LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual

Version 8.0

Copyright and Trademarks

LispWorks Objective-C and Cocoa Interface User Guide and Reference Manual

Version 8.0

December 2021

Address	Telephone	Fax
LispWorks Ltd St. John's Innovation Centre Cowley Road Cambridge CB4 0WS England	From North America: 877 759 8839 (toll-free) From elsewhere: +44 1223 421860	From North America: 617 812 8283 From elsewhere: +44 870 2206189

www.lispworks.com

Contents

1 Introduction to the Objective-C Interface

1.1 Introduction

Objective-C is a C-like object-oriented programming language that is used on macOS to implement the Cocoa API. The LispWorks Objective-C interface is an extension to the interface described in the *Foreign Language Interface User Guide and Reference Manual* to support calling Objective-C methods and also to provide defining forms for Objective-C classes and methods implemented in Lisp. This manual assumes that you are familiar with the LispWorks FLI, the Objective-C language and the Cocoa API where appropriate, and it uses the same notation and conventions as the *Foreign Language Interface User Guide and Reference Manual*.

Note: the LispWorks Objective-C interface is only available on the Macintosh.

The remainder of this chapter describes the LispWorks Objective-C interface, which is generally used in conjunction with the Cocoa API (see **3 The Cocoa Interface**). Examples in this chapter assume that the current package uses the **`objc`** package.

1.1.1 Initialization

Before calling any of the Objective-C interface functions, the runtime system must be initialized. This is done by calling **`ensure-objc-initialized`**, optionally passing a list of foreign modules to be loaded. For example, the following will initialize and load Cocoa:

```
(objc:ensure-objc-initialized
  :modules
  '("/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation"
    "/System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa"))
```

1.2 Objective-C data types

The Objective-C interface uses types in the same way as the LispWorks FLI, with a restricted set of FLI types being used to describe method arguments and results. Some types perform special conversions to hide the FLI details (see **1.3.3 Special argument and result conversion** and **1.4.3.1 Special method argument and result conversion**).

1.2.1 Objective-C pointers and pointer types

Objective-C defines its own memory management, so most interaction with its objects occurs using foreign pointers with the FLI type descriptor **`objc-object-pointer`**. When an Objective-C object class is implemented in Lisp, there is an additional object of type **`standard-objc-object`** which is associated with the foreign pointer (see **1.4 Defining Objective-C classes and methods**).

There are a few specific Objective-C pointer types that have a direct translation to FLI types:

Pointer types in Objective-C

Objective-C type	FLI type descriptor
`Class`	**`objc-class`**
`SEL`	**`sel`**
`id`	**`objc-object-pointer`**
`char *`	**`objc-c-string`**

Other pointer types are represented using the **`:pointer`** FLI type descriptor as normal.

When using pointers to struct types, the type must be defined using **`define-objc-struct`** rather than **`fli:define-c-struct`**.

1.2.2 Integer and boolean types

The various integer types in Objective-C have corresponding standard FLI types. In addition, the Objective-C type **`BOOL`**, which is an integer type with values **`NO`** and **`YES,`** has a corresponding FLI type **`objc-bool`** with values **`nil`** and **`t`**.

1.2.3 Structure types

Structures in Objective-C are like structures in the FLI, but are restricted to using other Objective-C types for the slots. The macro **`define-objc-struct`** must be used to define a structure type that is suitable for use as an Objective-C type.

1.3 Invoking Objective-C methods

Objective-C methods are associated with Objective-C objects or classes and are invoked by name with a specific set of arguments.

1.3.1 Simple calls to instance and class methods

The function **`invoke`** is used to call most methods (but see **1.3.4 Invoking a method that returns a boolean**, **1.3.5 Invoking a method that returns a structure** and **1.3.6 Invoking a method that returns a string or array** for ways of calling more complex methods). This function has two required arguments:

- the foreign pointer whose method should be invoked, and:
- the name of the method (see **1.3.2 Method naming**).

The remaining arguments are passed to the method in the specified order. See **1.3.3 Special argument and result conversion** for information about how the arguments are converted to FLI values.

For example, a call in Objective-C such as:

```
[window close]
```

would be written using **`invoke`** as:

```
(invoke window "close")
```

In addition, **`invoke`** can be used to call class methods for specifically named classes. This is done by passing a string naming the Objective-C class instead of the object.

For example, a class method call in Objective-C such as:

```
[NSObject alloc]
```

would be written using **`invoke`** as:

```
(invoke "NSObject" "alloc")
```

1.3.2 Method naming

Methods in Objective-C have compound names that describe their main name and any arguments. Functions like **`invoke`** that need a method name expect a string with all the name components concatenated together with no spaces.

For example, a call in Objective-C such as:

```
[box setWidth:10 height:20]
```

would be written using **`invoke`** as:

```
(invoke box "setWidth:height:" 10 20)
```

1.3.3 Special argument and result conversion

Since the LispWorks Objective-C interface is an extension of the FLI, most conversion of arguments and results is handled as specified in the *Foreign Language Interface User Guide and Reference Manual*. There are a few exceptions to make it easier to invoke methods with certain commonly used Objective-C classes and structures as shown in the **Special argument and result conversion for `invoke`**. See the specification of **`invoke`** for full details.

Special argument and result conversion for **`invoke`**

Type	Special argument behavior	Special result behavior
`NSRect`	Allows a vector to be passed.	Converts to a vector.
`NSPoint`	Allows a vector to be passed.	Converts to a vector.
`NSSize`	Allows a vector to be passed.	Converts to a vector.
`NSRange`	Allow a cons to be passed.	Converts to a cons.
`BOOL`	Allow **`nil`** or **`t`** to be passed.	None. See **1.3.4 Invoking a method that returns a boolean**.
`id`	Depending on the Objective-C class, allows automatic conversion of strings and arrays.	None. See **1.3.6 Invoking a method that returns a string or array**.
`Class`	Allows a string to be passed.	None.
`char *`	Allows a string to be passed.	Converts to a string.

1.3.4 Invoking a method that returns a boolean

When a method has return type **`BOOL`** on a Macintosh with an Intel CPU, the value is converted to the integer **`0`** or **`1`** because Objective-C cannot distinguish this type from the other integer types. Often it is more convenient to receive the value as a Lisp boolean and this can be done by using the function **`invoke-bool`**, which returns **`nil`** or **`t`**.

For example, a call in Objective-C such as:

```
[box isSquare] ? 1 : 2
```

could be written using **`invoke-bool`** as:

```
(if (invoke-bool box "isSquare") 1 2)
```

1.3.5 Invoking a method that returns a structure

As mentioned in **1.3.3 Special argument and result conversion**, when **`invoke`** is used with a method whose return type is one of the structure types listed in **Special argument and result conversion for `invoke`**, such as **`NSRect`**, a vector or cons containing the fields of the structure is returned. For other structure types defined with **`define-objc-struct`**, the function **`invoke-into`** must be used to call the method. This takes the same arguments as **`invoke`**, except that there is an extra initial argument, *result*, which should be a pointer to a foreign structure of the appropriate type for the method. When the method returns, the value is copied into this structure.

For example, a call in Objective-C such as:

```
{
  NSRect rect = [box frame];
  ...
}
```

could be written using **`invoke-into`** as:

```
(fli:with-dynamic-foreign-objects ((rect cocoa:ns-rect))
  (objc:invoke-into rect box "frame")
  ...)
```

In addition, for the structure return types mentioned in **Special argument and result conversion for `invoke`**, an appropriately sized vector or cons can be passed as *result* and this is filled with the field values.

For example, the above call could also be written using **`invoke-into`** as:

```
(let ((rect (make-array 4)))
  (objc:invoke-into rect box "frame")
  ...)
```

1.3.6 Invoking a method that returns a string or array

The Objective-C classes **`NSString`** and **`NSArray`** are used extensively in Cocoa to represent strings and arrays of various objects. When a method that returns these types is called with **`invoke`**, the result is a foreign pointer of type **`objc-object-pointer`** as for other classes.

In order to obtain a more useful Lisp value, **`invoke-into`** can be used by specifying a type as the extra initial argument. For a method that returns **`NSString`**, the symbol **`string`** can be specified to cause the foreign object to be converted to a string. For a method that returns **`NSArray`**, the symbol **`array`** can be specified and the foreign object is converted to an array of foreign pointers. Alternatively a type such as **`(array string)`** can be specified and the foreign object is converted to an array of strings.

For example, the form:

```
(invoke object "description")
```

will return a foreign pointer, whereas the form:

```
(invoke-into 'string object "description")
```

will return a string.

1.3.7 Invoking a method that returns values by reference

Values are returned by reference in Objective-C by passing a pointer to memory where the result should be stored, just like in the C language. The Objective-C interface in Lisp works similarly, using the standard FLI constructs for this.

For example, an Objective-C method declared as:

```
- (void)getValueInto:(int *)result;
```

might called from Objective-C like this:

```
int getResult(MyObject *object)
{
  int result;
  [object getValueInto:&result];
  return result;
}
```

The equivalent call from Lisp can be made like this:

```
(defun get-result (object)
  (fli:with-dynamic-foreign-objects ((result-value :int))
    (objc:invoke object "getValueInto:" result-value)
    (fli:dereference result-value)))
```

The same technique applies to in/out arguments, but adding code to initialize the dynamic foreign object before calling the method.

1.3.8 Invoking a method that uses vector types

In order to invoke a method that uses vector types (see "Vector types" in the *Foreign Language Interface User Guide and Reference Manual*), calls to **invoke** etc need to specify the argument and result types of the method. This is because vector types are not compatible with the Objective-C Runtime type encoding API.

