CHAPTER 4 TYPE SPECIFIERS

4.2. Type Specifier Lists

The special form "the" operates as an assertion in interpreted code, but has no effect on compiled code.

Although it is not an error to ask `(typep x 'foo)` when `foo` is not yet a defined type at compile time, the compiler will produce much more efficient code if type `foo` is known at compile time.

4.8. Type Conversion Function

The function `coerce` operates only on the types explicitly listed in the book.

[This page intentionally left blank]

CHAPTER 5 PROGRAM STRUCTURE

5.1.2. Variables

Unbound special variables have `il:nobind` in their value cell. If you try to access an unbound variable in interpreted code, an error is signaled. If you try to use one in compiled code, `il:nobind` is returned as its value.

5.2. Functions

5.2.2. Lambda-Expressions

In this release, argument number checking is performed in the interpreter, but not in all compiled code. For those compiled functions that do not check, if the function is called with fewer arguments than the function requires, the remaining required arguments will have value `nil`; if called with more arguments than the function permits (required plus optionals), the extra arguments are ignored.

Compiled functions also do not check for unexpected keywords, or malformed keyword/value pairs, though the interpreter does.

`lambda-parameters-limit` ⇒ 512.

5.3.1. Defining Named Functions

`xcl:definline` *name arg-list* `&body` *body* [*Macro*]

`xcl:definline` is exactly like `defun` except that it also arranges for the compiler to expand inline any calls to the named function. In future releases, this will be accomplished via the inline declaration mechanism. In Lyric, however, the `xcl:defoptimizer` facility is used. As a result, users should take care not to use `xcl:definline` for recursive functions, as this will cause the compiler to loop indefinitely, expanding the recursive calls.

5.3.2. Declaring Global Variables and Named Constants

`xcl:defglobalvar` *name* `&optional` *initial-value doc-string* [*Macro*]

`xcl:defglobalvar` is exactly like `defvar` except that it declares the variable name to be *global* instead of special. Note that if you change a variable from a *global* to a *special*, all functions using that variable must be recompiled. See section 9.1 for more information on global declarations.

`xcl:defglobalparameter` *name initial-value* `&optional` *doc-string* [*Macro*]

`xcl:defglobalparameter` is analogous to `defparameter` except that it declares the parameter as global instead of special.

CHAPTER 6 PREDICATES

Predicates are required to return `nil` for false and non-`nil` for true. There are some types such that `typep` for the type and the specific predicate for the type are equivalent in truth value only (some predicates return `t`, some return the object itself).

The function `subtypep` is defined to return two values. The first value is the value of the predicate, and the second is the certainty of the result. `subtypep` could always return `(values nil nil)` and be legal; however, that wouldn't be a very useful implementation of `subtypep`. Xerox Common Lisp's `subtypep` is guaranteed to handle the following cases:

1. Any two datatypes (including structures defined with `defstruct` with no `:type` option) will return a definite answer.
2. "Built-in" Common Lisp types will return a definite answer.
3. `nil` is `subtypep` of everything.
4. Everything is `subtypep` of `t`.
5. No non-nil type is `subtypep` of `nil`.
6. `and` and `or` of any of the previous expressions are handled properly.

Though `equalp` is required to work for objects "with components", it is not specified if this includes structures. However, there is no other interesting way to test equality of structures. So our implementation defines `equalp` to mean all components are `equalp`. (This is analagous to Interlisp-D's `il:equalall`). Of course, this means `equalp` may not terminate if comparing circular structures, just as `equal` may not terminate given circular lists.

[This page intentionally left blank]

CHAPTER 7 CONTROL STRUCTURE

7.3. Function Invocation

`Call-arguments-limit` ⇒ 512.

7.5. Establishing New Variable Bindings

`xcl:destructuring-bind` *pattern form* `&body` *body* [*Macro*]

Executes *body* with the variables in the s-expression *pattern* bound to elements of the list structure returned by *form*. It is analogous to `multiple-value-bind`. *pattern* can be any arglist acceptable to `defmacro` except that the `&environment` keyword may not be used. For example:

```
(xcl:destructuring-bind
      ((vl  v2) &key a b)
       '((1 2) :b 3 :a 4)
      ...body...)
```

is equivalent to:

```
(let ((vl 1)
     (v2 2)
     (a  4)
     (b  3))
   ...body...)
```

Note: `xcl:destructuring-bind` currently does no error checking for too many or too few elements being returned by *form*.

7.6. Conditionals

`case` *keyform* {({({*key*}*) | *key*} {*form*}*)}* [*Macro*]

Be careful about using `nil` as a keylist in the case macro. `nil` is interpreted as the list of no keys, not as the single key `nil`. Thus, any clause whose car is `nil` will never be selected. To use `nil` as a key, use the keylist `(nil)` instead.

Wrong:

```
(case expression
 ...
(nil ... code for expression being nil...)
  ...
  )
```

Right:

```
(case expression
  ...
  ((nil) ... code for expression being nil...)
  ...
  )
```

7.9.1. Constructs for Handling Multiple Values

```
multiple-values-limit ⇒ 512.
```

CHAPTER 8 MACROS

While the Common Lisp construct `defmacro` does remove any function definition the given symbol may have, it does not remove any Interlisp macro definition that might exist on the `il:macro`, `il:bytemacro`, or `il:dmacro` properties of the symbol. If a given symbol has both a Common Lisp and Interlisp macro definition, the one to be used depends upon the compiler or interpreter in use. The Common Lisp interpreter and the new XCL compiler will both use the Common Lisp macro. The Interlisp interpreter and compiler will use the Interlisp macro. Because of this potential for confusion, it is strongly recommended that, when providing a Common Lisp `defmacro` definition for a symbol, any existing Interlisp macro definition for that symbol should be removed.

