metabang-bind user guide

Contents

Introduction

`bind` combines *let*, *destructuring-bind*, *multiple-value-bind* *and** a whole lot more into a single form. It has two goals:

1. reduce the number of nesting levels
2. make it easier to understand all of the different forms of destructuring and variable binding by unifying the multiple forms of syntax and reducing special cases.

`bind` is extensible. It handles the traditional multiple-values, destructuring, and let-forms as well as property-lists, classes, and structures. Even better, you can create your own binding forms to make your code cleaner and easier to follow (for others *and* yourself!).

Simple bindings are as in _let*_. Destructuring is done if the first item in a binding is a list. Multiple value binding is done if the first item in a binding is a list and the first item in the list is the keyword ':values'.

Summary

Some examples

Bind mimics let in its general syntax:

```
(bind (&rest bindings) <body>)
```

where each `binding` can either be an symbol or a list. If the binding is an atom, then this atom will be bound to nil within the body (just as in let). If it is a list, then it will be interpreted depending on its first form.

```
(bind (a
       (...))
  ...)
```

Bind as a replacement for let

You can use `bind` as a direct replacement for `let*`:

```
(bind ((a 2) b)
  (list a b))
=> (2 nil)
```

As in `let*`, atoms are initially bound to `nil`.

Bind with multiple-values and destructuring

Suppose we define two silly functions:

```
(defun return-values (x y)
  (values  x y))

(defun return-list (x y)
  (list x y))
```

How could we use bind for these:

```
(bind (((:values a b) (return-values 1 2))
       ((c d) (return-list 3 4)))
  (list a b c d))
=> (1 2 3 4)
```

Note that `bind` makes it a little easier to ignore variables you don't care about. Suppose I've got a function `ijara` that returns 3 values and I happen to need only the second two. Using `destructuring-bind`, I'd write:

```
(destructuring-bind (foo value-1 value-2)
    (ijira)
  (declare (ignore foo))
  ...)
```

With `bind`, you use `nil` or `_` in place of a variable name and it will make up temporary variables names and add the necessary declarations for you.

```
(bind (((_ value-1 value-2) (ijira)))
  ...)
```

Bind with property lists

A property-list or `plist` is a list of alternating keywords and values. Each keyword specifies a property name; each value specifies the value of that name.

```
(setf plist
  '(:start 368421722 :end 368494926 :flavor :lemon
    :content :ragged)
```

You can use `getf` to find the current value of a property in a list (and `setf` to change them). The optional third argument to `getf` is used to specify a default value in case the list doesn't have a binding for the requested property already.

```
(let ((start (getf plist :start 0))
      (end (getf plist :end))
      (fuzz (getf plist :fuzziness 'no)))
  (list start end fuzz))
=> (368421722 368494926 no)
```

The binding form for property-lists is as follows:

```
(:plist property-spec*)
```

where each property-spec is an atom or a list of up to three elements:

- atoms bind a variable with that name to a property with the same name (converting the name to a keyword in order to do the lookup).
- lists with a single element are treated like atoms.

- lists with two elements specify the variable in the first and the name of the property in the second.
- Lists with three elements use the third element to specify a default value (if the second element is #_, then the property name is taken to be the same as the variable name).

Putting this altogether we can code the above let statement as:

```
(bind (((:plist (start _ 0) end (fuzz fuzziness 'no))
  plist))
=> (list start end fuzz))
```

(which takes some getting used to but has the advantage of brevity).

Bind with structures

Structure fields are accessed using a concatenation of the structure's `conc-name` and the name of the field. Bind therefore needs to know two things: the conc-name and the field-names. The binding-form looks like

```
(:structure <conc-name> structure-spec*)
```

where each `structure-spec` is an atom or list with two elements:

- an atom specifies both the name of the variable to which the structure field is bound and the field-name in the structure.
- a list has the variable name as its first item and the structure field name as its second.

So if we have a structure like:

```
(defstruct minimal-trout
  a b c)

(setf trout (make-minimal-trout :a 2 :b 3 :c 'yes))
```

We can bind these fields using:

```
(bind (((:structure minimal-trout- (my-name a) b c)
        trout))
  (list my-name b c))
=> (2 3 yes)
```

Bind with classes

You can read the slot of an instance with an accessor (if one exists) or by using slot-value{footnote Note that if an accessor exists, it will generally be much faster than slot-value because CLOS is able to cache information about the accessor and the instance.}. Bind also provides two slot-binding mechanisms: `:slots` and `:accessors`. Both look the same:

```
(:slots slot-spec*)
(:accessors accessor-spec*)
```

Where both slot-spec and accessor-spec can be atoms or lists with two elements.

- an atom tells bind to use it as the name of the new variable *and* to treat this name as the name of the slot or the name of the accessor, respectively.
- If the specification is a list, then bind will use the first item as the variable's name and the second item as the slot-name or accessor.