This is done by passing a list as the *method* argument. For example, yuo can invoke the following methods of **MDLTransform** in the Model I/O API:

```
;; Call -(vector_float3)translationAtTime:(NSTimeInterval)time;
(invoke ptr '("translationAtTime:"
              (:double)
              :result-type fli:vector-float3)
        20d0)

;; -(void)setTranslation:(vector_float3)translation
;;          forTime:(NSTimeInterval)time;
```

```
(objc:invoke ptr '("setTranslation:forTime:"
                   (fli:vector-float3 :double))
             #(22d0 32d0 42d0)
             20d0)
```

1.3.9 Determining whether a method exists

In some cases, an Objective-C class might have a method that is optionally implemented and **invoke** will signal an error if the method is missing for a particular object. To determine whether a method is implemented, call the function **can-invoke-p** with the foreign object pointer or class name and the name of the method.

For example, a call in Objective-C such as:

```
[foo respondsToSelector:@selector(frame)]
```

could be written using **can-invoke-p** as:

```
(can-invoke-p foo "frame")
```

1.3.10 Memory management

Objective-C uses reference counting for its memory management and also provides a mechanism for decrementing the reference count of an object when control returns to the event loop or some other well-defined point.

The following functions are direct equivalents of the memory management methods in the **NSObject** class:

Helper functions for memory management

Function	Method in **NSObject**
retain	**retain**
retain-count	**retainCount**
release	**release**
autorelease	**autorelease**

In addition, the function **make-autorelease-pool** and the macro **with-autorelease-pool** can be used to make autorelease pools if the standard one in the event loop is not available.

1.3.11 Selectors

Some Objective-C methods have arguments or values of type **SEL**, which is a pointer type used to represent selectors. These can be used in Lisp as foreign pointers of type **sel**, which can be obtained from a string by calling **coerce-to-selector**. The function **selector-name** can be used to find the name of a selector.

For example, a call in Objective-C such as:

```
[foo respondsToSelector:@selector(frame)]
```

could be written using **can-invoke-p** as in **1.3.9 Determining whether a method exists** or using selectors as follows:

```
(invoke foo "respondsToSelector:" (coerce-to-selector "frame"))
```

If **`*selector*`** is bound to the result of calling:

```
(coerce-to-selector "frame")
```

then:

```
(selector-name *selector*)
```

will return the string **`"frame"`**.

1.4 Defining Objective-C classes and methods

The preceding sections covered the use of existing Objective-C classes. This section describes how to implement Objective-C classes in Lisp.

1.4.1 Objects and pointers

When an Objective-C class is implemented in Lisp, each Objective-C foreign object has an associated Lisp object that can obtained by the function **`objc-object-from-pointer`**. Conversely, the function **`objc-object-pointer`** can be used to obtain a pointer to the foreign object from its associated Lisp object.

There are two kinds of Objective-C foreign object, classes and instances, each of which is associated with a Lisp object of some class as described in the following table:

Objective-C objects and associated Lisp objects

Objective-C type	FLI type descriptor	Class of associated Lisp object
`Class`	**`objc-class`**	**`standard-class`**
`id`	**`objc-object-pointer`**	subclass of **`standard-objc-object`**

The implementation of an Objective-C class in Lisp consists of a subclass of **`standard-objc-object`** and method definitions that become the Objective-C methods of the Objective-C class.

1.4.2 Defining an Objective-C class

An Objective-C class implemented in Lisp and its associated subclass of **`standard-objc-object`** should be defined using the macro **`define-objc-class`**. This has a syntax similar to **`cl:defclass`**, with additional class options including **`:objc-class-name`**to specify the name of the Objective-C class.

If the superclass list is empty, then **`standard-objc-object`** is used as the default superclass, otherwise **`standard-objc-object`** must be somewhere on class precedence list or included explicitly.

For example, the following form defines a Lisp class called **`my-object`** and an associated Objective-C class called **`MyObject`**.

```
(define-objc-class my-object ()
  ((slot1 :initarg :slot1 :initform nil))
  (:objc-class-name "MyObject"))
```

The class **`my-object`** will inherit from **`standard-objc-object`** and the class **`MyObject`** will inherit from **`NSObject`**. See

1.4.4 How inheritance works for more details on inheritance.

The class returned by **(find-class 'my-object)** is associated with the Objective-C class object for **MyObject**, so:

```
(objc-object-pointer (find-class 'my-object))
```

and:

```
(coerce-to-objc-class "MyObject")
```

will return a pointer to the same foreign object.

When an instance of **my-object** is made using **make-instance**, an associated foreign Objective-C object of the class **MyObject** is allocated by calling the class's **"alloc"** method and initialized by calling the instance's **"init"** method. The **:init-function** initarg can be used to call a different initialization method.

Conversely, if the **"allocWithZone:"** method is called for the class **MyObject** (or a method such as **"alloc"** that calls **"allocWithZone:"**), then an associated object of type **my-object** is made.

Note: If you implement an Objective-C class in Lisp but its name is not referenced at run time, and you deliver a runtime application, then you need to arrange for the Lisp class name to be retained during delivery. See **define-objc-class** for examples of how to do this.

1.4.3 Defining Objective-C methods

A class defined with **define-objc-class** has no methods associated with it by default, other than those inherited from its ancestor classes. New methods can be defined (or overridden) by using the macros **define-objc-method** for instance methods and **define-objc-class-method** for class methods.

Note that the Lisp method definition form is separate from the class definition, unlike in Objective-C where it is embedded in the **@implementation** block. Also, there is no Lisp equivalent of the **@interface** block: the methods of an Objective-C class are just those whose defining forms have been evaluated.

When defining a method, various things must be specified:

- The method name, which is a string as described in **1.3.2 Method naming**.
- The return type, which is an Objective-C FLI type.
- The Lisp class for which this method applies.
- Any extra arguments and their Objective-C FLI types.

For example, a method that would be implemented in an Objective-C class as follows:

```
@implementation MyObject
- (unsigned int)areaOfWidth:(unsigned int)width
                height:(unsigned int)height
{
  return width*height;
}
@end
```

could be defined in Lisp for instances of the **MyObject** class from **1.4.2 Defining an Objective-C class** using the form:

```
(define-objc-method ("areaOfWidth:height:" (:unsigned :int))
    ((self my-object)
     (width (:unsigned :int))
     (height (:unsigned :int)))
    (* width height))
```

The variable **self** is bound to a Lisp object of type **my-object**, and **width** and **height** are bound to non-negative integers. The area is returned to the caller as a non-negative integer.

1.4.3.1 Special method argument and result conversion

For certain types of argument, there is more than one useful conversion from the FLI value to a Lisp value. To control this, the argument specification can include an *arg-style,* which describes how the argument should be converted. If the *arg-style* is specified as **:foreign** then the argument is converted using normal FLI rules, but by default certain types are converted differently:

Special argument conversion for **define-objc-method**

Argument type	Special argument behavior
cocoa:ns-rect	The argument is a vector.
cocoa:ns-point	The argument is a vector.
cocoa:ns-size	The argument is a vector.
cocoa:ns-range	The argument is a cons.
objc-bool	The argument is **nil** or **t**.
objc-object-pointer	Depending on the Objective-C class, allows automatic conversion to a string or array.
objc-c-string	The argument is a string.

Likewise, result conversion can be controlled by the *result-style* specification. If this is :**foreign** then the value is assumed to be suitable for conversion to the *result-type* using the normal FLI rules, but if *result-style* is :**lisp** then additional conversions are performed for specific values of *result-type*:

Special result conversion for **define-objc-method**

Result type	Special result types supported
cocoa:ns-rect	The result can be a vector.
cocoa:ns-point	The result can be a vector.
cocoa:ns-size	The result can be a vector.
cocoa:ns-range	The result can be a cons.
objc-bool	The result can be **nil** or **t**.
objc-object-pointer	The result can be a string or an array. An autoreleased **NSString** or **NSArray** is allocated.
objc-class	The result can be a string naming a class.

1.4.3.2 Defining a method that returns a structure

When a the return type of a method is a structure type such as **cocoa:ns-rect** then the conversion specified in **Special result conversion for define-objc-method** can be used. Alternatively, and for any other structure defined with **define-objc-struct**, the method can specify a variable as its *result-style*. This variable is bound to a pointer to a foreign structure of the appropriate type and the method should set the slots in this structure to specify the result. For example, the following definitions show a method that returns a structure:

```
(define-objc-struct (pair
                     (:foreign-name "_Pair"))
  (:first :float)
  (:second :float))

(define-objc-method ("pair" (:struct pair) result-pair)
    ((this my-object))
  (setf (fli:foreign-slot-value result-pair :first) 1f0
        (fli:foreign-slot-value result-pair :second) 2f0))
```

1.4.4 How inheritance works

1.4.2 Defining an Objective-C class introduced the **define-objc-class** macro with the **:objc-class-name** class option for naming the Objective-C class. Since this macro is like **cl:defclass**, it can specify any number of superclasses from which the Lisp class will inherit and also provides a way for superclass of the Objective-C class to be chosen:

- If some of the Lisp classes in the class precedence list were defined with **define-objc-class** and given an associated Objective-C class name, then the first such class name is used. It is an error for several such classes to be in the class precedence list unless their associated Objective-C classes are also superclasses of each other in the same order as the precedence list.
- If no superclasses have an associated Objective-C class, then the **:objc-superclass-name** class option can be used to specify the superclass explicitly.
- Otherwise **NSObject** is used as the superclass.

