stripped off, including the si:rename-within encapsulation if any, and so no renamings will be done.