
Style Guide Documentation

Release 1.0

Dylan Hackers

April 29, 2012

CONTENTS

1 Purpose of this document	3
2 Controversial comments	5
3 80 columns law	7
4 Consistency	9
5 Naming	11
6 Dot notation	13
7 Symbols versus keywords	15
8 End words	17
9 Semicolons	19
10 General indentation	21
11 Operators on newline	23
11.1 Calls	23
12 if then else	25
13 let	27
14 select and case	29
15 Macros	31
16 for loop	33
17 Local methods	35
18 Parameter lists	37
19 Return values	39
20 Method constants	41
21 Generic function definitions	43

- Purpose of this document
- Controversial comments
- 80 columns law
- Consistency
- Naming
- Dot notation
- Symbols versus keywords
- End words
- Semicolons
- General indentation
- Operators on newline
- if then else
- let
- select and case
- Macros
- for loop
- Local methods
- Parameter lists
- Return values
- Method constants
- Generic function definitions
- Class definitions

PURPOSE OF THIS DOCUMENT

This document describes a coding style that the Dylan group recommends. There are still some areas of disagreement, and there is still room to change this style guide. In some places the Dylan book group decided to use a different style, which is noted in this guide.

CONTROVERSIAL COMMENTS

- scott thinks that matching definition names (e.g., *end class <bolt>*) are a bad idea because they are hard to maintain in the face of name changes.
- keith likes long case slot specifications always, but jonathan is willing to abbreviate in groups.
- keith thinks its ok to have parameters on same line as function name with return values on the next line (but jonathan disagrees). Dylan book does not split a line between the parameter list and return values.
- haahr claims that there is no space after function name in a function definition. we need to chase this up. Dylan book has a space after function name in a function definition.
- scott uses % prefixes on slot accessors that have short (or “unqualified” names) that are not part of the exported API. Dylan book does not do this.
- 6 character rule seems like overkill (but jonathan thinks that it does help in practice but is willing to get rid of it to simplify the pretty printing rules).
- _ is a weird notation, but does have precedent in syntax-case and prolog for dont care. keith likes the return values to be named for better documentation. Dylan book names return values.
- tucker disagrees with consistency rule. he thinks that there should be a consistent style on where you break and how you indent, but he disagrees that that means you have to break there on all the similar nearby statements, even when they don’t need it.

80 COLUMNS LAW

Lines should not exceed 80 columns in length.

CONSISTENCY

Formatting style should be consistent.

A broader pattern I tend to go for is consistency of indentation within a group of forms. If the shape of one or two forms in a group force a long-form indentation I'll often reformat the rest to give them a consistent look. For example, here is a sequence of `let` bindings.

```
let abigmobynamewithabiginitialization
  = 123456789 * 123456789 * 123456789 * 123456789;

let a-tiny-name
  = 123456789;
```

Here is another examples of a group, a bunch of generic function definitions forming a protocol:

```
//// Match protocol.
define generic match
  (pattern, fragment, more-patterns, more-fragments, env, fail) => ();

define generic match-empty
  (pattern, more-patterns, env, fail) => ();

define generic folds-separator?
  (pattern) => ();
```


NAMING

- full names
- lowercase
- angle bracket notation for types
- slot-names (keith)

There isn't a recommended Dylan style, although the majority of Dylan code I've seen from outside Harlequin just uses the unqualified name of the property as the slot name.

However, previous discussions here, apart from provoking complaints about the way Dylan works in this area, seem to have suggested that qualifying slot names is the right thing in most cases in order to avoid widespread problems with name clashes later on (at least for exported names). That's not to say it's caught on yet...

I do think the pure Lisp approach is very bad indeed though, and shouldn't be used. I disagree that it makes the uses of accessors more intelligible, particularly if you're calling accessors from different superclasses on the same object in a block of code. And, as you say, it rather breaks abstraction of implementation.

One possible compromise is to choose a prefix for a whole group of classes beneath a given root. For example, the compiler's intermediate representation consists of a number of different graph nodes and we could choose a blanket prefix for slots of classes in that heterarchy such as "dfm-" (our IR is known as the DFM).

Dylan book style: We will discuss this issue in book, but use shorter names. They won't be exported, generally, so it won't be an issue.

DOT NOTATION

Use for stateless property accessors that return a single value.

I now tend to use dot notation quite widely for any logical “property access”, even if computed. That is, `foo.size` is acceptable but the imperative `foo.initialize` and `foo.close` aren’t for me.