For example, both of these definitions define an Objective-C class that inherits from **MyObject**, via **my-object** in the case of **my-special-object** and explicitly for **my-other-object**:

```
(define-objc-class my-special-object (my-object)
  ()
  (:objc-class-name "MySpecialObject"))

(define-objc-class my-other-object ()
  ()
  (:objc-class-name "MyOtherObject")
  (:objc-superclass-name "MyObject"))
```

The set of methods available for a given Objective-C class consists of those defined on the class itself as well as those inherited from its superclasses.

1.4.5 Invoking methods in the superclass

Within the body of a **define-objc-method** or **define-objc-class-method** form, the local macro **current-super** can be used to obtain a special object which will make **invoke** call the method in the superclass of the defining class. This is equivalent to using **super** in Objective-C.

For example, the Objective-C code:

```
@implementation MySpecialObject
- (unsigned int)areaOfWidth:(unsigned int)width
                height:(unsigned int)height
{
  return 4*[super areaOfWidth:width height:height];
}
@end
```

could be written as follows in Lisp:

```
(define-objc-method ("areaOfWidth:height:" (:unsigned :int))
    ((self my-special-object)
     (width (:unsigned :int))
     (height (:unsigned :int)))
  (* 4 (invoke (current-super) "areaOfWidth:height:"
                               width height)))
```

1.4.6 Abstract classes

An abstract class is a normal Lisp class without an associated Objective-C class. As well as defining named Objective-C classes, **define-objc-class** can be used to define abstract classes by omitting the **:objc-class-name** class option.

The main purpose of abstract classes is to simulate multiple inheritance (Objective-C only supports single inheritance): when a Lisp class inherits from an abstract class, all the methods defined in the abstract class become methods in the inheriting class.

For example, the method **"size"** exists in both the Objective-C classes **MyData** and **MyOtherData** because the Lisp classes inherit it from the abstract class **my-size-mixin**, even though there is no common Objective-C ancestor class:

```
(define-objc-class my-size-mixin ()
  ())

(define-objc-method ("size" (:unsigned :int))
    ((self my-size-mixin))
  42)

(define-objc-class my-data (my-size-mixin)
  ()
  (:objc-class-name "MyData"))

(define-objc-class my-other-data (my-size-mixin)
  ()
  (:objc-class-name "MyOtherData"))
```

1.4.7 Instance variables

In a few cases, for instance when using nib files created by Apple's Interface Builder, it is necessary to add Objective-C instance variables to a class. This can be done using the **:objc-instance-vars** class option to **define-objc-class**. For example, the following class contains two instance variables, each of which is a pointer to an Objective-C foreign object:

```
(define-objc-class my-controller ()
  ()
  (:objc-class-name "MyController")
  (:objc-instance-vars
   ("widthField" objc:objc-object-pointer)
   ("heightField" objc:objc-object-pointer)))
```

Given an instance of **my-controller**, the instance variables can be accessed using the accessor

`objc-object-var-value`.

1.4.8 Memory management

Objective-C uses reference counting for its memory management, but the associated Lisp objects are managed by the Lisp garbage collector. When an Objective-C object is allocated, the associated Lisp object is recorded in the runtime system and cannot be removed by the garbage collector. When its reference count becomes zero, the object is removed from the runtime system and the generic function **`objc-object-destroyed`** is called with the object to allow cleanup methods to be implemented. After this point, the object can be removed by the garbage collector as normal.

1.4.9 Using and declaring formal protocols

Classes defined by **`define-objc-class`** can be made to support Objective-C formal protocols by specifying the **`:objc-protocols`** class option. All the standard formal protocols from macOS 10.4 are predefined.

Note: It is not possible to define new protocols entirely in Lisp on macOS 10.5 and later, but existing protocols can be declared using the **`define-objc-protocol`** macro.

2 Objective-C Reference

alloc-init-object *Function*

Summary

Allocates and initializes a foreign Objective-C object.

Package

`objc`

Signature

`alloc-init-object` *class* **`=>`** *pointer*

Arguments

class⇓ A string or Objective-C class pointer.

Values

pointer⇓ A foreign pointer to new Objective-C object.

Description

The function **`alloc-init-object`** calls the Objective-C **`"alloc"`** class method for *class* and then calls the **`"init"`** instance method to return *pointer*. This is equivalent to doing:

```
    (invoke (invoke class "alloc") "init")
```

See also

`invoke`

autorelease *Function*

Summary

Invokes the Objective-C **`"autorelease"`** method.

Package

`objc`

Signature

`autorelease` *pointer* **`=>`** *pointer*

Arguments

pointer⇓ A pointer to an Objective-C foreign object.

Values

pointer The argument *pointer*.

Description

The function **`autorelease`** calls the Objective-C **`"autorelease"`** instance method of *pointer* to register it with the current autorelease pool. The pointer is returned.

See also

`release`
`retain`
`make-autorelease-pool`
`with-autorelease-pool`

can-invoke-p *Function*

Summary

Checks whether a given Objective-C method can be invoked.

Package

`objc`

Signature

`can-invoke-p` *class-or-object-pointer method* **`=>`** *flag*

Arguments

class-or-object-pointer⇓

A string naming an Objective-C class, a pointer to an Objective-C foreign object or the result of calling **`current-super`**.

method⇓ A string naming the method to invoke.

Values

flag⇓ A boolean.

Description

The function **`can-invoke-p`** is used to check whether an Objective-C instance or class method can be invoked (is defined) for a given class or object.

If *class-or-object-pointer* is a string, then it must name an Objective-C class and the class method named *method* in that class is checked. If *class-or-object-pointer* is the result of calling **`current-super`** then the instance method named *method* is checked for the superclass of the current method. Otherwise *class-or-object-pointer* should a foreign pointer to an Objective-C object or class and the appropriate instance or class method named *method* is checked. The value of *method* should be a concatenation of the message name and its argument names, including the colons, for example **`"setWidth:height:"`**.

The return value *flag* is **`nil`** if the method cannot be invoked and **`t`** otherwise.

See also

`invoke`

coerce-to-objc-class *Function*

Summary

Coerces its argument to an Objective-C class pointer.

Package

`objc`

Signature

`coerce-to-objc-class` *class* **`=>`** *class-pointer*

Arguments

class⇓	A string or Objective-C class pointer.

Values

class-pointer	An Objective-C class pointer.

Description

The function **`coerce-to-objc-class`** returns the Objective-C class pointer for the class specified by *class*. If *class* is a string, then the registered Objective-C class pointer is found. Otherwise *class* should be a foreign pointer of type **`objc-class`** and is returned unchanged.

This is the opposite operation to the function **`objc-class-name`**.

See also

`objc-class`
`objc-class-name`

coerce-to-selector *Function*

Summary

Coerces its argument to an Objective-C method selector.

Package

`objc`

Signature

`coerce-to-selector` *method* **`=>`** *selector*

Arguments

method⇓ A string or selector.

Values

selector A selector.

Description

The function **`coerce-to-selector`** returns the selector named by *method*. If *method* is a string, then the registered selector is found or a new one is registered. Otherwise *method* should be a foreign pointer of type **`sel`** and is returned unchanged.

This is the opposite operation to the function **`selector-name`**.

See also

`sel`
`selector-name`

current-super *Local Macro*

Summary

Allows Objective-C methods to invoke their superclass's methods.

Package

`objc`

Signature

`current-super =>` *super-value*

Values

super-value⇓	An opaque value.

Description

The local macro **`current-super`** returns a value which can be passed to **`invoke`** to call a method in the superclass of the current method definition (like **`super`** in Objective-C). **`current-super`** can also be passed to **`can-invoke-p`**. When used within a **`define-objc-method`** form, instance methods in the superclass are invoked and when used within a **`define-objc-class-method`** form, class methods are invoked. *super-value* has dynamic extent and it is an error to use **`current-super`** in any other contexts.

Examples

See **1.4.5 Invoking methods in the superclass**.

See also

`define-objc-method`
`define-objc-class-method`
`invoke`
`can-invoke-p`

define-objc-class *Macro*

Summary

Defines a class and an Objective-C class.

Package

`objc`

Signature

`define-objc-class` *name* **`(`***superclass-name***`*)`** **`(`***slot-specifier***`*)`** *class-option***`*`** **`=>`** *name*

Arguments

name⇓	A symbol naming the class to define.
superclass-name⇓	A symbol naming a superclass.
slot-specifier⇓	A slot description as used by **`cl:defclass`**.
class-option⇓	A class option as used by **`cl:defclass`**.

Values

name	A symbol naming the class to define.

Description

The macro **`define-objc-class`** defines a **`standard-class`** called *name* which is used to implement an Objective-C class. Normal **`cl:defclass`** inheritance rules apply for slots and Lisp methods.

Each *superclass-name* argument specifies a direct superclass of the new class, which can be another Objective-C implementation class or any other **standard-class**, provided that **standard-objc-object** is included somewhere in the overall class precedence list. The class **standard-objc-object** is the default superclass if no others are specified.

*slot-specifier*s are standard **cl:defclass** slot definitions.

*class-option*s are standard **cl:defclass** class options. In addition the following options are recognized:

(:objc-class-name *objc-class-name***)**

This option makes the Objective-C class name used for instances of *name* be the string *objc-class -name*. If none of the classes in the class precedence list of *name* have a **:objc-class-name** option then no Objective-C object is created.

(:objc-superclass-name *objc-superclass-name***)**

This option makes the Objective-C superclass name of the Objective-C class defined by the **:objc-class-name** option be the string *objc-superclass-name*. If omitted, the *objc-superclass-name* defaults to the *objc-class-name* of the first class in the class precedence list that specifies such a name or to **"NSObject"** if no such class is found. It is an error to specify a *objc-superclass-name* which is different from the one that would be inherited from a superclass.

