Style Guide Documentation

Release 1.0

Dylan Hackers

December 10, 2018

CONTENTS

CHAPTER

ONE

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.

CHAPTER

TWO

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 precendent 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.

CHAPTER

THREE

LINE LENGTH

Lines should be kept to a *reasonable* length. Small monitors are rare these days so a strict 80 column rule is no longer necessary, even considering side-by-side diffs or split screen editing. However, please try to keep lines short in general, unless it requires unnatural contortions, because it is often easier to read that way. Use your judgement.

CHAPTER

FOUR

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) => ();
```

CHAPTER

FIVE

NAMING

Naming is hard. There is no single rule that works for everything. Here are some rules that should always be followed:

- Follow the naming conventions in the Dylan Reference Manual.
- Use lowercase, not uppercase or mixed case.

  Example: `join-segments` not `JoinSegments` or `JOIN-SEGMENTS`
- Use dash (hyphen, -) to separate words in a name, not underscore (_).

  Example: `run-jobs` not `run_jobs`
- Prefix a name with percent (e.g., `%do-scary-stuff`) to indicate an "internal" function. This roughly signals to the caller "you'd better know what you're doing".

Here are some hints for naming things in Dylan. These are guidelines only and need not be strictly followed:

- Within each Dylan module there is a single namespace for all bindings, whether they're variables, functions, constants, classes, or macros, so full names are to be preferred.
- Use verb-noun to name functions. Slot names are a notable exception to this rule. See next item.
- Naming slots poses some special challenges, perhaps best explained with an example. This might be the naive implementation of an abstract `<request>` class for a high-level networking library:

```
define abstract class <request> (<object>)
  slot client, init-keyword: client:;
  slot time-received, init-keyword: time-received:;
  ...
end;
```

  The problem is that both "client" and "time-received" are fairly reasonable names for local variables so they could easily be shadowed accidentally. Also, they're probably too short and general to be exported. Common practice would be to do something like this instead:

```
define abstract class <request> (<object>)
  slot request-client, init-keyword: client:;
  slot request-time-received, init-keyword: time-received:;
  ...
end;
```

  This leads to code such as

```
request.request-client := ...;
foo(request.request-time-received);
```

  which may look odd at first due to the duplication of "request", but this is an accepted pattern.

  (Note that this pattern may not work for mixin classes, but there is likely a better name anyway in such cases.)

There's an additional wrinkle when a subclass gets involved:

```
// Bad (breaks abstraction)
define class <http-request> (<request>)
  slot http-request-headers, init-keyword: headers:;
  ...
end;
```

Note that naming the slot `http-request-headers` would break abstraction because the caller now has to know which slots are in `<request>` and which are in `<http-request>` and prefix them appropriately. So instead it is better to use the same prefix for a whole group of classes, in this case "request-":

```
// Good (consistent prefix)
define class <http-request> (<request>)
  slot request-headers, init-keyword: headers:;
  ...
end;
```

- Use a plural noun to name variables bound to collections.

  Example: `*cats*`

- Do not include the type in the name. This way it won't be necessary to change the name if the implementation type changes.

  Example: `*frobnoids*` not `*frobnoid-list*`

CHAPTER

SIX

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.

CHAPTER

SEVEN

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:;
```

CHAPTER

EIGHT

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.

CHAPTER

NINE

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;
```

CHAPTER

TEN

GENERAL INDENTATION

Avoid boxing your code and having big right column:

No:

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

Yes:

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

Use two space indentation:

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

CHAPTER

ELEVEN

OPERATORS ON NEWLINE

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

```
supercalifragilisticexpealidocious
  | wasthatashovelfull
  | ofraisensorsyrup

superfragilisticespealidoscious
  := somereallylongexpressionthatdoesnotfitabove;

define variable lilgirlscryalldatime
  = bigboysdontcry;

let superfragilisticespealidoscious
  = someexpressionthatclearlydoesnotfitabove;
```

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);
```

CHAPTER

TWELVE

IF THEN ELSE

General case:

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

Abbreviated use:

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

CHAPTER

THIRTEEN

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;
```

CHAPTER

FOURTEEN

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;
```

CHAPTER

FIFTEEN

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;
```

CHAPTER

SIXTEEN

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
```

CHAPTER

SEVENTEEN

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;
```

CHAPTER

EIGHTEEN

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;
```

CHAPTER

NINETEEN

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;
```

CHAPTER

TWENTY

METHOD CONSTANTS

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

CHAPTER

TWENTYONE

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>);
```

CHAPTER

TWENTYTWO

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;
```