SYMBOLS VERSUS KEYWORDS

Use keywords only for keyword parameters. Do this:

```
make(<file-stream>, direction: #"input");
```

instead of this:

```
make(<file-stream>, direction: input:);
```

It's reasonable to use keyword syntax to specify a received keyword, for example in a slot specification or in a parameter list:

```
slot point-x, init-keyword: x;
```


END WORDS

“End words” are the optional text that follows `end` in statements. End words should always be used for top-level definitions, but otherwise it is up to the programmer. This section provides some guidelines for how to decide when to use them and when not to. If in doubt, err on the side of using them.

Generally speaking, end words should be used if the beginning of the block they terminate is more than about 15 lines away. They become more useful the more deeply nested the code is. Sometimes this might indicate a need to break the code down into multiple (possibly local) functions.

Pros:

- The compiler warns when end words don’t match. This could alert the programmer to mistakes in nesting.
- When reading source code in a flat file, the end word gives more context by telling you what the previous definition is.

Cons:

- The compiler warns when end words don’t match. This sometimes results in otherwise unnecessary maintenance.
- End words increase verbosity of the code.

SEMICOLONS

Last expression can go without semicolon only where the value is used. This is actually a useful little practice since if you want to add a form to the end of a body whose value is significant you're forced to think a little more.

```
define method empty? (vector :: <vector>)
  vector.size = 0
end method empty?;

define method add (vector :: <vector>, object)
  let new-vector :: <vector>
    = make(vector.class-for-copy, size: vector.size + 1);
  replace-subsequence!(new-vector, vector);
  new-vector[vector.size] := object;
  new-vector
end method add;
```


GENERAL INDENTATION

Avoid boxing your code and having big right column:

No:

```
define method yukyukyukyukyukyukyuk (blahblahblahblah :: <foo>,
                                     tolosetrack :: <bar>,
                                     concerned? :: <boolean>)
  ...
end method yukyukyukyukyukyukyuk;
```

Yes:

```
define method yukyukyukyukyukyukyuk
  (blahblahblahblah :: <foo>, tolosetrack :: <bar>,
   concerned? :: <boolean>)
  ...
end method yukyukyukyukyukyukyuk;
```

Use two space indentation:

```
begin
  tell-da-world(bigfish, smallpond);
  world
end
```


OPERATORS ON NEWLINE

In long expressions where line breaks are necessary, put operators on a new line and indent two spaces:

```
supercalifragilisticxpealidocious
  | wasthatashovelfull
  | ofraisensorsyrup

superfragilisticxpealidoscious
  := somereallylongexpressionthatdoesnotfitabove;

define variable lilgirlscryalldatime
  = bigboysdontcry;

let superfragilisticxpealidoscious
  = someexpressionthatclearlydoesnotfitabove;
```

11.1 Calls

Usually is on same line with arguments single spaced and no space between the function and its argument list:

Yes:

```
funkie(a, b, c);

longfunkiefunctionnamesuperfraligistic(a, b, c);
```

Function name up to 6 characters keep parens on same line:

```
values(0,
  sequence.size,
  sequence-next-state,
  sequence-finished-state?,
  sequence-current-key,
  stretchy-vector-current-element,
  stretchy-vector-current-element-setter,
  identity-copy-state)
```

Function name more than 6 characters break to newline:

```
redirect-computations!
  (old-c, new-c, previous-computations, next-computations);
```

More arguments:

```
redirect-computations!  
(old-c, new-c, previous-computations, next-computations,  
  areallylongidthatrequireswrappingtheargs);
```

IF THEN ELSE

General case:

```
if (expr)
  then statements ...
else
  else statements ...
end if;
```

Abbreviated use:

```
if (expr) x else y end;
```


LET

let statements should generally have the smallest scope necessary. They do not increase the indentation level:

```
let x = xxxxx;  
let y = yyyyy;  
let z = f!(x, y);  
inc!(x, z);  
z + z;
```


SELECT AND CASE

The aligned =>'s help make the cases stand out:

```
case
  count > 0 & test(item, target)
    => grovel(count - 1, src-index + 1, dst-index);
  otherwise
    => vector[dst-index] := item;
    grovel(count, src-index + 1, dst-index + 1)
end case;
```

Abbreviated use:

```
case
  *blue?*   => 2;
  *yellow?* => 3;
end case;
```

Long expression:

```
select (supercalifragilisticexbealidocious
  + someexpressionthatclearlydoesnotfitabove)
  1 => 2;
  2 => 3;
end select;
```