(:objc-instance-vars *var-spec****)**

This options allows Objective-C instance variables to be defined for this class. Each *var-spec* should be a list of the form:

(*ivar-name ivar-type***)**

where *ivar-name* is a string naming the instance variable and *ivar-type* is an Objective-C FLI type. The class will automatically contain all the instance variables specified by its superclasses.

(:objc-protocols *protocol-name****)**

This option allows Objective-C formal protocols to be registered as being implemented by the class. Each *protocol-name* should be a string naming a previously defined formal protocol (see **define-objc-protocol**). The class will automatically implement all protocols specified by its superclasses.

Notes

If *name* is not referenced at run time and you deliver an application relying on your class, then you need to arrange for *name* to be retained during delivery. This can be achieved with the Delivery keyword **:keep-symbols** (see the *Delivery User Guide*), but a more modular approach is shown in the example below.

Examples

Suppose your application relies on a class defined like this:

```
(objc:define-objc-class foo ()
  ()
  (:objc-class-name "Foo"))
```

If your Lisp code does not actually reference **foo** at run time then you must take care to retain your class during Delivery. The best way to achieve this is to keep its name on the plist of some other symbol like this:

```
(setf (get 'make-a-foo 'owner-class) 'foo)
```

Here **`make-a-foo`** is the only code that makes the **`Foo`** Objective-C object, so it is the best place to retain the Lisp class **`foo`** (that is, only if **`make-a-foo`** is retained).

See also

`standard-objc-object`
`define-objc-method`
`define-objc-class-method`
`define-objc-protocol`
1.4.2 Defining an Objective-C class

define-objc-class-method *Macro*

Summary

Defines an Objective-C class method for a specified class.

Package

`objc`

Signature

`define-objc-class-method (`*name result-type* **`&optional`** *result-style***`) (`***object-argspec* **`{`***argspec***`}*) {`***form***`}*`**

object-argspec **`::= (`***object-var class-name* **`[`***pointer-var***`])`**

argspec **`::= (`***arg-var arg-type* **`[`***arg-style***`])`**

Arguments

name⇓	A string naming the method to define.
result-type⇓	An Objective-C FLI type.
result-style⇓	An optional keyword specifying the result conversion style, either **`:lisp`** or **`:foreign`**.
form⇓	A form.
object-var⇓	A symbol naming a variable.
class-name⇓	A symbol naming a class defined with **`define-objc-class`**.
pointer-var⇓	An optional symbol naming a variable.
arg-var⇓	A symbol naming a variable.
arg-type⇓	An Objective-C FLI type.
arg-style⇓	An optional symbol or list specifying the argument conversion style.

Description

The macro **`define-objc-class-method`** defines the Objective-C class method *name* for the Objective-C classes associated with *class-name*. *name* should be a concatenation of the message name and its argument names, including the colons, for example **`"setWidth:height:"`**.

If the **`define-objc-class`** definition of *class-name* specifies the **`(:objc-class-name`** *objc-class-name***`)`** option, then the method is added to the Objective-C class *objc-class-name*. Otherwise, the method is added to the Objective-C class of every subclass of *class-name* that specifies the **`:objc-class-name`** option, allowing a mixin class to define methods that become part of the implementation of its subclasses (see **1.4.6 Abstract classes**).

When the method is invoked, each *form* is evaluated in sequence with *object-var* bound to the (sub)class of *class-name*, *pointer-var* (if specified) bound to the receiver foreign pointer to the Objective-C class and each *arg-var* bound to the corresponding method argument.

See **`define-objc-method`** for details of the argument and result conversion (using *arg-type*, *arg-style*, *result-type* and *result-style*).

*form*s can use functions such as **`invoke`** to invoke other class methods on *pointer-var*. The macro **`current-super`** can be used to obtain an object that allows class methods in the superclass to be invoked (like **`super`** in Objective-C).

See also

`define-objc-class`
`define-objc-method`
`current-super`

define-objc-method *Macro*

Summary

Defines an Objective-C instance method for a specified class.

Package

`objc`

Signature

`define-objc-method (`*name result-type* **`&optional`** *result-style***`)(`***object-argspec* **`{`***argspec***`}*) {`***form***`}*`**

object-argspec **`::= (`***object-var class-name* **`[`***pointer-var***`])`**

argspec **`::= (`***arg-var arg-type* **`[`***arg-style***`])`**

Arguments

name⇓	A string naming the method to define.
result-type⇓	An Objective-C FLI type.
result-style⇓	An optional keyword specifying the result conversion style, either **`:lisp`** or **`:foreign`**, or a symbol naming a variable.
form⇓	A form.
object-var⇓	A symbol naming a variable.
class-name⇓	A symbol naming a class defined with **`define-objc-class`**.
pointer-var⇓	An optional symbol naming a variable.
arg-var⇓	A symbol naming a variable.
arg-type⇓	An Objective-C FLI type.

arg-style⇓ An optional symbol or list specifying the argument conversion style.

Description

The macro **`define-objc-method`** defines the Objective-C instance method *name* for the Objective-C classes associated with *class-name*. *name* should be a concatenation of the message name and its argument names, including the colons, for example **`"setWidth:height:"`**.

If the **`define-objc-class`** definition of *class-name* specifies the **`(:objc-class-name`** *objc-class-name***`)`** option, then the method is added to the Objective-C class *objc-class-name*. Otherwise, the method is added to the Objective-C class of every subclass of *class-name* that specifies the **`:objc-class-name`** option, allowing a mixin class to define methods that become part of the implementation of its subclasses (see **1.4.6 Abstract classes**).

When the method is invoked, each *form* is evaluated in sequence with *object-var* bound to the object of type *class-name* associated with the receiver, *pointer-var* (if specified) bound to the receiver foreign pointer and each *arg-var* bound to the corresponding method argument.

Each argument has an *arg-type* (its Objective-C FLI type) and an optional *arg-style*, which specifies how the FLI value is converted to a Lisp value. If *arg-style* is **`:foreign`**, then *arg-var* is bound to the FLI value of the argument (typically an integer or foreign pointer). Otherwise, *arg-var* is bound to a value converted according to *arg-type*:

`cocoa:ns-rect` If *arg-style* is omitted or **`:lisp`** then the rectangle is converted to a vector of four elements of the form **`#(`***x y width height***`)`**. Otherwise the argument is a foreign pointer to a **`cocoa:ns-rect`** object.

`cocoa:ns-size` If *arg-style* is omitted or **`:lisp`** then the size is converted to a vector of two elements of the form **`#(`***width height***`)`**. Otherwise the argument is a foreign pointer to a **`cocoa:ns-size`** object.

`cocoa:ns-point` If *arg-style* is omitted or **`:lisp`** then the point is converted to a vector of two elements of the form **`#(`***x y***`)`**. Otherwise the argument is a foreign pointer to a **`cocoa:ns-point`** object.

`cocoa:ns-range` If *arg-style* is omitted or **`:lisp`** then the range is converted to a cons of the form **`(`***location* **`.`** *length***`)`**. Otherwise the argument is a foreign pointer to a **`cocoa:ns-range`** object.

`objc-object-pointer` If *arg-style* is the symbol **`string`** then the argument is assumed to be a pointer to an Objective-C **`NSString`** object and is converted to a Lisp string or **`nil`** for a null pointer.

If *arg-style* is the symbol **`array`** then the argument is assumed to be a pointer to an Objective-C **`NSArray`** object and is converted to a Lisp vector or **`nil`** for a null pointer.

If *arg-style* is the a list of the form **`(array`** *elt-arg-style***`)`** then the argument is assumed to be a pointer to an Objective-C **`NSArray`** object and is recursively converted to a Lisp vector using *elt-arg-style* for the elements or **`nil`** for a null pointer.

Otherwise, the argument remains as a foreign pointer to the Objective-C object.

`objc-c-string` If *arg-style* is the symbol **`string`** then the argument is assumed to be a pointer to a foreign string and is converted to a Lisp string or **`nil`** for a null pointer.

After the last *form* has been evaluated, its value is converted to *result-type* according to *result-style* and becomes the result of the method.

If *result-style* is a non-keyword symbol and *result-type* is a foreign structure type defined with **`define-objc-struct`** then the variable named by *result-style* is bound to a pointer to a foreign object of type *result-type* while *form*s are evaluated. *form*s must set the slots in this foreign object to specify the result.

If *result-style* is :**`foreign`** then the value is assumed to be suitable for conversion to *result-type* using the normal FLI rules.

If *result-style* is `:lisp` then additional conversions are performed for specific values of *result-type*:

`cocoa:ns-rect` If the value is a vector of four elements of the form `#(`*x y width height*`)`, the *x*, *y*, *width* and *height* are used to form the returned rectangle. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-rect`** and is copied.

`cocoa:ns-size` If the value is a vector of two elements of the form `#(`*width height*`)`, the *width* and *height* are used to form the returned size. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-size`** and is copied.

`cocoa:ns-point` If the value is a vector of two elements of the form `#(`*x y*`)`, the *x* and *y* are used to form the returned point. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-point`** and is copied.

`cocoa:ns-range` If the value is a cons of the form `(`*location* `.` *length*`)`, the *location* and *length* are used to form the returned range. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-range`** object and is copied.

`(:signed :char)` or `(:unsigned :char)`

If the value is `nil` then `NO` is returned.If the value is `t` then `YES` is returned. Otherwise the value must be an appropriate integer for *result-type*.

`objc-object-pointer` If the value is a string then it is converted to a newly allocated Objective-C `NSString` object which the caller is expected to release.

If the value is a vector then it is recursively converted to a newly allocated Objective-C `NSArray` object which the caller is expected to release.

If the value is `nil` then a null pointer is returned.

Otherwise the value should be a foreign pointer to an Objective-C object of the appropriate class.