Support we had a class like:

```
(defclass wicked-cool-class ()
  ((a :initarg :a :accessor its-a)
   (b :initarg :b :accessor b)
   (c :initarg :c :accessor just-c)))
```

If we don't mind using the slot-names as variable names, then we can use the simplest form of `:slots`:

```
(bind (((:slots a b c)
    (make-instance 'wicked-cool-class
                :a 1 :b 2 :c 3)))
  (list a b c))
==> (1 2 3)
```

We can also change the names within the context of our bind form:

```
(bind (((:slots a b (dance-count c))
    (make-instance 'wicked-cool-class
                :a 1 :b 2 :c 3)))
  (list a b dance-count))
==> (1 2 3)
```

Similarly, we can use `:accessors` with variable names that are the same as the accessor names...

```
(bind (((:accessors its-a b just-c)
    (make-instance 'wicked-cool-class
                :a 1 :b 2 :c 3)))
  (list its-a b just-c))
==> (1 2 3)
```

or that are different:

```
(bind (((:accessors (a its-a) b (c just-c))
    (make-instance 'wicked-cool-class
                :a 1 :b 2 :c 3)))
  (list a b c))
==> (1 2 3)
```

{anchor array-bindings}

Bind with arrays

Tamas Papp had the idea of letting `bind` handle arrays too. For example,

```
(bind ((#(a b c) #(1 2 3)))
  (list a b c))
==> (1 2 3)
```

One quick method definition and a few unit-tests later and bind does!

Bind with regular expressions

If you have CL-PPCRE or run with Allegro Common Lisp, you can use `bind` with regular expressions too. The syntax is

```
(:re expression &rest vars) string)
```

and will bind each grouped item in the expression to the corresponding var. For example:

```
(bind (((:re "(\\w+)\\s+(\\w+)\\s+(\\d{1,2})\\.(\\d{1,2})\\.(\\d{4})"
         fname lname nil month year) "Frank Zappa 21.12.1940"))
  (list fname lname month year))
```

The body of bind form will be evaluated even if the expression does not match.

Bind with `flet` and `labels`

Bind can even be used as a replacement for `flet` and `labels`. The syntax is

```
(:flet function-name (arguments*)) definition)

(:labels function-name (arguments*)) definition)
```

for example:

```
(bind (((:flet square (x)) (* x x)))
  (square 4))
==> 16

(bind (((:labels my-oddp (x))
   (cond ((<= x 0) nil)
     ((= x 1) t)
     (t (my-oddp (- x 2))))))
    (my-oddp 7))
==> t
```

Note that bind currently expands each binding-form into a new context. In particular, this means that

```
(bind (((:flet x (a)) (* a 2))
       ((:flet y (b)) (+ b 2)))
  ...)
```

expands as

```
(flet ((x (a) (progn (* a 2))))
  (flet ((y (b) (progn (+ b 2))))
    ...))
```

rather than

```
(flet ((x (a) (progn (* a 2)))
       (y (b) (progn (+ b 2))))
    ...)
```

Generally speaking, this shouldn't make much of a difference.

`bind` and declarations

`bind` handles declarations transparently by splitting them up and moving them to the correct place in the expansion. For example

```
(bind (((:values a b) (foo x))
       (#(d e) (bar y)))
  (declare (type fixnum a d)
       (optimize (speed 3)))
  (list a b d e))
```

becomes

```
(multiple-value-bind (a b)
    (foo x)
  (declare (type fixnum a) (optimize (speed 3)))
  (let ((#:values-258889 (bar y)))
    (let* ((d (row-major-aref #:values-258889 0))
           (e (row-major-aref #:values-258889 1)))
      (declare (optimize (speed 3)))
      (list a b d e))))
```

because `bind` knows to keep the variable declarations (like `type`) with their variables and to repeat other declarations (like `optimize`) at each level.

`bind` keeps track of variables declarations that are not used. The configuration variable `*unused-declarations-behavior*` controls what `bind` does.

More bindings

Since bind is extensible and I'm fallible, there are probably things bind can do that haven't made it into this guide. Use the following commands to see what bind can do:

{docs binding-forms}

{docs binding-form-docstring}

{docs binding-form-groups}

{docs binding-form-synonyms}

lambda-bind

Eric Schulte contributed `lambda-bind` (note, he called it `lambdab` but I dislike abbreviations so...):

{docs lambda-bind}

Extending bind yourself

Bind's syntax is extensible: the work for each binding-specification is handled by a generic function. This means that you can evolve bind to fit your program for whatever sort of data-structure makes sense for you. To make a binding form, you can either define a method for `bind-generate-bindings` or you can use the `defbinding-form` macro.

{docs bind-generate-bindings}

{docs defbinding-form}

There are many more examples included in the source code.