Xerox Common Lisp diverges from *Common Lisp: the Language* on the issue of destructuring in `&body` parameters. We implement an extension to that syntax to allow for easy parsing of the bodies of certain macros, such as `defun` and `defmacro`. Some uses of `&body` are interpreted as implicit calls to the XCL function `parse-body`:

`parse-body` body *environment* `&optional` *doc-string-allowed-p* [*Macro*]

The given *body* should be a list of forms in the syntax of a standard Common Lisp `lambda` body, described in *Common Lisp: the Language* as

{*declaration* | *doc-string*}* {*form*}*

The *environment* should be a lexical environment such as those acquired through the use of the `&environment` keyword. *doc-string-allowed-p* defaults to `t`. `parse-body` returns three values:

1. A list of the non-declaration, non-doc-string *form*'s found in the *body*
2. A list of the `declare` forms found in the *body*
3. The documentation string found, if any, or `nil`.

`parse-body` works by examining the forms in *body* one by one, macroexpanding them if necessary, trying to find the first non-declaration, non-string form. The tail of *body* beginning with that form is returned as the first value, a list of all of the declarations found is the second value and, if any documentation string is found, it is returned as the third value. If *doc-string-allowed-p* is `nil`, any strings found will be assumed to be the first form in the tail and the search will end. Note that Common Lisp allows macros to expand into either documentation strings or declarations. Because of this, `parse-body` will always have macroexpanded the first form in the tail. The original, unexpanded form is returned as the `car` of the first value, though.

Because of the usefulness of `parse-body`, and the frequency with which constructs like the following are used:

```
(defmacro define-foo (arg-list &body body
                               &environment env)
          (multiple-value-bind (code decls doc)
                                  (parse-body body env)
              ...))
```

the syntax of the `&body` keyword was changed to allow the following code, with the same meaning:

```
(defmacro define-foo (arg-list &body (code decls doc))
      ...)
```

This frees the programmer from having to specify an `&environment` parameter when it will only be used in a call to `parse-body`.

The full syntax of the XCL `&body` keyword is as follows:

`&body` *symbol*

This is treated exactly like `&rest` *symbol*.

`&body (`*symbol-or-list*`)`

When `&body` precedes a list of length one, it is treated exactly like `&rest` *symbol-or-list*.

`&body (`*symbol-or-list* *symbol-or-list* [*symbol-or-list*]`)`

When followed by a list of length two or three, it is treated as an implicit call to the function `parse-body`, described above. The *body* argument is the list that would have been bound to a simple `&rest` parameter, the *environment* argument is given by what would have been supplied to an `&environment` parameter, and *doc-string-allowed-p* is passed as `t` if, and only if, the third element of the `&body` list is provided. Each of the *symbol-or-list*'s is matched against the corresponding returned value of parse-body. This allows full destructuring on each of those three values, even though it is only likely to be useful at all for the first one.

When this third, so-called "parsing version" of &body is used, no &key parameters are allowed. Also, as described in *Common Lisp: the Language*, only one of &body and &rest may be used in a single argument list. Note also that this extension to Common Lisp contradicts the statement on page 145 of *Common Lisp: the Language* that &body "is identical in function to &rest."

Compatibility Note

All Interlisp nlambda functions appear to be macros from the point of view of the Common Lisp function macro-function. Those Interlisp nlambda functions that actually evaluate some of their arguments have also been defined as real Common Lisp macros. Thus, all calls to Interlisp nlambda functions are treated properly by the Common Lisp interpreter and the new XCL compiler.

[This page intentionally left blank]

CHAPTER 9 DECLARATIONS

9.1. Declaration Syntax

Inline declarations are ignored in this release. The macro `xcl:definline` creates a function, calls to which will be expanded inline.

Xerox Common Lisp supports an additional declaration `xcl:global`.

`(xcl:global` *var1 var2* `...)` specifies that all of the variables named are to be considered *global*, i.e., it is a declaration that the variables are never dynamically bound. All references to such a variable are compiled to fetch the top level binding directly. This declaration for global variables is analogous to the special declaration for special variables.

9.2. Declaration Specifiers

Xerox Common Lisp supports an additional declaration, `xcl:global`.

`(xcl:global` *var1 var2* `...)` specifies that all of the variables named are to be considered *global*. This specifier pervasively affects variable references. The affected references refer directly to the top-level value of the variable, bypassing a search for any intermediate bindings. This can in some cases lead to a significant performance improvement, especially if the references are in deeply-nested, frequently-executed code. Variables declared `xcl:global` may not be bound. The declaration `xcl:global` is analogous to the `special` declaration for special variables.

`inline` declarations are ignored in the Lyric release. Inline functions can be defined using the `xcl:definline` macro, described in Section 5.3.1 of Steele's *Common Lisp: the Language*.

[This page intentionally left blank]

CHAPTER 10 SYMBOLS

Symbol print names are limited to 255 characters.

`symbol-name` returns a string displaced to the symbol pname. Strings returned from `symbol-name` may be destructively modified without affecting the symbol pname.

Interlisp users should note that `cl:gensym` and `il:gensym` are not the same. `il:gensym` always creates a symbol in the Interlisp package, while `cl:gensym` creates uninterned symbols.

[This page intentionally left blank]