`objc-class` The value is coerced to a Objective-C class pointer as if by **`coerce-to-objc-class`**. In particular, this allows strings to be returned.

*form*s can use functions such as **`invoke`** to invoke other methods on *pointer-var*. The macro **`current-super`** can be used to obtain an object that allows methods in the superclass to be invoked (like `super` in Objective-C).

Examples

See **1.4.3 Defining Objective-C methods**.
See **1.4.5 Invoking methods in the superclass**.
See **1.4.6 Abstract classes**.

See also

`define-objc-class`
`define-objc-class-method`
`current-super`
`define-objc-struct`

define-objc-protocol *Macro*

Summary

Defines an Objective-C formal protocol.

Package

`objc`

Signature

`define-objc-protocol` *name* **`&key`** *incorporated-protocols instance-methods class-methods*

Arguments

name⇓	A string naming the protocol to define.
incorporated-protocols⇓	A list of protocol names.
instance-methods⇓	A list of instance method specifications.
class-methods⇓	A list of class method specifications.

Description

The macro **`define-objc-protocol`** defines an Objective-C formal protocol named by *name* for use in the **`:objc-class-protocols`** option of **`define-objc-class`**.

If *incorporated-protocols* is specified, it should be a list of already defined formal protocol names. These protocols are registered as being incorporated within *name*. The default is for no protocols to be incorporated.

If *instance-methods* or *class-methods* are specified, they define the instance and class methods respectively in the protocol. Each should give a list of method specifications, which are lists of the form:

`(`*name result-type arg-type***`*)`**

with components:

name	A string naming the method. *name* should be a concatenation of the message name and its argument names, including the colons, for example **`"setWidth:height:"`**.
result-type	The Objective-C FLI type that the method returns.
arg-type	The Objective-C FLI type of the corresponding argument of the method.

The receiver and selector arguments should not be specified by the *arg-type*s. All the standard Cocoa Foundation and Application Kit protocols from the macOS 10.4 SDK are predefined by LispWorks.

Notes

It is not possible to define new protocols entirely in Lisp on macOS 10.5 and later, but **`define-objc-protocol`** can be used to declare existing protocols.

See also

define-objc-class

define-objc-struct *Macro*

Summary

Defines a foreign structure for use with Objective-C.

Package

objc

Signature

define-objc-struct (*name* **{***option***}*) {***slot***}***

option **::= (:foreign-name** *foreign-name***) | (:typedef-name** *typedef-name***)**

slot **::= (***slot-name slot-type***)**

Arguments

name⇓	A symbol naming the foreign structure type.
foreign-name⇓	A string giving the foreign structure name.
typedef-name⇓	A symbol naming a foreign structure type alias.
slot-name⇓	A symbol naming the foreign slot.
slot-type⇓	An FLI type descriptor for the foreign slot.

Description

The macro **define-objc-struct** defines a foreign structure type named *name* with the slots specified by each *slot-name* and *slot-type*. In addition, **(:struct** *name***)** becomes an Objective-C type that can be used with **invoke**, **invoke-into** and **define-objc-method** or **define-objc-class-method**.

foreign-name must be specified to allow the Objective-C runtime system to identify the type.

If *typedef-name* is specified,it allows that symbol to be used in place of **(:struct** *name***)** when using the type in a **define-objc-method** or **define-objc-class-method** form.

See also

invoke-into
define-objc-method
define-objc-class-method

description *Function*

Summary

Calls the Objective-C **`"description"`** instance method.

Package

`objc`

Signature

`description` *pointer* **`=>`** *string*

Arguments

pointer⇓	A pointer to an Objective-C foreign object.

Values

string	A string.

Description

The function **`description`** calls the Objective-C **`"description"`** instance method of *pointer* and returns the description as a string.

ensure-objc-initialized *Function*

Summary

Initializes the Objective-C system if required.

Package

`objc`

Signature

`ensure-objc-initialized &key` *modules*

Arguments

modules⇓	A list of strings.

Description

The function **`ensure-objc-initialized`** must be called before any other functions in the **`objc`** package to initialize the Objective-C system. It is safe to use the defining macros such as **`define-objc-class`** and **`define-objc-method`** before calling **`ensure-objc-initialized`**.

modules can be a list of strings specifying foreign modules to load. Typically, this needs to be the paths to the Cocoa **`.dylib`** files to make Objective-C work. See **`fli:register-module`**.

Note: Do not call **`ensure-objc-initialized`** in a LispWorks for iOS Runtime application, because this has already been done by LispWorks when the application starts.

invoke *Function*

Summary

Invokes an Objective-C method.

Package

`objc`

Signature

`invoke` *class-or-object-pointer method* **`&rest`** *args* **`=>`** *value*

Arguments

class-or-object-pointer⇓

A string naming an Objective-C class, a pointer to an Objective-C foreign object or the result of calling **`current-super`**.

method⇓ A string naming the method to invoke or a list as specified below.

args⇓ Arguments to the method.

Values

value The value returned by the method.

Description

The function **`invoke`** is used to call Objective-C instance and class methods.

If *class-or-object-pointer* is a string, then it must name an Objective-C class and the class method named *method* in that class is called. If *class-or-object-pointer* is the result of calling **`current-super`** then the instance method named *method* is invoked for the superclass of the current method. Otherwise *class-or-object-pointer* should be a foreign pointer to an Objective-C object or class and the appropriate instance or class method named *method* is invoked.

If *method* is a string then it should be a concatenation of the message name and its argument names, including the colons, for example **`"setWidth:height:"`**.

Otherwise *method* must be a list in one of two forms:

- **`(`***method-name arg-types***`)`**
- **`(`***method-name arg-types* **`:result-type`** *result-type***`)`**

method-name must be a string, as described when *method* is a string above. *arg-types* must be a list of FLI argument types, each one matching the corresponding argument to the method. *result-type* must be the FLI result type of the method, which defaults to **`:void`** if omitted. This is primarily intended for invoking methods using vector types, which are not compatible with the Objective-C Runtime type encoding API. See **1.3.8 Invoking a method that uses vector types**.

Each argument in *args* is converted to an appropriate FLI Objective-C value and is passed in order to the method. This conversion is done based on the signature of the method as follows:

`NSRect`

If the argument is a vector of four elements of the form **`#(`***x y width height***`)`**, the *x*, *y*, *width* and *height* are used to form the rectangle. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-rect`** nd is copied.

`NSSize`

If the argument is a vector of two elements of the form **`#(`***width height***`)`**, the *width* and *height* are used to form the size. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-size`** and is copied.

`NSPoint`

If the argument is a vector of two elements of the form **`#(`***x y***`)`**, the *x* and *y* are used to form the point. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-point`** and is copied.

`NSRange`

If the argument is a cons of the form **`(`***location* **`.`** *length***`)`**, the *location* and *length* are used to form the range. Otherwise it is assumed to be a foreign pointer to a **`cocoa:ns-range`** object and is copied.

other structures

The argument should be a foreign pointer to the appropriate struct object and is copied.

`BOOL`

If the argument is **`nil`** then **`NO`** is passed, if the argument is **`t`** then **`YES`** is passed. Otherwise the argument must be an integer (due to a limitation in the Objective-C type system, this case cannot be distinguished from the **`signed char`** type).

`id`

If the argument is a string then it is converted to a newly allocated Objective-C **`NSString`** object which is released when the function returns.

If the argument is a vector then it is recursively converted to a newly allocated Objective-C **`NSArray`** object which is released when the function returns.

If the argument is **`nil`** then a null pointer is passed.

Otherwise the argument should be a foreign pointer to an Objective-C object of the appropriate class.

`Class`

The argument is coerced to an Objective-C class pointer as if by **`coerce-to-objc-class`**. In particular, this allows strings to be passed as class arguments.

`char *`

If the argument is a string then it is converted to a newly allocated foreign string which is freed when the function returns.

Otherwise the argument should be a foreign pointer.

struct *structname* *

The argument should be a foreign pointer to a struct whose type is defined by **`define-objc-struct`** with **`:foreign-name`** *structname*.

other integer and pointer types

All other integer and pointer types are converted using the normal FLI rules.

When the method returns, its value is converted according to its type:

`NSRect`

A vector of four elements of the form **`#(`***x y width height***`)`** is created containing the rectangle.

`NSSize`

A vector of two elements of the form **`#(`***width height***`)`** is created containing the size.

`NSPoint` A vector of two elements of the form `#(`*x y*`)` is created containing the point.

`NSRange` A cons of the form `(`*location* `.` *length*`)` is created containing the range.

other structures Other structures cannot be returned by value using **`invoke`**. See **`invoke-into`** for how to handle these types.

`BOOL` If the value is **`NO`** then **`0`** is returned, otherwise **`1`** is returned. See also **`invoke-bool`**.

`id` An object of type **`objc-object-pointer`** is returned.

`char *` The value is converted to a string and returned.

other integer and pointer types

All other integer and pointer types are converted using the normal FLI rules.

See also

`invoke-bool`
`invoke-into`
`can-invoke-p`

invoke-bool *Function*

Summary

Invokes an Objective-C method that returns a **`BOOL`**.

Package

`objc`

Signature

`invoke-bool` *class-or-object-pointer method* **`&rest`** *args* **`=>`** *value*

Arguments

class-or-object-pointer⇓

A string naming an Objective-C class, a pointer to an Objective-C foreign object or the result of calling **`current-super`**.

method⇓ A string naming the method to invoke or a list as specified by **`invoke`**.

args⇓ Arguments to the method.

Values

value The value returned by the method.