MACROS

```
define macro collecting
  { collecting () ?body end }
  => { collecting (_collector)
      ?body;
      collected(_collector)
      end }
  { collecting (as ?expression) ?body end }
  => { collecting (_collector as ?expression)
      ?body;
      collected(_collector)
      end }
  { collecting (?vars) ?body end }
  => { ?vars;
      ?body }
vars:
  { ?var, ... }
  => { ?var; ... }
  { }
  => { }
end macro;
```


FOR LOOP

Put each iteration clause on a line by itself:

```
for (elementincollectionnumberone in collection1,  
     elementincollectionnumbertwo in collection2)  
    ...  
end for
```

If the iteration clauses are utterly trivial, they may be on one line:

```
for (f in foo, b in bar)  
    ...  
end for
```


LOCAL METHODS

```
method (y)
  local method strip (x)
    ...
  end method strip,
  method chars (x)
    ...
  end method chars;
  strip(chars(y))
end method;
```

Tight for space:

```
method (y)
  local
    method strip (x)
      ...
    end method strip,
    method chars (x)
      ...
    end method chars;
  strip(chars(y))
end method;
```

Abbreviated use:

```
method (y)
  local strip (x) ... end,
    chars (x) ... end;
  strip(chars(y))
end method;
```

A single recursive method:

```
method (y)
  local stripchars (x)
    ...
  end;
  stripchars(y)
end method;
```


PARAMETER LISTS

Right after function name:

```
define method vector (#rest rest)
  rest
end method vector;
```

Indentation, style A:

```
define method union
  (seq-1 :: <sequence>, seq-2 :: <sequence>, #key test = \==)
  remove-duplicates(concatenate(seq-1, seq-2), test: test)
end method union;
```

Optional parameters: Use the same aesthetic applied to indenting operators continued across lines, indent #key names as follows:

```
define method print
  (object :: <multiple-value-combination>,
   #key stream = *standard-output*, verbose? :: <boolean> = #t,
   depth :: false-or(<integer>))
  ...
end method print;
```


RETURN VALUES

No semicolon.

Parenthesis notation.

If both parameter list and return values fit on the first line:

```
define method past? (time :: <offset>) => (result :: <boolean>)
  time.total-seconds < 0;
end method past?;
```

If parameter list and return values do not both fit on the first line:

```
define method element-setter
  (new-value, list :: <list>, key :: <small-integer>) => (new-value)
end method element-setter;
```

If parameter list and return values do not both fit on the same line:

```
define method decode-total-seconds
  (time :: <time-of-day>)
  => (hour :: <integer>, min :: <integer>, sec :: <integer>)
  decode-total-seconds(time.total-seconds);
end method decode-total-seconds;

define method convert-expressions
  (env :: <environment>, argument-forms)
  => (first :: <computation>, last :: <computation>, temporaries)
end method convert-expressions;
```

Optional parameters split across a line:

```
define method fill!
  (sequence :: <mutable-sequence>, value :: <object>,
   #key start: first = 0, end: last)
  => (sequence :: <mutable-sequence>)
end method fill!;
```

Complicated cases

The following is preferred:

```
define method \<
  (a :: <double-float>, b :: <ratio>) => (res :: <boolean>)
  a < as(<double-float>, b)
end method \<;
```

Over this:

```
define method \< (a :: <double-float>, b :: <ratio>)
  => (res :: <boolean>)
  a < as(<double-float>, b)
end method \<;
```

Use other return value name to convey more meaning if possible.

```
define method reverse! (list :: <list>) => (list :: <list>)
  ...
end method reverse!;

define generic munge (list :: <list>) => (new-list :: <list>);

define generic munge! (list :: <list>) => (list :: <list>);
```

Use `_` for poetry impaired or where the function name corresponds exactly to the return value name

```
define method first (s :: <sequence>, #rest keys, #key default) => (_)
  ...
end method first;
```

METHOD CONSTANTS

```
define constant curry
  = method (...) => (...)
  ...
end method;
```


GENERIC FUNCTION DEFINITIONS

```
define open generic choose
  (pred :: <function>, seq :: <sequence>) => (elts :: <sequence>);

define open generic choose-by
  (pred :: <function>, test-seq :: <sequence>, val-seq :: sequence)
=> (_ :: <sequence>);
```


CLASS DEFINITIONS

Lots of direct superclasses:

```
define class <z>
  (<a>, <b>, <c>)
  ...
end class <z>;
```

Long slot initializations:

```
define class <entry-state> (<temporary>)
  slot name, init-keyword: name;;
  slot me-block, init-keyword: block;;
  slot exits :: <stretchy-vector> = make(<stretchy-vector>),
    init-keyword: exits;;
end class;
```