Description

The function **`invoke-bool`** is used to call Objective-C instance and class methods that return the type **`BOOL`**. It behaves identically to **`invoke`**, except that if the return value is **`NO`** then **`nil`** is returned, otherwise **`t`** is returned. The meaning of *class-or-object-pointer*, *method* and *args* is identical to **`invoke`**.

See also

`invoke`
`invoke-into`

invoke-into *Function*

Summary

Invokes an Objective-C method that returns a specific type or fills a specific object.

Package

`objc`

Signature

`invoke-into` *result class-or-object-pointer method* **`&rest`** *args* **`=>`** *value*

Arguments

result⇓ A symbol or list naming the return type or an object to contain the returned value.

class-or-object-pointer⇓

A string naming an Objective-C class, a pointer to an Objective-C foreign object or the result of calling **`current-super`**.

method⇓ A string naming the method to invoke or a list as specified by **`invoke`**.

args⇓ Arguments to the method.

Values

value The value returned by the method.

Description

The function **`invoke-into`** is used to call Objective-C instance and class methods that return specific types which are not supported directly by **`invoke`** or for methods that return values of some foreign structure type where an existing object should be filled with the value. The meaning of *class-or-object-pointer*, *method* and *args* is identical to **`invoke`**.

The value of *result* controls how the value of the method is converted and returned as follows:

the symbol **`string`** If the result type of the method is **`id`**, then the value is assumed to be an Objective-C object of class **`NSString`** and is converted a string and returned. Otherwise no special conversion is performed.

the symbol **`array`**

If the result type of the method is **`id`**, then the value is assumed to be an Objective-C object of class **`NSArray`** and is converted a vector and returned. Otherwise no special conversion is performed.

a list of the form **`(array`** *elt-type***`)`**

If the result type of the method is **`id`**, then the value is assumed to be an Objective-C object of class **`NSArray`** and is recursively converted a vector and returned. The component *elt-type* should be either **`string`**, **`array`** or another list of the form **`(array`** *sub-elt-type***`)`** and is used to control the conversion of the elements.

Otherwise no special conversion is performed.

the symbol **`:pointer`**

If the result type of the method is **`unsigned char *`**, then the value is returned as a pointer of type **`objc-c-string`**.

Otherwise no special conversion is performed.

a list of the form **`(:pointer`** *elt-type***`)`**

If the result type of the method is **`unsigned char *`**, then the value is returned as a pointer with element type *elt-type*.

Otherwise no special conversion is performed.

a pointer to a foreign structure

If the result type of the method is a foreign structure type defined with **`define-objc-struct`** or a built-in structure type such as **`NSRect`**, the value is copied into the structure pointed to by *result* and the pointer is returned. Otherwise no special conversion is performed.

an object of type **`vector`**

If the result type of the method is **`id`**, then the value is assumed to be an Objective-C object of class **`NSArray`** and is converted to fill the vector, which must be at least as long as the **`NSArray`**. The vector is returned.

If the result type of the method is **`NSRect`**, **`NSSize`** or **`NSPoint`** then the first 4, 2 or 2 elements respectively of the vector are set to the corresponding components of the result. The vector is returned.

Otherwise no special conversion is performed.

an object of type **`cons`**

If the result type of the method is **`NSRange`** then the **`car`** of the cons is set to the *location* of the range and the **`cdr`** of the cons is set to the *length* of the range. The cons is returned.

Otherwise no special conversion is performed.

See also

`invoke`
`invoke-bool`
`define-objc-struct`

make-autorelease-pool *Function*

Summary

Makes an autorelease pool for the current thread.

Package

`objc`

Signature

`make-autorelease-pool =>` *pool*

Values

pool	A foreign pointer to an autorelease pool object.

Description

The function **`make-autorelease-pool`** returns a new Objective-C autorelease pool for the current thread. An autorelease pool is provided automatically for the main thread when running CAPI with Cocoa, but other threads need to allocate one if they call Objective-C methods that use **`autorelease`**.

See also

`autorelease`
`with-autorelease-pool`

objc-at-question-mark *FLI Type Descriptor*

Summary

A foreign type corresponding to '@?' character pair in the type encoding of a method.

Package

`objc`

Syntax

`objc-at-question-mark`

Description

The FLI type **`objc-at-question-mark`** is corresponds to the '@?' character pair in the type encoding of a method.

According to the documentation this is an illegal combination, but experimentally it is used by Apple. It seems to be used when the argument should be a pointer to a (Clang) block, which is the foreign type **`fli:foreign-block-pointer`** in LispWorks. Since this is not documented, it cannot be relied on.

Notes

At the time of writing **`objc-at-question-mark`** is an alias for the FLI type **`:pointer`**.

See also

`objc-class-method-signature`

objc-bool *FLI Type Descriptor*

Summary

A foreign type for the Objective-C type **`BOOL`**.

Package

`objc`

Syntax

`objc-bool`

Description

The FLI type **`objc-bool`** is a boolean type for use as the Objective-C type **`BOOL`**. It converts between **`nil`** and **`NO`** and between non-nil and **`YES`**.

See also

`invoke-bool`

objc-c++-bool *FLI Type Descriptor*

Summary

A foreign type corresponding to the C++ bool or the C99 _Bool type.

Package

`objc`

Syntax

`objc-c++-bool`

Description

The FLI type **`objc-c++-bool`** corresponds to the C++ bool or C99 _Bool types (the 'B' character in the type encoding defined by the Type Encodings section of Apple's Objective-C Runtime Programming Guide). Note that most boolean values are specified using the Objective-C BOOL type (**`objc-bool`** in LispWorks), so **`objc-c++-bool`** is not commonly used. However, on Macs based on Apple silicon, the Objective-C BOOL type is the C99 _Bool type, so you may see

objc-c++-bool in error messages or foreign template definitions.

Notes

At the time of writing **objc-c++-bool** is an alias for the FLI type **(:boolean :standard)**.

See also

objc-class-method-signature

objc-class *FLI Type Descriptor*

Summary

A foreign type for pointers to Objective-C class objects.

Package

objc

Syntax

objc-class

Description

The FLI type **objc-class** is a pointer type that is used to represent pointers to Objective-C class objects. This is like the **Class** type in Objective-C.

See also

objc-object-pointer

objc-class-method-signature *Function*

Summary

Tries to find the relevant method, and returns its signature.

Package

objc

Signature

objc-class-method-signature *class-spec method-name* **=>** *arg-types*, *result-type*, *type-encoding*

Arguments

class-spec⇓	A string, an **objc-object-pointer** or an **objc-class** pointer.
method-name⇓	A string.

Values

arg-types⇓	A list.
result-type	A foreign type descriptor.
type-encoding	A string.

Description

The function **`objc-class-method-signature`** tries to find the relevant method, and returns its signature.

class-spec needs to be a string naming a class, an **`objc-object-pointer`** foreign pointer (which specifies its class), or an **`objc-class`** pointer.

method-name specifies the method name. It can be either a class method or an instance method.

The first return value is a list of the argument types (that is, foreign types). Note that the first and second arguments of all Objective-C methods are the object/class and the method selector (name). These are are typed as **`objc-object-pointer`** and **`sel`**, so *arg-types* always starts with these two symbols.

The second return value is the result type of the method.

The third return value is a string which is the type encoding of the signature of the method, as stored internally by the Objective-C runtime system.

If **`objc-class-method-signature`** fails to locate the method, it returns **`nil`**.

See also

`objc-class`
`objc-object-pointer`
`sel`

objc-class-name *Function*

Summary

Returns the name of an Objective-C class.

Package

`objc`

Signature

`objc-class-name` *class* **`=>`** *name*

Arguments

class⇓	A pointer to an Objective-C class.

Values

name	A string.

Description

The function **`objc-class-name`** returns the name of the Objective-C class *class* as a string.

This is the opposite operation to the function **`coerce-to-objc-class`**.

See also

`objc-class`
`coerce-to-objc-class`

objc-c-string *FLI Type Descriptor*

Summary

A foreign type for the Objective-C type **`char *`**.

Package

`objc`

Syntax

`objc-c-string`

Description

The FLI type **`objc-c-string`** is a pointer type for use where the Objective-C type **`char *`** occurs as the argument in a method definition. It converts the argument to a string within the body of the method.

See also

`define-objc-method`

objc-object-destroyed *Generic Function*

Summary

Called when an Objective-C is destroyed.

Package

`objc`

Signature

`objc-object-destroyed` *object*

Method signatures

`objc-object-destroyed (`*object* **`standard-objc-object)`**

Arguments

object⇓ An object of type **`standard-objc-object`**.

Description

When an Objective-C foreign object is destroyed (when the reference count becomes zero) and its class was defined by **`define-objc-class`**, the runtime system calls the generic function **`objc-object-destroyed`** with *object* being the associated object of type **`standard-objc-object`** to allow cleanups to be done.

The built-in primary method specializing *object* on **`standard-objc-object`** does nothing, but typically **`:after`** methods are defined to handle class-specific cleanups. This function should not be called directly.

Defining a method for **`objc-object-destroyed`** is similar to implementing **`"dealloc"`** in Objective-C code.

See also

`release`
`standard-objc-object`

objc-object-from-pointer *Function*

Summary

Finds the Lisp object associated with a given Objective-C foreign pointer.

Package

`objc`

Signature

`objc-object-from-pointer` *pointer* **`=>`** *object*

Arguments

pointer⇓ A pointer to an Objective-C foreign object.

Values

object⇓ The Lisp object associated with *pointer*.

Description

The function **`objc-object-from-pointer`** returns the Lisp object *object* associated with the Objective-C foreign object referenced by *pointer*. For an Objective-C instance, *object* is of type **`standard-objc-object`** and for an Objective-C class it is the **`standard-class`** that was defined by **`define-objc-class`**.

Note that for a given returned *object*, the value of the form:

```
(objc-object-pointer object)
```

has the same address as *pointer*.

See also

define-objc-class
standard-objc-object
objc-object-pointer

objc-object-pointer *Function*

Summary

Returns the Objective-C foreign pointer associated with a given Lisp object.

Package

objc

Signature

objc-object-pointer *object-or-class* **=>** *pointer*

Arguments

object-or-class⇓	An instance of **standard-objc-object** or a class defined by **define-objc-class**.

Values

pointer⇓	A pointer to an Objective-C foreign object or class.

Description

The function **objc-object-pointer** returns the Objective-C foreign pointer associated with a given Lisp object. If *object* is an instance of **standard-objc-object** then *pointer* will have foreign type **objc-object-pointer**. Otherwise, *object* should be a class defined by **define-objc-class** and the associated Objective-C class object is returned as a foreign pointer of type **objc-class**.

Note that for a given returned *pointer*, the value of the form:

```
(objc-object-from-pointer pointer)
```

is *object-or-class*.

See also

standard-objc-object
define-objc-class
objc-object-pointer
objc-class
objc-object-from-pointer

objc-object-pointer *FLI Type Descriptor*

Summary

A foreign type for pointers to Objective-C foreign objects.

Package

`objc`

Syntax

`objc-object-pointer`

Description

The FLI type **`objc-object-pointer`** a pointer type that is used to represent pointers to Objective-C foreign objects. This is like the **`id`** type in Objective-C.

See also

`objc-object-from-pointer`
`objc-class`

objc-object-var-value *Accessor*

Summary

Accesses an Objective-C instance variable.

Package

`objc`

Signature

`objc-object-var-value` *object var-name* **`&key`** *result-pointer* **`=>`** *value*

`(setf objc-object-var-value)` *value object var-name* **`&key`** *result-pointer* **`=>`** *value*

Arguments

object⇓	A object of type **`standard-objc-object`**.
var-name⇓	A string.
result-pointer⇓	A foreign pointer or **`nil`**.
value⇓	A value.

Values

value⇓ A value.

Description

The accessor **`objc-object-var-value`** gets or gets the value of the instance variable *var-name* in the Objective-C foreign object associated with *object*. The type of *value* depends on the declared type of the instance variable. If this type is a foreign structure type, then *result-pointer* should be supplied to the reader, giving a pointer to a foreign object of the correct type that is filled with the value.

Note that it is only possible to access instance variables that are defined in Lisp by **`define-objc-class`**, not those inherited from superclasses implemented in Objective-C.

See also

`standard-objc-object`
`define-objc-class`

objc-unknown *FLI Type Descriptor*

Summary

A foreign type corresponding to '?' character in the type encoding of a method.

Package

`objc`

Syntax

`objc-unknown`

Description

The FLI type **`objc-unknown`** corresponds to '?' character in the type encoding of a method.

In general, you do not need to use this, but you may see it in the result of **`objc-class-method-signature`**.

Notes

At the time of writing **`objc-unknown`** is an alias for the FLI type **`:void`**.

See also

`objc-class-method-signature`

release *Function*

Summary

Invokes the Objective-C **`"release"`** method.

Package

`objc`

Signature

`release` *pointer*

Arguments

pointer⇓	A pointer to an Objective-C foreign object.

Description

The function **`release`** calls the Objective-C **`"release"`** instance method of *pointer* to decrement its retain count.

See also

`retain`
`autorelease`
`retain-count`

retain *Function*

Summary

Invokes the Objective-C **`"retain"`** method.

Package

`objc`

Signature

`retain` *pointer* **`=>`** *pointer*

Arguments

pointer⇓	A pointer to an Objective-C foreign object.

Values

pointer	An argument *pointer*.

Description

The function **`retain`** calls the Objective-C **`"retain"`** instance method of *pointer* to decrement its retain count. *pointer* is returned.

See also

`release`
`autorelease`
`retain-count`

retain-count *Function*

Summary

Invokes the Objective-C **`"retainCount"`** method.

Package

`objc`

Signature

`retain-count` *pointer* **`=>`** *retain-count*

Arguments

pointer⇓	A pointer to an Objective-C foreign object.

Values

retain-count	An integer.

Description

The function **`retain-count`** calls the Objective-C **`"retainCount"`** instance method of *pointer* to return its retain count.

See also

`retain`
`release`

sel *FLI Type Descriptor*

Summary

A foreign type for Objective-C method selectors.

Package

`objc`

Syntax

`sel`

Description

The FLI type **`sel`** is an opaque type used to represent method selectors. This is like the **`SEL`** type in Objective-C.

A selector can be obtained from a string by calling the function **`coerce-to-selector`**.

See also

`coerce-to-selector`
`define-objc-method`

selector-name *Function*

Summary

Returns the name of a method selector.

Package

`objc`

Signature

`selector-name` *selector* **`=>`** *name*

Arguments

selector⇓ A string or selector.

Values

name A string.

Description

The function **`selector-name`** returns the name of the method selector *selector*. If *selector* is a string then it is returned unchanged, otherwise it should be a foreign **`sel`** pointer and its name is returned.

This is the opposite operation to the function **`coerce-to-selector`**.

See also

`sel`
`coerce-to-selector`

standard-objc-object *Abstract Class*

Summary

The class from which all classes that implement an Objective-C class should inherit.

Package

`objc`

Superclasses

`standard-object`

Initargs

`:init-function`	An optional function that is called to initialize the Objective-C foreign object.
`:pointer`	An optional Objective-C foreign object pointer for the object.

Readers

`objc-object-pointer`

Description

The abstract class **`standard-objc-object`** provides the framework for subclasses to implement an Objective-C class. Subclasses are typically defined using **`define-objc-class`**, which allows the Objective-C class name to be specified. Instances of such a subclass have an associated Objective-C foreign object whose pointer can be retrieved using the **`objc-object-pointer`** accessor. The function **`objc-object-from-pointer`** can be used to obtain the object again from the Objective-C foreign pointer.

There are two ways that subclasses of **`standard-objc-object`** can be made:

- Via **`make-instance`**. In this case, the Objective-C object is allocated automatically by calling the Objective-C class's **`"alloc"`** method. If the *init-function* initarg is not specified, the object is initialized by calling its **`"init"`** method. If the *init-function* initarg is specified, it is called during initialization with the newly allocated object and it should call the appropriate initialization method for that object and return its result. This allows a specific initialization method, such as **`"initWithFrame:"`**, to be called if required.
- Via the Objective-C class's **`"allocWithZone:"`** method (or a method such as **`"alloc"`** that calls **`"allocWithZone:"`**). In this case, an instance of the subclass of **`standard-objc-object`** is made with the value of the *pointer* initarg being a pointer to the newly allocated Objective-C foreign object.

See also

`define-objc-class`
`objc-object-destroyed`
`objc-object-from-pointer`
`objc-object-pointer`

trace-invoke *Function*

Summary

Traces the invocation of an Objective-C method.

Package

`objc`

Signature

`trace-invoke` *method*

Arguments

method⇓ A string.

Description

The function **`trace-invoke`** sets up a trace on **`invoke`** for calls to the Objective-C method named *method*. Use **`untrace-invoke`** to remove any such tracing.

See also

`invoke`
`untrace-invoke`

untrace-invoke *Function*

Summary

Removes traces of the invocation of an Objective-C method.

Package

`objc`

Signature

`untrace-invoke` *method*

Arguments

method⇓ A string.

Description

The function **`untrace-invoke`** removes any tracing on **`invoke`** for calls to the Objective-C method named *method*.

See also

invoke
trace-invoke

with-autorelease-pool *Macro*

Summary

Evaluates forms in the scope of a temporary autorelease pool.

Package

objc

Signature

with-autorelease-pool (*option****)** *form**** =>** *values*

Arguments

option⇓	There are currently no options.
form⇓	A form.

Values

values	The values returned by the last *form*.

Description

The macro **with-autorelease-pool** creates a new autorelease pool and evaluates each *form* in sequence. The pool is released at the end, even if a non-local exit is performed by *form*s. An autorelease pool is provided automatically for the main thread when running CAPI with Cocoa, but other threads need to allocate one if they call Objective-C methods that use **autorelease**.

option must be empty.

Examples

The **"description"** method returns an autoreleased **NSString**, so to make this function safe for use anywhere, the **with-autorelease-pool** macro is used:

```
(defun object-description (object)
  (with-autorelease-pool ()
    (invoke-into 'string object "description")))
```

See also

autorelease
make-autorelease-pool

3 The Cocoa Interface

3.1 Introduction

Cocoa is an extensive macOS API for access to a variety of operating system services, mostly through Objective-C classes and methods. These can be used via the Objective-C interface described in the preceding chapters, but there are a few foreign structure types and helper functions defined in the **`cocoa`** package that are useful.

3.2 Types

There are four commonly used structure types in Cocoa that have equivalents in the Objective-C interface. In addition, each one has a helper function that will set its slots.

Cocoa structure types and helper functions

Objective-C type	FLI type descriptor	Helper function to set the slots
`NSRect`	**`cocoa:ns-rect`**	**`cocoa:set-ns-rect*`**
`NSPoint`	**`cocoa:ns-point`**	**`cocoa:set-ns-point*`**
`NSSize`	**`cocoa:ns-size`**	**`cocoa:set-ns-size*`**
`NSRange`	**`cocoa:ns-range`**	**`cocoa:set-ns-range*`**

3.3 Observers

Cocoa provides a mechanism called notification centers to register observers for particular events. The helper functions **`cocoa:add-observer`** and **`cocoa:remove-observer`** can be used to add and remove observers.

3.4 How to run Cocoa on its own

This section describes how you can run LispWorks as a Cocoa application, either by saving a LispWorks development image with a suitable restart function, or by delivering a LispWorks application which uses a nib file generated by Apple's Interface Builder.

3.4.1 LispWorks as a Cocoa application

The following startup function can be used to make LispWorks run as a Cocoa application. Typically, before calling **`"run"`** you would create an application delegate with a method on **`applicationDidFinishLaunching:`** to initialize the application's windows.

```
(defun init-function ()
  (mp:initialize-multiprocessing
   "main thread"
   '()
   #'(lambda ()
       (objc:ensure-objc-initialized
        :modules
        '("/System/Library/Frameworks/Foundation.framework/Versions/C/Foundation"
          "/System/Library/Frameworks/Cocoa.framework/Versions/A/Cocoa"))
       (objc:with-autorelease-pool ()
         (let ((app (objc:invoke "NSApplication"
                                 "sharedApplication")))
           (objc:invoke app "run"))))))
```

To use this, a bundle must be created, calling **`init-function`** on startup. For example, the following build script will create **`lw-cocoa-app.app`**:

```
(in-package "CL-USER")
(load-all-patches)
(example-compile-file
 "configuration/macos-application-bundle.lisp" :load t)
(save-image (when (save-argument-real-p)
              (write-macos-application-bundle "lw-cocoa-app"))
            :restart-function 'init-function)
```

See "Saving a LispWorks image" in the *LispWorks® User Guide and Reference Manual* for information on using a build script to create a new LispWorks image.

3.4.2 Using a nib file in a LispWorks application

For a complete example demonstrating how to build a standalone Cocoa application which uses a nib file, see these two files:

```
(example-edit-file "objc/area-calculator/area-calculator")
```

```
(example-edit-file "objc/area-calculator/deliver")
```

The area calculator example connects the nib file generated by Apple's Interface Builder to a Lisp implementation of an Objective-C class which acts as the MVC controller.

4 Cocoa Reference

add-observer *Function*

Summary

Adds an observer to a notification center.

Package

`cocoa`

Signature

`add-observer` *target selector* **`&key`** *name object center*

Arguments

target⇓	A pointer to an Objective-C foreign object.
selector⇓	A selector of type **`sel`**.
name⇓	A string or **`nil`**.
object⇓	A pointer to an Objective-C foreign object or **`nil`**.
center⇓	A notification center.

Description

The function **`add-observer`** calls the Objective-C instance method **`"addObserver:selector:name:object:"`** of *center* to add *target* as an observer for *selector* with the given *name* and *object*, which both default to **`nil`**.

If *center* is omitted then it defaults to the default notification center.

See also

`remove-observer`

ns-not-found *Constant*

Summary

A constant similar to the Cocoa constant **`NSNotFound`**.

Package

`cocoa`

Description

The constant **`ns-not-found`** has the same value as the Cocoa Foundation constant **`NSNotFound`**.

ns-point *FLI Type Descriptor*

Summary

A foreign type for the Objective-C structure type **`NSPoint`**.

Package

`cocoa`

Syntax

`ns-point`

Description

The FLI type **`ns-point`** is a structure type for use as the Objective-C type **`NSPoint`**. The structure has two slots, **`:x`** and **`:y`**, both of foreign type **`:float`**.

When used directly in method definition or invocation, it allows automatic conversion to/from a vector of two elements of the form **`#(`***x y***`)`**.

See also

`ns-rect`
`set-ns-point*`

ns-range *FLI Type Descriptor*

Summary

A foreign type for the Objective-C structure type **`NSRange`**.

Package

`cocoa`

Syntax

`ns-range`

Description

The FLI type **`ns-range`** is a structure type for use as the Objective-C type **`NSRange`**. The structure has two slots, **`:location`** and **`:length`**, both of foreign type **`(:unsigned :int)`**.

When used directly in method definition or invocation, it allows automatic conversion to/from a cons of the form **`(`***location* **`.`** *length***`)`**.

See also

set-ns-range*

ns-rect *FLI Type Descriptor*

Summary

A foreign type for the Objective-C structure type **`NSRect`**.

Package

`cocoa`

Syntax

`ns-rect`

Description

The FLI type **`ns-rect`** is a structure type for use as the Objective-C type **`NSRect`**. The structure has two slots, **`:origin`** of foreign type **`ns-point`** and **`:size`** of foreign type **`ns-size`**.

When used directly in method definition or invocation, it allows automatic conversion to/from a vector of four elements of the form **`#(`***x y width height***`)`**.

See also

`ns-point`
`ns-size`
`set-ns-rect*`

ns-size *FLI Type Descriptor*

Summary

A foreign type for the Objective-C structure type **`NSSize`**.

Package

`cocoa`

Syntax

`ns-size`

Description

The FLI type **`ns-size`** is a structure type for use as the Objective-C type **`NSSize`**. The structure has two slots, **`:width`** and **`:height`**, both of foreign type **`:float`**.

When used directly in method definition or invocation, it allows automatic conversion to/from a vector of two elements of the form **#(***width height***)**.

See also

ns-rect
set-ns-size*

remove-observer *Function*

Summary

Removes an observer from a notification center.

Package

cocoa

Signature

remove-observer *target* **&key** *name object center*

Arguments

target⇓	A pointer to an Objective-C foreign object.
name⇓	A string or **nil**.
object⇓	A pointer to an Objective-C foreign object or **nil**.
center⇓	A notification center.

Description

The function **remove-observer** calls the Objective-C instance method **"removeObserver:name:object:"** of *center* to remove *target* as an observer with the given *name* and *object*, which both default to **nil**.

If *center* is omitted then it defaults to the default notification center.

See also

add-observer

set-ns-point* *Function*

Summary

Set the slots in a **ns-point** structure.

Package

cocoa

Signature

`set-ns-point*` *point x y* **`=>`** *point*

Arguments

point⇓	A pointer to a foreign object of type **`ns-point`**.
x⇓	A real.
y⇓	A real.

Values

point	A pointer to a foreign object of type **`ns-point`**.

Description

The function **`set-ns-point*`** sets the slots of the foreign **`ns-point`** structure pointed to by *point* to the values of *x* and *y*. *point* is returned.

See also

`ns-point`
`set-ns-rect*`

set-ns-range* *Function*

Summary

Set the slots in a **`ns-range`** structure.

Package

`cocoa`

Signature

`set-ns-range*` *range location length* **`=>`** *range*

Arguments

range⇓	A pointer to a foreign object of type **`ns-range`**.
location⇓	A positive integer.
length⇓	A positive integer.

Values

range	A pointer to a foreign object of type **`ns-range`**.

Description

The function **`set-ns-range*`** sets the slots of the foreign **`ns-range`** structure pointed to by *range* to the values of *location* and *length*. *range* is returned.

See also

ns-range

set-ns-rect* *Function*

Summary

Set the slots in a **`ns-rect`** structure.

Package

`cocoa`

Signature

`set-ns-rect*` *rect x y width height* **`=>`** *rect*

Arguments

rect⇓	A pointer to a foreign object of type **`ns-rect`**.
x⇓	A real.
y⇓	A real.
width⇓	A real.
height⇓	A real.

Values

rect	A pointer to a foreign object of type **`ns-rect`**.

Description

The function **`set-ns-rect*`** sets the slots of the foreign **`ns-rect`** structure pointed to by *rect* to the values of *x*, *y*, *width* and *height*. *rect* is returned.

See also

`ns-rect`
`set-ns-point*`
`set-ns-size*`

set-ns-size* *Function*

Summary

Set the slots in a **`ns-size`** structure.

Package

`cocoa`

Signature

`set-ns-size*` *size width height* **`=>`** *size*

Arguments

size⇓	A pointer to a foreign object of type **`ns-size`**.
width⇓	A real.
height⇓	A real.

Values

size	A pointer to a foreign object of type **`ns-size`**.

Description

The function **`set-ns-size*`** sets the slots of the foreign **`ns-size`** structure pointed to by *size* to the values of *width* and *height*. *size* is returned.

See also

`ns-size`
`set-ns-rect*`

5 Self-contained examples

This chapter enumerates the set of examples in the LispWorks library relevant to the content of this manual. Each example file contains complete, self-contained code and detailed comments, which include one or more entry points near the start of the file which you can run to start the program.

To run the example code:

1. Open the file in the Editor tool in the LispWorks IDE. Evaluating the call to **`example-edit-file`** shown below will achieve this.
2. Compile the example code, by **`Ctrl+Shift+B`**.
3. Place the cursor at the end of the entry point form and press **`Ctrl+X Ctrl+E`** to run it.
4. Read the comment at the top of the file, which may contain further instructions on how to interact with the example.

5.1 Example definitions

This file contains various example definitions used in this manual:

```
(example-edit-file "objc/manual")
```

5.2 Displaying Cocoa classes in CAPI windows

5.2.1 Using Web Kit to display HTML

This example demonstrates the use of **`capi:cocoa-view-pane`** containing a **`WebView`** from Apple's Web Kit and allowing an HTML page to be viewed:

```
(example-edit-file "objc/web-kit")
```

5.2.2 Showing a movie using NSMovieView

This example demonstrates the use of **`capi:cocoa-view-pane`** containing a **`NSMovieView`** and allowing a movie file to be opened and played:

```
(example-edit-file "objc/movie-view")
```

5.3 nib file example

This example connects a nib file (as generated by Apple's Interface Builder) to a Lisp implementation of an Objective-C class which acts as the MVC controller:

```
(example-edit-file "objc/area-calculator/area-calculator")
```

Use this script to build it as a standalone Cocoa application:

```
(example-edit-file "objc/area-calculator/deliver")
```

Index